MEAP Unedited Draft

Erlang AND OTP IN ACTION

Martin Logan
Eric Merritt
Richard Carlsson

MANNING

MEAP Edition
Manning Early Access Program

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1 The foundations of Erlang/OTP

Welcome to our book about Erlang and OTP in action! You probably know already that Erlang is a programming language—and as such it is pretty interesting in itself—but our focus here will be on the practical and the "in action", and for that we also need the *OTP framework*. This is always included in any Erlang distribution, and is actually such an integral part of Erlang these days that it is hard to say where the line is drawn between OTP and the plain standard libraries; hence, one often writes "Erlang/OTP" to refer to either or both.

But why should we learn to use the OTP framework, when we could just hack away, rolling our own solutions as we go? Well, these are some of the main points of OTP:

Productivity
Using OTP makes it possible to produce production-quality systems in very short time.

Stability
Code written on top of OTP can focus on the logic, and avoid error prone re-implementations of the typical things that every real-world system will need: process management, servers, state machines, etc.

Supervision
The application structure provided by the framework makes it simple to supervise and control the running systems, both automatically and through graphical user interfaces.

Upgradability
The framework provides patterns for handling code upgrades in a systematic way.

Reliable code base
The code for the OTP framework itself is rock-solid and has been thoroughly battle tested.

Despite these advantages, it is probably true to say that to most Erlang programmers, OTP is still something of a secret art, learned partly by osmosis and partly by poring over the more impenetrable sections of the documentation. We would like to change this. This is to our knowledge the first book focused on learning to use OTP, and we want to show that it can be a much easier experience than you might think. We are sure you won't regret it.

In this first chapter, we will present the core features on which Erlang/OTP is built, and that are provided by the Erlang programming language and run-time system:

- Processes and concurrency
- Fault tolerance
- Distributed programming
- Erlang's core functional language

The point here is to get you acquainted with the thinking behind all the concrete stuff we'll be diving into from chapter 2 onwards, rather than starting off by handing you a whole bunch of facts up front. Erlang is different, and many of the things you will see in this book will take some time to get accustomed to. With this chapter, we hope to give you some idea of why things work the way they do, before we get into technical details.

1.1 – Understanding processes and concurrency

Erlang was designed for *concurrency*—having multiple tasks running simultaneously—from the ground up; it was a central concern when the language was designed. Its built-in support for concurrency, which uses the *process* concept to get a clean separation between tasks, allows us to create fault tolerant architectures and fully utilize the multi-core hardware that is available to us today.

Before we go any further, we should explain exactly what we mean by the words "process" and "concurrency".

1.1.1 – Processes

Processes are at the heart of concurrency. A *process* is the embodiment of an ongoing activity: an agent that is running a piece of program code, concurrent to other processes running their own code, at their own pace.

They are a bit like people: individuals, who don't share things. That's not to say that people are not generous, but if *you* eat food, *I* don't get full, and furthermore, if you eat *bad* food, *I* don't get sick from it. You have your own brain and internals that keep you thinking and living independently of what I do. This is how processes behave; they are separate from one another and are guaranteed not to disturb one another through their own internal state changes.

Figure illustrating processes running their own code (some running the same code, at different points)

A process has its own working memory and its own mailbox for incoming messages. Whereas *threads* in many other programming languages and operating systems are concurrent activities that share the same memory space (and have countless opportunities to step on each other's toes), Erlang's processes can safely work under the assumption that nobody else will be poking around and changing their data from one microsecond to the next. We say that *processes encapsulate state*.

PROCESSES: AN EXAMPLE

Consider a web server: it receives requests for web pages, and for each request it needs to do some work that involves finding the data for the page and either transmitting it back to the place the request came from (sometimes split into many chunks, sent one at a time), or replying with an error message in case of failure. Clearly, each request has very little to do with any other, but if the server accepted only one at a time and did not start handling the next request until the previous was finished, there would quickly be thousands of requests on queue if the web site was a popular one.

If the server instead could start handling requests as soon as they arrived, each in a separate process, there would be no queue and most requests would take about the same time from start to finish. The state encapsulated by each process would then be: the specific URL for the request, who to reply to, and how far it has come in the handling as yet. When the request is finished, the process disappears, cleanly forgetting all about the request and recycling the memory. If a bug should cause one request to crash, only that process will die, while all the others keep working happily.

Figure illustrating the web server processes example?

When Erlang was invented, its focus was on handling phone calls; these days, it's mostly Internet traffic, but the principles are the same.

THE ADVANTAGES OF ERLANG-STYLE PROCESSES

Because processes cannot directly change each other's internal state, it is possible to make significant advances in error handling. No matter how bad code a process is running, it cannot corrupt the internal state of your other processes. Even at a fine-grained level within your program, you can have the same isolation that you see between the web browser and the word processor on your computer desktop. This turns out to be very, very powerful, as we will see later on when we talk about process supervision.

Since processes can share no internal data, they must communicate by copying. If one process wants to exchange information with another, it sends a message; that message is a read-only copy of the data that the sender has. These fundamental semantics of message passing make *distribution* a natural part of Erlang. In real life, you can't share data over the wire—you can only copy it. Erlang process communication always works as if the receiver gets a personal copy of the message, even if the sender happens to be on the same computer—this means that network programming is no different from coding on a single machine!

This *transparent distribution* allows Erlang programmers to look at the network as simply a collection of resources—we don't much care about whether process X is running on a different machine than process Y, because they are going to communicate in the exact same way no matter where they are located.

1.1.2 – Concurrency explained

So what do we really mean by "concurrent"? Is it just another word for "in parallel"? Well, almost but not exactly, at least when we're talking about computers and programming.

One popular semi-formal definition reads something like "those things that don't have anything that forces them to happen in a specific order are said to be concurrent". For example, given the task to sort two packs of cards, you could sort one first, and then the other, or if you had some extra arms and eyes you could sort both in parallel. There is nothing that requires you to do them in a certain order; hence, they are concurrent tasks, they can be done in either order, or you can jump back and forth between the tasks until they're both done, or, if you have the extra appendages (or perhaps someone to help you), you can perform them simultaneously in true parallel fashion.

Figure showing concurrent vs. order-constrained tasks

This may sound strange: shouldn't we say that tasks are concurrent only if they are actually happening at the same time? Well, the point with that definition is that they *could* happen at the same time, and we are free to schedule them at our convenience. Tasks that *need* to be done simultaneously together are not really separate tasks at all. Some tasks, though, are separate but non-concurrent and must be done in order, such as breaking the egg before making the omelet.

One of the really nice things that Erlang does for you is that it helps you with the physical execution: if there are extra CPUs (or cores or hyperthreads) available, it will use them to run more of your concurrent processes in parallel—if not, it will use what CPU power there is to do them all a bit at a time. You will not need to think about such details, and your Erlang programs automatically adapt to different hardware—they just run more efficiently if there are more CPUs, as long as you have things lined up that can be done concurrently.

Figure showing Erlang processes running on a single core and on a multicore machine

But what if your tasks are not concurrent? If your program must first do X, then Y, and finally Z? Well, that is where you need to start thinking about the real dependencies in the problem you are out to solve. Perhaps X and Y can be done in any order as long as it is before Z. Or perhaps you can start working on a part of Z as soon as parts of X and Y are done. There is no simple recipe, but surprisingly often a little bit of thinking can get you a long way, and it gets easier with experience.

Rethinking the problem in order to eliminate unnecessary dependencies can make the code run more efficiently on modern hardware. However, that should usually be your second concern. The most important effect of separating parts of the program that don't really need to be together will be that it makes your code less confused, more readable, and allows you to focus on the real problems rather than on the mess that follows from trying to do several things at once. This means higher productivity and fewer bugs.

1.1.3 – Programming with processes in Erlang

When you build an Erlang program you say to yourself, "what activities here are concurrent; can happen independently of one another?" Once you sketch out an answer that question, you can start building a system where every single instance of those activities you identified becomes a separate process.

In contrast to most other languages, concurrency in Erlang is very cheap. Spawning a process is about as much work as allocating an object in your average object-oriented

language. This can take some getting used to in the beginning, because it is such a foreign concept! Once you do get used to it however, magic starts to happen. Picture a complex operation that has six concurrent parts, all modeled as separate processes. The operation starts, processes are spawned, data is manipulated, a result is produced, and at that very moment the processes involved simply disappear magically into oblivion, taking with them their internal state, database handles, sockets, and any other stuff that needs to be cleaned up that you don't want to have to do manually.

Figure of processes being set up, running, and disappearing

In the rest of this section we are going to take a brief look at the characteristics of processes. We will show how quick and easy it is to start them, how lightweight they are, and how simple it is to communicate between them. This will enable us to talk in more detail about what you can really do with them and how they are the basis of the fault tolerance and scalability that OTP provides.

1.1.4 – Creating a process: "spawning"

Erlang processes are *not* operating system "threads". They are much more lightweight, implemented by the Erlang run-time system, and Erlang is easily capable of spawning hundreds of thousands of processes on a single system running on commodity hardware. Each of these processes is separate from all the other processes in the run-time system; it shares no memory with the others, and in no way can it be corrupted by another process dying or going berserk.

A typical thread in a modern operating system reserves some megabytes of address space for its stack (which means that a 32-bit machine can never have more than about a thousand simultaneous threads), and it still crashes if it happens to use more stack space than expected. Erlang processes, on the other hand, start out with only a couple of hundred bytes of stack space each, and they grow or shrink automatically as required.

Figure illustrating lots of Erlang processes?

The syntax for starting processes is quite straightforward, as illustrated by the following example. We are going to spawn a process whose job is to execute the function call `io:format("erlang!")` and then finish, and we do it like this:

```
spawn(io, format, ["erlang!"])
```

That's all. (Although the `spawn` function has some other variants, this is the simplest.) This will start a separate process which will print the text "erlang!" on the console, and then quit.

In chapter 2 we will get into details about the Erlang language and its syntax, but for the time being we hope you will simply be able to get the gist of our examples without further explanation. One of the strengths of Erlang is that it is generally pretty easy to understand the code even if you've never seen the language before. Let's see if you agree.

1.1.5 – How processes talk

Processes need to do more than spawn and run however—they need to communicate. Erlang makes this communication quite simple. The basic operator for sending a message is `!`, pronounced "bang", and it is used on the form "Destination ! Message". This is message passing at its most primitive, like mailing a postcard. OTP takes process communication to another level, and we will be diving into all that is good with OTP and messaging later on, but for now, let's marvel at the simplicity of communicating between two independent and concurrent processes illustrated in the following snippet..

```
run() ->
    Pid = spawn(fun ping/0),
    Pid ! self(),
    receive
        pong -> ok
    end.

ping() ->
    receive
        From -> From ! pong
    end.
```

Take a minute and look at the code above. You can probably understand it without any previous knowledge of Erlang. Points worth noting are: another variant of the `spawn` function, that here gets just a single reference to "the function named `ping` that takes zero arguments" (`fun ping/0`); and also the function `self()` that produces the identifier of the current process, which is then sent on to the new process so that it knows where to reply.

That's it in a nutshell: process communication. Every call to `spawn` yields a fresh process identifier that uniquely identifies the new child process. This process identifier can then be used to send messages to the child. Each process has a "process mailbox" where incoming messages are stored as they arrive, regardless of what the process is currently busy doing, and are kept there until it decides to look for messages. The process may then search and

retrieve messages from this mailbox at its convenience using a `receive` expression, as shown in the example (which simply grabs the first available message).

In this section we told you that processes are independent of one another and cannot corrupt one another because they do not share anything. This is one of the pillars of another of Erlang's main features: fault tolerance, which is a topic we will cover in some more detail in the next section.

1.2 – Erlang's fault tolerance infrastructure

Fault tolerance is worth its weight in gold in the real world. Programmers are not perfect, nor are requirements. In order to deal with imperfections in code and data, just like aircraft engineers deal with imperfections in steel and aluminum, we need to have systems that are fault tolerant, that are able to deal with mistakes and do not go to pieces each time an unexpected problem occurs.

Like many programming languages, Erlang has *exception handling* for catching errors in a particular piece of code, but it also has a unique system of *process links* for handling process failures in a very effective way, which is what we're going to talk about here.

1.2.1 – How process links work

When an Erlang process dies unexpectedly, an *exit signal* is generated. All processes that are *linked* to the dying process will receive this signal. By default, this will cause the receiver to exit as well and propagate the signal on to any other processes it is linked to, and so on, until all the processes that are linked directly or indirectly to each other have exited. This cascading behavior allows us to have a group of processes behave as a single application with respect to termination, so that we never need to worry about finding and killing off any left-over processes before we can restart that entire subsystem from scratch.

Previously, we mentioned cleaning up complex state through processes. This is basically how it happens: a process encapsulates all its state and can therefore die safely without corrupting the rest of the system. This is just as true for a group of linked processes as it is for a single process. If one of them crashes, all its collaborators also terminate, and all the complex state that was created is snuffed out of existence cleanly and easily, saving programmer time and reducing errors.

Instead of thrashing around desperately to save a situation that you probably will not be able to fix, the Erlang philosophy is "let it crash"—you just drop everything cleanly without

affecting the rest of your code and start over, logging precisely where things went pear-shaped and how. This can also take some getting used to, but is a powerful recipe for fault tolerance and for creating systems that are possible to debug despite their complexity.

1.2.2 – Supervision and the trapping of exit signals

One of the main ways fault tolerance is achieved in OTP is by overriding the default propagation of exit signals. By setting a *process flag* called `trap_exit`, we can make a process *trap* any incoming exit signal rather than obey it. In this case, when the signal is received, it is simply dropped in the process' mailbox as a normal message on the form `{'EXIT', Pid, Reason}` that describes in which other process the failure originated and why, allowing the trapping process to check for such messages and take action.

Such a signal trapping process is sometimes called a *system process*, and will typically be running code that is very different from that run by ordinary *worker* processes, which do not usually trap exit signals. Since a system process acts as a bulwark that prevents exit signals from propagating further, it insulates the processes it is linked to from each other, and can also be entrusted with reporting failures and even restarting the failed subsystems. We call such processes *supervisors*.

Figure illustrating supervisor, workers, and signals

The point of letting an entire subsystem terminate completely and be restarted is that it brings us back to a state known to function properly. Think of it like rebooting your computer: a way to clear up a mess and restart from a point that ought to be working. But the problem with a computer reboot it is that it is not granular enough. Ideally, what you would like to be able to do is reboot just a part of the system, and the smaller, the better. Erlang process links and supervisors provide a mechanism for such fine-grained "reboots".

If that was all, though, we would still be left to implement our supervisors from scratch, which would require careful thought, lots of experience, and a long time shaking out the bugs and corner cases. Fortunately for us, the OTP framework already provides just about everything we need: both a methodology for structuring our applications using supervision, and stable, battle-hardened libraries to build them on.

OTP allows processes to be started by a supervisor in a prescribed manner and order. A supervisor can also be told how to restart its processes with respect to one another in the event of a failure of any single process, how many attempts it should make to restart the

processes within a certain period of time before it ought to give up, and more. All you need to do is to provide some parameters and hooks.

But a system should not be structured as just a single-level hierarchy of supervisors and workers. In any complex system, you will want a *supervision tree*, with multiple layers, that allows subsystems to be restarted at different levels.

1.2.3 – Layering processes for fault tolerance

Layering brings related subsystems together under a common supervisor. More importantly, it defines different levels of working base states that we can revert to. In the diagram below, you can see that there are two distinct groups of worker processes, A and B, supervised separately from one another. These two groups and their supervisors together form a larger group C, under yet another supervisor higher up in the tree.

Figure illustrating a layered system of supervisors and workers

Let's assume that the processes in group A work together to produce a stream of data that group B consumes. Group B is however not required for group A to function. Just to make things concrete, let's say group A is processing and encoding multimedia data, while group B presents it. Let's further suppose that a small percent of the data entering group A is corrupt in some way not predicted at the time the application was written.

This malformed data causes a process within group A to malfunction. Following the "let it crash" philosophy, that process dies immediately without so much as trying to untangle the mess, and because processes are isolated, none of the other processes have been affected by the bad input. The supervisor, detecting that a process has died, restores the base state we prescribed for group A, and the system picks up from a known point. The beauty of this is that group B, the presentation system, has no idea that this is going on, and really does not care. So long as group A pushes enough good data to group B for the latter to display something of acceptable quality to the user, we have a successful system.

By isolating independent parts of our system and organizing them into a supervision tree, we can create little subsystems that can be individually restarted, in fractions of a second, to keep our system chugging along even in the face of unpredicted errors. If group A fails to restart properly, its supervisor might eventually give up and escalate the problem to the supervisor of the entire group C, which might then in a case like this decide to shut down B as well and call it a day. If you imagine that our system is in fact running hundreds of

simultaneous instances of C-like subsystems, this could correspond to dropping a single multimedia connection due to bad data, while all the rest keep streaming.

However, there are some things that we are forced to share as long as we are running on a single machine: the available memory, the disk drive, even the processor and all related circuitry, and perhaps most significantly, a single power cord to a single outlet. If one of these things breaks down or is disconnected, no amount of layering or process separation will save us from inevitable downtime. This brings us to our next topic, which is distribution—the Erlang feature that will allow us to achieve the highest levels of fault tolerance, and also to make our solutions scale.

1.3 – Distributed Erlang

Erlang programs can be distributed very naturally over multiple computers, due to the properties of the language and its copy-based process communication. To see why, take for example two threads in a language such as Java or C++, running happily and sharing memory between them as a means of communication as pictured in the diagram below.

Figure showing threads sharing memory

Assuming that you manage to get the locking right, this is all very nice, and quite efficient, but only until you want to move one of the threads to separate machine. Perhaps you want to make use of more computing power or memory, or just prevent both threads from vanishing if a hardware failure takes down one machine. When this moment comes, the programmer is often forced to fundamentally restructure the code to adapt to the very different communication mechanism he now needs to use in this new distributed context. Obviously, it will require a large programming effort, and will most likely introduce subtle bugs that may take years to weed out.

Erlang programs, on the other hand, are not much affected by this kind of problem. As we explained in Section 1.1.1, the way Erlang avoids sharing of data and communicates by copying makes the code immediately suitable for splitting over several machines. The kind of intricate data-sharing dependencies between different parts of the code that you can get when programming with threads in an imperative language, occur only very rarely in Erlang. If it works on your laptop today, it could be running on a cluster tomorrow.

At one employer we had quite a number of different Erlang applications running on our network. We probably had at least 15 distinct types of self-contained OTP applications that

all needed to cooperate to achieve a single goal. Integration testing this cluster of 15 different applications running on 15 separate virtual machine emulators, although doable, would not have been the most convenient undertaking. Without changing a single line of code, we were able to simply invoke all of the applications on a single Erlang VM and test them. They communicated with one another on that single node in exactly the same manner, using exactly the same syntax as when they were running on multiple nodes across the network. This concept is known as *location transparency*. It basically means that when you send a message to a process using its unique ID as the delivery address, you do not need to know or even care about where that process is located—as long as the receiver is still alive and running, the Erlang runtime system will deliver the message to its mailbox for you.

Figure illustrating location transparency

The fact that it is usually straightforward to distribute an Erlang application over a network of nodes also means that scalability problems become an order of magnitude easier to attack. You still have to figure out which processes will do what, how many of each kind, on which machines, how to distribute the workload, and how to manage the data, but at least you won't need to start with questions like "how on earth do I split my existing program into individual parts that can be distributed and replicated?", "how should they communicate?" and "how can I handle failures gracefully?"

1.4 – Functional programming: Erlang's face to the world

For many readers of this book, functional programming may be a new concept. For others it will not be. It is by no means the defining feature of Erlang—concurrency has that honor—but it is an important aspect of the language. Functional programming and the mindset that it teaches you are a very natural match to the problems encountered in concurrent and distributed programming, as many others have recently realized. (Need we say more than "Google MapReduce"?)

In the next chapter, we are going to go through the important parts of the Erlang programming language. For many of you, the syntax is going to feel quite strange—it borrows mainly from the Prolog tradition, rather than from C. But different as it may be, it is not complicated. Bear with it for a while, and it will become second nature. Once familiar with it, you will be able to open any module in the Erlang kernel and understand most of what it does, which is the true test of syntax: at the end of the day, can you read it?

2

Erlang Essentials

This book is not mainly about the Erlang programming language in itself, but before we move on to programming with Erlang/OTP design patterns, we want to go through some language basics to make sure everyone is on the same level, and also to serve as a quick reference as you work your way through the chapters. These are the things we think every Erlang programmer should be aware of. If you already know Erlang, you can skim this chapter, but we'll try to make sure there are some useful nuggets for you too.

If this is your very first contact with Erlang, we hope that the material here should be enough for you to digest the rest of the book, but before you start using Erlang for a real project you should also arm yourself with a more thorough guide to Erlang programming; the chapter ends with some pointers to further reading material.

To get the most out of this chapter, you should have a working Erlang installation on your computer. If your operating system is Windows, just open a web browser and go to `www.erlang.org/download.html`, then download and run the latest version from the top of the "Windows binary" column. For other operating systems, and further details on installing Erlang, see Appendix A.

2.1 – The Erlang shell

An Erlang system is a more interactive environment than you may be used to. With most programming languages, you either compile the program to an OS executable which you then run, or you run an interpreter on a bunch of script files or byte code compiled files. In either case, this runs until the program finishes or crashes, and then you get back to the operating system again, and you can repeat the process (possibly after editing the code).

Erlang, on the other hand, is more like an operating system within your operating system. Although Erlang starts pretty quickly, it is not designed for start-stop execution—it is designed for running continuously, and for interactive development, debugging, and upgrading. Optimally, the only reason for restarting Erlang is because of a hardware failure,

operating system upgrade, or similar (but sometimes things can get complicated, and the easiest way out may be a restart).

Interaction with an Erlang system happens mainly through the *shell*. The shell is your command central; it is where you can try out short snippets to see how they work, but it is also where you do incremental development, interactive debugging, and control a running system in production. To make you comfortable with working in the shell, our examples are written so that you can try them out as you read them. Let's start a shell right away!

2.1.1 – Starting the shell

We assume that you have downloaded and installed Erlang/OTP as we said above. If you are using Linux, Mac OS X, or any other UNIX-based system, just open a console window and run the `erl` command. If you are using Windows, you should click on the Erlang icon that the installer created for you; this runs the program called `werl`, which opens a special console for Erlang that avoids the problems of running `erl` interactively under the normal Windows console.

You should see something like the following:

```
Erlang (BEAM) emulator version 5.6.5 [smp:2] [async-threads:0]

Eshell V5.6.5  (abort with ^G)
1>
```

The "`1>`" is the prompt. This will change to "`2>`", etc., as you enter commands. You can use the up and down arrows or the Ctrl-P/Ctrl-N keys to move up and down among previously entered lines, and a few other Emacs-style key bindings also exist, but most normal keys behave as expected.

It is also possible to start the Erlang system with the `-noshell` flag, like this (on your operating system command line):

```
erl -noshell
```

In this case, the Erlang system will be running, but you cannot talk to it via the console. This is used for running Erlang as a batch job or daemon.

2.1.2 – Entering expressions

First of all, what you enter at the shell prompt isn't "commands" as such, but *expressions*, the difference being that an expression always has a *result*. When the expression has been evaluated, the shell will print the result. It will also remember it so that you can refer to it later, using the syntax `v(1)`, `v(2)`, etc. For example, type the number `42`, followed by a period, then press return and you should see the following:

```
Eshell V5.6.5  (abort with ^G)
1> 42.
```

```
42
2>
```

What happened here was that when you pressed return, Erlang evaluated the expression "`42`", printed the result (the value 42), and finally printed a new prompt, this time with the number 2.

ENDING WITH A PERIOD

The period or full stop character before you pressed enter must always be used to tell the shell that it has seen the end of the expression. If you press enter without it, the shell will keep prompting for more characters (without incrementing the prompt number), like this:

```
2> 12
2> + 5
2> .
17
3>
```

So if you forget the period character at first, don't worry, all you need to do is type it in and press enter. As you see, simple arithmetic expressions work as expected. Now let's try referring back to the previous results:

```
3> v(1).
42
4> v(2).
17
5> v(2) + v(3).
59
6>
```

By default, the shell keeps only the latest 20 results.

ENTERING QUOTED STRINGS

When you enter double- or single-quoted strings (without going into detail about what this means, for now), a particular gotcha worth bringing up right away is that if you forget a closing quote character and press return, the shell will expect more characters and will print the same prompt again, much like above when we forgot the period. If this happens, type a single closing quote character, followed by a period, and press enter again. For example if we happen to do this:

```
1> "hello there.
1>
```

the period doesn't end the expression—it is part of the string. To get the shell out of this state, we do the following:

```
1> ".
```

```
"hello there.\n"
2>
```

Note that the result above was a string that contained a period and a newline, which is probably not what we wanted, so we can use the up-arrow or Ctrl-P to go back and edit the line, inserting the missing quote character in the right place this time:

```
2> "hello there".
"hello there"
3> v(2).
"hello there"
4>
```

The shell keeps previous results so you can refer to them regardless of whether they are numbers, strings, or any other kind of data.

2.1.3 – Shell functions

There are some functions like `v(N)` above that are only available in the shell, and not anywhere else in Erlang. These *shell functions* usually have very short (and somewhat cryptic) names. If you want a list of the available shell functions, just enter `help()` (which is a shell function in itself). It's a confusing list for the beginner, so here are the ones that you should know about from start:

- `help()` – prints the available shell functions
- `h()` – prints the history of entered commands
- `v(N)` – fetches the value computed at prompt N
- `cd(Dir)` – changes current directory (`Dir` should be a double-quoted string)
- `ls()` and `ls(Dir)` – print a directory listing
- `pwd()` – print working directory (current directory)
- `q()` – quit (shorthand for `init:stop()`)
- `i()` – print information about what the system is running
- `memory()` – print memory usage information

Please try out a few of these right now, for example listing or changing the current directory, printing the history, and printing the system and memory information. Look briefly at the output from running `i()` and note that much like in an operating system, there is a whole bunch of things going on in the background apart from the shell prompt that you see.

2.1.4 – Escaping from the shell

There are some different ways of leaving the shell (and stopping the entire Erlang system). You should be familiar with all of them, because they all have their uses in managing and debugging a system.

CALLING Q() OR INIT:STOP()

The safest method is to run the shell function `q()`, shown in the previous section. This is a short-cut for the function `init:stop()` (which you can call directly if you like), that shuts down the Erlang system in a controlled manner, telling running applications to stop and giving them some time to respond. This usually takes a couple of seconds, but can need more time on a running production system with a lot to clean up.

THE BREAK MENU

If you are more impatient, and don't have anything important running that you are afraid to interrupt, you can bring up the low-level BREAK menu by pressing Ctrl-C on UNIX-like systems, or Ctrl-Break on Windows in the `werl` console. It looks like this:

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
```

The interesting options here are (a) to abort the system (hard shutdown), (c) to go back to the shell, and (v) to print the running Erlang version. The others print a lot of raw information about the system, that you may find useful for debugging once you have become an Erlang master, and (k) even lets you browse through the current activities within Erlang and kill off any offenders, if you really know what you are doing. Note that the shell as such does not really know about the BREAK menu, so it will not refresh the prompt when you go back using (c), until you press enter.

CTRL-G

The third and most useful escape is the "User switch command" menu, which is reached by pressing Ctrl-G. It will present you with this cryptic text:

```
User switch command
 -->
```

Type `h` or `?` and press return, and you will see this list:

```
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q                 - quit erlang
  ? | h             - this message
```

Entering `c` at the "`-->`" prompt will get you back to the shell. Entering `q` will cause a hard shutdown, like (a) in the BREAK menu—don't confuse `q` here with the system-friendly shell function `q()` described previously! Also note that the BREAK menu is more low-level and can be called up while you are in the Ctrl-G menu, but not the other way around.

The remaining options here are for job control, which we will give a brief introduction to in the next section.

2.1.5 – Job control basics

Suppose you are sitting at your Erlang shell prompt, and you happen to write something stupid that will run forever (or longer than you care to wait, anyhow). We all do this now and then. You *could* make the Erlang system shut down by one of the methods described above, and restart it, but the nicer and more Erlang-like way (especially if there are some important processes running on this system that you really would prefer not to interrupt) is to simply kill the current job and start a new one, without disturbing anything else.

To simulate this situation, enter the following at your Erlang shell prompt, followed by a period and newline:

```
timer:sleep(infinity)
```

(That didn't need any explanation, I hope.) Right, so now the shell is locked up. To get out of this mess, you bring up the "User switch command" menu with Ctrl-G as we described in the previous section, and start by entering `j` to list current jobs. There should be only one job right now, so you'll see something like this:

```
User switch command
 --> j
   1* {shell,start,[init]}
 -->
```

Ok, now we enter an `s`, to start a new shell job (on the local system) like the one you had before, and after that we list our jobs again:

```
 --> s
 --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
 -->
```

To connect to the new job, we could enter "`c 2`", to be explicit, but since the "`*`"-marker indicates that job number 2 is already the default choice, it is enough to say "`c`":

```
--> c
Eshell V5.7.2  (abort with ^G)
1>
```

…and we're back at the wheel again! But wait, what about the old job? Hit Ctrl-G again and list the jobs, and you'll see that it's still hanging around. Let's kill it by entering "`k 1`", and then go back to the shell again so we can get on with making more mistakes:

```
User switch command
 --> j
   1  {shell,start,[init]}
   2* {shell,start,[]}
 --> k 1
 --> j
   2* {shell,start,[]}
 --> c
```

When you do this sort of thing, just be very careful about which job it is you're killing, in case you have several things in progress in different jobs. When you kill a job, all the history, previous results and other things associated with that shell job will disappear. To keep better track of jobs, you can specify a name when you start a new job, that will show up in the list:

```
 --> s important
 --> j
   2  {shell,start,[]}
   3* {important,start,[]}
 --> c
```

We will see more of the Ctrl-G menu in chapter 8 when we talk about distributed Erlang and how to use remote shells. This is as simple as it is powerful, and is the single most important tool for remote controlling and debugging production systems.

Now that you have a feel for how to work in the Erlang console, it is time to start playing around with the actual programming language.

2.2 – Data types in Erlang

Understanding basic data representation conventions is an essential part of learning any programming language. Erlang's built-in data types are straightforward and relatively few, but you can achieve quite a lot with them. Data in Erlang is usually referred to as "terms". Do try entering some examples of terms while you read this section. (Don't forget to add a period before you press return.) Let's start with the simplest ones.

2.2.1 – Numbers and arithmetic

Erlang has two numerical data types: integers and floating-point numbers ("floats"). Conversion is done automatically by most of the arithmetic operations, so you don't usually need to do any explicit type coercion.

INTEGERS

Integers in Erlang can be of arbitrary size. If they are small enough, they are represented in memory by a single machine word; if they get larger (so-called "bignums"), the necessary space is allocated automatically. This is completely transparent to the programmer, and means that you never need to worry about truncation or wrap-around effects in arithmetic—those things simply cannot happen.

Normally, integers are written as you would expect (and you can try entering some really large numbers just for fun):

```
101
-101
1234567890 * 9876543210 * 9999999999
```

You can also write integers in any base between 2 and 36 (corresponding to digits 0-9 plus characters A-Z/a-z), although bases except 2, 16, and possibly 8 are rarely seen in practice. This notation was borrowed from the Ada programming language:

```
16#FFffFFff
2#10101
36#ZZ
```

Also, the following `$`-prefix notation yields the character code (ASCII/Latin-1/Unicode) for any character (try it):

```
$9
$z
$\n
```

We will see a little more of this notation when we discuss strings later on.

FLOATS

Floats are handled using 64-bit IEEE 754-1985 representation ("double precision"), and the syntax is the same as used by most programming languages, with the exception that while many languages allow a floating-point number to begin with just a period, as in "`.01`", Erlang requires that it starts with a digit, as in "`0.01`".

```
3.14
-0.123
299792458.0
6.022137e23
6.6720e-11
```

There are no "single precision" floating-point numbers in Erlang, but people with a C/C++ background sometimes misinterpret what we mean by "floats" in Erlang.

ARITHMETIC AND BITWISE OPERATIONS

Normal infix notation is used for the common arithmetic operators, and `+`, `-`, `*` work as you would expect. If either or both of the arguments of a binary arithmetic operation is a float, the operation will be made in floating point, and Erlang automatically converts any integer arguments to floating point as necessary. For example, `2 * 3.14` yields the float `6.28`.

For division, there are two choices. First, the `/` operator always yields a floating point number, so for example `4/2` yields `2.0`, not `2`. Integer division (truncating) is performed by the `div` operator, as in `7 div 2`, yielding 3.

The remainder of an integer division is given by the `rem` operator, as in `15 rem 4`, yielding 3. (This can differ from what a "modulo" operator would yield, if negative numbers are involved.)

Other floating-point arithmetic functions are found in the standard library module `math`; these are named directly after the corresponding functions in the C standard library, for example `math:sqrt(2)`.

There are also some additional integer operators for bitwise operations: `N bsl K` shifts the integer N K steps to the left, and `bsr` performs a corresponding arithmetic right shift. The bitwise logic operators are named `band`, `bor`, `bxor`, and `bnot`. For example, `X band (bnot Y)` would mask away those bits from X that are set in Y.

From numerical data, let's move on to something equally primitive: bits and bytes.

2.2.2 – Binaries and bitstrings

A *binary* is a sequence of unsigned 8-bit bytes, used for storing and processing chunks of data (often data that comes from a file or has been received over some network protocol). A *bitstring* is a generalized binary whose length in bits is not necessarily a multiple of 8; it could for instance be 12 bits long, consisting of "one and a half" bytes.

Arbitrary bitstrings are a more recent addition to the language, while whole-byte binaries have been around for many years, but to a programmer there is very little difference on the surface of things, except that you can do some really nifty things these days that used to be impossible before. Since the syntax is the same, and the name "binary" is so ingrained, you rarely hear people (including us) talk about "bitstrings" unless they want to make a point about the more flexible length.

The basic syntax for a binary is:

```
<<0, 1, 2, …, 255>>
```

that is, a comma-separated list of integers in the range 0 to 255, enclosed in << … >>. There must not be any space between the two delimiter characters on either side, as in "< <". A binary can contain any number of bytes; for example, <<>> is an empty binary.

Strings may also be used to make a binary, as in:

```
<<"hello", 32, "dude">>
```

This is the same as writing the corresponding sequence of bytes for the 8-bit character codes (ASCII/Latin-1) of the strings. Hence, this notation is limited to 8-bit characters, but it is often quite useful for things like text-based protocols.

These short examples only show how to create proper binaries, whose length in bits is divisible by eight. Erlang has an advanced and somewhat intricate syntax for constructing

new binaries or bitstrings as well as for matching and extracting data from them. We will show some examples of this later, in Section 2.10.

Our next topic is something almost as primitive as numbers and bits to an Erlang programmer: *atoms*.

2.2.3 – Atoms

In Erlang, an "atom" is a special kind of string constant that is identified only by the characters in the string, so that two atoms are always considered to be exactly the same if they have the same character representation. Internally, however, these strings are stored in a table and are referred to by the table index, so that checking atoms for equivalence at runtime amounts to comparing two small integers, and each time you use an atom, it only takes up one word of memory. (The actual index number used for any particular atom is automatically assigned at run-time and can vary from one run of the system to the next; there is no way, and no need, for the user to know this.)

Atoms in Erlang play a role similar to `enum` constants in Java or C: they are used as labels. The difference is that you don't need to declare them in advance; you can invent them as you go and use them anywhere you like. (Try entering some of the below examples in the shell.) In the Lisp programming language, they are known as "symbols". Programming with atoms is easier, more readable, and more user friendly than using numeric constants.

Normally, atoms are written starting with a lowercase letter, like the following:

```
ok
error
foo
undefined
trap_exit
```

After the initial letter, you can use uppercase letters, digits, underscores, and @-characters, like this:

```
route66
atoms_often_contain_underscore
pleaseDoNotUseCamelCaseInAtomsItLooksAwful
vader@deathstar
```

For anything else, you need to use single-quotes (and you can of course single-quote the above atoms as well—this is sometimes done for clarification, e.g. in documentation):

```
'$%#*!'
'Blanks and Capitals can be quoted'
'Anything inside single-quotes\n is an atom'
```

You should think about atoms as special labels, not as any old strings. Their length is limited to 255 characters, and there is an upper limit on the number of atoms you can have

in a system: currently, just over a million (1048576 to be exact). This is usually not a problem, but you should avoid dynamic generation of unique atoms such as `'x_4711'`, `'x_4712'`, etc., in a system that is expected to run for a long time (days, months, years). Atoms are not removed from the table again until the system restarts, even if they are no longer used by anyone.

Now that we have gone through the primitives, it is time to look at how we can create more complicated data structures, starting with tuples.

2.2.4 – Tuples

A tuple (or "n-tuple", as generalized from "triple", "quadruple", etc.) is a fixed-length ordered sequence of other Erlang terms. Tuples are written within curly braces, like this:

```
{1, 2, 3}
{one, two, three, four}
{from, "Russia", "with love"}
{complex, {nested, "structure", {here}}}
{}
```

As you see, they can contain zero (`{}`), one (`{here}`) or more elements, and the elements may be all of the same type, or of wildly different types, and the elements can themselves be tuples or any other data type.

A standard convention in Erlang is to label tuples to indicate what type of data they contain, by using an atom as the first element, as in `{size, 42}`, or `{position, 5, 2}`. These are called *tagged tuples*.

Tuples are the main way of constructing compound data structures or returning multiple values in Erlang, like structs in C or objects in Java, but the entries are not named, they are numbered (from 1 to N). Accessing an element of a tuple is a constant-time operation, just as fast (and safe) as accessing en entry in a Java array. The *record syntax*, explained later, allows you to declare names for the entries of tuples, so you don't have to work directly with indices. Also, *pattern matching* makes it easy to refer to the different parts of a tuple using variables, so it is indeed rare that you need to access an entry directly by its index.

The standard library contains modules that implements some more complicated *abstract data types*, such as arrays, sets, dictionaries (i.e., "associative arrays" or "hash maps"), etc., but under the hood they are mostly implemented using tuples in various ways.

2.2.5 – Lists

Lists are truly the work-horse of Erlang's data types—as they are in most functional programming languages, for that matter. This has to do with their simplicity, efficiency, and flexibility, but also with that they follow naturally from the idea of *referential transparency* (see Appendix B for details). Lists are used to hold an arbitrary number of items. They are written within square brackets, and in the simplest form they look like this:

```
[]
```

```
[1, 2, 3]
[one, two, three]
[[1,2,3],[4,5,6]]
[{tomorrow, "buy cheese"},
 {soon, "fix trap door"},
 {later, "repair moon rocket"}]
```

that is, as simply a sequence of zero or more other Erlang terms (which may be other lists). The empty list, `[]`, is also known as *nil*, a name that comes from the Lisp programming language world; it is in fact more like an atom, in that it is a special value that only takes a single word of memory to represent.

ADDING TO A LIST

What you can do with lists that you can't do as easily and efficiently with tuples, is create a new, longer list from an existing list in such a way that the old list is simply a part of the new one. This is signaled with the "`|`" character (a "vertical bar"). For example:

```
[ 1 |  []  ]
```

this combines the empty list on the right of the "`|`" with the additional element 1 on the left, yielding the list [1]. Try typing it in the shell and see for yourself how these examples work. Continuing in the same manner:

```
[ 2 |  [1]  ]
```

yields the list [2,1] (note the order: we are adding to the left). We can even add more than one element at a time, but only on the left hand side of the `|`:

```
[ 5, 4, 3 |  [2,1]  ]
```

which gives us [5,4,3,2,1]. This is actually done by adding first 3, then 4, and finally 5, as with 1 and 2 above, but the compiler does the job of splitting it into smaller steps for us.

For lists of arbitrary lengths, we can use the `++` operator to append them. For example:

```
[1,2,3,4] ++ [5,6,7,8]
```

yields the list `[1,2,3,4,5,6,7,8]`. This happens in exactly the same way: by starting with `[4|[5,6,7,8]]`, then `[3|[4,5,6,7,8]]`, etc., and finally `[1|[2,3,4,5,6,7,8]]`. The list that was on the right hand side of `++` is never modified—we don't do that sort of "destructive updates" in Erlang—it just gets included in the resulting list, technically via a pointer. This also means that the ++ operator does not care how long the right-hand side list is, because it never has to do anything with it.

The list on the left-hand side is a different thing, though. To create the resulting list in the way we described above, we must first of all find the end of the left-hand side list (the

element `4` in this case), and then start building the result backwards from there. This means that *the length of the left-hand side list decides how much time `++` takes*. For this reason, we always try to add new (shorter) stuff to the left of the list, even if it means the final list will be in reverse order. It is much cheaper to finish up with a quick call to reverse the list afterwards (please trust us here) than it is to repeatedly go through a list that keeps getting longer and longer every time just so we can add something to its end.

2.2.6 – Strings

A double-quoted string in Erlang is merely an alternative way of writing a list of character codes. For example:

```
"abcd"
"Hello!"
" \t\r\n"
""
```

are exactly equivalent to

```
[97,98,99,100]
[72,101,108,108,111,33]
[32,9,13,10]
[]
```

which could also have been written as

```
[$a, $b, $c, $d]
[$H, $e, $l, $l, $o, $!]
[$\ , $\t, $\r, $\n]
[]
```

(if you recall the $-syntax from the section about integers, a few pages back). This correspondence is reflected in the names of some of the standard library functions in Erlang, such as `atom_to_list(A)`, which returns the list of characters of any atom A.

STRINGS AND THE SHELL

The Erlang shell tries to maintain the illusion that strings are different from plain lists, by checking if they contain only printable characters. If they do, it prints them as double-quoted strings, and otherwise as lists of integers. This is more user friendly, but occasionally doesn't do what you want (for example, when an expression returns a list of numbers that by coincidence look like printable characters, and you see a string of line noise as result).

A useful trick in that case is to append a zero to the start of the list, to force the shell to print the real representation. For example, even if `v(1)` is a shown as a string, `[0 | v(1)]` will not be. (You can of course use the string formatting functions in the standard library to pretty-print the value, giving you full control, but how much fun is that?)

Now that we know how to work with data structures, we'll quickly go over the remaining primitive data types: identifiers and funs.

2.2.7 – Pids, ports, and references

These three "identifier" data types are closely related, so we present them here together.

PIDS (PROCESS IDENTIFIERS)

As you know by now, Erlang supports programming with processes; indeed, for any code to run at all there must be an Erlang process running it. Every process has a unique identifier, usually referred to as a *pid*. Pids are a special data type in Erlang, and should be thought of as opaque objects. When the shell prints them, however, they show up on the form "`<0.35.0>`", that is, as three integers enclosed in angle brackets. You cannot enter this into the shell and create a pid using this syntax; it is only shown for debugging purposes, so that you can compare pids easily.

Although pids are expected to be unique for the lifetime of the system (until you restart Erlang), in practice the same identifier may be reused when the system has been running for a very long time and some hundred million processes have come and gone. This is rarely considered a problem.

The function `self()` always gives you the pid of the process that is currently running (the one that called self()). You can try it out in the shell—that's right, the shell is also a process in Erlang.

PORT IDENTIFIERS

A *port* is much like a process, except that it can also communicate with the world outside Erlang (and cannot do much else—in particular, it can't run any code). Hence, *port identifiers* are very closely related to pids, and the shell prints them on the form "`#Port<0.472>`". We will get back to ports later in this book.

REFERENCES

The third data type of this family is *references* (often called just "refs"). They are created with the function `make_ref()` (try it out!), and are printed by the shell on the form "`#Ref<0.0.0.39>`". References are used as unique one-off labels or "cookies".

2.2.8 – Functions as data: "funs"

Erlang is said to be a *functional* programming language, and an expected feature of such a language is that it should be able to handle functions as data, that is, pass a function as input to another function, return a function as the *result* of another function, put a function in a data structure and pick it up later, etc. Of course, you must also be able to *call* a function that you have gotten that way. In Erlang, such a function-as-data object is called a "fun" (and sometimes a "closure" or even "lambda expression").

We will discuss funs in more detail later, and just note for now that the shell will print them on the form "`#Fun<…>`", with some information for debugging purposes between the angle brackets. You cannot create a fun using this syntax.

That was the last in our list of built-in data types. We will now discuss something that unites them all: the comparison operators.

2.2.9 – Comparing terms

The different data types in Erlang have one thing in common: they can all be compared and ordered, using built-in operators like `<`, `>`, and `==`. The normal orderings on numbers of course hold, so that `1 < 2` and `3.14 > 3` and so on, and atoms and strings (as well as any other lists) and tuples are ordered lexicographically, so that `'abacus' < 'abba'`, `"zzz" > "zzy"`, `[1,2,3] > [1,2,2,1]`, and `{fred,baker,42} < {fred,cook,18}`.

So far, it all seems pretty normal, but on top of that you also have an ordering between values of different types, so that for example `42 < 'aardvark'`, `[1,2,3] > {1,2,3}`, and `'abc' < "abc"`. That is, all numbers come before all atoms, all tuples come before all lists, and all atoms come before all tuples and lists (strings are really lists, remember?).

You don't have to memorize which data types come before which in this order. The important thing to know is that you can compare *any* two terms for order, and you will get the same result always. In particular, if you sort a list of various terms of different types (a list of mixed numbers, strings, atoms, tuples, …) by using the standard library function `lists:sort(…)`, you will always get a properly sorted list as result, where all the numbers come first, then all atoms, and so on. You can try it out in the shell: for example, `lists:sort([b,3,a,"z",1,c,"x",2.5,"y"])`.

LESS-THAN/GREATER-THAN OR EQUALS

One of those little differences in syntax between Erlang and most other programming languages except Prolog is that the less-than-or-equals operator is *not* written "<=", for the reason that this looks too much like an arrow pointing to the left (and that symbol is indeed reserved for use as a left-arrow). Instead, less-than-or-equals is written "=<". The greater-than-or-equals operator is written ">=" as in most languages. The only thing you need to remember is that *comparisons never look like arrows*.

EQUALITY COMPARISONS

There are two kinds of operators for equality comparisons in Erlang. The first one is the *exact equality*, written "=:=", which returns `true` only if both sides are exactly the same. For example, `42 =:= 42`. The negative form (*exact inequality*) is written "=/=", as for example in `1 =/= 2`.

Exact equality is the preferred kind of equals-operator when you are comparing terms in general (and it is also the one that is used in pattern matching, which we will talk about later). However, it means that integers and floating-point numbers are considered to be different, even if they are as similar as could be. For instance, `2 =:= 2.0`, returns `false`.

If you are comparing numbers in general (or perhaps tuples containing numbers, like vectors) in a mathematical way, you should instead use the *arithmetic equality* operator, written "==". This will compare numbers by coercing integers to floating-point as necessary. Hence, `2 == 2.0` returns true. The negative form (*arithmetic inequality*) is written "/="; for example, `2 /= 2.0` returns false. Remember, though, that comparing floating-point

numbers for equality is always a somewhat suspicious thing to do, because the tiny rounding errors involved in the floating-point representation may cause values that "ought to be equal" to differ ever so slightly, and then == will return `false`. It is usually a better idea to use <, >, =<, or >= to compare numbers when they may be in floating point. Those operators are also "arithmetic", by the way, i.e., they always coerce integers to floats when necessary. That's why we could compare 3 and 3.14 using > previously.

If you use "==" (the coercing, arithmetic equality operator) when it is not warranted—which is more or less always except when you're actually doing maths—you will only be making it harder for program analysis tools to help you find out if your program is doing something it shouldn't. You may also be masking errors at run-time so that they are not detected as early as they could be, and instead show up much later, perhaps as weird data in a file or database (like a year showing up as 1970.0, or a month as 2.0).

That said, seasoned Erlang programmers usually avoid using the equality comparison operators at all, and do as much as possible through pattern matching, which we will talk about a little further below.

2.2.10 – Understanding lists

Lists are different enough in Erlang compared to most common programming languages that we need to give them some special treatment before we can leave the topic of data types.

THE STRUCTURE OF A LIST

Basically, lists are created from A) the empty list ("nil"), and B) so-called *list cells* which add one element at a time on top of an existing list, building a singly-linked list in memory. Each such cell uses only two words of memory: one for the value (or a pointer to the value), known as the "head", and one for a pointer to the rest of the list, called the "tail"; see figure 2.?. List cells are sometimes called "cons cells" (from "list constructor") by people with a background in Lisp or functional programming, and the action of adding a cons cell is known as "consing" if you want to be really geeky.

Figure 2.? A list cell—the primitive building block of lists

Although there is no real technical difference between the "head" and the "tail" elements of a list cell, they are by convention always used so that the first (the head) contains the payload of the cell, and the second (the tail) is the rest of the list. This convention is used in the syntax as well as in all library functions that operate on lists.

A common gotcha for beginners (and even old-timers get snagged on this in the form of a typo sometimes) is to mistakenly write a comma instead of the intended vertical bar. Consider this: What is the actual difference between the following two expressions?

```
[ 1, 2, [3,4] ]
[ 1, 2 | [3,4] ]
```

(See if you get the point before you read on – try entering them in the shell if you need more clues.)

The answer is that the first is a list with three elements, the last of which happens to be another list (with two elements). The second expression is a list with four elements, made up from stacking two elements on top of the list [3,4]. Figure 2.? illustrates the structure of these two examples. Make sure you understand it before you read on—it is central to everything you do with lists. Also see Appendix B for a deeper discussion about lists and referential transparency.

Figure 2.? The list cell structures in the above example

You should learn to love the list. But remember, lists are mainly good for temporary data, for example as a collection that you are processing, as a list of results that you are compiling, or as a string buffer. For long-term data storage, you may want to use a different representation when size matters, such as binaries for storing larger amounts of constant string data.

As you will see moving forward, quite a large part of the data processing you will do in Erlang, including string manipulation, comes down to traversing a list of items, much like you traverse collections and arrays in most other languages. Lists are your main intermediate data structure.

IMPROPER LISTS

A final note is on the difference between a *proper list* and an *improper list*. Proper lists are those that you have seen so far. They are all built up with an empty list as the innermost "tail". This means that starting from the outside we can peel off one cell at a time, and know that we must finally end up with the tail being an empty list.

An improper list, however, has been created by adding a list cell on top on something that is not a list to begin with. For example:

```
[ 1 |  oops  ]
```

This will create a list cell with a non-list tail (in this case, an atom `'oops'`). Erlang does not forbid it, and does not check for such things at runtime, but you should generally regard such a thing, if you see it, as a programming error and rewrite the code.

The main problem with it is that any functions that expect to receive a proper list will crash (throw an exception) if they try to traverse an improper list and end up with finding a non-list as the tail. Don't be tempted to use list cells this way even if you think you have a clever idea—it is bug-prone, and confuses both humans and program analysis tools.

That said, there are one or two valid uses for creating improper lists, but they are considered advanced programming techniques and will have to be covered by another book.

2.3 – Modules and functions

So far, you've seen some basic Erlang expressions: code snippets that can be evaluated to produce some value. But real Erlang programs are not just one-liners that you can enter in the shell. To give your code some structure in life and a place to call home, Erlang has *modules*, which are containers for program code. Each module has a unique name, which is specified as an atom. Erlang's standard library contains a large number of pre-defined modules, such as the `lists` module that contains many functions for working with lists.

2.3.1 – Calling functions in other modules (remote calls)

When you want to call a function that resides in some other module, you need to qualify the function name with the name of the module it is in, using a colon character as separator. For instance, to reverse a list `[1,2,3]` using the function `reverse` in the standard library module `lists`, we write as follows (and you can try this in the shell):

```
lists:reverse([1,2,3])
```

This form is called a "remote call" (calls a function in a different module), as opposed to a "local call" (calls a function in the same module). This should not be confused with "Remote Procedure Call", which is a concept in distributed programming and is a completely different thing altogether (asking another process or computer to run a function for you).

2.3.2 – Functions of different arity

The number of arguments a function takes is referred to as its *arity*. For example, a function which takes one argument is a *unary* function; one that takes two arguments is a *binary* function; one that takes three arguments it a *ternary* function, and so on. We have seen a couple of functions already such as `self()` that are *nullary*, that is, that take no arguments.

The reason we bring this up is that the arity of functions is more important in Erlang than in most other programming languages. Erlang does not have function overloading as such, but instead, it treats functions of different arities as completely separate even if they have the same atom as identifier. In fact, the full name of a function must always include the arity (written with a slash as separator). For example, the list-reversing function above is really `reverse/1`, or if we want to be particular about which module it resides in, we'll write `lists:reverse/1`. Note, though, that you can only use this syntax where the language actually expects a function name; if you just write `hello/2` as an expression, Erlang will interpret this as an attempt to divide the atom `'hello'` by 2 (and will not like it).

To show how this works, there is in fact a function `lists:reverse/2`, which does almost the same as `reverse/1`, but it also appends the list given as its second argument to the final result, so that `lists:reverse([10,11,12], [9,8,7])` will result in the list `[12,11,10,9,8,7]`. In some languages, this function might have had to be called "reverse_onto" or similar to avoid a name collision, but in Erlang we can get away with using

the same atom for both. Do not abuse this power when you write your own functions – it should be done in a systematic way so that it is easy for your users to remember how to call your functions. If you create functions that differ only in arity but produce wildly different results, you will not be thanked for it. When in doubt, opt for giving the functions clearly different names.

At any rate, always remember that in order to exactly specify which function we are talking about, we need to give the arity, not just the name.

2.3.3 – Built-in functions and standard library modules

Like any other programming language, Erlang comes with a *standard library* of useful functions. These are spread over a large number of modules; however, some standard library modules are more commonly used than others. In particular, the module named `erlang` contains those functions that are central to the entire Erlang system, that everything else builds on. Another useful module that we have seen already is the `lists` module. The `io` module handles basic text input and output. The `dict` module provides hash-based associative arrays (dictionaries), and the `array` module provides extensible integer-indexed arrays. And so forth.

Some functions are involved with so low-level things that they are really an intrinsic part of the language and the run-time system. These are commonly referred to as *built-in functions*, or BIFs, and like the Erlang run-time system itself, they are implemented in the C programming language. (Though some may disagree on the details, this is the most common definition.) In particular, all the functions in the `erlang` module are BIFs. Some BIFs, like our friend lists:reverse/1 from the previous section, could in principle be written directly in Erlang (like most of the other functions in the `lists` module), but have been implemented in C for efficiency reasons. In general, though, the programmer does not have to care about how the functions are implemented—they look the same to the eye—but the term "BIF" is used quite often in the Erlang world, so it's useful to know what it refers to.

Finally, even the operators of the language, such as +, are really built-in functions, and belong to the `erlang` module. For example, you can write `erlang:'+'(1,2)` for addition.

AUTOMATICALLY IMPORTED FUNCTIONS

A few functions (all found in the module `erlang`) are both important and very commonly used, in Erlang programs as well as in the shell. These are automatically *imported*, which means that you don't need to write the module name explicitly. We have already seen the function `self()`, which returns the process identifier (the "pid") of the process that calls it. This is actually a remote call to `erlang:self()`, but because it is one of the auto-imported functions, you don't need to prefix it with `erlang:`. Other examples are `spawn(…)`, which starts a new process, and `length(…)`, which computes the length of a list.

2.3.4 – Creating modules

To make ourselves a new module that can be used, we need to:

- Write a source file

- Compile it
- Load it, or at least put it in the load path

The first step is easy—start your favorite text editor, open a new file, and start typing. Give the module a name, and save the file using the same name as for the module, plus the suffix ".erl". For example, Code listing 2.1 shows how such a file (named "my_module.erl") might look. Create this file using your text editor now:

Code listing 2.1 – my_module.erl

```
%% This is a simple Erlang module

-module(my_module).

-export([pie/0]).

pie() ->
    3.14.
```

To start with the easiest part of the above, the part that says "`pie() -> 3.14.`" is a *function definition*. It creates a function "`pie`" that takes no arguments and returns the floating-point number 3.14. The arrow "`->`" is there to separate the function *head* (the name and arguments) from its *body* (what the function does). Note that we don't need to say "return" or any such keyword: *a function always returns the value of the expression in the body*. Also note that we must have a "`.`" at the end of the function definition, just like we had to write a "`.`" after each expression we entered in the shell.

The second thing to note is the *comment* on the first line. Comments in Erlang are introduced with the "`%`"-character, and we'll say more about them in a moment.

The very first item in a module, apart from any comments, must always be the *module declaration*, on the form "`-module(…).`" *Declarations* are basically anything that's not a function or a comment. They always begin with a hyphen ("`-`"), and just like function definitions, they must end with a "`.`" character. The module declaration is always required, and the name it specifies must match the name of the file (apart from the ".erl" suffix).

The line we saved for last is the one that says "-export([…])." This is an *export declaration*, and tells the compiler which functions (separated by commas) should be visible from the outside. Functions not listed here will be kept internal to the module (so you can't call them from the shell). In this example, we only had one function, and we want that to be available, so we put it in the export list. As we explained in the previous section, we need to state the arity (0, in this case) as well as the name in order to identify exactly what function we're referring to, hence "`pie/0`".

COMMENTS

There is only one kind of source-code comment in Erlang. These are introduced with the % character, and go on until the end of the line. Of course, %-characters within a quoted string or atom don't count. For example:

```
% This is a comment and it ends here.

"This % does not begin a comment"

'nor does this: %'    %<-but this one does
```

You can write comments in the shell, if you like, but there is very little point to it, which is why we have not talked about them earlier.

Style-wise, comments that follow after some code on the same line are usually written with only a single %-character, while comments that are on lines of their own are typically written starting with two %:s, like this:

```
%% This is your average standalone comment line.
%% Also, longer comments may require more lines.

frotz() -> blah.   % this is a comment on a line of code
```

(Some people even like to start with three %:s on comment lines that describe things on a whole-file level, such as comments at the top of the source file.)

One good reason to stick to these conventions is that syntax-aware editors such as Emacs and ErlIDE can be made to know about them, so that they will indent comments automatically according to how many %:s they begin with.

Now that we have a source file that defines a module, we next need to compile it.

2.3.5 – Compiling and loading modules

When you *compile* a module, you produce a corresponding file with the extension ".beam" instead of ".erl", which contains instructions on a form that the Erlang system can load and execute. This is a more compact and efficient representation of the program than the source code, and it contains everything that the system needs to load and run the module. In contrast, a source code file might require that additional files are available, via include declarations (more about those later). All such files that make up the complete source code for the module have to be read at the time the module is compiled. The single .beam file, then, is a more "definite" form for a module, although it cannot be easily read by a human, and cannot be edited by hand – you have to edit the source file instead and re-compile it.

COMPILING FROM THE SHELL

The simplest way to compile a module when you are playing around and testing things, is to use the shell function `c(…)`, which compiles a module and also loads it (if the compilation worked) so you can try it out immediately. It looks for the source file relative to the current directory of the Erlang shell, and you don't even need to say ".erl" at the end of the name. For example, if you start Erlang in the same directory as the file you created above, you can do the following:

```
1> c(my_module).
```

```
{ok,my_module}
2> my_module:pie().
3.14
3>
```

The result `{ok,my_module}` from the call to `c(…)` is just an indicator that the compilation worked, creating a module called `my_module`, which has now been loaded. We can call the function `pie` we exported from it to check that it works.

If you look in the directory of the source file (you can use the shell function `ls()` to do this), you will see that there is now a new file called `my_module.beam` alongside the source file `my_module.erl`. This is the compiled version of the module, also called an *object file*.

MODULE LOADING AND THE CODE PATH

If you now exit the Erlang shell (using the shell function `q()`, for example), and then restart Erlang again in the same directory, you can try calling your module directly without compiling it first (assuming the compilation above worked):

```
1> my_module:pie().
3.14
2>
```

How did this work, then? It's quite simple: whenever Erlang tries to call a module that hasn't been loaded into the system yet, it will automatically try to load it from a correspondingly named .beam file, if it can find one. The directories it will look in are listed in the *code path*, and by default, this includes the current directory (where it found your .beam file and loaded it). To check out the current setting of the code path, call the function `code:get_path()`. This should return a list, at the start of which you'll see `"."`, meaning the current directory. The default code path also includes all the standard library directories. In addition, you can use the functions in the `code` module to modify the path as you like.

THE STAND-ALONE COMPILER, ERLC

In a real software project, you typically want to script your builds using some external build tool, such as GNU Make. In this case, you can use the standalone `erlc` program to run the compiler from your operating system command line. For example:

```
erlc my_module.erl
```

(You can of course run this by hand if you like. Try it out!) This is a bit different from the shell function we used above. Here you need to give the full file name including the ".erl" extension. You can also use options much like you would with a C compiler; for instance, to specify the output directory (where to put the .beam file), you can write something like:

```
erlc my_module.erl -o ./ebin
```

(You may have noticed in the `code:get_path()` example above that all the standard library directories in the path had names ending in "/ebin". This is the normal convention for Erlang, to keep the .beam files in a subdirectory called `ebin`. We'll get back to this in more detail later when we talk about applications.)

If you're on Windows, there is a slight complication: the installation program does not set up the `PATH` environment variable to point to the `erl` and `erlc` programs; you'll have to do this yourself if you want to run them from your `cmd.exe` command line. They can be found in the `bin` subdirectory of the directory where Erlang was installed—the path will probably look something like "`C:\Program Files\erl5.7.3\bin`". Also remember that the `erl` command does not play very well with `cmd.exe`—it's good for running Erlang applications from scripts, but as an interactive environment, you want to run `werl`, like when you click the Erlang icon.

COMPILED MODULES VS. EVALUATION IN THE SHELL

There is a difference between what happens with expressions that you enter in the Erlang shell, and code that you put in a module (and compile, load, and run). A .beam file, as we said, is an efficient, ready-to-deploy representation of a module. Furthermore, all the code in a .beam file was compiled together at the same time, in the same context. It can do things relating to the module it is in, such as specifying which functions are exported and not, or find out what the name of the module is, or declare other things that should hold for the module as a whole.

Code that you enter in the shell, however, consists basically of one-off expressions, to be forgotten fairly soon. It is never part of any module. Therefore, it is not possible to use declarations (like "`-export([…]).`" or "`-module(…).`" that we saw above) in the shell; there is no module context for such declarations to apply to.

The shell simply parses expressions and evaluates them by interpreting them on the fly. This is much less efficient (by several orders of magnitude) than running compiled code, but of course, that doesn't matter much when all it has to do is to perform a call to a function in some existing compiled module (which will run at normal speed), like when you say "`lists:reverse([1,2,3])`". In this case, all the shell does is to prepare the list `[1,2,3]`, and then pass it over to the `reverse` function (and of course print the result afterwards). Even if it does this at a comparatively slow speed, it is still much too fast for a human to notice.

It is possible, though, by use of things such as list comprehensions (we'll explain those further on) or clever use of recursive funs (a neat trick, but may cause the brains of novices to implode), to write code in the shell that is more or less entirely evaluated by the shell's interpreter from start to end, and that actually does some significant amount of work. In that case, it will be very notably slower than if you had written it in a module instead. So, remember this: *never do measurements on code that you have hacked together in the shell*. If you want sane numbers from your benchmarks, you must write them as modules, not as shell one-liners. Don't draw conclusions about efficiency from what you see in the shell.

As an aside, it may happen that in some odd corner case, code evaluated in the shell behaves slightly different from the same code when compiled as part of a module. In such a case, it is the compiled version that is the gold standard. The shell just tries its best to do the exact same thing when it interprets the expressions.

2.4 – Variables and pattern matching

Variables in Erlang are a bit different from variables in most other programming languages, which is why we have postponed introducing them until now. The thing about them is not that they are more difficult than in other languages; it's that they are so much simpler! So simple, in fact, that your first reaction might be "how do I do anything useful with them?"

2.4.1 – Variable syntax

The most visible difference is that in Erlang, variables begin with an uppercase letter! (We have already reserved names-that-begin-with-lowercase for writing atoms, remember?) Here are some examples of variables, using "CamelCase" to separate word parts, which is the normal style for variables in Erlang:

```
Z
Name
ShoeSize12
ThisIsARatherLongVariableName
```

You can also begin a variable with an underscore character. In that case, the second character is by convention usually an uppercase character:

```
_SomeThing
_X
_this_may_look_like_an_atom_but_is_really_a_variable
```

There is a small difference in functionality here: the compiler normally warns you if you assign a value to a variable, and then don't use that variable for anything. This catches a lot of silly mistakes, so don't turn off that warning. Instead, when you want to use a variable for something just to making the program more readable, you can use one that starts with an underscore. (We will see how you might want to write variables like this when we talk about pattern matching below.) The compiler will not complain if those are unused. Also, any variables that are not used will simply be optimized away, so they carry no extra cost: you can use them freely to annotate your program for better readability.

2.4.2 – Single assignment

The next surprise is that Erlang's variables are strictly single assignment. This means that when you "assign" a value to a variable, or as we say in Erlang country, we *bind* the variable to a value, then that variable will hold the same value throughout its entire *scope* (i.e., that part of the program code where the variable "exists"). The same variable name can of course

be reused in different places in the program, but then we are talking about different variables with distinct and non-overlapping scopes, like "Paris" can be one thing in Texas and another thing in France.

In most other programming languages, what's called a "variable" is really a kind of box-with-a-name, and you can change the contents of the box from one point in the program to the next. This is really rather odd if you think about it, and it's certainly not what you learned in algebra class. Erlang's variables, on the other hand, are just like those you knew from mathematics: a name for some value, that doesn't change behind your back (which is why you could solve equations). Of course, the values are really stored somewhere in the computer's memory, but *you* don't have to care about micro-management issues like creating those little boxes and moving things between them or reusing them to save space. The Erlang compiler handles all of that for you, and does it well.

For some more details about single assignment and the concept of *referential transparency*, see Appendix B.

THE = OPERATOR AND USING VARIABLES IN THE SHELL

The simplest form of assignment in Erlang is through the = operator. This is really a "match" operator, and as we shall see below, it can do a bit more than just straightforward assignment, but for now, here's an example that you can try in the shell:

```
1> X = 42.
42
2> X.
42
3> X+1.
43
4>
```

That probably worked as you expected. Variables in the shell are a bit particular, though. Their scope is "as long as the shell is still running, unless you say otherwise". To forget all bound variables, you can call the shell function `f()`, like this:

```
4> f().
ok
5> X.
* 1: variable 'X' is unbound
6> X = 17.
17
7> X.
17
8>
```

As you see, once forgotten, X can be reused for something else. What if we try to reuse it without forgetting the old value?

```
8> X = 101.
```

```
** exception error: no match of right hand side value 101
9>
```

Oops. The single assignment was enforced, and we got an exception instead. The error message indicates that a "match" was taking place. What if we try to "re-assign" the same value that X already has?

```
9> X = 17.
17
10>
```

No complaints this time. That's what the "match" part means: if X already has a value, it just checks that the right-hand side is the same (comparing them for *exact equivalence*; look back to Section 2.2.9 if you've forgotten what this means). If you want to forget just X, and not all the other variable bindings you might have made in the shell, use `f(X)`, like this:

```
10> Y = 42.
42
11> f(X).
ok
12> X.
* 1: variable 'X' is unbound
13> Y.
42.
14>
```

Just remember that the above is how variable scope works *in the shell*. Within a module, scopes are tied to function definitions and similar things, and there is no way to "forget" variable bindings prematurely; we will get into more detail in Section 2.5.2.

VARIABLES AND UPDATES

Once you accept that you can't "update" a variable with a new value, you will probably wonder how to change anything. After all, much of what a program does is compute new data from old, for instance, adding 1 to a number. The short answer is, if you need to keep track of another value, then give it another name. For example, if you have a variable X that holds some integer, and want to give a name to the value you get if you add 1 to X, you could say something like "X1 = X + 1":

```
1> X = 17.
17
2> X1 = X + 1.
18
3>
```

Maybe you can come up with a more descriptive name, like NewX or IncrementedX, and depending on the code at hand, this might or might not be better (overlong names can also be bad for readability), but the normal fallback when you're out of inspiration is to use

names like X1, X2, X3, for "modified" variants of X. (If you're wondering what to do with variables in a loop, it will have to wait until we discuss recursive functions, later on.)

But usually, you should avoid situations where you need a lot of different variables for almost-the-same-thing-only-modified-a-bit. You should try to split the code into separate functions instead, where each function can have its own "X" and works on only one step of the whole problem. This makes for much more readable and sane code in the long run.

2.4.3 – Pattern matching: assignment on steroids

Pattern matching is one of the utterly indispensible features of Erlang. Once you get used to it, you wonder how you could ever be without it, and the thought of programming in a language that doesn't have pattern matching becomes rather depressing. (Trust us.)

At the one and same time, pattern matching serves the following important purposes:

- Choosing control flow branches
- Performing variable assignments (bindings)
- Decomposing data structures (selecting and extracting parts)

THE = OPERATOR IS REALLY A PATTERN MATCH

In the previous section, we called = a "match" operator. This is because what it does is really *pattern matching*, rather than just "assignment". On the left-hand side, we have a *pattern*, and on the right-hand side we have a plain old expression. To evaluate the match, the right-hand side expression is evaluated first, to get a value. That value is then *matched* against the pattern (a bit like when you match a string against a regular expression). If the pattern doesn't match at all, as in "`17 = 42`", or "`true = false`", the match fails and throws an exception containing the reason code `badmatch`. In the shell, this will be presented to you as the error message "no match of right hand side value …".

If the match works, however, the program will continue with any expressions that follow after it, but now any variables that occurred in the pattern on the left-hand side will have the exact same values as the corresponding parts of the value from the right-hand side. (If the pattern is only a single variable, as in "`X = 42`", then the corresponding part is of course the entire right-hand side value.) To illustrate, try the following in the shell:

```
1> {A, B, C} = {1970, "Richard", male}.
{1970,"Richard",male}
2> A.
1970
3> B.
"Richard"
4> C.
male
5>
```

It should not be hard to see what is happening here. The pattern `{A,B,C}` matched the right-hand side tuple, and as a result, the variables were bound to the corresponding elements, so we can refer to them afterwards. It couldn't be any simpler.

This shows another common kind of match:

```
1> {rectangle, Width, Height} = {rectangle, 200, 100}.
{rectangle,200,100}
2> Width.
200
3> Height.
100
4>
```

Here, the pattern required that the first element of the tuple was an atom `rectangle` (used as a label). Since the right-hand side tuple had a matching atom as its first element, and had three elements as required, the match succeeded, and the variables `Width` and `Height` became bound.

A variable can occur several times in a pattern, and this is sometimes useful when you want to specify that two fields must have the same value:

```
1> {point, X, X} = {point, 2, 2}.
{point,2,2}
2> X.
2
3>
```

If the fields aren't exactly equal, the match will fail:

```
1> {point, X, X} = {point, 1, 2}.
** exception error: no match of right hand side value {1,2}
2>
```

Because of the single-assignment variables, it is impossible to give X both the value 1 and 2, no matter which one you start with.

2.4.4 – More about patterns

Patterns look like expressions, but are more limited. They can only contain variables, constants, and constant data structures like lists and tuples; no operators, function calls, funs, etc. They can be arbitrarily complicated and nested, though. For example, let's first create a list containing some information about users of some system (only one, right now):

```
1> Users = [{person, [{name,"Martin","Logan"}, {shoe_size,12},
{tags,[jujitsu,beer,erlang]}]}].
…
2>
```

(The shell will print back the value of what we just entered; we skipped that for clarity). Now, let's extract some selected data about the first user in the list:

```
2> [ {person, [{name,_,Surname},_,{tags, Tags}]} | _ ] = Users.
…
3> Surname.
"Logan"
4> Tags.
[jujitsu,beer,erlang]
5>
```

First of all, note the use of a single underscore ("_") to indicate a *don't-care pattern*. In other words, where we wrote "_", we don't care what value the right-hand side has at that point, and we don't want to know. We can have several underscores in the same pattern, as above, but they don't have to have the same values in all places (like variables do). These don't-care patterns are sometimes referred to as "anonymous variables", but they are not really variables at all, just placeholders.

Second, look at the outermost part of the pattern, in particular the last part before the =. This has the form `[ … | _ ]`. To understand this, we need to recall what we said about lists in Section 2.2.10: lists are made up of list cells, forming a chain, and a single list cell is written `[…|…]`. You can also visualize the cells as layers upon layers, like an onion, with an empty list as the center, and each layer carrying some data.

The pattern above can then be read out as follows:

> Something that consists of an outermost list cell, whose inner layers I don't care about for now (the "`| _ ]`" part of the pattern), but whose payload (in the "`[ … |`" part of the pattern) has the shape `{person, […, …, …]}`, i.e., is a 2-tuple tagged as `person`, whose second element is a list of exactly three elements; the first of these elements is a 3-tuple labeled `name`, and I want the third element of that—let's call it `Surname`; the third is a 2-tuple labeled `tags`, and I want the second element of that one—let's call it `Tags`.

Congratulations if you got through all that. But it does show that what a pattern can express in a single line and less than 50 characters, can be a rather long-winded business if you spell it out (which you need to do to get the same effect in most other languages). Patterns are natural, compact, readable, and very powerful.

MATCHING STRING PREFIXES USING ++

As you recall, strings in Erlang (written within double-quotes) are really just lists of character codes. This makes matching on string prefixes particularly easy. First, take this example of matching a simple list prefix:

```
[1,2,3 | Rest] = [1,2,3,4,5,6,7]
```

Since the left-hand side pattern has the same three initial elements as the list on the right-hand side, they match, and as a result, the variable `Rest` will be bound to the list cells that follow the 3, that is, `Rest = [4,5,6,7]`.

But strings are just lists of character codes, and we could get the code point for a character through the `$`-syntax (e.g., `$A` yields 65), so the following also works:

```
[$h, $t, $t, $p, $: | Rest] = "http://www.erlang.org"
```

which should give us the string `"//www.erlang.org"` in `Rest`. This is nice, but can be nicer. We said before that operators are not allowed in patterns. There is one exception: the `++` operator, which is used to append strings. This can be used in patterns if and only if its left argument is a constant string. Our example can then be written as simply as:

```
"http://" ++ Rest = "http://www.erlang.org"
```

(To make things more interesting, we included the two slashes in the pattern as well here, giving us `"www.erlang.org"` in `Rest`.) There is no magic behind this; it's just that what the `++` operator would do if you said `"abc" ++ SomeString`, is to create a list on the form `[$a, $b, $c | SomeString]`, like the pattern we wrote by hand above. Obviously, for the compiler to be able to do this expansion, the left argument of `++` must be a constant string at compile time.

The ease with which you can do basic string matching is probably the reason why you don't tend to see a lot of regular expressions in Erlang programs. They would usually be overkill, and have more overhead, for straightforward matches like this.

2.5 – Functions and clauses

That was a lot of basics before we could start talking about functions! But we really wanted to make sure you understand variables and pattern matching first, because after that, you'll have no trouble at all understanding how functions work in Erlang. Not that the concept of functions is much different compared to other languages, but in Erlang, functions and pattern matching are intimately connected. Though this can be very intuitive, it can at the same time take some getting used to.

You saw a simple function definition already, in Section 2.3.4, when we introduced modules. From now on, we will not be doing so many examples in the shell as we will be writing our own modules (though we will use the shell for compiling and running our code). For these initial examples, we will simply continue using the module you created back then, the one called "`my_module`", and add some new functions to it so we can try them out. Just remember to add the function name (with the correct arity) to the "`-export([…])`" list, and re-compile the module using `c(my_module)` before you try to call the function. If you see an error message saying something like "undefined function my_module:xxx/N", it is

probably because you forgot to do either of these things. If you get the error message "undefined shell command xxx/N", you forgot to write "`my_module:`" before the function name when you tried to call it.

2.5.1 – A function with side effects: printing text

Let's start out with something basic: taking some input and printing some text to the console. The standard library function io:format(…) is the normal way of writing text to the standard output stream in Erlang. It takes two arguments: the first is a format string, and the second is a list of terms to be printed. We'll use this in our own function, simply called "print", which has a variable "Term" as parameter:

```
print(Term) ->
    io:format("The value of Term is: ~p.~n", [Term]).
```

Write this function in your module, add `print/1` to the export list, compile the module again from the shell using `c(my_module)`, and then call `my_module:print("hello")`. You should see the following:

```
1> c(my_module).
{ok,my_module}
2> my_module:print("hello").
The value of Term is: "hello".
ok
3>
```

The escape code "~p" in the format string above means "pretty-print an Erlang term". This means that lists of printable characters will be displayed as double-quoted strings, and also that if the term is too large to fit on one line, it will be split up over several lines (with suitable indentation). Try calling your new print function with some different values from the data types that you got to know in Section 2.2. What happens if you try to print the list `[65,66,67]`?

(The escape code "~n" used above means "insert a line break", so we get a new line after the message, but you probably figured that out already.)

If you now change the "~p" to "~w" (do this!) and recompile, and then call the function again with `my_module:print("hello")` as before, you will see this instead:

```
5> c(my_module).
{ok,my_module}
6> my_module:print("hello").
The value of Term is: [104,101,108,108,111].
ok
7>
```

What's that ugly list? Well, a string is a list of character codes, remember? The escape code "~w" means "print an Erlang term in its raw form", without fancy line breaking and without printing lists as double-quoted strings even if they contain only printable character codes.

The function io:format(…) is an example of a function that has *side effects*. It does return a result, as you can see (the atom 'ok'), but its main purpose is to have an effect on the environment around it. (And by extension, the same is true for your `print` function.) In Erlang, practically all side effects can be seen as *messages* (and they usually are in practice too). In this case, the `io:format` function just prepares the text to be printed, and then sends it off as a message to the console driver before it returns 'ok'.

Finally, note that you have already started doing *interactive development* in the Erlang shell: you changed your program, recompiled and loaded the new version, and tried it out, without ever stopping and restarting the Erlang environment itself. If your Erlang system had happened to be doing something of interest in the background (like serving up web pages to customers), it would still be merrily chugging along while you keep fixing things.

2.5.2 – Multiple clauses and pattern matching for choice

Next, we'll look at where pattern matching comes in. In Erlang, a function can consist of more than one *clause*. Where the example in the previous section had a single clause, the following example has two clauses:

```
either_or_both(true, _) ->
    true;
either_or_both(_, true) ->
    true;
either_or_both(false, false) ->
    false.
```

Our function `either_or_both/2` here is an example of a Boolean function—one that operates on the values `true` and `false` (which are ordinary atoms in Erlang, remember?). As its name indicates, we want it to behave like the built-in `or` operator: if either of the arguments, or both, are `true`, the result should also be `true`, otherwise the result should be `false`. And no non-Boolean inputs are accepted.

Note that the clauses are separated by semicolons (`;`), and that only the last clause is terminated by a period (`.`). All the clauses must begin with the same name, and have the same number of arguments, and they must be defined together—you cannot put another function definition between two clauses of the same function.

CLAUSE SELECTION

When the function is called, Erlang will try the clauses in top-down order using pattern matching: first, it will match the incoming arguments against the patterns in the first clause; if they don't match, the next clause is tried, and so on. In our example, it means that if the first argument is `true`, the first clause will always be chosen (no matter what the second argument is—note that we have used a don't-care pattern in its place).

However, if the first clause doesn't match, then if the second argument is `true`, the second clause of our example will be chosen. But if that clause doesn't match either, the third clause is tried, and if that still doesn't match, we'll get a run-time exception of the type `function_clause` to indicate that the arguments didn't match *any* of the clauses of the function in this call.

Now, take another look at these clauses and think about what we know at each point. If we ever get to the second clause, we *know* that the first argument is not `true` (because otherwise the first clause would match). Likewise, if we get as far as the third clause, we know that *neither* of the arguments can be `true`. The only valid possibility left at that point is that both arguments are `false` (if the function was called with only `true` or `false` as argument values).

It is good programming practice to make this knowledge explicit in the code—that's why we don't accept anything else than `(false, false)` in the last clause. It means that if someone calls this function with some unexpected value like `foo` or `42`, he will get a run-time exception (`function_clause`), which is what we *want*: it means that he will *fail early*, so he will get a chance to detect his mistake and fix his code as soon as possible, and so that bad data does not propagate further throughout the system. If we had tried to be "nice" and said `(_, _)` in the last clause, to just return `false` in all remaining cases, then a call such as `either_or_both(foo, bar)` would also return `false` without any hint of a problem.

2.5.3 – Guards

There is still a possibility of some nonsense slipping through here, though. If someone would call the above function as `either_or_both(true, 42)` or as `either_or_both(foo, true)`, then it would quietly return `true` as if all was well in the world. We can add some extra requirements to the clauses to plug this hole, using *guards*:

```
either_or_both(true, B) when is_boolean(B) ->
    true;
either_or_both(A, true) when is_boolean(A) ->
    true;
either_or_both(false, false) ->
    false.
```

A clause guard begins with the keyword `when` and ends at the `->` arrow. It contains one or more tests, separated by commas if there are more than one, and all of them have to be true for the clause to be selected. As you can see, we need to use variables in the patterns to be able to refer to them in the guard, so we used the names A and B here instead of don't-care patterns. The `is_boolean(…)` test is one of those built-in functions that you can call without specifying the module name (they live in the `erlang` module). There are similar tests for all of the primitive data types: `is_atom(…)`, `is_integer(…)`, etc. The `is_boolean(…)` test simply checks that the value is one of the atoms `true` and `false`.

Apart from such *type tests*, the number of things you can do within a guard is strictly limited. You can use most of the operators (`+`, `-`, `*`, `/`, `++`, etc.), and some of the built-in

functions (like `self()`), but you cannot call your own functions, or functions in another module. This is partly for efficiency reasons—you want to be sure that clause selection is fast—but mostly because of possible side effects. It is important that if a guard *fails* (turns out to be false), we should be able to go on and try the next clause *as if nothing had happened*. If you for example would be able to somehow send a message from inside a guard, but then that guard fails, it would be impossible to "undo" that message—someone might already have seen it and as a consequence performed some visible change to the world, like changing a file or printing some text. Erlang does not allow this, and that makes guards (and clauses in general) much easier to reason about, reorder, and refactor.

Make sure you add this function to your module and try it out, first the initial version and then the one with guards. Give them some different inputs and check their behavior. We hope you see how this ability to experiment with functions in Erlang, test them interactively, and incrementally improve them without restarting the runtime environment can be a big boost both for productivity and for creativity.

2.5.4 – Patterns, clauses, and variable scope

Let's take another example of pattern matching, to show how we use it to both select clauses and extract the interesting data at the same time. The following function assumes that we are using tagged tuples to represent information about different geometric shapes:

```
area({circle, Radius}) ->
    Radius * Radius * math:pi();
area({square, Side}) ->
    Side * Side;
area({rectangle, Height, Width}) ->
    Height * Width.
```

If you for instance call `my_module:area({square, 5})`, you should get 25. If you pass it `{rectangle, 3, 4}` it returns 12, and so on. Pattern matching decides which clause will be selected, but it also binds the variables to the elements of the data structure so that we can refer to these values in the body of each clause. Note that as opposed to the `either_or_both` function above, the order of the clauses in this function does not really matter, because only one can match at any time; we say that they are *mutually exclusive*.

The *scope*, or lifetime, of a variable that is bound in the head of a function clause is the entire clause, up until the semicolon or period that ends that clause. For example, in the `area` function above, we used different variable names for Radius (of a circle) and Side (of a square), but we could also have called both X if we wanted, because they live in separate clauses. On the other hand, the Height and Width variables (for a rectangle) must have distinct names, since they have overlapping scope. You never need to declare variables in Erlang—you just use them as you need them; however, this convention means that you cannot reuse the same name within the same clause.

When a variable goes out of scope, the value that it referred to will become a candidate for garbage collection (i.e., the memory it used will be recycled) if no other part of the program still needs it. That's another thing you rarely need to think about in Erlang.

2.6 – Case- and if-expressions

If function clauses were the only way to make control flow branches in Erlang, you'd have to invent a new function name for each little choice to be made in your program. Although it might be very pedagogical, it could also be pretty annoying. Fortunately, Erlang provides `case`-expressions for this purpose. They also have one or more clauses, but can only have one pattern per clause (so no parentheses are needed).

For example, the `area` function from Section 2.5.4 could also be written using a `case` expression:

```
area(Shape) ->
    case Shape of
        {circle, Radius} ->
            Radius * Radius * math:pi();
        {square, Side} ->
            Side * Side;
        {rectangle, Height, Width} ->
            Height * Width
    end.
```

Note that we had to give the input to `area` a name, so we could refer to it as the value the case should switch on (`case Shape of …`). Also note that all the clauses are separated by semicolons, just like with function clauses, and that the entire case-expression must end with the keyword `end`. (There is no semicolon after the last clause—they are separators, not terminators.) In this particular case, the new version of the function is arguably *less* readable, because of the extra variable and the `case`/`of`/`end` keywords, and most Erlang programmers would prefer the original (even if that repeats the function name three times).

When you want to switch on multiple items using a `case` expression, you have to group them using tuple notation. For example, the `either_or_both` function from Section 2.5.3 could be written as follows:

```
either_or_both(A, B) ->
    case {A, B} of
        {true, B} when is_boolean(B) ->
            true;
        {A, true} when is_boolean(A) ->
            true;
        {false, false} ->
            False
    end.
```

As you can see, you can use guards (`when …`) in case-clauses as well. Again, you may or may not prefer the original version of the function as being more succinct.

2.6.1 – Boolean if-then-else switches in Erlang

Surprise: there aren't any! You simply use a `case`-expression instead, like the following:

```
case either_or_both(X, Y) of
    true  -> io:format("yes~n");
    false -> io:format("no~n")
end
```

Although it can be tempting to use an underscore as a catch-all pattern in the last case, don't do that. Spell out both the `true` and the `false` cases. This ensures that your program fails early, in case the input to the switch happens to be something else than `true`/`false`, and also helps program analysis tools see what the intention of the programmer was.

2.6.2 – if-expressions

As a special case, `if`-expressions are a stripped down variant of `case`-expressions, without a specific value to switch on, and without patterns. You can use an if-expression when you want to have one or more clauses that only depend on what is in the guards. For example:

```
sign(N) when is_number(N) ->
    if
       N > 0 -> positive;
       N < 0 -> negative;
       true  -> zero
    end.
```

This could also have been written using a `case` with a dummy switch value (and using don't-care underscores as patterns in all clauses):

```
sign(N) when is_number(N) ->
    case dummy of
       _ when N > 0 -> positive;
       _ when N < 0 -> negative;
       _ when true  -> zero
    end.
```

Hopefully, that also makes you see why the last catch-all clause in the `if`-expression was written "`true -> …`". If the guard test is always true, the clause will always match.

The if-expressions were added to the language a long time ago, a bit on a whim. They are not used very often, because most switches tend to have some kind of pattern involved anyway. Although they come in handy on occasion, a long-standing complaint from Erlang programmers has been that they are mostly a waste of the keyword "`if`". It is just one of those things that are hard to change in a language in retrospect. As a beginner, what you need to remember is that the conditions you switch on cannot be any old expressions—they are *guard tests*, and as such they are limited (see Section 2.5.3).

2.7 – Funs

We introduced funs briefly in Section 2.2.8, but it is not until now, after we have shown how functions and clauses work, that we are ready to talk about how funs are created.

2.7.1 – Funs as aliases for existing functions

If you have a function in the same module, for example, `either_or_both/2`, and you want to refer to it so you can say to some other part of the program "please call this function", then you can create a fun by saying:

```
fun either_or_both/2
```

and like any value, you can bind it to a variable:

```
F = fun either_or_both/2
```

or pass it directly to another function:

```
yesno(fun either_or_both/2)
```

and if you get a fun-value from somewhere, you can call it like any old function, like this:

```
yesno(F) ->
    case F(true, false) of
        true  -> io:format("yes~n");
        false -> io:format("no~n")
    end.
```

This means that it is simple to parameterize behavior. The same function (`yesno`) can be used in different ways depending on which funs you give it. In this example, the input parameter F is expected to be some kind of condition that takes two Boolean arguments, but apart from that, it could be doing anything.

HIGHER ORDER FUNCTIONS

The function `yesno/1` above is what is called a *higher order function*: one that gets a fun as input, or returns a fun as result, or both. Funs and higher order functions are very useful indeed; they are used for all sorts of things that "delegates" and "adapters" and "commands" and "strategies", etc., are used for in Object-Oriented languages.

Note that these "local" alias funs are similar in implementation to *anonymous* funs (explained below), and are *tied to the current version of the module*. See "Local funs have a short best-before date" in the next section for more details.

REMOTE ALIAS FUNS

If you want to create a fun that refers to a function which exists in some *other* module, you can use the following syntax (it also works for exported functions in the same module):

```
fun other_module:some_function/2
```

These "remote" alias funs have a different behavior with respect to code loading: they are not tied to any particular version of the function they refer to. Instead, they will always be directed to the latest available version of the function whenever the fun is called. Such fun-values are merely symbolic references to the function, and can be stored for any period of time and/or passed between Erlang systems without problems.

2.7.2 – Anonymous funs

While the alias funs discussed above are quite useful, the real power comes with the syntax for *anonymous* funs, also known as "lambda expressions". Like the above funs, they start with the `fun` keyword, and like a case-expression they end with the `end` keyword. Between those keywords, they simply look like one or more function clauses without any function names. For example, here is the simplest possible anonymous fun: it takes no arguments and always returns zero:

```
fun () -> 0 end
```

Here, on the other hand, is a more complicated one—it does exactly the same thing as the `area` function from Section 2.5.4, but it has no name:

```
fun ({circle, Radius}) ->
        Radius * Radius * math:pi();
    ({square, Side}) ->
        Side * Side;
    ({rectangle, Height, Width}) ->
        Height * Width
end
```

Obviously, to make any use of anonymous funs we either have to bind them to a variable, or pass them directly to some other function, like we did with the `yesno/1` function in Section 2.7.1., for example:

```
yesno( fun (A, B) ->  A or B  end )
```

LOCAL FUNS HAVE A SHORT BEST-BEFORE DATE

When you create an anonymous fun, or a fun as an alias for a local function, the fun-value is tied to that particular version of the code. If you reload the module that it belongs to more than once, the fun will no longer work: it will throw an exception if someone attempts to call it. Hence, it is not a good idea to keep such fun-values around

for a long time (e.g., by storing them in a database). Also, if you send them in a message to a different Erlang system, then that system must have the exact same version of the code for the fun to work. Remote alias funs are better for such purposes.

CLOSURES

The word "closure" is often used interchangeably with "fun" or "lambda expression", but more specifically it refers to the common and extremely useful case when you are accessing variables within the "`fun … end`" that are bound *outside* the fun. The fun-value will then also encapsulate the current values of those variables.

To make it more concrete, let's say you have a list of text strings, and a function `to_html` (not shown here) that will create an HTML document containing these strings marked up as a bullet list or similar. Furthermore, for every second string (after it has properly HTML-escaped it), it will apply a fun that you provide, to wrap the string in some additional markup of your choice. We could for example make every other line bold, like this:

```
to_html(Items, fun (Text) -> "<b>" ++ Text ++ "</b>" end)
```

But suppose we want to make the exact kind of emphasis a parameter; something that is passed to this part of our program from somewhere else, in a variable. We can then simply use that variable in the fun:

```
render(Items, Em) ->
    to_html(Items,
            fun (Text) ->
                "<" ++ Em ++ ">" ++ Text ++ "</" ++ Em ++ ">"
            end).
```

Such a fun will include, as a snapshot, the current values of those variables that it uses that are bound outside the fun itself. Because Erlang's single assignment and referential transparency properties guarantee that these values cannot be changed by anyone, we know that whether we call the fun right away, or not until later on, it will have exactly the same values for these variables as when it was created. (Of course, we can create multiple instances of the same fun, each with possibly different values for the externally bound variables, but each of those instances lives its own isolated life.)

The fun above is the meeting place for three different sources of information: the caller of `render`, who says whether to use `"b"` or `"i"` or something else for every second string; the `to_html` function that does the main job of transforming the items to HTML; and the `render` function itself, which specifies exactly how the additional markup is added (that is, what the fun really does). Note that it is a requirement from `to_html` that the callback fun should take one argument, which must be a string. The interface of the callback becomes part of the interface of `to_html`.

2.8 – Exceptions, try and catch

We have mentioned exceptions without further explanation up until now. What, then, is an exception? You could say that it is an alternative way of returning from a function, with the difference that it just keeps going back to the caller, and to the caller's caller, etc., until either someone catches it or it reaches the initial call of the process (in which case the process dies).

There are three classes of exceptions in Erlang:

error

This is the "run-time error" kind of exception, caused by things like division by zero, failing match operations, no matching function clause, et cetera. They also have the property that if a process dies because of an error exception, it will be reported to the Erlang error logger.

exit

This kind of exception is used to signal "this process is giving up". It is generally expected not to be caught, but to cause the process to die and let others know why it quit. Exit can also be used for normal termination, to quit and signal "job done, all OK". In either case, process termination due to exit (for whatever reason) is not considered unexpected, so it is not reported to the error logger.

throw

This kind of exception is for user-defined purposes. You can use throws to signal that your function encountered something unexpected (like a missing file, or just bad input), or to perform a so-called "non-local return" or "long jump" out of a deep recursion. A throw that is not caught by the process will mutate into an error exception with the reason "`nocatch`", terminating the process and logging it.

2.8.1 – Throwing (raising) exceptions

For each of the above classes, there is a corresponding built-in function to throw (or "raise") such an exception:

```
throw(SomeTerm)
exit(Reason)
erlang:error(Reason)
```

Since `throw` and `exit` are quite common, they are auto-imported: you don't need to prefix them with "erlang:". In normal code you typically don't need to raise `error` exceptions (but it can be a good thing to do if you are writing a library and want to throw errors like "`badarg`" just like Erlang's standard library functions).

As a special case, if a process calls `exit(normal)`, and the exception is not caught, that process will terminate as if it had simply finished the job it was spawned to do. This means that other (linked) processes will not regard it as an abnormal termination (as they will for all other exit reasons).

2.8.2 – Using try…catch

In modern Erlang, you use a `try`-expression to handle exceptions that may occur in a piece of code. In most ways it works like a case-expression, and in the simplest form it looks like this:

```
try
    some_unsafe_function(…)
catch
    oops          -> got_throw_oops;
    throw:Other   -> {got_throw, Other}
    exit:Reason   -> {got_exit, Reason}
    error:Reason -> {got_error, Reason}
end
```

Between `try` and `catch`, we have the *body* or *protected section*. Any exception that occurs within the body and tries to propagate out from it will be caught and matched against the clauses listed between `catch` and `end`. If it doesn't match any of the clauses, the exception will continue as if there was no `try`-expression at all around the body. Similarly, if the body is evaluated without raising an exception, its result will become the result of the whole expression, as if the `try` and `catch…end` had not been there. The only difference is when there is an exception, and it matches one of the clauses. In that case, the result becomes whatever the matching clause returns.

The patterns of these clauses are a bit special—they may contain a colon (:) to separate the exception class (`error`, `exit`, or `throw`) from the thrown term. If you leave out the class, it defaults to `throw`. You shouldn't normally catch `error` and `exit` unless you know what you are doing—it goes against the idea of failing early, and you could be masking a real problem. Sometimes, though, you just want to run some code that you don't trust too much, and catch anything that it throws. You can use the following pattern for catching *all* exceptions:

```
_:_  ->  got_some_exception
```

(or "`Class:Term -> …`" if you want to inspect the data in the exception).

Also, note that once you get to the `catch` part, the code is no longer protected. If a new exception happens in a catch-clause, it will simply propagate out of the entire `try`-expression.

2.8.3 – try…of…catch

There is a longer form of `try` that is useful when you need to do quite different things in the successful cases and the exception cases. For instance, if you want to continue doing some work with the value you got in the successful case, but want to print an error message and give up in the exception case, you can add an "`of…`" section, like this:

```
try
```

```
    some_unsafe_function(…)
of
    0 -> io:format("nothing to do~n");
    N -> do_something_with(N)
catch
    _:_ -> io:format("some problem~n")
end
```

Because it is quite common that the thing you immediately want to do in the successful case (apart from giving a name to the result) is to switch on the value that you got, we simply let you write one or more clauses between `of` and `catch`, just like those in a case-expression, for what should happen if the `try…of` part succeeds. Just note that the `of…` part, just like the catch-clauses, is no longer protected—if an exception occurs there, it will not be caught by this try-expression.

2.8.4 – after

Finally (if you'll pardon the Java reference), you can add an `after` section to any `try`-expression. Its purpose is to guarantee that a piece of code is executed for the sake of its side effects, no matter what happens in the rest of the try-expression, just before we are about to leave it. This usually involves deallocating a resource in some way or other; for example, to guarantee that a file is closed, as in this example:

```
{ok, FileHandle} = file:open("foo.txt", [read]),
try
    do_something_with_file(FileHandle)
after
    file:close(FileHandle)
end
```

Here, if the match "`{ok,FileHandle}=…`" works, we know we have successfully opened the file. We then immediately enter a try-expression whose `after` section ensures that the file will be closed, even if an exception occurs.

Note that if you have an `after` part, you don't need a `catch` part (but you can of course have that, and an `of` part too if you like). In either case, the `after` part is not executed until the entire try-expression is ready, and this includes the situation where a new exception is thrown from the `of` part or from one of the catch-clauses. If so, that exception is temporarily put on hold while the `after` part runs, and is then re-thrown. If the `after` part should throw an exception, that takes over and any suspended exception is forgotten.

2.8.5 – Getting a stack trace

Normally, the execution stack trace is not included in the part of the exception that you can see, but it is stored internally. You can inspect the stack trace of the latest thrown exception of the current process by calling the built-in function `erlang:get_stacktrace()`.

The stack trace is a list, in reverse order (last call first), of the calls nearest the top of the stack when the exception occurred. Each function is represented as `{Module, Function, Args}`, where Module and Function are atoms, and Args is either the arity of the function, or the actual list of arguments to the call, depending on what information was available at the time. Typically, only the topmost call might have an argument list.

Note that if you call erlang:get_stacktrace() and get an empty list, it simply means that no exception has been caught by this process yet.

2.8.6 – Re-throwing

It might happen that you need to examine an exception more closely before you decide whether to catch it or not. Although this is unusual, you can then catch it first and re-throw it if necessary, using the built-in function erlang:raise(Class, Reason, Stacktrace). Here, Class must be one of error, exit, or throw, and Stacktrace should be what you got from erlang:get_stacktrace(). For example:

```
try
  do_something(…)
catch
    Class:Reason ->
        Trace = erlang:get_stacktrace(),
        case analyze_exc(Class, Reason) of
            true  -> handle_exc(Class, Reason, Trace),
            false -> erlang:raise(Class, Reason, Trace)
        end
end
```

In the above, we catch *any* exception, analyze it, and either handle it ourselves or re-throw it. It is however both messy and inefficient (since it requires creating the symbolic stack trace as a list of tuples) and should only be done if you see no better solution.

2.8.7 – Plain old catch

Before `try`–expressions were added to the language, there was just `catch`. You'll see a lot of this in older code, since it was the only way of handling exceptions. It works like this: "`catch Expression`" will evaluate Expression (which can be any expression), and if it produces a result (i.e., doesn't throw an exception), you'll simply get that result. Otherwise, if there is an exception, it will be caught and presented as the result of the `catch`, using different conventions depending on the exception class. This shell dialog demonstrates the different cases:

```
1> catch throw(foo).
foo
2> catch exit(foo).
{'EXIT',foo}
3> catch foo=bar.
{'EXIT',{{badmatch,bar},[{erl_eval,expr,3}]}}
```

In brief, for `throw` you get the thrown term as it is, for `exit` you get the reason in a tagged tuple ('EXIT' is an atom in all-uppercase, to be "hard to fake by mistake"), and for `error`, you get it tagged and packed up along with the stack trace. This might look handy, but there is a lot of confusion going on that makes it hard or impossible to tell exactly what has happened, and how to proceed. You should avoid plain `catch`, but you'll likely need to understand what it is doing when you see it in older code.

2.9 – List comprehensions

A "comprehension" is a compact notation for describing operations on sets or sequences of items (like lists). You may already know it from ordinary mathematical set notation, where e.g., {x | x ∈ ***N***, x > 0} is read as "all values x such that x comes from the natural numbers (denoted by ***N***), and x is greater than zero", or in other words, all positive natural numbers.

If you're not already familiar with set notation, take a close look at the above example, and you'll quickly see how it works. The vertical bar "|" separates the *template* part that describes how the individual elements are made up, from the *generators and conditions* part that specifies what the sources for elements are and what restrictions we have. In this example, the template is simply "x", we have a generator that says "for all values x in N", and a condition that only those x that are greater than zero may be part of the result. Pretty simple stuff, really, but it's a very efficient way of expressing these kinds of operations.

2.9.1 – List comprehension notation

Erlang is a programming language, though, and not pure mathematics. We can use the same ideas in our notation, but we must be a bit more concrete. In particular, the order of elements becomes more important, as well as what data structures we use. In Erlang, the first choice for representing a sequence of items is of course a *list*, so we get a *list comprehension*. We just have to adapt the notation a bit. For example, if we have an existing list of integers, both positive and negative, we can easily create a new list containing only the positive ones (preserving their relative order in the list), like this:

```
[ X  ||  X <- ListOfIntegers,  X > 0 ]
```

Note that we must use double vertical bars "`||`", because the single vertical bar is already used for plain list cells. Apart from that, we write `[…]` as usual for a list. We don't have "∈" on our keyboard, so a left arrow "`<-`" is used to denote a generator, and anything else to the right of the "`||`" that's not a generator must be a conditional, such as "X > 0". The template part can be any expression, and can use any variables that are bound to the right of the vertical bars (such as X, which is bound by the generator) or that are bound outside the list comprehension.

Furthermore, if you have more than one generator in the comprehension, it will cause it to go through all combinations of elements, as if you had nested loops. This is rarely useful, but on occasion it can turn out to be just what you need.

2.9.2 – Mapping, filtering, and pattern matching

A single list comprehension can perform any combination of *map* and *filter* operations, where "map" means that we perform some operation on the elements before we put them in the resulting list. For example, the following list comprehension selects only positive even numbers from the source list (`rem` is the "remainder" operation), and squares them:

```
[ math:pow(X,2) || X <- ListOfIntegers, X > 0, X rem 2 == 0 ]
```

The greatest power, however, comes via pattern matching. In a generator, the left hand side of the "<–" arrow does not have to be a variable—it can be any pattern, just like in a match operation ("="). This means that generators already have a built-in condition: *only those elements that match the pattern are considered*; any others are silently skipped. Furthermore, patterns let us extract parts of the elements for use in conditions or in the template section. For example, assume that we have a list of tuples representing geometric shapes, as in the `area` function of Section 2.5.4. We can then select, say, only those rectangles whose area is at least 10 (and no other shapes), and create a corresponding list of areas, like this:

```
[ {area, H*W} || {rectangle, H, W} <- Shapes, H*W >= 10 ]
```

You should learn to use list comprehensions when you can. Apart from being efficient, they are generally the most compact and readable way of expressing this type of operations.

2.10 – Bit syntax and bitstring comprehensions

We introduced binaries and general bitstrings in Section 2.2.2, but we only showed examples of how to create plain binaries (whose length in bits is divisible by 8, i.e., that can be viewed as a sequence of whole bytes). However, in modern Erlang, bitstrings can be of any length. The so-called *bit syntax* allows you to form new bitstrings of exactly the size and layout you want, and reversely, it can be used in patterns to match and extract segments from a bitstring (for example, binary data read from a file or from a socket). In combination with *comprehensions*, this notation becomes extremely powerful.

2.10.1 – Building a bitstring

A bitstring is written as `<<Segment1, …, SegmentN>>`, with zero or more segment specifiers between the double less-than/greater-than delimiters. The total length of the bitstring, in bits, is exactly the sum of the lengths of the segments.

A segment specifier can be on one of the following forms:

```
Data
Data:Size
Data/TypeSpecifiers
Data:Size/TypeSpecifiers
```

where Data must be an integer, a floating-point number, or another bitstring. You can specify the size of the segment, as an integer number of "units", and you can specify the segment type, which decides what Data is expected to be and how it should be encoded or decoded. For example, a simple binary like `<<1,2,3>>` has three segments that all have plain integers as data, and no size or type specifiers. In this case, the type defaults to be `integer`, and the default size for integers is 1. The unit for the `integer` type is 8 bits, so each segment will be encoded as an 8-bit unsigned byte. Similarly, `<<"abc">>` is just a short-hand for `<<$a,$b,$c>>`, that is, a sequence of 8-bit integer character codes (in Latin-1). If an integer needs more bits than the segment has room for, then it will be truncated to fit, so `<<254,255,256,257>>` becomes `<<254,255,0,1>>`.

The type of a segment depends only on what you have specified; it doesn't depend on what the actual Data happens to be from one time to the next. You might think that would be handy, but it goes against the "fail early" philosophy and could land you in some nasty situations, e.g., with bogus binary data written to a file. This means that we cannot for instance concatenate two bitstrings like this:

```
B1 = <<1,2>>,
B2 = <<3,4>>,
<<B1, B2>>
```

because by default it is assumed that B1 and B2 are integers. However, if we say that B1 and B2 are bitstrings, it works:

```
<<B1/bits, B2/bits>>
```

yielding <<1,2,3,4>>, as we wanted.

You can control the encoding and decodingTypeSpecifiers part of a segment (after a "`/`"). It consists of one or more atoms, separated by "`-`" as in "`integer-unsigned-big`". The order of the atoms is not significant. The current set of specifiers you can use is:

- `integer`, `float`, `binary`, `bytes`, `bitstring`, `bits`, `utf8`, `utf16`, `utf32`
- `signed`, `unsigned`
- `big`, `little`, `native`

and as a special case, you can include "`unit:Integer`". These can be combined in various ways, picking at most one from each group above. (`bits` is an alias for `bitstring` and `bytes` is an alias for `binary`). For the types `integer`, `float`, and `bitstring`, the size unit is 1 bit, while for `binary` the unit is 8 bits (whole bytes).

There are a lot more details that we don't have room to go through here, so you'll need to check up on the official documentation or read another book about Erlang programming if

you want to start working with binaries, but this should be enough to give you the idea. We'll just mention one important case:

UTF ENCODINGS IN BITSTRINGS

As a quite recent addition to Erlang, you can specify one of `utf8`, `utf16`, and `utf32` as the type of a bitstring segment, as you probably noted in the list above. These let you work with UTF-encoded characters in bitstrings. For example:

```
<<"Motörhead"/utf8>>
```

You cannot specify a size for such a segment, since the size is determined by the input. In the above example, the result uses 10 bytes to encode 9 characters.

2.10.2 – Pattern matching with bit syntax

Just as you can both construct and deconstruct tuples with the same syntax, you can deconstruct the data in a bitstring by using the same bit syntax. This makes parsing funny file formats and protocol data a much simpler task, and is much less error prone than doing manual bit shifting and masking. To just show a classic example, here is how you can parse the contents of an IP packet header, using a pattern in a function clause:

```
ipv4(<<Version:4, IHL:4, ToS:8, TotalLength:16,
       Identification:16, Flags:3, FragOffset:13,
       TimeToLive:8, Protocol:8, Checksum:16,
       SourceAddress:32, DestinationAddress:32,
       OptionsAndPadding:((IHL-5)*32)/bits,
       RemainingData/bytes >>) when Version =:= 4 ->
    …
```

Any incoming packet that is large enough to match, and whose Version field is 4, would be decoded into the above variables, most as integers, except for OptionsAndPadding (a bitstring whose length depends on the previously decoded IHL field) and the RemainingData segment that would simply contain all the data following the header. Extracting a binary from another like this does not involve copying the actual data, so it is a cheap operation.

2.10.3 – Bitstring comprehensions

The idea of list comprehensions, which exists in many functional programming languages, has been extended in Erlang to work with the bit syntax. A *bitstring comprehension* looks much like a list comprehension, but is enclosed in `<<…>>` rather than in `[…]`. For example, if we have a list of small numbers, all between 0 and 7, we can pack them into a bitstring using only 3 bits per number, like this:

```
<<  <<X:3>> ||  X <- [1,2,3,4,5,6,7]  >>
```

This returns a bitstring that the shell prints as "<<41,203,23:5>>". Note the 23:5 at the end—it means that the total length is 8 + 8 + 5 = 21 bits, which makes sense since we had 7 elements in the input list.

So how can we decode such a bitstring? With another bitstring comprehension, of course! The difference is that we need to use a generator written with a "<=", to pick out parts of the input (which is now a bitstring), instead of "<-" which only picks elements from lists:

```
<<  <<X:8>> ||  <<X:3>>  <=  <<41,203,23:5>>  >>
```

This yields the binary <<1,2,3,4,5,6,7>>, so we have successfully recoded a 3-bit-per-number format into an 8-bit-per-number format. But if we wanted the result as a list, not as another bitstring? Well, we can use a list comprehension with a bitstring generator!

```
[ X ||  <<X:3>> <= <<41,203,23:5>> ]
```

This produces the corresponding list [1,2,3,4,5,6,7]. We invite you to play a little with the bit syntax in the shell to get the hang of it. There are many interesting things you can do with the bit syntax and a little creativity.

2.11 – Record syntax

In order to keep the amount of strange syntax down in the previous sections, we have postponed explaining one of the more important parts of Erlang: the record syntax.

Tuples are the main building blocks in Erlang for most kinds of structured data, but software engineering-wise, they are not as flexible as we would like. Imagine that we write a whole program (perhaps many modules) around the fact that our representation of a customer (for example) is a tuple with five elements. If we then should find that we need to add another field, which is pretty likely as these things happen, we will be forced to go through all the code and edit every occurrence of such a tuple, both where they are created and in any patterns that match on them. Not to mention that this is rather error prone: what if you write a four-tuple instead of a five-tuple somewhere, or forget to add a field to one instance when you update all the others? To remedy this problem (but without sacrificing the speed and small memory footprint that you get with tuples), the *record syntax* was invented.

2.11.1 – Record declarations

The record syntax lets you work with "records", which are simply tagged tuples, in a way that avoids most of the problems with adding or removing fields and remembering in which order they occur in the tuple. The first thing you need to do is to write a *record declaration*, which looks like this:

```
-record(customer, {name="<anonymous>", address, phone}).
```

This tells the compiler that we will be working with 4-tuples (three fields plus the tag), where the first element is always the atom `customer`. The other fields will be in the same order as in the record declaration, so `name` is always the second field.

2.11.2 – Creating records

To create a new record-tuple, we use some variant of the following syntax:

```
#customer{}

#customer{phone="55512345"}

#customer{name="Sandy Claws", address="Christmas Town", phone="55554321"}
```

We always need to give the record name after the `#`, so the compiler can match it to the record declaration above. Within the `{…}`, we can choose to give values for any of the fields (or none), and in any order. (The compiler will make sure they are ordered as in the declaration.) Those fields that we didn't give values for will be set to the default, which is the atom `undefined` unless we have specified a default value in the declaration.

2.11.3 – Record fields and pattern matching

Assume that we bind the variable R to the second of the above examples. We can now access the individual fields by using a dot-notation:

```
R#customer.name       ->  "<anonymous>"
R#customer.address    ->  undefined
R#customer.phone      ->  "55512345"
```

As before, we need to specify the record name in order to tell the compiler "treat the tuple in R as a customer record". However, the most common way of extracting fields from a record is to use pattern matching. The following function takes a customer record as input, and checks that the phone number is not undefined:

```
print_contact(#customer{name=Name, address=Addr, phone=Phone})
          when Phone =/= undefined ->
    io:format("Contact: ~s at ~s.~n", [Name, Phone]).
```

It's just like matching on a tuple, except that you don't need to care about the exact number of fields, and their order. If you add or reorder fields in the record declaration, you can recompile the code and it will work as it did before.

2.11.3 – Updating record fields

As we have pointed out before, we don't really "update" parts of existing data structures in Erlang, at least not in place. What we do is create a new, slightly modified copy of the old data. For instance, if we want to update a tuple with 4 elements, we create a new 4-tuple and copy those elements that should be kept unchanged. That might sound expensive, but in

fact we never copy more than a single word per element—a "shallow" copy. And tuple creation is a quite fast operation: Erlang is optimized for creating (and recycling) large amounts of small tuples and list cells at high speed—they are "scratchpad" data just as much as they are used to represent more permanent data structures.

The notation for updating fields is very similar to that for creating new records, except that we need to say where the old record comes from. Say that we had the second customer record from the above section in R. The following will create a copy of R with the name and address fields updated:

```
R#customer{name="Jack Skellington", address="Hallowe'en"}
```

It is important to keep in mind that R itself is not changed in any way by this, and we cannot "reassign" R to hold the new value; if we want to put it in a variable, we would have to use another name such as R1. On the other hand, if it turns out that no part of the program is using the tuple in R anymore after the creation of R1, then R will be recycled automatically. (A clever compiler can sometimes see that it is safe to reuse the existing tuple in R to create the new R1, if it's just going to become garbage anyway after that.)

2.11.4 – Where to put the record declarations

For records that are only used within a single module, you usually write the record declarations at the top of the module, along with the export declarations and similar things that belong in the "header" of the module. But if you need to use the exact same record declaration in several modules, you have to do something different. Since you don't want multiple definitions of the same records duplicated over several source files (making it hard to remember to keep them all in sync), you should put those definitions that need to be shared in a separate so called *header file*, which will then be read by all the modules that need it. This is handled by the preprocessor, which is our next topic.

2.12 – Preprocessing and include files

Erlang has a preprocessor very similar to the one used in C and C++, which means that it is a *token-level* preprocessor. It works on the sequence of tokens produced by splitting the source file into separate words and symbols, rather than on the characters of the text. This makes it a bit easier to reason about, but also limits what it can do.

The preprocessor always runs as part of the compilation process, and performs three important tasks: *macro expansion*, *file inclusion*, and *conditional compilation*. We will look at these in order:

2.12.1 – Defining and using macros

You can define a macro with or without parameters using the `define` directive, as in the following examples:

```
-define(PI,  3.14).
```

```
-define(pair(X,Y),  {X, Y}).
```

Macro names can be written as Erlang variables or atoms, but it is traditional to use all-uppercase for constants, and mostly lowercase for other macros. To expand a macro at some point in the source code (following its definition), you must prefix it with a question mark:

```
circumference(Radius) -> Radius * 2 * ?PI.

pair_of_pairs(A, B, C, D) -> ?pair( ?pair(A, B), ?pair(C, D) ).
```

The above code will then be expanded to the following, just before proper compilation starts:

```
circumference(Radius) -> Radius * 2 * 3.14.

pair_of_pairs(A, B, C, D) -> { {A, B}, {C, D} }.
```

Macros are not a substitute for using proper functions, but are an escape route when normal functions won't do for the kind of abstraction you want to perform: when you need to be absolutely sure that the expansion is performed at compile time, or the syntax does not allow you to use a function call.

UN-DEFINING A MACRO

The undef directive can be used to remove a macro definition (if there is one). For example, after the following lines:

```
-define(foo, false).
-undef(foo).
-define(foo, true).
```

the `foo` macro will be defined to `true`.

USEFUL PRE-DEFINED MACROS

The preprocessor pre-defines certain macros for your convenience, and the most useful of them is probably the `MODULE` macro. It always expands to the name of the module that is being compiled, in the form of an atom. You can also use the `FILE` and `LINE` macros to get the current position in the source file. For example:

```
current_pos() -> [{module, ?MODULE}, {file, ?FILE}, {line, ?LINE}].
```

Macro definitions, like record declarations (as we mentioned in section 2.11), should be placed in a header file when the same definition needs to be shared between multiple source files. This brings us to the next feature:

2.12.2 – Include files

An Erlang source code file can include another file by using an *include directive*, which has the form "`-include("…").`". The text of the included file is read by the preprocessor and is inserted at the point of the include directive. Such files generally only contain declarations, not functions, and since they are typically included at the top of the source file for the module they are known as *header files*. By convention, an Erlang header file has the file name extension ".hrl".

To locate a file specified by a directive such as "`-include("some_file.hrl").`", the Erlang compiler will search in the current directory for the file called `some_file.hrl`, but also in any other directories that are listed in the *include path*. You can add directories to the include path by using the `-I` flag to `erlc`, or by an option `{i,Directory}` to the `c` shell function, as in:

```
1> c("src/my_module", [ {i, "../include/"} ]).
```

THE INCLUDE_LIB DIRECTIVE

If your code depends on a header file that is part of some other Erlang application or library, you would have to know where that application is installed so you can add its header file directory to the include path. In addition, the install path may contain some version number, so if you upgraded that application you might need to update the include path as well. Erlang has a special include directive for avoiding most of this trouble, called `include_lib`. For example,

```
-include_lib("kernel/include/file.hrl").
```

This will look for the file relative to the locations of the installed applications that the Erlang system knows about (in particular, all the standard libraries that came with the Erlang distribution). For example, the path to the `kernel` application could be something like "`C:\Program Files\erl5.6.5\lib\kernel-2.12.5`". The include_lib directive will match this path (stripping the version number) to the leading "`kernel/`" part of the file name, and look for an `include` subdirectory containing `file.hrl`. Even if the Erlang installation is upgraded, your source code doesn't need to be modified.

2.12.3 – Conditional compilation

Conditional compilation means that certain parts of the program may simply be skipped by the compiler, depending on some condition. This is often used to create different versions of the program, such as a special version for debugging. The following preprocessor directives control which parts of the code may be skipped, and when:

```
-ifdef(MacroName).

-ifndef(MacroName).
```

```
-else.

-endif.
```

As the names indicate, `ifdef` and `ifndef` test if a macro is defined, or is not defined. For each ifdef or ifndef, there must be a corresponding endif to end the conditional section. Optionally, a conditional section may be divided in two halves by an else. For example, the following code exports the function foo/1 only if the DEBUG macro is defined (to any value):

```
-ifdef(DEBUG).
-export([foo/1]).
-endif.
```

To control this from the command line or from your build system, you can define a macro either by giving an option `{d,MacroName,Value}` to the shell `c` function, or you can pass the option `-Dname=value` to the `erlc` command. Since the actual macro value doesn't really matter here, `true` is usually used.

Because Erlang's parser works on one period-terminated declaration (called a "form") at a time, you *cannot use conditional compilation in the middle of a function definition*, because the period after the `ifdef` would be read as the end of the function. Instead, you can conditionally define a macro, and use the macro within the function, like this:

```
-ifdef(DEBUG).
-define(show(X), io:format("The value of X is: ~w.~n", [X])).
-else.
-define(show(X), ok).
-endif.

foo(A) ->
    ?show(A),
    …
```

If the above is compiled with `DEBUG` defined, the `foo` function will print the value of A on the console before it continues with whatever the function is supposed to do. If not, the first thing in the function will simply be the atom `ok`, which is just a constant, and since it is not used for anything the compiler will optimize it away as if it had not been there.

2.13 – Processes

In chapter 1 we introduced processes, messages, and the concept of process links and signals, and we presented process identifiers ("pids") in Section 2.2.7. In this section, we will go through the most important things you should know about working with Erlang processes.

2.13.1 – Summary of process operations

The following are the main process-related operations. You'll recognize some of them from chapter 1, but there are a few new ones as well:

Spawning a process

```
Pid = spawn(fun …)
Pid = spawn(Module, Function, ListOfArgs)
```

Spawning and simultaneously linking a process

```
Pid = spawn_link(…)
```

Process ID of current process

```
Pid = self()
```

Sending messages

```
Pid ! Message
```

Checking the mailbox

```
receive … end
```

(See Section 2.13.2 below for details.)

Sending an exit signal to a process

```
exit(Pid, Reason)
```

(Does not terminate the sender. If Reason is `kill`, the signal is untrappable.)

Setting the trap_exit flag

```
process_flag(trap_exit, true)
```

(Incoming exit signals will be converted to harmless messages, with the exception of untrappable signals.)

Throwing an exception to end the process (unless caught)

```
exit(Reason)
```

Unidirectional monitoring of a process

```
Ref = monitor(process, Pid)
```

(If the monitored process identified by `Pid` dies, a message containing the unique reference `Ref` is sent to the process who set up the monitor.)

2.13.2 – Receiving messages, selective receive

The receiving process can extract messages from the mailbox queue using a `receive`-expression. Although incoming messages are queued up strictly in order of arrival, the receiver can decide which message to extract, and simply leave the others in the mailbox for later. This ability to ignore messages that are currently irrelevant (e.g., may have arrived early) is a key feature of Erlang's process communication. The general form for `receive` is:

```
receive
    Pattern1 when Guard1 -> Body1;
    …
    PatternN when GuardN -> BodyN
after Time ->
    TimeoutBody
end
```

The `after…` section is optional; if omitted, the receive will never time out. Otherwise, `Time` must be an integer number of milliseconds or the atom `infinity`. If `Time` is 0, it means that the receive will never block at all; in all other cases, then if no matching message is found in the process' mailbox, the receive will wait for such a message to arrive, or the timeout to occur, whichever happens first. The process will be suspended while it is waiting, and will only wake up in order to inspect new messages.

Each time a `receive` is entered, it starts by looking at the oldest message (at the head of the queue), tries to match it against the clauses just as in a `case`-expression, and if no clause matches it moves on to the next message. If the pattern of a clause matches the current message, and the guard succeeds (if there is one), the message is removed from the mailbox and the corresponding clause body is evaluated. If no message is found and a timeout occurs, the timeout-body is evaluated instead, and the mailbox remains unchanged.

2.13.3 – Registered processes

On each Erlang system, there is a local process registry—a simple name service, where processes can be registered. The same name can only be used by one process at a time, which means that this can only be used for "singleton" processes: typically system-like services of which there are at most one at a time on each run-time system. In fact, if you start the Erlang shell and call the built-in function `registered()`, you will see something like the following:

```
1> registered().
[rex,kernel_sup,global_name_server,standard_error_sup,
 inet_db,file_server_2,init,code_server,error_logger,
 user_drv,application_controller,standard_error,
 kernel_safe_sup,global_group,erl_prim_loader,user]
2>
```

Quite a bunch, as you can see. An Erlang system is indeed much like an operating system in itself, with a set of important system services running. (One is even called `init`…) We can find the pid currently registered under a name using the built-in `whereis` function:

```
2> whereis(user).
<0.24.0>
3>
```

We can even send messages directly to a process using only the registered name:

```
1> init ! {stop, stop}.
```

(Did you try it? It was a dirty trick, relying on knowledge about the format of messages between system processes. There are no guarantees that it won't change some day.)

If we start our own processes, we may register them with the `register` function:

```
1> Pid = spawn(timer, sleep, [60000]).
<0.34.0>
2> register(fred, Pid).
true
3> whereis(fred).
<0.34.0>
4> whereis(fred).
undefined
5>
```

(Note that in this example, the process we start will be finished after 60 seconds, and the name will then automatically go back to being undefined again.)

Suppose that a registered processes dies and the service it performed is restarted. The new process will then have a different process identifier. Instead of individually telling every process in the system that the service "`rex`" (for example) has a new pid, we can simply update the process registry. (However, there is a period during which the name does not point anywhere, until the service has been restarted and re-registered).

2.13.4 – Delivery of messages and signals

The messages sent between Erlang processes with the "`!`" operator are in fact just a special case of a more general system of signals. The *exit signals* that are sent from a dying process to its linked neighbors are the other main kind of signal, but there are a few others that are not directly visible to the programmer, such as the *link requests* that are sent when we try to link two processes. (Imagine that they are on different machines, and you'll see why they need to be signals. Since the link is bidirectional, both sides must know about the link.)

For all signals, there are a couple of basic delivery guarantees:

- If a process P1 sends out two signals S1 and S2, in that order, to the same

destination process P2 (regardless of what else it does between S1 and S2, and how far apart in time the signals are sent), then they will arrive in the same relative order at P2 (if both arrive). This means, among other things, that an exit signal can never overtake the last message sent before the process dies, and that a message can never overtake a link request. This is fundamental for process communication in Erlang.

- A best effort attempt is made to deliver all signals. Within the same Erlang run-time system, there is never a danger of losing messages in transit between two processes. However, between two Erlang systems connected over a network, it may happen that a message is dropped if network connection is lost (somewhat depending on the transport protocol). It the connection is then restored, it is possible that S2 in the above example eventually arrives, but S1 was lost.

For the most part, this simply means that you don't have to think much about message ordering and delivery—things tend to work much as you expect.

2.13.5 – The process dictionary

Each process has, as part of its state, its own private *process dictionary*, a simple hash table where you can store Erlang terms using any values as keys. The built-in functions `put(Key, Value)` and `get(Key)` are used to store and retrieve terms. We shall not say much about the process dictionary except give you some reasons why you should avoid using it, no matter how tempting it might seem.

- The simplest point is that it makes programs harder to reason about. You can no longer just look at the code and get the whole story. Instead, it suddenly depends on which process is running the code, and what its current state is.
- A more important point is that it makes it hard or impossible to do a handover from one process to the next, where you do half of the work in one process and the rest in another. The new process will simply not have the correct data in its own dictionary, unless you make sure to pack it up and ship it from the first process to the second.
- Finally, if you write some kind of library, and you use the process dictionary to store certain information between calls from the client (much like a web server uses cookies), it means that you force the client to use a single process for the duration of the session—if the client tries to call your API from a second process, it won't have the necessary context.

Using the process dictionary can in some cases be justified, but in general, there is a better solution to the problem of storing data (that even lets you share the information between processes if you like). It goes by the strange name of "ETS tables".

2.14 –ETS tables

ETS stands for "Erlang Term Storage". An ETS table, then, is simply a table containing Erlang terms (that is, any Erlang data), that can also be shared between processes. But surely that sounds like it goes against the fundamental ideas of referential transparency and avoiding

sharing. Are we suddenly smuggling in destructive updates through the back door? Two words: process semantics.

2.14.1 – Why ETS tables work like they do

The main design philosophy behind the ETS tables in Erlang is that such a table should look and feel almost exactly as if it was a separate process. Indeed, they could have been implemented as processes and still have the same interface. In practice, though, they are implemented in C as part of the Erlang run-time system, so they are lightweight and fast, and the interface functions are BIFs. This extra focus on efficiency is motivated because so many other things in Erlang are built on top of these ETS tables.

Of course we still want to avoid sharing data (when we can), and we particularly want to avoid the situation where things unexpectedly change behind someone's back. On the other hand, if you can implement a form of storage based on the normal semantics of processes and message passing, you know that there is nothing fundamentally fishy is going on, so why shouldn't you use it—in particular when it is something we will always need in one form or another: efficient hash tables for storing data.

An ETS table basically works like a simplistic database server: it is isolated from everything else, and holds information that is used by many. The difference compared to arrays in Java or C or similar is that the clients are quite aware that they are talking to a separate animal with a life of its own, and that what they read right now might not be what they read from the same index later. But at the same time, they can be assured that the data that they *have* read will not be subject to mysterious changes. If you look up an entry in the table, you get the currently stored tuple. Even if someone immediately afterwards updates that position in the table with a new tuple, the data you got will not be affected in the least. By comparison, if you look up a stored object in a Java array, it may be possible for another thread to look up the same object moments later and modify it in some way that will affect you. In Erlang, we try to keep it obvious when we are referring to data that is subject to change over time, and when we are referring to plain immutable data.

2.14.2 – Basics of using ETS tables

ETS tables are created and manipulated via the standard library `ets` module. To create a new table, use the function `ets:new(Name, Options)`. The name must be given as an atom, and Options must be a list. Unless the `named_table` option is specified, the name isn't actually used for anything in particular (and you can reuse the same name in as many tables as you like), but it can still be a useful indicator when you are debugging a system and find a mysterious table lurking somewhere, so it is better to for instance use the name of the current module, rather than "table" or "foo", which will not be of much help.

The function `ets:new/2` returns a table identifier that you can use to perform operations on the table. For example, the following creates a table and stores a couple of tuples:

```
T = ets:new(mytable,[]),
ets:insert(T, {17, hello}),
```

```
ets:insert(T, {42, goodbye})
```

Now, another similarity with databases is that an ETS table only stores "rows", that is, tuples. If you want to store any other Erlang data, you need to wrap it in a tuple first. This is because one of the tuple fields is always used as the index in the table, and by default it's the first field. (This can be changed with an option when you create the table.) Thus, we can look up rows in our table by their first elements, like this:

```
ets:lookup(T, 17)
```

which returns `[{17, hello}]`. But hang on, why is it in a list? Well, a table doesn't need to be a *set* of rows (where every key is unique), which is the default; it can also be a *bag* (where several rows can have the same key, but there cannot be two completely identical rows), or even a *duplicate bag*, where there can be several identical rows as well. In those cases, a lookup might return more than one row as result. In any case, you always get an empty list if no matching row was found.

There is really quite a lot to learn about what you can do with ETS tables. There are many parameters that can be specified when you create them, and there are many powerful interface functions for searching, traversing, and more. We will meet them again later in chapter 6.

2.15 – Recursion: it's how we loop

You may have noted that apart from list comprehensions, there has been a notable lack of iterative constructs in this presentation of the language. This is because Erlang relies on recursive function calls for such things. Although it's not really difficult, only different, there are some details and techniques that you should know about, both to make your path easier if you are not used to this way of thinking, and to help you produce solid code that avoids the common pitfalls.

To get started with recursion, let's take something really simple, like adding up the numbers from 0 to N. This is easily expressed as follows: to sum the numbers from 0 to N, you take the sum from 0 to N-1, add the number N, and you're done. Unless N is already 0, in which case the sum is 0. Or, as an Erlang function (add it to `my_module.erl`):

```
sum(0) -> 0;
sum(N) -> sum(N-1) + N.
```

Couldn't be simpler, right? Never had to do recursion before? Don't worry; it's actually a very natural concept for a human. (Reasoning about nested for-loops with `break` and `continue` and whatnot—now *that* often requires superhuman attention to detail.) But to understand how you can write all kinds of iterative algorithms using recursion, we need to go through some basics. This is important stuff, so please bear with us.

2.15.1 – From iteration to recursion

All problems that can be expressed recursively can also be written as loops (if you do your own bookkeeping). Which approach you choose is really a question of how easy your programming language makes it to write your code depending on the choice, and how efficient the result is. Some languages, like old Basic dialects, Fortran-77, or machine language assembler, don't have any support at all for recursion as a programming technique. Many, like Pascal, C/C++, Java, etc., allow you to write recursive functions, but because of limitations and inefficiencies in the implementation, recursion still isn't quite as useful as it could be. This situation is probably what most programmers are used to. But Erlang is different: it *only* uses recursion to create "loops", and the implementation is quite efficient.

A PLAIN OLD LOOP

Sometimes, you start out with a loop-style piece of code (maybe just in your head), and you want to implement it in Erlang. Perhaps something like the following typical code in C or Java for computing the sum from 0 to n, where n is the input parameter:

```
int sum(int n) {

  int total = 0;
  while (n != 0) {
      total = total + n;
      n = n - 1;
  }
  return total;

}
```

The code above is a very "procedural" way of expressing the algorithm: mentally, we can go through the program step by step and see how it keeps changing state until it is finished.

But let's think about how we would *describe* that algorithm to someone else using plain text, as concisely as we can. We'd probably come up with something like this:

1. You have the number N already. Let Total be zero, initially.
2. If N is not zero yet:
    - Add N to Total
    - Decrement N by one
    - Repeat step 2
3. You're done; the result is in Total.

LOOPING IN A FUNCTIONAL WAY

Now consider this alternative way of stating point 2 above:

2. If N is not zero yet, repeat this same step with:
    - Total+N as the new value for Total

- N-1 as the new value for N.

Seen this way, point 2 is a *recursive function* with two parameter variables, N and Total. It doesn't use any other information. On each recursive call, we pass on what the values for the next "iteration" should be, and we can forget the previous values. This way of saying "and then we do the same thing again, but with different values" is really a natural way for humans to reason about iteration. (Kids don't usually have a problem with it at all; it's us grown-ups, perhaps damaged by years of procedural coding, who can find it mind-bending.) Let's see how this step would look if we wrote it down in Erlang, very much to the letter:

```
step_two(N, Total) when N =/= 0  ->  step_two(N-1, Total+N).
```

Pretty readable, don't you think? (Add this function to `my_module.erl`.) Note that we never say "`N = N-1`" or similar—that just doesn't work in Erlang: you can't demand that a number should be the same as itself minus one. Instead, we say "call `step_two` with N-1 for the new N, and Total+N for the new Total". But we're missing something, right? What do we do when we're done? Let's add another clause to this function (and give it a better name):

```
do_sum(N, Total) when N =/= 0 -> do_sum(N-1, Total+N);
do_sum(0, Total) -> Total.
```

This incorporates step 3 from our verbal description into the code. When the first clause no longer matches (when the guard test turns false), the second clause will be used instead, and all it has to do is return the result that's found in Total.

INITIALIZING THE LOOP

Now we just have to do something about step 1: giving an initial value of 0 to Total. This obviously means calling `do_sum(N, 0)` from somewhere. But where? Well, we really have a "step zero", which is the problem description in itself: "to compute the sum of the numbers from 0 to N". That would be `do_sum(N)`, right? So, to compute `do_sum(N)`, we merely have to compute `do_sum(N, 0)`, or:

```
do_sum(N) -> do_sum(N, 0).
```

Note what we're doing here: we are creating one function called `do_sum/1` (note the period that ends the definition above), that takes a single argument. It calls our other function `do_sum/2`, which takes two arguments. Recall that to Erlang, these are completely different functions. In a case like this, the one with fewer arguments acts as a "front end", while the other should not be directly accessed by our users; hence, we should only put `do_sum/1` in the export list of the module. (We named these functions `do_sum` so that we don't clash with the `sum` function from the start of this chapter. You should try typing in both and check that `sum/1` and `do_sum/1` give the same results for the same values of N.)

FINAL TOUCHES

Now let's summarize this two-part implementation and improve it a little:

```
do_sum(N) -> do_sum(N, 0).

do_sum(0, Total) -> Total;
do_sum(N, Total) -> do_sum(N-1, Total+N).
```

Can you see what we did there? Rather that following the literal translation from our text above that said "N is not zero" in the recursive case, we simply changed the order of the clauses so that the *base case* (the clause that doesn't do any recursion) is tested first, each time around. For this particular algorithm, that made the code even simpler: in the first clause, try to match N against 0. If it doesn't match, well, then N isn't zero, so we can use the second case without any guard test.

That was a long section, but we still have some post-mortem studies to do, to understand a couple of important points and give names to the techniques we just used. We'll start by discussing the two kinds of recursion used in `sum` and `do_sum` above.

2.15.2 – Understanding tail recursion

Recursive calls can be divided into two categories: *tail recursive* and *non*-tail recursive (or *body recursive* as they are sometimes called). The function `sum` at the start of the chapter was an example of a non-tail recursive function (it contained a non-tail recursive call). In many other programming languages, this is the only kind of recursion you ever think about, because in other cases you typically use some kind of loop construct instead.

However, the function `do_sum/2` above, that we got from reasoning about loops, was a *tail recursive function*. All its recursive calls are so-called "*tail calls*". The thing about tail calls is that they can easily be spotted, and in particular, the compiler can always tell just by looking at the code whether a call is a tail call or not, so it can treat them specially.

What is this difference, then? Well, it's simply that a tail call is one where there is nothing left for the function to do when that call is done (except returning). Compare the bodies of these two function clauses of `sum` and `do_sum`, respectively:

```
sum(N) -> sum(N-1) + N.

do_sum(N, Total) -> do_sum(N-1, Total+N).
```

See how in `sum`, once the call to `sum(N-1)` is done, there is still some work left to do before it can return, namely, adding N. On the other hand, in `do_sum`, when the call to `do_sum(N-1, Total+N)` is done, no more work is needed—the value of that recursive call is the value that should be returned. *Whenever that is the case, the call is a tail call, or "last call".* In fact, it doesn't matter if the call is recursive (back to the same function again) or not—that's really just a special case, but it's the most important one. Can you spot the last call in the body of `sum` above? (That's right, it's the call to '+'.)

YOU CAN RELY ON TAIL CALL OPTIMIZATION

You are probably aware that behind the scenes, each process uses a *stack* to keep track of what it needs to go back and do later, while it is running the program (such as "remember to go back to this spot and add N afterwards"). The stack is a last-in-first-out data structure, like a heap of notes stacked on top of each other, and of course, if you just keep adding more and more things to remember, you will run out of memory. That's not a good thing if you want your server to run forever, so how can Erlang use only recursive calls for loops? Doesn't that add more and more stuff to the stack on each call? The answer is no, because Erlang guarantees *tail call optimization*.

What tail call optimization means is that when the compiler sees that a call is a tail call (the last thing that needs to be done before returning), it can generate code to throw away the information about the current call from the stack *before* the tail call is performed. Basically, the current call has no more real work to do, so it says to the function that it is about to tail-call: "Hey! When you're finished, hand over your result directly to my parent. I'm going to retire now." Hence, *tail calls don't make the stack grow*. (As a special case, if the tail call is a recursive call back to the *same* function, it can reuse much of the info on top of the stack, rather than throwing away the note just to recreate it.) Essentially, a tail call becomes "clean up if needed, and then jump". Because of this, tail recursive functions can run forever without using up the stack, *and* they can be just as efficient as a "while"-loop.

2.15.3 – Accumulator parameters

If you compare the behavior of `sum` and `do_sum` above, for the same number N, `sum` is going to do half of the work counting down to zero and making notes on the stack about what numbers to add later, and the other half going back through the notes adding up the numbers until the stack is empty. `do_sum`, on the other hand, will only use a single note on the stack, but keeps replacing it with newer information until it sees that N is zero, and then it can simply throw away that note as well and return the value Total.

In this example, Total is an example of an *accumulator parameter*. Its purpose is to "accumulate" information in a variable (as opposed to keeping information on the stack and returning to it later). When you write a tail-recursive version of a function, you usually need at least one such extra parameter, and sometimes more. They must be initialized at the start of the loop, so you will need one function as front end and one as the main loop. At the end, they will either be part of the final return value, or be thrown away if they only hold temporary information during the loop.

2.15.4 – Some words on efficiency

A tail recursive solution is often more efficient than the corresponding non-tail recursive solution, but not always; it depends a lot on what the algorithm does. Whereas the non-tail recursive function can be "sloppy" and leave it to the system to handle the stack and remember everything necessary to come back and finish up, the tail recursive version needs to be "pedantic" and keep everything it needs to complete the job in its accumulator

variables, often in the form of data structures like lists. If the non-tail recursive function is a drunkard who drops papers behind him so he can find his way back home, the tail recursive function is a traveler who pushes everything he owns in front of him on his cart. If everything you need for the final result is a number, as in our `sum`/`do_sum` example, the traveler wins big, because the load is light and he can move quickly. However, if the result requires tracking essentially the same information that the drunkard gets "for free", then the traveler has to do complicated data management and may turn out to be a bit slower.

In general, some problems will be more straightforward to solve using a non-tail recursive implementation, while some problems are more obvious to solve in a tail recursive fashion. It can be a nice intellectual exercise to try both variants, but for production code our advice is that if you have a choice between writing a tail recursive or a non-tail recursive implementation of a function, then pick the approach that will be more readable and maintainable and that you feel sure that you can implement correctly. Once that is working, leave it and go do something else. Don't spend time on premature optimization, in particular at the cost of readability.

Of course, in many cases the choice is obvious: a function that must loop "forever" simply has to be tail recursive. We say that it runs "in constant space", that is, it does not use more memory as time passes, even if it never returns.

2.15.5 – Tips for writing recursive functions

When you're new to programming with recursion, it can often feel like your mind just goes blank when you try to see how to solve a problem; like you just don't know where to start. There are a couple of methods that can help you get going.

To demonstrate, we'll use a concrete problem that you will often need to solve in one form or another: to go through the elements of a data structure. We'll look at lists here, but the same thinking applies to all recursive data structures, such as trees of tuples. Our task here will be to *reverse a list*, or rather, to create a new, reversed version of a given list (which can be of any length). For this, we will obviously have to visit all the elements of the original list, because they all need to be in the result.

LOOK AT EXAMPLES

The first thing you can do, if you're unsure about where to start, is to write down a couple of simple examples of inputs and the desired results. For reversed lists, we might have these examples:

```
[]      -> []
[x]     -> [x]
[x,y]   -> [y,x]
[x,y,z] -> [z,y,x]
```

Trivial, indeed, but having something written down is often better for seeing recurring patterns than just mulling it over in your head, and it makes the problem more concrete. It

might also make you remember some special cases. If you are into test-driven development, you would write the examples as tests right away.

BASE CASES

The next thing you can do is to write down the base cases, and what should happen in those. (The base cases are simply those cases that won't require any recursive calls. Usually there is only one such case, but sometimes there are more.) For reversing lists, we might consider the first two of the above as base cases. Let's write a couple of clauses to get started with our new function (in `my_module`), that we will call `rev/1`:

```
rev([])  -> [];
rev([X]) -> [X].
```

This is far from complete, but at least we can try it out right away for simple cases like `rev([])`, `rev([17])`, `rev(["hello"])`, and `rev([foo])`. That is, it should not matter what type of elements we have in the list; it's the only order that matters.

But after this step, it gets more difficult: we must get the recursive cases right.

THE SHAPE OF THE DATA

We now have to look at the remaining cases, and how they are constructed. If we have a list that is not one of the base cases above, it must have at least a couple of elements, i.e., it has the shape [A, B, …]. Recall that a list is really made up of list cells: "`[…|…]`" (look back at Section 2.2.5 for reference if you need to). If we write out the individual cells, our list here actually has the following form:

```
[ A |  [B | …]  ]
```

In other words, it has a cell for each element. Suppose we write this down in Erlang as the first clause of our rev function (do this!):

```
rev([A |  [B | TheRest]  ]) -> not_yet_implemented;
```

We might get some warnings about unused variables A, B, and TheRest, and the body of the clause doesn't do anything useful except return an atom saying this isn't implemented yet, but at least we can check that our rev function now seems to "work" for lists with 2 or more elements.

Now, we just need to figure out what to do in this case. It's going to be something with a recursive call to `rev`, we know that much.

IMAGINE YOU HAVE A WORKING FUNCTION ALREADY

If you can't see a solution from the examples and the structure of the data (this gets much easier after a little practice), or you can almost see it but can't quite get the details right, then a useful trick is to say to yourself "I have a working version of this function already,

somewhere, and I'm simply writing a new version that does the same thing (but better)". While you are working on your new function, you're allowed to use the old one for experimenting.

Suppose we try to think like this: we have a function `old_rev/1` that we can use. Great! So to replace the `not_yet_implemented` above with something useful, what could we do? We have the variables A, B, and TheRest there, and we want to end up with the same list only backwards. So if we could reverse TheRest (using `old_rev`), and then put B and A at the end (recall that ++ appends two lists), we should get the correct result, right? Like this:

```
rev([A |  [B | TheRest] ]) -> old_rev(TheRest) ++ [B, A];
```

That was easy enough. Now it actually looks like our function should be computing the right result for all lists, regardless of length. But if it's fully working, that means it's just as good as `old_rev`, so let's just use our own `rev` instead! The entire function then becomes:

```
rev([A |  [B | TheRest] ]) -> rev(TheRest) ++ [B, A];
rev([])  -> [];
rev([X]) -> [X].
```

And it really works on its own, as you can see if you call `my_module:rev([1,2,3,4])`. Nice! Next, let's think a bit about how we can know that it will work correctly on all lists.

PROVING TERMINATION

It might be easy to see at a glance that your function must sooner or later terminate, that is, that it will not loop forever, regardless of input. But for a more complicated function, it can be harder to see that it will *always* eventually reach a point where it will return a result.

The main line of reasoning that you can follow to convince yourself that your function will terminate, is that of *monotonically decreasing arguments*. This means that, assuming that your base cases are the smallest possible inputs that your function will accept and the recursive cases handle all inputs that are larger than that, then, if each recursive case always passes on a smaller argument to the recursive call than what it got as input, you know that the arguments must therefore eventually end up as small as the base cases, so the function *must* terminate. The thing that should make you suspicious and think twice is if a recursive call could pass on arguments that are as large as the inputs, or larger. (If a function is recursive over several arguments, which happens, then in each step at least one of these should be getting smaller, and none of them getting larger.) Of course, arguments that are not part of the loop condition can be disregarded entirely, such as accumulator parameters.

In our `rev` example, no lists can be smaller than those in the base cases, and if we look at the recursive case, we see that when we call `rev(TheRest)` recursively, then TheRest will have fewer elements than the list we got as input, which started with A and B. Hence, we are working on successively smaller lists, so we know we cannot loop forever.

When we are recursing over numbers, as in our `sum` example from the start of this chapter, it can be easy to miss the fact that there is no smallest possible integer. If we look back at the definition of `sum`, we see that the recursive case is indeed always passing on a smaller number than its input, so it must eventually become zero or less. But if it was already smaller than zero to begin with, e.g., if you called `sum(-1)`? Then it will simply keep calling itself with -2, -3, -4, etc., until we run out of memory trying to represent a huge negative number. To prevent this from happening, we could make sure that no clause matches if the input is negative, by adding a guard "`when N > 0`" to the recursive case.

You might also need to reinterpret what "small" and "large" means in the context of your function. If you for instance recurse over a number N starting at 1 and ending at 100, you need to think of 100 as the "smallest" case, and of N+1 as "smaller" than N. The important thing is that on each recursive call, you keep moving towards the base cases.

MINIMIZE THE BASE CASES

Though it doesn't do any harm for the functionality to have unnecessarily many base cases, it can be confusing for the next person working on the code. If you have more than one, you should try to see if some of them can be easily eliminated. We started out with two: `[]` and `[X]`, because it seemed easiest. But if we look at what `[X]` actually means in terms of list cells, we see that it can be written as

```
[ X |  []  ]
```

and since `rev` already can handle the empty list, we see that we could handle rev([X]) by doing rev([]) ++ [X], even if it looks a little redundant. But that means that we don't need two separate cases for lists of 1 element and lists of 2 or more elements. We can join those two rules into a single recursive case, to give us a cleaner solution:

```
rev([X | TheRest]) -> rev(TheRest) ++ [X];
rev([]) -> [].
```

(Note that the order of these clauses doesn't matter: a list is either empty or it isn't, so only one clause can match. However, it's a bit useless to check for empty lists first, because if you recurse over a list of 100 elements, it will be nonempty 100 times and empty once.)

RECOGNIZING QUADRATIC BEHAVIOR

So, we have a working function that reverses a list. All fine? Not quite. If the input is a long list, this implementation will take much too long time. Why? Because we have run into the dreaded "quadratic time" behavior. Quadratic in the sense that if it takes T units of time (for whatever unit you like to measure it in) to run the function on some list, then it will take 4T units of time to run it on a twice as long list, 9T units for a list that's three times as long, etc. You might not notice it for shorter lists, but it can quickly get out of hand. Say that you have a function that gets a list of all the files in a directory, and does something with that list. It's working fine, but you've never tried it on directories with more than a hundred files, and that

took 1/10th of a second, which didn't seem like a problem. But if the algorithm is quadratic in time, and you one day use it on a directory containing ten thousand files (a hundred times larger than before), then that will take 100 x 100 = 10 000 times as long time, or a thousand seconds (over 15 minutes), rather than the 10 seconds it would have taken if the algorithm had been proportional, or "linear", in time. Your customers will not be happy.

Why is our implementation of `rev` quadratic in time? It's because for each recursive call (once for every element in the list) we also use the `++` operator, which in itself takes time in direct proportion to the length of the list on its left-hand side (if you recall from Section 2.2.5). Let's say `++` takes time T if the left list has length 1. In our `rev`, the left-hand side of `++` is the list returned by the recursive call, which has the same length as the input. This means that if we run `rev` on a list of length 100, the first call will take time 100T, the second 99T, the third 98T, etc., until we're down to 1. (Each call will also take a little time to pick out X and TheRest and perform the recursive call, but that's so small in comparison to 100T that we can ignore it here.)

What does 100T + 99T + 98T + … + 2T + 1T amount to? It's like the area of a right triangle whose legs have length 100: the area is 100 x 100 / 2, or half of a square of side 100. In general, then, the time for `rev` on a list of length N is proportional to N x N / 2. Since we're mostly interested in how it grows when N gets larger, we say that it is *quadratic*, because it grows just like N x N. The "divided by two" factor simply pales in comparison to the main behavior of the function.

Figure: sum of times for N iterations of a quadratic function = area of triangle

Note that this sort of thing can happen in any language, using any kind of loop or iteration over some kind of collection; it is not because of recursion, it is because we have to do something N times, and for *each* of those times we do something else that takes time proportional to N, so that the times add up in a triangle-like way. The only consolation is that it could be worse: if your algorithm takes *cubic* time, you'll be looking at a 28 hour wait in our example above. If it takes *exponential* time, waiting is probably not an option.

AVOIDING QUADRATIC TIME

What can we do with our `rev` function to avoid the quadratic time behavior? We can't use `++` with the "varying" list on the left hand side, at any rate. What if we tried a tail-recursive approach? We go through the list, but at each step we push all the stuff we need "in front of us" in the arguments so that when we reach the end of the list, we'll be done, and there is nothing on the stack to go back to, just like we did with `do_sum` in Section 2.15.1. How would that look for recursion over lists? We can use the same basic division into base case and recursive case as `rev`, but we'll call the new function `tailrev`, and we'll need an accumulator parameter that will contain the final result when we reach the end, like this:

```
tailrev([X | TheRest], Acc) -> not_yet_implemented;
tailrev([], Acc) -> Acc.
```

Now for the not_yet_implemented part: we want it to do a tail call, so it should have the shape "`tailrev(TheRest, …)`", and the second argument should be something to do with Acc and the element X. We know that a "cons" operation (adding an element to the left of a list) is a very cheap and simple operation to do, and we know we want Acc to become the final reversed list. Suppose we do `[X | Acc]`, adding the element to the left of Acc, basically "writing the new list from right to left" as we traverse the old list:

```
tailrev([X | TheRest], Acc) -> tailrev(TheRest, [X | Acc]);
tailrev([], Acc) -> Acc.
```

For each element we see, as long as the list is not yet empty, we tack the element on to the left side of Acc. But what should Acc be initialized to? The easiest way to see this is by looking at what should happen with the simplest case that is not already a base case. Suppose we call `tailrev([foo], Acc)`, to reverse a list of just one element. This would match the first clause above, binding X to `foo` and TheRest to `[]`, so the body of the clause would then become "`tailrev([], [foo | Acc])`". In the next step, the base case `tailrev([], Acc)` will match, and it should return the final Acc. This means that the original Acc must be an empty list, so that `[foo | Acc] = [foo]`. Our complete implementation will then be the following, with `tailrev/1` as the main entry point:

```
tailrev(List) -> tailrev(List, []).

tailrev([X | TheRest], Acc) -> tailrev(TheRest, [X | Acc]);
tailrev([], Acc) -> Acc.
```

Why is this implementation linear in time (in proportion to the length of the list) rather than quadratic? It's because for each element of the list, we only perform operations that have fixed cost (such as adding to the left of the list); therefore, if the list has L elements, the total time will be L times C, for some small constant C, and the algorithm will never blow up in our face like the quadratic version did when the input got big.

LOOK OUT FOR LENGTH

A common beginners' mistake made by many who are used to programming in languages like Java, where it is a fixed-cost operation to get the length of a list, is to use the built-in function `length` in guards, like this:

Don't do this: length(List) traverses the whole list

```
loop(List) when length(List) > 0 ->
    do_something;
loop(EmptyList) ->
    done.
```

A function like that will use quadratic time in proportion to the length of the list, because it has to *traverse the list from start to end* to count the number of elements, each time. This will add up like a triangle, just like the time for ++ in the previous section. But if all you wanted to know was if the list is nonempty, you could easily do it with pattern matching:

Do this: pattern match to check for nonempty lists

```
loop([SomeElement | RestOfList]) ->
    do_something;
loop([]) ->
    done.
```

A match like that takes a small, fixed amount of time. You can even use matching to check for lists that are at least of a certain length, like this:

Pattern matching to check for lists of various lengths

```
loop([A, B, C | TheRest]) -> three_or_more;
loop([A, B | TheRest]) -> two_or_more;
loop([A | TheRest]) -> one_or_more;
loop([]) -> none.
```

Just note that you need to check for the longer lists first, because if you check for "two or more" before the others, it will match all lists of length three as well, and so on. We say that we check for the *most specific* patterns first.

2.16 – Erlang Programming Resources

To learn more about the Erlang language, get a better grasp of functional and concurrent programming techniques, and learn more about the available libraries and tools, the following are the most important resources.

2.16.1 – Books

There are, as of this writing, two modern books about Erlang the programming language. The first, which kicked off a whole new wave of interest for Erlang all over the world, is *Programming Erlang – Software for a Concurrent World* by Joe Armstrong (Pragmatic Bookshelf, 2007). It is a very good introduction to the language and to concurrent programming in general, and gives a number of interesting examples of the kinds of programs one can write easily in Erlang.

The second, more recent addition, is *Erlang Programming* by Cesarini and Thompson (O'Reilly, 2009), which dives deeper into the language details, conventions and techniques for functional programming and concurrent programming, and the libraries and tools that are part of the Erlang ecosystem.

Finally and mainly of historical interest, you might be able to find a copy of *Concurrent Programming in Erlang*, 2nd ed., by Armstrong, Virding, Wikström and Williams (Prentice Hall, 1996), but it is outdated with regard to language features, libraries and tools.

2.16.2 – Online material

The main website for Erlang is `www.erlang.org`, where you can download the latest release of Open Source Erlang, read online documentation, find official news from the OTP development team at Ericsson, subscribe to mailing lists, etc.

There is also a community website at `www.trapexit.org`, with mailing list archives, a wiki for tutorials, articles, cookbooks, links and more. The site `www.planeterlang.org` summarizes Erlang-related feeds from various sources, to help you keep up to date.

The main mailing list for Erlang is `erlang-questions@erlang.org`, where you can generally find answers to the most complicated questions, from experienced and professional users like yourself. The archives go back more than ten years and are a treasure trove of information.

Finally, searching for "erlang" at `stackoverflow.com` is a good complement to the erlang-questions mailing list for finding answers to various questions.

2.17 – Summary

We have covered quite a lot in this chapter, from the Erlang shell, via data types, modules and functions, pattern matching, guards, funs and exceptions, to list comprehensions, the bit syntax, the preprocessor, ETS tables, recursion and more. To be sure, there is much more to learn about the finer details of writing Erlang programs, but what we have covered provides a solid footing for you to move forward with. If you skimmed this part even though you're not too familiar with Erlang, don't worry, you can come back here for reference when and if you need it.

In the following chapters, we will explain new concepts as they are brought up, assuming that you have some previous knowledge to stand on. Now, we are going to dive into OTP, where we will remain for the rest of this book and where you as an Erlang programmer will hopefully choose to stay for a lot longer.

3

Writing a TCP Based RPC Service

What! No hello world?

That's right, no hello world. If you are reading this book you are an intermediate programmer and have probably already done an Erlang hello world. Chapter two provided a nice review of the basics of the language and now it is time to do something concrete. In the spirit of getting down and dirty with real world Erlang we say, NO, to hello world! Instead we will do something immediately useable. We are going to build a TCP enabled RPC server! If you don't know what RPC is, we will explain. RPC stands for remote procedure call. An RPC server allows you to call procedures i.e. functions remotely. The TCP enabled RPC server will allow a person connected to a running Erlang node to run Erlang commands, and inspect the results with no more than a simple TCP utility like good old Telnet. The TCP RPC server will be a nice first step toward making our software accessible for post-production diagnostics. This application as written would constitute a security hole if included in a running production server, but it would not be difficult to limit the modules or functions that this utility could access thus closing the hole. We will not do that in this chapter however. The creation of this basic service is going to be used as a vehicle for explaining the first, most fundamental, most powerful, and most frequently used of the OTP behaviours (yes that is spelled correctly); the Generic Server. Behaviours and the code they plug into within OTP greatly enhance the overall stability, readability, and functionality of software that incorporates them. We are going to cover implementing our first behaviour in this chapter and we will also be learning about basic TCP socket usage with the gen_tcp module(not a behaviour). This book is for intermediate level Erlang programmers and so we are going to start in the thick of it. It will be necessary to pay strict attention, but we promise it will be gentle enough to fully understand. When you are done you have taken a great leap forward in terms of being able to create reliable software. By the end of this chapter we will have taken what you

learned from Chapter 2 and used it to create a working Erlang program that will eventually be a piece of a production quality service because once we complete our program here we will move into chapter 4 where we will hook this program further into the OTP framework completing an Erlang application that is capable of being composed with other such applications to form a running Erlang service (also known as a Release). Next we get a bit more explicit about what it is we will be building in this chapter.

3.1 – What we are we creating?

Our RPC server will allow us to listen on a TCP socket and accept a single connection from an outside TCP client. Once connected it will allow a client to run functions via a simple ASCII text over TCP. Figure 3.1 illustrates the design and function of the RPC server.

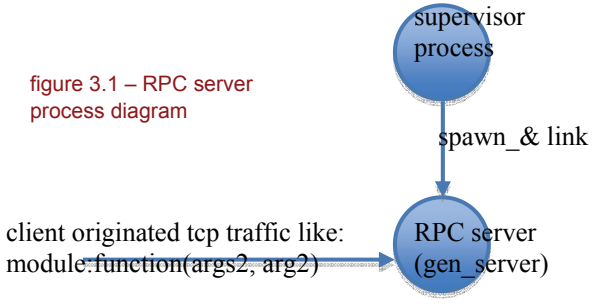


figure 3.1 – RPC server process diagram

What we see above in figure 3.1 are two processes. One is the supervisor, more on that later, for now just know that it spawns another process that will actually do the work of being an RPC server. This second process will create a TCP listen socket and listen for a connection. When it receives a connection it will take in ASCII text that looks just like erlang function calls on the erlang shell and it will executed those calls and return the result back over the TCP stream. This functionality is useful for any number of things including remote administration and diagnostics in a pinch. Again, the RPC server will understand a basic protocol over that TCP stream which looks just like standard Erlang function calls. The next example shows first the generic format for this protocol expressed in an Erlangy way followed by an actual example.

```
Module:Function(Args).
```

for example

```
lists:flatten("Hello", "Dolly").
```

The RPC server will interpret the text, in doing so it will convert the ASCII text forming the module, the function, and the arguments transforming them into valid Erlang terms, then

executing Module:Function(Args) and finally returning the results as ASCII text back over the TCP stream which involves transforming Erlang terms back into an ASCII string. Accomplishing this will require knowledge of a number of fundamental Erlang/OTP concepts like modules, functions, messaging, processes, and of course behaviours, all of which we will get into in the next couple of sections.

3.1.1 – The fundamentals are critical

You should already have a basic grasp of modules, functions, messaging and processes these concepts were addressed in Chapters 1 and 2. We will spend some more time on each of those concepts in this chapter as well as introduce the newer concept of behaviours. In Erlang code is housed in modules. This means that processes and functions live within modules. Processes are spawned around functions. If a process is spawned aound the io:format/2 function then it will execute that code and go away when it is done. This is the nature of all processes. Processes communicate with each other by sending messages. Figure 3.2 illustrates the relationships between modules, functions, messages and processes.

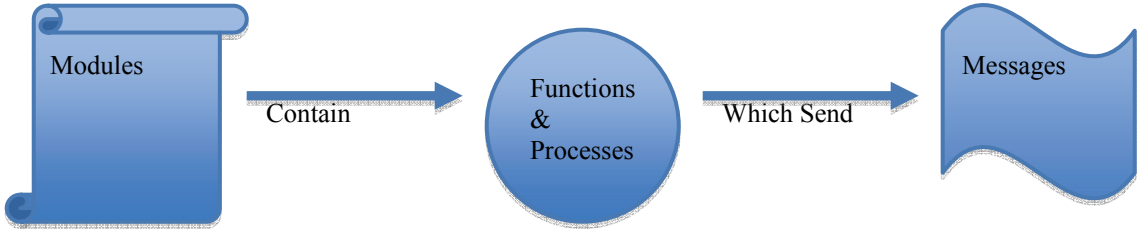


figure 3.2 – modules, functions processes and messages relationship diagram

Taking just a second here to review these concepts in some depth:

MODULES

Modules are containers for code. The guard access to functions by either making them private or exporting them for public use. Modules are one per file and the module name and the file name must be the same. A module named "test" must reside in a file called "test.erl".

PROCESSES

Processes are the fundamental units of concurrency in Erlang. They communicate with eachother through messages which are the only way they can influence one another. Processes are spawned around functions as in spawn(io, format, ["~p~n", ["hi there"]]).

When the function a process is spawned around exists so does the process. A process spawned around io:format/2 as illustrated a second ago will be very short lived. A process spawned around a function like spawn(fun() -> loop() -> loop() end) will last forever. It is generally a good idea to spawn no more than a single type of process per module. Processes are also the fundamental container for state in Erlang. Data that needs to be kept track of and modified over time can be placed inside a process.

MESSAGES

Messages are the mechanism by which processes interact with one another. Messages are sent from one process to another and are copies of data from one processes state to that of another; always. Messages do not pass by reference they are for all practical purposes always a copy. Messages are received in the process mailbox and retrieved by entering a receive block.

FUNCTIONS

Functions do the work. They are the sequential part of Erlang. They are executed within processes which represent the concurrent part. Functions are contained in modules where they can be either exported or private.

With that refresher done we move now into the concept of behaviours.

3.2.1 – Behaviour basics

The module we create in this chapter will not house just any old type of process. We will be creating the most common and useful type of Erlang/OTP process, a "Generic Server" process which is behaviour. Behaviours have quite a number of advantages:

1. Developer writes less code, sometimes much less.
2. Code is more solid and reliable because well tested generic behaviour code is leveraged.
3. Code fits into larger patterns. Generic servers provide out of the box hooks to fit into that larger OTP framework which provides many powerful features for free.
4. Code is easier to read because it follows a well known pattern.

The word behaviour in Erlang is a somewhat overloaded term. It describes a set of functionality that is best broken down and described as three constituent parts:

1. Behaviour interface
2. Behaviour implementation
3. Behaviour container

The behaviour interface is a specific set of functions and return structures. The generic server behaviour contains 6 functions; init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3.

The second part of a behaviour is the behaviour implementation. This is what the user of a behaviour defines. This is where the domain specific code lies. A behaviour implementation is a module that exports the functions defined by the behaviour interface. A behaviour implementation module contains a –behaviour attribute which indicates which behaviour is being implemented. This instructs the compiler to check the module over to ensure that the full interface has been implemented i.e. exported. Code listing 3.1 shows the parts of the header and the interface functions that must be implemented for a valid generic server.

Code listing 3.1 – a raw gen_server behaviour implementation module

```
-behaviour(gen_server).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

If any of the above functions are missing the behaviour implementation is incomplete and not fully conformant with the behaviour interface. The compiler will issue warnings in this case. We will get into what each of these functions do in more detail through actually implementing them as part of our RPC server. For now understand that it is within these functions that the RPC server, the application specific code, will reside.

The third and final constituent part of a behaviour is the behaviour container. The behaviour container is a single process that is spawned around behaviour specific code that ships with Erlang/OTP. For the generic server this code sits within the gen_server module which resides in the standard library known as stdlib. The behaviour container is what provides much of the benefit that comes with using a behaviour. Behaviour containers handle much of what is challenging about writing canonical, concurrent, fault tolerant OTP code. The container handles things like synchronous messaging, process initialization and process termination and cleanup. The container also provides hooks into larger OTP patterns and structures like code change and supervision.

A behaviour interface is used as the contract for the behaviour implementation to leverage the power of the behaviour container. Together these three constituent parts form a full behaviour which allows a developer to:

1. Writes less code, sometimes much less.
2. Harness well tested behaviour code for more powerful functionality.
3. Fit into larger OTP patterns and structures for free.
4. Write more readable code that follows a well known behaviour interface.

With a basic understanding of the basics we can now move on into the actual implementation of the RPC server which will utilize all of what we have just covered. What we will be doing from here on out is all related to the business of implementing the TCP RPC server. This exercise is going to cover a lot; at one level it is about how to use behaviours. We will be coding up a behaviour implementation, which of course conforms to a behaviour interface, and we will be demonstrating some of how the gen_server behaviour container is leveraged to provide us the functionality that we are looking for. At another level what we will be going over is even more fundamental. We will be using processes, modules, functions, and messaging in the context of OTP.

3.2 – Implementing the TCP RPC server

As an intermediate level Erlang programmer you have some familiarity modules, processes, functions, and messaging. The context in which you have familiarity with them is most likely a more informal, pure Erlang, context. We will be revisiting them in this chapter in their OTP context. If you have not read anything about Erlang and this is your first book, then you are probably a very experienced programmer or you would not have made it this far. You don't need any prior knowledge of modules, processes, functions, or messaging to grasp what we will cover in this chapter, it only makes it easier. In fact, it is our contention, that writing non-OTP erlang code is really an advanced topic, and something that should be done only when you really really have to. So perhaps not ever having done it the non-OTP way is a blessing because you will pick up the right habits straight away with regard to

things OTP and even with respect to the rigorous approach we take to module structure and layout, inline documentation and commenting.

Since it is a modules that will contain our behaviour implementation we will start with a little bit about module creation and layout.

3.2.1– Canonical Behaviour Implementation Module Layout

One of the nice things about behaviours is that they give you a certain amount of consistency. When looking at a behaviour implemenation module there are aspects of the module that you see common to all behaviour implementation such as the behaviour interface functions or the start or start_link functions. To make the files even more canonical, and recognizable, you can adopt the canonical behaviour implementation module layout that we will elaborate upon here.

The fundamental layout consists of four parts. These parts in the order that they appear in the file are detailed in table 3.1.

Section	**Description**	**Functions Exported**	**Edoc**
Header	Module attributes and boilerplate	N/A	Yes, file level
API	Programmer interface, the way the world interacts with the module	Yes	Yes, function level
Implementation of Behaviour Interface functions	The implementation of the functions specified by the behaviour interface	Yes	Optional
Internal functions	Helper functions for the API and behaviour interface functions	No	Optional

table 3.1

To define our behaviour we will take a look at each of these sections in turn, with the exception of internal functions which can be sprinkled throughout both the API and the behaviour interface functions section; although they are more heavily present typically in the behaviour interface function section. We will cover all the nuances of implementing each

section here from how to document them with edoc, idioms typically used, naming conventions and so forth. So, first up is the module header.

3.2.2 - Module Header

Before we can even create the header we need to create a file to hold it. So, since we are going to be building the tcp rpc server let's create a file named tr_server.erl where we will place all this code.

MODULE NAMING CONVENTION AND ERLANGS FLAT NAMESPACE

Erlang has a flat name space, i.e. it does not support packages officially. Unoficially it does but they are not well integrated and should not be used. Having a flat namespace means module names can collide. It is easy to get two modules in a single vm named the same thing if modules get named things like "server". To avoid these clashes the standard practice is to prefix modules. Since we are calling our project here TCP RPC we take the first two letters of those words and for T.R. hence the tr_server.

Open up the file and the first thing that we will place inside of it is the file level header comment block.

```
%%%-------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008 Erlware
%%% @doc This module defines a server process that listens for incoming
%%%      TCP connections and allows the user to execute commands via
%%%      that TCP stream
%%% @end
%%%-------------------------------------------------------------------
```

The first thing to notice is that each comment line, which begins with a % sign in Erlang actually uses three % signs. This is a convention, file level comments, comments that apply to the whole file not just something in the file usually use three percent signs. This may be the first time you have seen an file level header containing edoc. The Edoc documentation generation library comes with the standard Erlang/OTP distro. This is not a book on edoc so we will not get too deep into it's use, it has good documentation that you can find on the web. We should spend a little time on it though because it is the defacto standard for in-code documentation and it is something that we use very heavily in our own code. All edoc tags begin with an @ sign. Table 3.2 describes the tags in this header.

Tag	Description
@author	Author information and email address

@copyright	Date and attribution
@doc	Description of what is contained in the file. can contain valid html and other special markup. See edocs' own docs for details.
@end	Marks the end of the documentation, used here so that the %%%----… line is not included in the generated documentation.

table 3.2

The first non-comment addition to our file is the –module attribute. The name supplied therein must be the same as the file name; in our case tr_server like so:

```
-module(tr_server).
```

All the attributes that get placed into our header terminate with a period. Once we have the module attribute in the file the next order of business is the addition of the behaviour attribute. This indicates to the compiler which behaviour this module is an implementation for and allows it to warn if the implementation is not conformant i.e. behaviour interface functions are not exported. Our behaviour attribute is as follows:

```
-behaviour(gen_server).
```

Next up; export attributes. We will typically have two. The first one is for our API section and the second is for the behaviour interface functions which are also exported. Since we have not yet written or designed the API a place holder is enough for that but we do know what our behaviour inteface functions are and we can export them now.

```
%% API
-export([]).

%% Gen Server Callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
```

The first thing to notice is the phrasing of the second comment "Gen Server Calbacks". The word callbacks is often used to describe the behaviour interface functions. This is because the behaviour implementation module name is passed as an argument to the behaviour container upon startup. The container calls back into it through these interface functions to accomplish work. We will go into more detail about the actual user of each of the interface functions later on in the chapter. After the exports come a number of optional module

attributes. They are highlighted as code annotations below in code listing 3.1 which contains the full header for the tr_server.

Code Listing 3.1 – the full tr_server.erl header

```
%%%----------------------------------------------------------------------
%%% @author Martin & Eric <erlware-dev@googlegroups.com>
%%%  [http://www.erlware.org]
%%% @copyright 2008 Erlware
%%% @doc This module defines a server process that listens for incoming
%%%      TCP connections and allows the user to execute functions via
%%%      that TCP stream
%%% @end
%%%----------------------------------------------------------------------
-module(tr_server).

-behaviour(gen_server).

%% API
-export([
         start_link/1,
         start_link/0,
         get_count/0,
         stop/0
        ]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

 -define(SERVER, ?MODULE).      #1
-define(DEFAULT_PORT, 1055).

 -record(state, {port, lsock, request_count = 0}).      #2
```

Below the –export attributes we have two macro definitions. Macros are defined with the attribute –define as seen at code annotation #1. The convention is to make all macros capital letters. To use a macro just reference it within the code with a ? infront of the name in the –define attributes. So, if we add the macro ?DEFAULT_PORT to the code somewhere the compiler will automatically transform that to 1055. The final attribute in the header for tr_server is the record attribute seen at code annotation #2. This one deserves a little extra time devoted to fully explaining it.

If you are familiar with C or C++ programming you undoubtedly know of "structs". Structures complex datatypes that can be referenced by name. A record is just that. A record has a name and contains members that can be referenced by name. In our code we have defined:

```
-record(state, {port, lsock, request_count = 0}).
```

This is a record definition for a record named "state" that contains 3 named fields; "port", "lsock", and "request_count". The field request_count has a default value associated with it such that if this record is defined and that fields not explicitly set it will automatically be set to the default values defined here. All other fields if unset will be set to the atom `undefined`. The state record can be created in the following manner:

```
State = #state{port = 80}, %% Values aside from port's are set to defaults.
```

Now accessing these values is easy. If I want the value of request count I would use this expression:

```
State#state.request_count,
```

The value returned would be 0 on this case. To change the value of request count we could write the following:

```
NewState = State#state.request_count = 1,
```

The record syntax has proven confusing to many over the years. Please don't be upset if you don't get it right away. That said, don't shy away from it, we will be using it more throughout this text and it's usage will be come ever more clear. For now the basics above should be a nice starting point to move into more advanced syntax and reap the benefits of writing code with records. The next section to cover in creating our tr_server behaviour implementation module is the API.

3.2.2– The API section

The API is the face our module presents to the world. All the functionality that we desire to present for users of our behavour implementation will be presented in the (A)pplication (P)rogrammer (I)nterface. The main two uses of the API are

1. process startup functions
2. sending messages to the process spawned by the startup functions

To accomplish these tasks functionality that comes with the behaviour container is typically used. There are 3 primary pieces of gen server container functionality that pertain to the API or the tr_server. They are listed out in table 3.3.

Function Name	Associated behaviour interface (callback) function	Description
gen_server:start_link/4	Module:init/1	Startup a gen_server process
gen_server:call/2	Module:handle_call/3	Send a synchronous message to a gen_server process
gen_server:cast/2	Module:handle_cast/2	Send an asynchronous message to a gen_server process

table 3.3

GEN_SERVER PROCESS?

What do we mean when we say gen_server process as is stated three time in table 3.3? We are referring to the notion of *process type*. Processes of the same type are said to run, or make concurrent, the exact same code. This means two processes of the same type will understand and react to the same messages in the same way. The only difference between two processes of the same type is typically the data or state they contain. Processes of the same type usually have the same *spawn signature.* A spawn signature refers generally to the function around which the spawn call was invoked.

The API for the tr_server is listed in code listing 3.2.

Code LIsting 3.2 – tr_server API

```
%%%==========================================================================
 %%% API              #1
%%%==========================================================================

%%--------------------------------------------------------------------------
%% @doc
%% Starts the server
%%
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
%% @end
%%--------------------------------------------------------------------------
start_link(Port) ->
   gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], []).    #2

%% @spec start_link() -> {ok, Pid}
%% @equiv start_link(Port::integer())
start_link() ->
    start_link(?DEFAULT_PORT).
```

```
%%--------------------------------------------------------------------
%% @doc fetch the number of requests made to this server.
%% @spec get_count() -> {ok, Count}
%% where
%%  Count = integer()
%% @end
%%--------------------------------------------------------------------
get_count() ->
    gen_server:call(?SERVER, get_count).     #3

%%--------------------------------------------------------------------
%% @doc stops the server.
%% @spec stop() -> ok
%% @end
%%--------------------------------------------------------------------
stop() ->
    gen_server:cast(?SERVER, stop).     #4
```

The API section starts after a file level comment indicating the start of the section at code annotation number 1. What this API tells us is that tr_server can do three things:

1. It can be started with gen_server:start_link/4 at code annotation number 2
2. It can be queried for the number of requests it has processed with gen_sever:call/2 at code annotation number 3.
3. It can be stopped using gen_server:cast/2 at code annotation number 3.

This is fairly straightforward APIs can be much longer but then again sometimes even shorter than this. The important concept to understand is that the API is for the most part what defines how any given module can interact with the module exporting the API. A major function of a good Erlang/OTP API is to wrap the messaging protocol used to communicate with actual processes of the process type contained within the same module.

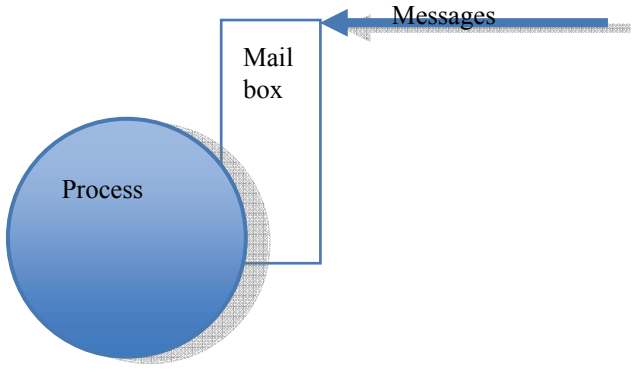
ONE PROCESS TYPE PER MODULE

One module may be used to spawn many processes but should only contain one process type. If multiple code segments are made concurrent from within a single module it becomes very hard to reason about the system as a whole when it contains multiple modules. The best practice then for clean Erlang/OTP systems is one process type per module.

We will see how the API wraps the messaging protocol that the process type we define in the tr_server module can understand as we dig into the API above. Before we get into messaging protocols and wrapping then in APIs we want to do a quick refresher on messaging.

QUICK MESSAGING REFRESH

Processes are the building blocks of any concurrent program. Processes can be communicated with via messages which copy data from one process to another. Messages are placed into a processes message queue or "mailbox". In the figure below you can see messages entering the process mailbox. The mailbox and the process itself are not synchronized. A process can continue to do work as its mailbox receives messages. A process is free to pull messages out of its mailbox when it is ready to do so.



The list of messages a process will handle is known as its protocol set or "alphabet". This alphabet ties directly in with a modules API because it is the API that defines and hides this alphabet from the rest of the world. With this concept of Erlang messaging fresh into your mind we are ready to move into the practice of implementing good APIs.

As stated earlier one function of the API in a module is to wrap the messages sent to a process with functions. So let us say that the process type we define within the tr_server module, we will start calling it process type tr_server from here on out, can understand the following messages:

1. get_count
2. stop

These two messages are simple, just single atoms, but even still their actual format should not be leaked out of the tr_server module they should instead be wrapped by functions with the same name. That is another pattern and hallmark of well written Erlang/OTP code. API functions that wrap messages are named the same thing as the message tag. To further illustrate this let's say we have a complex message of the form:

```
{calculate, {Number, Constant, {options, Options}}}
```

That is something we surely don't want to leak out of our module. Leaking this out would break good encapsulation rules and lead to fragile code. Instead we would wrap it in a function that has the same name as the message tag as such:

```
calculate(Number, Constant, Options) ->
```

This is basic encapsulation in Erlang/OTP. This way if you need to change the protocol a process understands you can do it without having to modify all the code that communicates with your process. It keeps things much more simple and clean. Now, with good style understood we can explain our API which embodies that style and practice. We are going to implement three API functions that will define for the world much of how they can interact with the tcp_rpc server.

API Function Name	Description
start_link/1	Starts the tcp_rpc server
get_count/0	Returns the number of transactions performed on the server
stop/0	Shuts down the server

table 3.1 – The tcp_rpc API

START_LINK(PORT) ->

This function is used to start our server process and where we indicate to the behaviour container what module will contain the implementation of the behaviour interface. The second argument to this function, the macro ?MODULE, tells the container that this very module contains the behaviour implementation. It takes a single argument, Port, which is the port that our server should listen for incoming TCP connections on. Our start_link/1 function is a wrapper around `gen_server:start_link` which is the function we use to start our generic server process.

```
gen_server:start_link({local, ?SERVER}, ?MODULE, [Port], []).
```

When the code above is executed as above it will block until the gen_server process is initialized by the init/1 behaviour interface (callback) function, more on that in section 3.2.3, and then return. At that point our RPC server will be up and running fully initialized and ready to accept messages. The `[Port]` code is how we pass our argument into our server. The tuple {local, ?SERVER} locally registers this process as the name the macro ?SERVER resolves to which is the module name. This makes this process type a singleton meaning only one process of type tr_server can exist at any one time and the registration also means

that we can message this process by name instead of having to hang on to its process ID. We make use of this in the get_count and stop functions that make up the rest of our API.

GET_COUNT() ->

```
get_count() ->
    gen_server:call(?SERVER, get_count).
```

get_count/0 sends a message to our server. The process ID or registered name to send the message to is the first argument in the gen_server:call/2 function and in this case it is the atom the macro ?SERVER resolves to. This is possible because our start_link function registered the process. The message that it sends is the atom `get_count`; it can be seen as the second argument to gen_server:call/2 above. This message is sent synchronously meaning that any call to get_count will block until the server sends a reply back to the caller. That reply will then be returned by the get_count function. All this is accomplished via `gen_server:call/2`. Another important thing to notice here is the fact that the message get_count has the name of the function that sends it get_count/0; this is not by accident as you know.

STOP() ->

The stop function uses `gen_server:cast/2` to send an asynchronous message, the atom `stop`, to the server.

```
gen_server:cast(?SERVER, stop).
```

This call returns immediately after the call to cast, it does not wait for any sort of return (this is the definition of asynchronous).

Before we move on to the next section on defining our behaviour interface callback functions I want to talk a little bit about the new edoc tag we used in the documentation that exists for each function. This can been seen in code listing 3.2. It is highly recommended that documentation exist for each and every API function as it does in our API. The additional tag that was used to document functions is the @spec tag. The @doc and @end tag have the same meaning they did at the file scope level but now apply at the function level. The @spec tag, the new tag, is used to specify the data types that apply to a given function as well as the return values that it sends back. The @spec tag for start link

```
%% @spec start_link(Port::integer()) -> {ok, Pid}
%% where
%%  Pid = pid()
```

indicates that the function takes a single argument that is an integer and returns the tuple {ok, Pid} where pid is a pid. Types are always written in function style notation, meaning they have () following them as in integer(). These types can be attached directly to Variables with the :: notation is we see for the variable Port or they can be added to a where clause as we have done for Pid. Notice also that function level comments contain %% two percent signs as opposed to three for file level comments.

The next thing we need to cover now is the third part of our behaviour implementation module layout and that is the behaviour interface functions section otherwise known, and henceforth referred to as, the callback function section.

3.2.4– The Callback Function Section

Each of the gen_server container functions we used in our API correspond to specific functions specified by the gen_server interface and implemented in the callback function section. To refresh your memory here is table 3.2 again with something extra added.

Function Name	**Associated behaviour interface (callback) function**	**Description**
gen_server:start_link/4	Module:init/1	Startup a gen_server process
gen_server:call/2	Module:handle_call/3	Send a synchronous message to a gen_server process
gen_server:cast/2	Module:handle_cast/2	Send an asynchronous message to a gen_server process
N/A	Module:handle_info/2	Messages sent to a gen_server that were not sent with one of it's standard container messaging functions like call or cast. This is for "out of band" messages.

table 3.4

Look at our tr_server:start_link/1 function back in code listing 3.2. That function hides that fact that we are calling gen_server:start_link. As we can see from the table above a call to gen_server:start_link will result in a call to tr_server:init/1. This means that our tr_server needs to export init/1 so that when start_link is called the behaviour container can find the callback function and ensure that our tr_server process has been properly initialized before the call to start link returns and unblocks the caller. Similarly our tr_server:get_count/2 function shields the user from having to worry about our protocol and the fact that the messaging is done with a gen_server:call/2. This function too will need a corresponding

callback exported which is tr_server:handle_call/2. This function will serve to handle the messaging sent by the gen_server:call/2 function which is employed by our get_count/0 function.

Notice that the handle_info callback function does not coorespond to any gen_server API function. This function is an important special case. This callback reacts to any messages sent to a gen server process mailbox that did not come from one of the gen_server messaging API functions for example any message sent via a plain old "!". In the case of our RPC server we are going to use it to receive TCP messages which will be pulled off the socket and sent to our processes as plain old messages. The next step here is for us to start defining all the functions we need to for our implementation to conform to the behaviour interface for the gen_server. We will define these functions now in code listing 3.3.

Listing 3.3 - Generic Server callback definitions

```
init([Port]) ->
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),      #1
      {ok, #state{port = Port, lsock = LSock}, 0}.

handle_call(get_count, _From, State) ->
    {reply, {ok, State#state.request_count}, State}.    #2

handle_cast(stop, State) ->
    {stop, ok, State}.      #3
```

The functions from the code above are where we do our real work, lets dive in and understand exactly what they do.

INIT/1 – CALLBACK FOR GEN_SERVER:START_LINK

init/1 is the first callback we encounter in our gen_server behaviour callback section. As mentioned above the `gen_server:start_link/4` function is how we spawn a new process. Init and start_link are really the first examples we will ecounter of how OTP helps us write industrial strength code with a minimum of effort. start_link prepares us to be hooked into the powerful fault tolerant process supervision structures of OTP. It also provides us with critical initialization functionality. When start_link is called it blocks the caller and makes a call to our init callback function. start_link stays blocked until our init function does what it needs to do in order to prepare our process's base operationali state and then returns. This enables you to ensure that your process is fully initialized before any other processes can possibly send it a message. We are going to break this function down line by line. The first thing we see is

```
init([Port]) ->
```

which indicates that init takes a single argument, a list, and that this list contains one element "Port". This was passed in through our start_link function. Next create our TCP listen socket with the help of gen_tcp (seen at code annotation #1). A listen socket is a

socket that we create and wait to accept actual TCP connections on. Once we accept a connection on our LSock we will have an active socket connection from which we can receive TCP datagram's. We set this socket to active true which tells gen_tcp to send all incoming TCP data to our process as messages in our process mailbox.

```
{ok, LSock} = gen_tcp:listen(Port, [{active, true}]),
```

Lastly we return from init with a tuple containing the atom ok, our process state in the form of a #state record, and a curious 0.

```
{ok, #state{port = Port, lsock = LSock}, 0}.
```

The 0 is a timeout value. What adding a timeout of 0 says to our generic server is that as soon as we are finished with init we should generate a timeout message which will force us to handle that timeout message as soon as init returns. The reason for this will be explained in section 3.2.5.

HANDLE_CALL/3 – CALLBACK FOR GEN_SERVER:CALL

The callback `tr_server:handle_call` is invoked every time a message is received in a gen_servers process mailbox that was sent from the gen_server:call function. The generic server call function is an example of how behaviours help you write more reliable code in less time. All messaging in Erlang is asynchronous but there are times when we need to have synchronous semantics when making function calls that ultimately sent messages underneath the scenes. The gen_server:call function builds up this synchronous request response functionality in a reliable way from Erlangs asynchronous messaging primitives. The function takes three arguments, the first is the message sent directly via gen_server:call, the second is From (lets not worry about that yet), and the third is the the state of the server. In our handle call method we interrogate the state to retrieve the request count and we return it (listing 3.3, code annotation #2). The return is done through the return tuple of the call function.

```
{reply, {ok, State#state.request_count}, State}.
```

Our tuple indicates to the generic server that we want to send a reply to the call, that the response the caller of gen_server:call should get back is `{ok, State#state.request_count}`, and finally that we want to cache the same state we started with, unchanged.

HANDLE_CAST/2 – CALLBACK FOR GEN_SERVER:CAST

Our api function named "stop" uses gen_server:cast to send an asynchronous message to the tr_server process. The message stop upon being received by our server causes it to terminate. Whenever gen_server:cast is used to send a message to our server that message

will be picked up by tr_server:handle_cast/2. Our handle_cast function takes two arguments, the first of which is the message sent via the cast and the second of which is our server state. When our handle_cast function sees the message stop, it returns the following return tuple:

```
{stop, ok, State}.
```

(Which can also be seen in context at listing 3.3, code annotation #3). This tells our server to stop and to return the reason for stoppage as "ok" which indicates a graceful shutdown. At this point we have covered most of the salient behaviour and relationships surrounding our use of the gen_server container and callback functions. There is more to the gen_server but we will see more on that throughout the book. There is one thing missing from our coverage of what we are currently using gen_server for which is the use of handle_info for out of band message handling. As we said earlier there are sometimes messages sent to a process that are not defined in the API. If for example someone sent a message to this process using pid() ! random_message that random message would not be handled in any of the callback functions we have defined thus far. It would be considered then and "out of band" message. Before we go on we must point out that sending messages that way will lead to confusing code; keep the alphabet, the protocol, hidden in the API and use the standard behaviour container functions for transporting your messages. There are some occasions however when receiving out of band messages is acceptable. Remember the purpose of this module is primarily to handle TCP streams, those are not sent from the API but instead come in off the socket connection we set up and are consequently considered out of band, but a quite acceptable use of out of band messaging as is the next usage, for timeouts. We got the tr_server process setup with a listen socket in tr_server:init/1 and then returned a timeout of 0 from that function. Returning a timeout of 0 indicates that a timeout should be generated instantly upon returning from the init/1 function. The timeout is the 3rd element in the return tuple below.

```
init([Port]) ->
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),      #1
      {ok, #state{port = Port, lsock = LSock}, 0}.
```

That timeout of 0 generated an out of band 'timeout' message to be sent to our process which is handled in the handle_info/2 generic server callback function. Back to TCP sockets; when a socket is set to `{active, true}` as ours is (code annoation #1) messages are delivered straight from the socket into our process mailbox. They are not sent as a result of a client calling one of the servers API functions. They are "out of band". Code listing 3.4 covers the writing of the handle info callback function.

Listing 3.4 - Handle Info Code

```
handle_info({tcp, Socket, RawData}, State) ->     #1
```

```
    RequestCount = State#state.request_count,
    try      #2
      MFA = re:replace(RawData, "\r\n$", "", [{return, list}]),  #3

        {match, [M, F, A]} =
      re:run(MFA,        #4
                    "(.*):(.*)\s*\\((.*)\s*\\)\s*.\s*$",
                    [{capture, [1,2,3], list}, ungreedy]),

      Result = apply(list_to_atom(M), list_to_atom(F), args_to_terms(A)),          #5
      gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Result]))      #6
    catch
        _C:E ->
            gen_tcp:send(Socket, io_lib:fwrite("~p~n", [E]))
    end,
    {noreply, State#state{request_count = RequestCount + 1}};
handle_info(timeout, #state{lsock = LSock} = State) ->          #7
        {ok, _Sock} = gen_tcp:accept(LSock),
     {noreply, State}.
```

That was quite a bit of code. Time to dive in and explain what it all means.

HANDLE_INFO/2 – OUT OF BAND MESSAGING CALLBACK

Remember, we are creating a server, that server has an API, an interface to the world. When functions in the API are called they are actually messaging a process. Messages may also come from other outside sources like a TCP socket. It is here in the gen_server/handle_info/2 callback function clauses that we handle those out of band messages. In the code above in listing 3.4 we handle we handle 2 messages of this type, we will talk about the timeout message first, even though it appears last in the code at code annotation #7. This message is created whenever a timeout is triggered between the invocation of callback functions. If you think back to our init function you will remember we added a timeout value of 0. Again by way of an example here is our init function, which contains a timeout of 0.

```
init([Port]) ->
    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),
    {ok, #state{port = Port, lsock = LSock}, 0}.
```

That tells our process that if 0 milliseconds elapse between the time the timeout value is returned and the next callback invocation then a timeout message should be generated. In this case that essentially means that before anything else can happen, i.e. info coming in on our socket, we generate a timeout message. That message is out of band, not prompted by "call" or "cast" or any of the other messaging functions generic servers understand so it is therefore handled in handle_info/2 which can be seen at code annotation #7 in code listing 3.4. This timeout clause of handle info does one important thing, it waits on a call to gen_tcp:accept on our listen socket for a TCP connection. Once that connection is made we are ready to receive messages off of the socket via that active socket mechanism. That

brings us to the other clause of the handle_info function at code annotation #1. This clause matches on the message {tcp, Socket, RawData}. This is the message sent by an active socket when data is pulled off the buffer. RawData is what we are interested in; it is that data that contains the instructions from the user. Notice that one of the first things we do is wrap most of the inner workings of this clause in a try clause; seen at code annotation #2. This clause is handling data from the outside world, data that is out of our control. We need to guard and check the data here so we don't have to do it later.

CHECK THE BORDERS

Checking data as it passes from the untrusted world into the trusted inner sanctum of your Erlang code is a fundamental Erlang design principle. Once you verify the data is what your code expects there is no need to error check it any further. After that you code for the correct case and allow supervision to take care of the rest. The reduction in code, and thus the reduction in programmer error, with this technique is quite significant. If there is something too terribly wrong you will see process restarts a plenty in your logs and you can correct the problem then.

The try clause will feed any exception to the waiting catch clause below it which will then send a generic error response back to the use via gen_tcp:send/2.

The first thing we do once in the try clause it to clean the data we get from the socket, using the "re" regular expressions module from stdlib to strip it of a newline carriage return pair.

```
MFA = re:replace(RawData, "\r\n$", "", [{return, list}]),
```

This leaves us with our protocol which should be the module to call followed by the function to call and then the arguments, if any, all of which are to be transformed into Erlang terms and executed. At code annotation number 4 we see the following lines.

```
{match, [M, F, A]} =
          re:run(MFA,
                 "(.*):(.*)\s*\\((.*)\s*\\)\s*.\s*$",
                 [{capture, [1,2,3], list}, ungreedy]),
```

This is another use of the powerful re module to split out the module, function, and argument portions of the text that comes in over the socket. For example if we sent:

```
lists:flatten("hello", "dolly").
```

this code would return and bind the variables M, F and A as follows:

```
M = "lists"
F = "flatten"
A = "\"hello\", \"dolly\""
```

The exact syntax and usage of regular expressions and the re module is beyond the scope of this book. The re module features perl compatible regular expression syntax, something that is very well documented on the web. Regular expressions are very powerful and we recommend you read up on them if you are not already familiar. This next block of code found at code annotation #5 handles the execution of the function we wish to call.

```
Result = apply(list_to_atom(M), list_to_atom(F), args_to_terms(A)),
```

apply/3 is used for dynamic function calls. As you can see we convert the string M for module, and the string F for function into atoms as prescribed by the apply/3 function. The third argument to the function in our code is the result of another function call, args_to_terms/1. This function is an internal function. Cannonically internal functions are place below all of the gen_server call back functions. We typically delineate that section of the file with a comment. All of this is below:

```
%%%===================================================================
%%% Internal functions
%%%===================================================================

args_to_terms([]) ->
    [];
args_to_terms(RawArgs) ->
    {ok, Toks, _Line} = erl_scan:string("[" ++ RawArgs ++ "]. ", 1),
    {ok, Args} = erl_parse:parse_term(Toks),
    Args.
```

This function returns either an [] list in the case that no arguments were supplied for the function we are to apply, or it does a little bit of parsing magic in order to turn the terms passed as arguments into one big list containing all the arguments to be applied. Placing all arguments in an enclosing list is because apply/3 wants all arguments in a list and it has the nice side effect of making it so we only have to scan and parse a single Erlang term. Taking the example of lists:flatten("hello", "dolly") from earlier we would have the following steps.

1. "\"hello\", \"dolly\"" is scanned as erl_scan:string("[\"hello\", \"dolly\"]"), notice the [ ] brackets now enclosing the arguments. The return value from erl_scan:string are Erlang tokens representing the term we would like to create.
2. The tokens are supplied to erl_parse:parse_term(Toks) which takes the tokens and converts them into a true Erlang term: ["hello", "dolly"]

Once the term is returned module, function, and args are executed by apply/3. In the future we will may want to limit the functions a user can call, and make that configurable but for now we keep it simple and dangerous allowing the user to execute any function the ERTS system has loaded.

The return value from the application of the desired function is then taken and converted into a string and sent over the socket as a response to the user. This code is printed below.

```
gen_tcp:send(Socket, io_lib:fwrite("~p~n", [Result]))
```

The technique used for doing this is worth examining. We use the pretty printing functionality ~p contained within the io_lib:fwrite/2 function to pretty print the term that is returned by apply/2. It is worth noting though that fwrite returns a deep list of characters which are not entirely readable to the human eye. The call to send this deep list over a socket automatically flattens this list. If the results of io_lib:fwrite were for human consumption it would be necessary to call lists:flatten/1 on the return value prior to displaying it.

In the event that something goes wrong with the processing we are doing on the data incoming off of the TCP socket that exception is caught as pictured below. The exception generated receives the same pretty printing treatement and is sent back as a string over the TCP socket.

```
    catch
        _C:E ->
            gen_tcp:send(Socket, io_lib:fwrite("~p~n", [E]))
    end,
```

Below we have the last line of code in our function clause which updates the request count in our server state #state record so that the API call to get count can return an actual count of the number of commands executed for a given session.

```
{noreply, State#state{request_count = RequestCount + 1}};
```

To update a record first access the old instance of it by using the record variable followed by a # sign and the record name, then set the desired field equal to the new value. This expression will return a new record with the updated value.

Our RPC server is coded and ready to try out, we need only include our code in the shell and call startlink which will start our Generic Server process. With our RPC server we can call any exported Erlang function the ERTS system has loaded, except one. The one we can't call from our TCP session is our very own tr_server:get_count/0. If a call is made there would be deadlock. This is one of the few tricky deadlock situations you will run into in an Erlang program. The good news is that once you can spot it, it is very easy to avoid. Basically make a synchronous call from within a server callback function with no timeout to the same server process. In this case it is handle_info/2 making a gen_server:call through the get_count/0 function to itself. The result will be that the gen_server waits forever for the response because it can only handle the synchronous call to get_count/0 after it has finished processing the current call to handle_info which is now dependent on get_count/0. Circular wait style deadlock. To make this more clear lets break this down into steps.

1. TCP in with function to call "tr_server:get_count()."
2. Handle info processes this and calls apply(tr_server, get_count, []).
3. Get count sends a message to the very same tr_server process. BUT that message is never picked up because we are still in the previous call to handle_info.
4. DEADLOCK

Recognizing that situation will almost guarantee that you will not deal with deadlock situations in Erlang – because that is about the most common one and like we said, it is fairly easy to spot. Copy everything share nothing asynchronous messaging makes deadlock a particularly difficult thing to get into, luckily. One last thing to mention, gen_server:call has an automatic timeout associated with it, and so we will eventually unblock, but that is not something you want to rely on, it is best to avoid this case all together. Our RPC server is done. Moving forward we need to give it a trial run to see how well it works.

3.3 - Running the RPC Server

The first step to getting this running is compiling the code. To do that we run the command erlc tr_server.erl. If that returns without any errors you will have a file named tr_server.beam in your current directory. Now do as I do:

```
Eshell V5.6.2  (abort with ^G)
1> tr_server:start_link(1055).
{ok,<0.33.0>}
```

Running erl first will yield you an Erlang shell prompt. From that prompt start the tr_server by calling its start_link function passing it any port. I choose to use 1055 because it is easy to remember (10 = 5 + 5). The call to start_link will return a tuple containing ok and a process identifier, it's the weird looking thing with the numbers in it. More on that later. Lets leave the shell now and open up a second shell prompt where we can run our telnet session.

```
Macintosh:~ martinjlogan$ telnet localhost 1055
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
init:stop().
ok
Connection closed by foreign host.
```

Our first session was a success – why you say? Let's inspect our results and see what exactly happened up there.

I use telnet to connect via TCP on port 1055 to my now running tr_server. Once connected I run the following command obeying our protocol, really Erlangs protocol, init:stop(). This should result in a function call of the form init:stop(), or actually, apply(init, stop, []), being generated and called. Since init:stop/0 returns the atom ok we expect to see the string "ok" sent back to our telnet session – which we do again. The next thing we see is "Connection closed by foreign host". This is printed by the telnet utility because the socket it was connected to was close by the server, our Erlang VM. The reason for that is that the function init:stop() shuts down ERTS. Wonderful in that we have just demonstrated that our RPC server works and we have also shown just how dangerous it can be! That was a meaty section. We have built in just a few lines of code a very powerful application that is quite useful in the real world.

3.4 – Summary

We have covered quite a bit in this chapter. We have learned a little more about processes, modules, functions, and messages. We have also learned about behaviours and the three parts that make them what they are the:

1. Behaviour Interface
2. Behaviour Implementation
3. Behaviour Container

We have covered the gen_server behaviour specifically at some depth and we did this all through a real world example in our tcp_rpc service. In the next chapter we are going to take our little stand alone generic RPC server and we are going to hook it into a structure that is going to render it enterprise grade. We are going to hook it into an OTP application. Once that is complete our little server will be part of an application that is versioned, fault tolerant, ready for use by others in their projects, and ready to go to production. Chapter 4 will take what you have learned in Chapter 2 and here in chapter 3 and will add on another layer in the basic understanding of Erlang/OTP as a framework by teaching you the fundamental fault tolerant structure, the supervisor, and by teaching you how to roll up your functionality into nice OTP packages. From there we will refine that understanding in Chapter 5 explaining the theory behind some of the things you have taken for granted up until that point. Once that is complete we will be ready to move on into creating something seriously powerful Erlang software.

4

OTP Packaging and Organization

The Erlang/OTP ecosystem's entire purpose is building stable, fault tolerant systems. We have already started to introduce some of the core concepts around using this ecosystem in Chapter 2. We are going to take you a bit further and teach you how to build up a fault tolerant; production quality system around the RPC_server you just finished building in Chapter 3.

We are going to do this by introducing two fundamental concepts. Those concepts are Applications and Supervisors. Applications are the way that we package related modules in Erlang. They are a little bit different than packaging systems in other languages because they have the ability to be alive and dynamic. Like everything else in a concurrent language, Erlang Applications, themselves, are concurrent. They can have a well defined lifecycle. They generally startup, do what they are designed to do and shutdown. This makes them much more interesting and, generally, more valuable then jars or gems or other packaging system you may be accustomed to.

Supervisors are another interesting feature of the ecosystem. They monitor processes. They watch a process and if anything goes wrong they restart the process. This is very similar to daemon managers, like daemon tools, on operating systems. Their whole purpose for existence is simply to make sure a process never permanently dies.

Throughout this chapter we aren't going to delve too deeply into the theory or practice around using Applications and Supervisors. We will mostly concentrate on the practical aspects of armoring the module we just created in the Chapter 3. That means wrapping it up as an Application and setting up a Supervisor for it. We will go over the most basic aspects of both tasks and talk about why they are important. We do this to get you over the hump of these concepts and not overload you. Later on in Part 2 of this book we will go into some real detail and talk about all the interesting options that are available for advanced supervision,

for handling code within applications and even for packaging multiple applications into a larger meta structure called a release.

4.1 OTP Applications

We will get started by talking about how to package up your code and get it to fit nicely into a normal OTP system. For some reason this topic tends to cause a lot of confusion to folks who are new to the system. We can understand that. When we first started working with Erlang, what is traditionally known as OTP was black magic, poorly documented and with very few examples. All the authors of this book travelled similar paths to gain knowledge of this powerful system. It involved a lot of trial and error and quite a few helpful hints from various old timers on the Erlang mailing list. Fortunately once you get your mind around the basic concepts it's really pretty simple. Just remember that Applications are fundamental ways to group related code. Applications can be just libraries, called "Library Applications", which are collections of related code to be used by other Applications. The OTP stdlib is a good example of a library application. Much more often however Applications are living things with explicit startup and shutdown semantics. These living Applications are referred to as "Active Applications". Active Applications must have a root supervisor whose job is to manage the processes contained within the Application. Generally when we refer to Applications in this chapter we are referring to them generically or to Active Applications unless we specifically state we are talking about Library Applications.

Active vs Library Applications

Both Active and Library Applications are Applications. They both share the same meta data formats and directory structures and both fit into the overall OTP application framework. The main difference between then is that Active Applications have a life cycle. The startup and they shutdown and between they do work. They have a top level supervisor and a tree of processes that do actual ongoing work. Library Applications by contrast are a collection of modules that don't actually start as part of the application. Library Applications are passive and are there to be used by other Applications.

4.1.1 Explaining Applications

Creating an Application consists mostly of setting up your directory structure and including the Application metadata. This metadata tells the system what it needs to know to start and stop the application. It also details information about the applications dependencies and what needs to be present or started beforehand. There is some coding involved as well but we can get to that in a minute.

The Drawbacks of Naming Them ‘’

> The main drawback to packages en Erlang is that they are not called packages, jars, gems or anything else that would be explicit and understandable. They are called 'Applications' instead. In one way it makes sense, but it also tends to cause a huge amount of confusion. When people are talking about Applications you never know whether they are talking about them in the Erlang sense or the broader sense used by the rest of the industry. To reduce the level of ambiguity, from here on out when we are talking about Applications we will always mean Applications in the Erlang/OTP sense.

Erlang, in general, doesn't have an archived format for its Applications. It just uses a very simple directory layout as shown in the following code snippet.

```
<application-name>-<application-version>
  |
  |- ebin
  |- include
  |- priv
  |- src
```

That's it! That is most of what you need to do to get a valid format for your Application. You should, of course, replace <application-name> with the name of your Application. In our case its tcp_rpc. You should also replace <application-version> with the version of your application. For tcp_rpc I am just going to choose 0.1.0. Each directory listed in table 4.1 is a place for something. Some of them are self-explanatory others need a bit more detail.

Table 4.1 Application Directories and Their Importance

Directory	Description
Ebin	This is the location where all of your complied code should end up. It is also the location of the dotapp file, which contains the application meta data. We will talk about it in more detail shortly.
include	This is where your public header files go. Basically any hrl file that is part of your public API should end up in this directory. Private hrl files that are only used within your code and are not intended for public consumption should end up in the same place as all the rest of your source.
priv	Ah, the priv dir. Anything that needs to be distributed with your application ends up in this directory. This ranges from templates to shared objects and dlls. The location of an application's priv directory is very easy to access. Just call the function

	code:priv_dir(<application-name>). This will return the full path of the priv dir as a string.
src	This is where you put all the source related to your application. All of your Erlang, ASN.1, YECC, MIB etc. It all goes here. Its not required that you distribute your source code with your application. However, as many of the Erlang applications you will use are open source they normally include src .

Now that we have our Application directory structure in the layout that Erlang expects we can work on adding the metadata that the system requires. The meta data consists of Erlang terms that describe the application in a file called <application-name>.app in the ebin directory. In Listing 4.2 we will create the meta data for our tcp_rpc Application.

4.1 Application Metadata

```
%% -*- mode: Erlang; fill-column: 75; comment-column: 50; -*-

{application, tcp_rpc,
 [{description, "RPC server for Erlang and OTP in action"},
  {vsn, "0.1.0"},
  {modules, [tr_app,
             tr_sup,
             tr_app]},
  {registered, [tr_sup]},
  {applications, [kernel, stdlib]},
  {mod, {tr_app, []}}]}.
```

This file is used by OTP to understand what code to load into the run time system (ERTS) for the application, how the application is started, and how it fits into the ERTS system with other applications. To understand how this works you should understand that Applications in Erlang are quite a bit different then what you are used to in other languages. In Java jars or ML modules the code is dead more or less, meaning that it doesn't do anything until it is called by some running code. In most Applications this isn't the case at all. The runtime system starts the Application when the system starts and stops it when the system is shut down. For this to function the overall OTP system must understand where your application fits and when and how to start it.

The format of this .app (pronounced “dot app”) file itself is straightforward. It’s just an Erlang term terminated by a period. It consists of a tuple of three elements, a three tuple. The first element is the atom 'application'. The second is the name of the application. In this case that’s 'tcp_rpc'. The final element in the tuple is a key value list of parameters, some of which are required others which are not. The ones we have included here are the most important ones that you need in most applications. Depending on the nature of your

application you may or may not need some of these parameters. We will get into a lot more detail and theory in Part 2 of this book. For now I will give you the information you need to know to get off the ground. Let's take a second and go over these parameters. Take a look at Table 4.2 which outlines the file parts.

Table 4.2 *.app File Parts

Part	Description
description	A short description of your application. Usually just a sentence or two, though it can be as long as you would like.
vsn	This describes the version of your application. The version can be anything you would like. Its formatted as a simple string. However, I suggest that you try to keep your versions to the normal <major>.<minor>.<patch>. Erlang and OTP wont have any problem with any type of version you put there. There are systems out there that try to parse this and they may not be as well written. There is also the advantage that everyone out there understands this format and may not understand your personal version format. In the end its up to you.
modules	This is a list of all the modules in your application. It's a bit tedious to maintain, but there are tools out there to help you with that maintenance. This is a simple list order doesn't matter at all.
registered	This is a bit more interesting then the others. As you know Erlang has the concept of registered processes. These are long lived processes that are registered at under an addressable name. They function like little services in your Application. OTP needs to know which process are registered for various purposes such as system upgrades and because 2 processes with the same registered name is not possible within a single running ERTS system. The system can warn early of duplicate registered names issue if the names are included in the .app files. The registered key value pair lists the registered names of processes in the system.
applications	This is a another interesting one. Applications have dependencies. Applications, because they are living systems, expect these dependencies to be started and available when they start. This entry is a list of all the applications that need to be started before this application can be started. Order doesn't matter here either. OTP is smart enough to look at the entire system and understand what needs to be started when.

mod	This is the place where you tell the OTP system how to start your application. This tuple contains the module name, along with some optional arguments that can be supplied on startup (don't use them use a config file instead this is an uncommon place to put config in general). The module supplied here needs to conform to the application behaviour which means it will export a start/2 function. The module and args supplied here get called along with the start/2 application behavour function to start your application. Shutdown works similarly.

There are other parameters that are available here for your use. We won't go into them now because they aren't really relevant to our tcp_rpc app. We get into a lot more detail later on in the book, for now we have covered the most critical and usrful terms.

We have the directory structure set up. We have the metadata in place. However, we haven't quite done everything we need to do to package up tcp_rpc. As mentioned earlier when describing the mod key value pair, we need a starting point for our application in the form of a module that implements the "application" behaviour. We will create this starting point in the next section.

4.1.2 The Application Behaviour

Every Active Application needs one 'application' behaviour implementation module. This behaviour provides the start up logic for the system. At a minimum this provides the point from which the root supervisor is started; the supervisor that is the ultimate originator of all the processes that will be started as part of the application. The application behaviour may or may not do other things depending on the needs of your system. As for the root supervisor, we will get into supervisors in just a bit. Right now with Listing 4.3 we concentrate on the application behaviour implementation.

4.2 Application behaviour

```
-module(tr_app).

 -behaviour(application).      #1

-export([
     start/2,
     stop/1
     ]).

start(Type, StartArgs) ->
   case tr_sup:start_link() of      #2
    {ok, Pid} ->
        {ok, Pid};
```

```
        Error ->
            Error
        end.

stop(State) ->
        ok.
```

The 'application' behaviour for our RPC server is pretty normal for an application. I have taken out most of the documentation just so we can get a nice overview of the code itself. We start out with the normal module attribute. We then specify that this implements the application behaviour via '-behaviour' attribute at code annotation number 1. It is not absolutely necessary that you provide this attribute, it mainly exists so that the Erlang compiler can check that you actually implement the functions required by the behaviour. During runtime whether or not you implement the behaviour is ignored. The functions are just called and if they don't exist an exception is generated. We advise you to implement the behaviour. If you start editing other peoples code you will find that those that implement the behaviours properly generally have systems with fewer errors that are easier to read and maintain.

Next we export the functions that we have implemented. To accomplish this we use the export atribute; typically one export attribute for the API of our module and one for the functions required by the behaviour that we are implementing. Here we are only really implementing a behaviour so we only have the one export.

The first real function that we care about is the start function. This is where the magic happens. The start function will be called when your Applications starts up and it must return the pid of the root supervisor. So at the very least the root Supervisor must be started (this happens at code annotation #2). You can, of course, do anything else you would like to do here as well, such as pull configuration from a config file, or initialize an ETS table, or any other simple one time per application type functionality. For tcp_rpc all we really need to do is start the root supervisor. You can see that we do that with the tr_sup:start_link function. We check the return value to make sure it works for us and then we return the value. If we don't get the value we expect we return an error.

Well we have implemented our Behaviour but now we hit on a small but annoying issue. What do we name our module! This is an important topic because all modules in Erlang live in a single namespace. This has certain implications for module naming and means that this module naming takes a bit more then the usual amount of thought.

NAMING

One thing you may have noticed within our examples is that we have been using modules named tr_*. As explained in chapter 2 Erlang has a flat namespace. That means that every module exits in the same global namespace. While this makes things simple it does open up the possibility of namespace collisions. These are always nasty little problems to debug and there really isn't a lot you can do about them when the occur, unless one of the modules

colliding is your own. Unfortunately, in practice this almost never happens. Its almost always two modules in different third party libraries when it does happen. In the interests of reducing the probability of these types of namespace collisions the community has adopted the practice of building a prefix into the module names. This is very similar to what has been done in other languages that lack a namespace. It's a bit inelegant but in general it works very well.

Prefix Naming

The authors typically choose the acronym one would form from the name of an application as the prefix for each of its modules. TCP RPC would be T.R hence tr_.

In practice the namespace consists of the initials of the application. It is separated from the actual module name by an underscore. We do this for all of our modules. For a few we have some standard names we use. For example, for the module that implements the 'application' behaviour interface we usually name it <namespace>_app and for the root supervisor we usually name it <namespace>_sup. For our application it means that we are going to have at least two modules, one called tr_app and another called tr_sup. ts_app is the module that implements the 'application' behaviour while tr_sup is our root supervisor. You have already seen these names in our various examples.

The modules tr_app and tr_sup alone aren't really very useful without some other processes and modules coded to actually do the work the application is supposed to accomplish. Fortunately we already put that together in the last chapter. We will take that module which luck would have it is already named correctly as tr_server and place it into our src directory. Our luck does not stop there with th name of tr_server, we happen to already be using a gen_server for it as well so it is going to plug right into our application without change as we will see when we get through section 4.2 on supervision.

So there are three things that we need to do to create an Application skeleton.

3. We must use the correct directory structure.
4. We must add the Application metadata in the form of the *.app file.
5. We must create a module that starts our Application and implements the Application Behaviour.

That essentially covers the basic skeleton, once that is done the work to flesh it out begins with creating a root supervisor. We also need to make sure that all of our worker processes are capable of being supervised. This basically means that they implement one of the OTP behaviours like gen_server, gen_event, gen_fsm, etc. What behavours do to render their processes supervisable is not deep magic, it involves returning proper expected values and establishing Erlang links between the behaviour process and the supervisor that started

it. For now it is not required to know the specifics, only that we need to implement processes that implement behaviours when we create applications.

That just about wraps up the salient points of OTP Applications as a structure in and of themselves but it doesn't completely finish the topic of creating an application that actually works and lives. An application that lives starts processes to do the work it is coded to accomplish. Those worker processes are spawned and supervised by processes that implement the supervisor behaviour. In the next section we will to get up to speed with supervisors.

4.2 Adding Fault Tolerance with Supervisors

Supervisors are one of the core things that make Erlang what it is. As was stated earlier, applications accomplish there work with processes. Those processes are started and supervised by supervisors. An Active Application is essentially a tree of processes, supervisors and workers, where the root of the tree is the root supervisor. Figure 4.1 illustrates an example process structure for an Application..

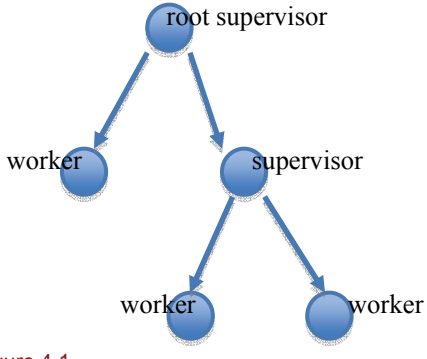


figure 4.1

The supervisors themselves are just behaviours and like all behaviours they have interfaces, interface implementations, and containers. If you are already using the OTP behaviours (and we have) then setting up a supervisor for your processes is pretty easy. Lets take a look at the supervisor implementation that we created for the our new tcp_rpc Application in Listing 4.3.

4.3 Supervisor

```
-module(tr_sup).

-behaviour(supervisor).

%% API
-export([start_link/0]).
```

```
%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).

start_link() ->
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

init([]) ->
    Server = {tr_server,{tr_server,start_link, []},
                  permanent,2000,worker,[tr_server]}
    {ok,{{one_for_one,0,1}, [Server]}}.
```

The supervisor:start_link/3 (code annotation #1) function is of a type we have already seen in our tcp_rpc with gen_server:start_link/4. This is very similar. It starts up our supervisor. Well technically we start up the supervisor directly giving it a bit of information about the behaviour. The two arguments that we pass are important, the first argument is a tuple that contains the name of registered process and where its registered. In the example above we are registering the a local process under the name tr_sup. The second argument is the name of the module that implements the supervisor (or tr_sup) and the third argument is the list of arguments that get passed to the init function on start up. We don't really need arguments in our init so we just leave it as an empty list.

The really interesting stuff in this case is the init function. This is where we tell the supervisor what to start and how to start it. Yes, that gobbledygook there goes a long way towards telling the supervisor what process to supervise and how. Let's just go over it a bit. As with the Application here there is a lot more that you can do with supervisors and we are going to go into that in a lot of detail in Part 2 of this book. For right now we should stick to gaining a good understanding of what it is we are doing in this instance.

4.2.1 - The Child Spec

The child specification is the description of the process you want the Supervisor to supervise. Consider it a recipe for the supervisor container on how you want your top level processes to be instantiated. In most types of supervisors the processes described in the Child Spec are started when the Supervisor starts and exist for the life of the Supervisor itself. Lets take a look at a simple spec in the following code snippet.

```
TelnetServer = {tr_server,{tr_server,start_link, []},
                  permanent,2000,worker,[tr_server]}
```

You will see an entry like this for every process a supervisor is going to manage. The tuple is the important part. It's just a tuple of six elements. The first element is the id that the supervisor will use to name and reference the process internally. In our case that name is tr_server, purposely the same name as our module, we do this for the same reason that we tag messages wrapped by API functions with the name of the function; no need to make things confusing. The second entry is a module function argument triple that will be used to

start the process. That is, it's a tuple of three elements where the first element is the name of the module, the second is the name of the function and the third is a list of arguments to be passed to the function, just the same as we applied arguments from the TCP stream in Chapter 2. The third element is the type of process under supervision. We have chosen permanent because we expect this to be a long lived process. There are two other options, transient and temporary, which geared towards less long lived processes. The forth option is the type of shutdown when the process is killed. Here we have used a number. That number indicates that this is a soft shutdown and the process will have that number of milliseconds to actually stop. So in this example after a kill request, for whatever reason, the tr_server process will have 2000 milliseconds to shutdown. You can make this as small or large as you would like, the right value depends on your requirements. The fifth value is also a type. It indicates whether the process is a supervisor or a worker as illustrated in figure 4.1. We can chain supervisors together into a supervisor hierarchy if we need to. Multiple is actually a pretty common approach to fine grained process control in more complex Applications. For this process we are just going to use worker. Worker process are defined as any process in a supervision tree that is not implement the supervisor behaviour Finally, the sixth option is the list of modules that this module uses. It's only used during hot code upgrades to the system. It indicates to the system in what order to upgrade modules. In general its not that heavily used and you can safely ignore it for now. With our child spec completed we now need to communicate to the supervisor behaviour container the supervision information we just pulled together. We do that through the return value from the init/1 callback function.

4.4.2 Returning from the supervisor init/1 function

The supervisor itself is an implementation of a Behaviour meaning that it consists of one process running code from a behaviour container module and a user defined callback module or behaviour implementation. This means that in order to communicate to the behaviour container we need to return information from our callback functions. The following code snippet contains what we return from the init/1 callback function to the behaviour container so that it can start the child processes we have specified.

```
{ok,{{one_for_one,0,1}, [Server]}}.
```

This is what must be returned from the function. It's a simple tagged return value that indicates to the supervisor a few which children to start and how. The first value us the atom 'ok'. The second is a two value tuple. The first value is a tuple of three elements. The first element is the Restart Strategy. We choose a one_for_one strategy here. That means that if a process dies only that process is restarted. There are more complex strategies available and we will talk about those later. The next two values are related. The first number indicates the maximum number of restarts allowed. The next number indicates the timeframe for those restarts. Lets say that the first number was ten and the next number was one thousand. In that case, we could have as many as ten restarts in any one thousand milliseconds. If we exceeded that restart count the supervisor itself would terminate and

propagate the failure up the supervision chain. The next value is the list of child descriptions. In this case, we just have the variable Server. We could, of course, have as many here as we would like and it's not uncommon to have half a dozen or so.

4.4 Starting Our Application

We have our Application complete. We have our directory structure laid out, our metadata in place and our Supervisor written. Lets see if we can't run the application we created. Go ahead and compile all the code in the src directory into the ebin directory. If you are on a unix like system the following two lines will do the trick. In general use erlc on all the .erl files to generate their corresponding .beam object code files and then ensure that those files are in the ebin directory.

```
cd ebin
erlc ../src/*erl
```

Once you do that start a shell in the ebin directory and run the start/1 command exported by the application behaviour continer. This is demonstrated in the next code snippet.

```
Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]

Eshell V5.5.5  (abort with ^G)
1> application:start(tcp_rpc).
ok
```

Look at that. It looks like it started. Well it's not really that surprising after all. We did everything right. To test this further try the same tests on the tcp_rpc application as we tried on the lone module at the end of chapter 2.

4.4 Summary

So we have gone over the major aspects of Application packaging and covered the differences that exist between Active and Library Applications. Hopefully, you have been taking these examples and trying them out as we have moved along through our code. You can get the code for this book online at github.com by searching for the title of this book. At a minimum we recommend that you compile and run it. Using these structures will increase the default level of fault tolerance for your system and help you produce consistent, reliable and understandable Erlang/OTP software. This does not end our discussion on packaging in OTP. In chapter 10 we will cover another type of package called a Release. Releases aggregate many Applications to form a running Erlang service. When we are done with that you will know how to start a full Erlang system the proper way and you will understand why the way we started our Application in listing 4.6 is only for manual testing and not to be used in production. That said, we have taken a very important step in creating production quality Erlang/OTP software with the knowledge of Applications we have gained in this chapter. We

will ultimately take this knowledge and use it to show you how to build production quality Erlang services in the second part of this book, but first, we are going to dive into a little bit of theory in chapter 5 so that we can take all the how you have just learned and flesh it out with some background on why.

5

Processes, Linking and the Platform

This chapter is fundamentally about giving you a background on models for concurrency so that you have a point of reference when thinking about the one that Erlang employes. It is about covering the fundamental aspects of the Erlang model and one of it's most important and unique features; process linking. Finally this chapter is about giving you some exposure to how Erlang implements its chosen concurrency model so as to give you an understanding of how your code will run on the Erlang/OTP platform as well as providing insight into the nature of the tool that is Erlang/OTP. This all means we are going to be diving just a bit into theory for the first and only time. In our minds getting down and dirty with the language and platform was more important, initially, than teaching you why things are the way they are. We think the right approach to teaching Erlangs' is to teach the pragmatic matters before diving into the somewhat more esoteric aspects of theory and implementation. However, just because we didn't go into detail on this topic earlier doesn't mean we don't think it's important. In fact, it's important enough that we are going to devote this whole chapter to it. Understanding these concepts should give you a firmer foundation on which to build your knowledge of Erlang and therefore build your Erlang based systems. Of course, this is going to require that we delve into a few details of how Erlang has been implemented. So this seemly disjointed chapter has the purpose of getting some of the foundational concepts of Erlang's concurrency and fault tolerance into your head in as quick and efficient a way as possible. It's up to you to absorb them and integrate the knowledge into your gestalt.

We are going to cover two main areas with an additional bit of miscellaneous coverage on interesting details of Erlangs implementation. The first bit will cover concurrent systems in

general and methods of handling shared state in particular. The second bit will cover Erlang's Process Model and one of the special and unique features therein; Process Linking. Pay attention there, this is one of the major things that make Erlang interesting and its high level of fault tolerance possible. We assume that's one of the reasons you picked up this book. The third and final bit of this chapter is going to cover some interesting bits of trivia around Erlang's implementation. We say trivia half jokingly. It would be much better to say that these bits about implementation are what the creators of Erlang got right when implementing Erlang and other languages (for the most part) haven't. So continue on brave adventurer and don't worry we will be back in the thick of hands on type development in the next chapter!

5.1 Concurrent Systems

Concurrent systems, of which Erlang/OTP is one, have started to gain in popularity over the last few years. The truth of that can be borne out by the fact that you hold this book in your hand (or are reading it on a device) at this very moment. The bane of all concurrent systems, the problem that all implementers have had to solve is the problem of Shared State. How do your processes share information with one another? It may seem like a simple enough problem, this sharing of information, but it's been wrestled with by some of the brightest minds out there with very, very few good results. There are many approaches that have been tried, four of which have gained a bit of market share. At the very least, they have appeared in a few reasonably interesting languages. In the interest of providing good background on the topic of concurency, we are going to go over a few of these. We will cover Shared Memory with Locks (what you are probably using already), Software Transactional Memory, Dataflow – Promises and Futures, and finally Asynchronous Message Passing. We will go over some of the good features and some of the bad. We will be going into a bit more detail on Asynchronous Message Passing as that happens to be the paradigm Erlang uses. In the end having knowledge about the possible approaches to state in concurrency will help you to appreciate Erlang all the more.

5.1.1 Four Approaches to Sharing State

We are going to briefly discuss four approaches to shared state that have gained mind share over the last few years. We wont spend too much time on any single one, but this will give you a nice overview of the approaches various languages and systems are taking, and highlight the difference between these approaches and Erlangs. These four approaches are Shared Memory With Locks, Software Transactional Memory (STM), Dataflow Programming, specifically Futures and Promises, and Asynchronous Message Passing. This is mostly about comparing and contrasting the types of concurrency that are available to you to help you understand why Erlang chose the model for concurrency that it did. Up first, the Shared Memory model.

5.1.2 Shared Memory With Locks

Shared Memory Communication is the GOTO of our time. Like GOTO of years past it's the current mainstream technique and has been for a long, long time. Just like GOTO, there are numerous gotchas and ways to shoot yourself in the foot. This approach to concurrency has succeeded in providing an entire generation of engineers with a deeply abiding fear of concurrency. This fear has hindered our ability to adapt to the new reality of multi-core systems. Having said that, we must now admit that, just like GOTOs, shared memory has a small low level niche where it probably can't be replaced. If you do work in that niche then you already know you need shared memory and if you don't you don't. By way of example, implementing business logic in services is not that niche.

Shared memory typically refers to a block of random access memory that can be accessed by several different concurrently running processes. This block of memory is protected by some type of guard that makes sure that the block of memory isn't being accessed by more then one process at any particular time. Generally, this causes the working processes to queue up on the guard, waiting until they can access that particular block of memory. These guards usually take the form of Locks, Mutexes, Semaphores, etc. There exists significant complexity in managing the serial access to the block of memory; there is complexity managing lock contention for heavily used resources. There is a very real possibility, and in many cases probability, of creating complex deadlocks in your code in a way that isn't easily visible to you as a developer. The number of stories of systems running for a week, a month, or even a year or more and then suddenly deadlocking are too many to count. This approach to concurrency may work for small numbers of processes. As the number of processes grows it quickly becomes unmanageable. This type of concurrency is found in all the current popular programming and scripting languages, C, C++, Java, Perl, Python, etc. Its ubiquitous, I believe, because it is fairly easy to implement and doesn't adversely affect the imperative model that these languages are based on. Unfortunately, the wide spread use of this method of concurrency has really hurt our ability to think about concurrent issues and make use of concurrency on a large scale.

5.1.3 Software Transactional Memory (STM)

The first non-traditional concurrency mechanism we are going to look at is Software Transactional Memory, or STM for short. The most popular embodiment of STM is currently available in the GHC implementation of Haskell. Wikipedia does in fact contain very well compiled and sited definitions for many computer science topics and so we will rely upon it here and also because pasting Simon Peyton Jones' original paper on the topic here proved to be a little too much text.

Software transactional memory (STM) is a concurrency control mechanism

analogous to database transactions for controlling access to shared memory in concurrent computing. It functions as an alternative to lock-based synchronization, and is typically implemented in a lock-free way. A transaction in this context is a piece of code that executes a series of reads and writes to shared memory. These reads and writes logically occur at a single instant in time; intermediate states are not visible to other (successful) transactions.

The Magic Ball

To use a bit of illustration, lets say we had a baseball that I wanted to get signed by every member of the 1949 Yankees. It's a very special ball in that I can make copies on demand and hand them out to whoever I would like. However, I am not giving them a permanent copy, I am just letting them sign it and hand it back. So ten people come up and ask to sign it. I magically make ten copies of the clean new ball. The fifth person I gave a ball too happens to get back to me the quickest. I replace the ball in my hand with the new ball number 5 just gave me. A few seconds later number 6 comes back having signed his ball. I look at the ball in my hand and the ball he is giving to me and notice that they are different. My ball has number 5s signature, the ball he is giving me has number sixes signature. Uh oh, that's not going to work. So I throw away number 6's ball and make a copy of the ball I currently have in my hand and hand it back to number 6 telling him to sign it. This repeates over and over again as the rest of the signers come back having signed an out of date ball. In essence here, I am the gatekeeper and I make sure that I always have the most recent 'copy' of the ball.

STM has a few benefits that aren't immediately obvious, from this description. First and foremost STM is optimistic. Every thread does what it needs to do without knowing or caring if another thread is working with the same data. At the end of a manipulation, if everything is good and nothing has changed, then the changes are committed. If problems or conflicts occur the change can be rolled back and retried. The cool part of this is that there isn't any waiting for resources. Threads can write to different parts of a structure without sharing any lock. The bad part about this is that you have to retry failed commits. There is also some, not insignificant, overhead involved with the transaction subsystem itself that causes a performance hit. Additionally in certain situations there may be a memory hit, ie if n processes are modifying the same amount of memory you would need O(n) times memory to support the transaction. In general, for the programmer, this approach is many times more manageable than the mainstream shared memory approach and may be a useful way of taking advantage of concurrency. We still consider this approach to be shared memory at its core but must admit that the position is an ongoing argument between us and a number of our colleagues.

5.1.4 Dataflow - Futures and Promises

Another approach to concurrency is the use of Futures and Promises. Its most visible implementation is in Mozart-Oz and even in the very main stream Java programming language. Once again the definition Wikipedia presents is a concise summary of the topic and so we will rely upon it once again.

> In computer science, futures and promises are closely related constructs used for synchronization in some concurrent programming languages. They both refer to an object that acts as a proxy for a result that is initially not known, usually because the computation of its value has not yet completed.
>
> - The Masses (i.e. Wikipedia)

Making Good Use of IOUs

Lets go back to my ball illustration to describe Futures. I want to get my baseball signed again, this time by Joe Dimaggio. I head out to Yankee Stadium (and back through time) to get his signature. I finally find him but he is a bit busy right now so he hands me a note saying that he will give me a signed ball. Shocked I sit and stare at the note until he comes back again and hands me the ball. The note, now being worthless gets discarded.

Promises are really very similar but we must change the illustration a bit to describe them. I want to get my baseball signed, of course still by Joe Dimaggio. I head out to Yankee Stadium (and back through time) to get his signature. I can't find Joe specifically but most of the 1949 Yankees are there. They get together and hand me a note promising me that I will get a ball signed by Joe. Shocked I sit and stare at the note until one of the team members comes back again and hands me the ball. The note, now being worthless gets discarded.

Our baseball stories are a bit stretched but they work to highlight the fundamental differences in Futures and Promises. A Future is a contract that a **specific** thread will, at some point in the future, provide a value to fill that contract. A promise is, well a promise, that at some point **some** thread will provide the promised value. This is very much a dataflow style of programming and is mostly found in those languages that support that style, like Mozart-Oz and Alice ML.
Futures and Promises are conceptually pretty simple. They make passing around data in concurrent systems intuitive. They also serve as a good foundation on which to build up more complex structures like channels. Those languages that support Futures and Promises usually support advanced ideas like unification and in that context Futures and Promises

work really well. However, although Futures and Promises remove the most egregious faults in shared state systems it's still a direct sharing of state with the attendant faults.

In the end both of these approaches involve shared state. They both do a reasonably good job at mitigating the most insidious problems of using shared state, but they just mitigate those problems, they don't eliminate them. The next mechanism takes a completely different approach to the problem. For that reason it does manage to eliminate most of the problems involved with shared memory concurrency. Of course, there is always a trade off and in this case the trade off the cost is in additional memory usage and copying costs; but we are getting ahead or ourselves, let us reset and start at the beginning and then proceed to the end in a linear fashion.

5.1.5 Asynchronous Message Passing Concurrency - Erlang's Approach

Erlang itself adopts this model. This form of concurrency is built around processes and message passing. Processes encapsulate state and communicate only through messages, which are not shared between them but instead copied to and from them. The communication is fundamentally asynchronous. Higher level semantics such as synchronous communication can be built up from the primitives. This is sometimes referred to as the actor model although that term is not historically used within the Erlang community to describe its own model.

> Back to baseball. I have this baseball again. This time instead of driving to the stadium (and going back in time) I decide to mail the ball to a service that specializes in getting baseballs signed by the Yankees. I put my ball in a box with a nice letter explaining what want and put it in the mail and I go about my business thinking from time to time about the ball. Two weeks later I get a box with my ball. It has a nice letter in it describing the ball and who signed it. Under the letter is the ball itself signed by the entire team!

Asynchronous Message Passing concurrency is about processes communicating by sending messages to one another. Semantically these messages are completely separate entities unrelated to whatever process they were sent from. This means that when you are writing code in a language or platform, like Erlang, that uses this form of concurrency you don't need to worry about access to shared state at all, you just need to worry about how the messages will flow through your system.

Of course, you don't get this for free. In many cases, message passing concurrency is built by doing a deep copy of the message before sending and then sending the copy instead of the actual message. The problem here is that that copy can be quite expensive for sufficiently large structures. This additional memory usage may have negative implications for your system if you are in any way memory constrained or are sending a lot of large messages. In practice, this means that you must be aware of and manage the size and complexity of the messages that you are sending and receiving. Once you get into and start using idiomatic Erlang you naturally wont send large messages. You will hide these types of large structures behind a process façade there is also a technique described in chapter 14 on performance that will allow you to avoid a copy when sending messages locally.

Much like Futures and Promises the most egregious 'shoot yourself in the head' possibilities of deadlocking are removed but dead locking is still possible but no longer probable. You must be aware of that in your design and implementation as you saw in Chapter 3 where we illustrated the number one Erlang deadlocking scenario and how to avoid it. If you are interested in more detail on the theory behind this approach take a look at Tony Hoare's 1978 book "Communicating sequential processes". This is what kicked off the topic and remains a solid description of the idea.
Of course the foundation to any concurrent system is some method of concurrency. Without some method of concurrency the idea of shared state looses all meaning. Erlang uses processes as its fundamental unit of concurrency. The term ‘processes’ carry a lot of baggage unfortunately thanks to the use of the term in various operating systems. We are going to spend the rest of this chapter talking about processes and inter-process communication in Erlang, some of their special properties and how they are implemented uniquely as part of Erlang. So without further adieu we move from the theory and implementation of non Erlang models to something quite a bit closer to home.

5.2 Processes and the Usefulness of Links Between Them

We are going to talk about Processes and messaging but also about something else Erlang provides in terms of process relationships; linking processes in Erlang. This will include writing some code to really illustrate how processes function and how this linking occurs and why it’s useful. It would be a good idea for you to take the code and actually build it on your system (you have Erlang installed already right?). Play around with it until you get an understanding of processes and how they get linked together.

5.2.1 Process Isolation

In Erlang processes are inviolate and untouchable – they are isolated. With very few exceptions the actions of one process can not influence the behavior of another. The only avenue of influence is the passing of messages. These messages can always be explicitly received by the target process which can then make a decision about the how to deal with the message. This idea is that processes are untouchable and inviolate is called "Process Isolation" and it is one of the fundamental principle on which Erlang's fault tolerance is built. No process in the system has the ability to cause another process to fail. All errors are isolated to the process producing them. This, by itself is interesting and important, but the addition of one other feature takes it even further. That additional feature is process linking.

5.2.2 Process Linking

Process linking is simply the ability for processes to forge bidirectional 'links' to another. This is a simple indicator to the system that if one process dies any process that links to it should fail as well. You may think that this violates the Process Isolation feature we just got finished describing but it is important to remember that linking is explicit and must be programmatically forged for a purpuse. That said the process receiving the exit signal from another process doesn't need to die. It may then elect to receive the message about the death of the linked process instead and take some action. It sounds so very simple and yet this model is at the heart of what makes Erlang a fault tolerant system. Take a moment and think about this before you read on. Try to come up with some ramifications of this behavior. Don't worry, We're not going anywhere.

Ah, back already? Good I hope you came up with a few ideas on your own. You may have even guessed where Erlang or better said, Erlang/OTP, makes heavy use of this. That's right in Supervisors. Erlang uses this process linking behavior to setup Supervisor hierarchies. These Supervisors watch other processes and restart them when something untoward happens. Now we are getting into really interesting stuff. Not only do we have processes that have very little influence on one another, we have a way to restart them individually when they die! Think about the power here. This is like a very, very fine grained Service Oriented Architecture. Your Erlang system may be made up of hundreds or thousands of these services each individually managed and capable of being stopped and restarted on their own with little or no effect to the system as a whole. This is how systems written in Erlang can achieve nine nines of reliability. Processes and linking also have another very important ramification, and that centers on the management of complex state. Picture a complex transaction involving many activities. When one of those activities goes awry rolling back all the state in a system without links and where data is not isolated within processes can be increadibly complex. In Erlang it is a matter of links and exist signal propagation among isolated processes as figure 5.1 illustrates.

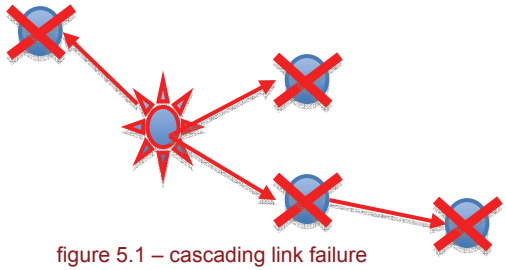


figure 5.1 – cascading link failure

as soon as a single part of the complex operation fails the others all cascade and fail leaving not a single traice of themselves in the rest of the system. This again adds to the ability for Erlang to manage complex logic with a high degree of stability and without unexpected behavior.

5.2.3 Playing With Process Isolation and Linking

It is important to really understand process isolation and linking, they are key to the OTP platform and a huge part of what makes programs written in Erlang so fault tolerant. To that end we are going to play with a few examples. In Listing 5.1 we create a simple module thats only purpose is to let us play with linking and isolation. In the interest of space I have removed the usual Erlang documentation and just added some annotations. In production code you should have the comments and documentation.

5.1 Example server

```
-module(link_ex).

-behaviour(gen_server).

-export([start_link/0, ping/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).


ping() ->       #1
    gen_server:cast(?SERVER, ping).
```

```
init([]) ->
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(ping, State) ->
    io:format("Got It!~n"),
    {noreply, State}.

handle_info(timeout, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

This is a good bit of code I know. Take a second to review it. It's a very simple gen_server that has one function as its main API. The ping function at code annotation #1 will cause our gen_server to print out a message informing you that it got the message. Copy this module to a file and compile it. Go into the directory where the beam file exists and launch the erl shell.

As in almost all gen_servers we start the process by calling the start_link function. As you can probably guess from the name this starts the process and links it to the calling process. In 5.2 you can see that I am calling process directly in the shell.

5.2 Getting Things Going

```
$> erl

Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]

Eshell V5.5.5  (abort with ^G)
1> link_ex:start_link().
{ok,<0.32.0>}
2> link_ex:ping().
okGot It!

3>
```

The exact output of link_ex:start_link() will be a bit different for you. The second part of the return value will be a pid and its going to be different from system to system and from

time to time. In Listing 5.2, you see that we print out the nice 'Got It!' message when we send call the ping/0 function.

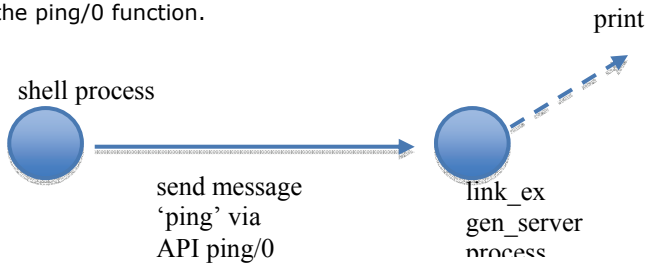


figure 5.2

Lets now send a message to the link_ex server that is not part of it’s alphabet or protocol as defined by the API, check out Listing 5.3.

5.3 Making Things Crash

```
3>  link_ex ! a_message_i_dont_understand.

=ERROR REPORT==== 7-Sep-2009::14:41:49 ===
** Generic server link_ex terminating
** Last message in was a_message_i_dont_understand
** When Server state == {state}
** Reason for termination ==
** {function_clause,[{link_ex,handle_info,
                              [a_message_i_dont_understand,{state}]},
                     {gen_server,handle_msg,5},
                     {proc_lib,init_p_do_apply,3}]}
** exception exit: function_clause
     in function  link_ex:handle_info/2
        called as link_ex:handle_info(a_message_i_dont_understand,{state})
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p_do_apply/3
4>
```

Using the servers registered name, link_ex, we send our gen_server a raw out of band message with Erlang’s ! messaging operator and… Oh No! Our server crashed what are we to do! Not much really we expected this. We don't handle the message that we are sending so we inevitably get a crash. It just so happens that the shell captures that crash and prints out a nice error message. Right now how it does that is black magic but it wont be for long.

Lets dig a bit deeper into process linking. We will create another gen_server who's only job is to pass messages to the original link_ex. We will watch them both crash. Wont that be fun. Lets get the code into a file to start though.

Second Example Server

```
-module(link_ex2).

-behaviour(gen_server).

-export([start_link/0, ping/0, ping_error/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

 ping_error() ->       #1
    gen_server:cast(?SERVER, ping_error).


ping() ->
    gen_server:cast(?SERVER, ping).

init([]) ->
   link_ex:start_link(),       #2
    {ok, #state{}}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(ping, State) ->
    link_ex:ping(),
    {noreply, State};
handle_cast(ping_error, State) ->
   link_ex ! a_message_i_dont_understand,       #3
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

As you can see this is very similar to our first example server. There are a few changes though. At code annotation #1 we have added an additional API function called ping_error/0 which will ultimately send the message 'a_message_i_dont_understand' over to the original link_ex server. This can be seen at code annotation #3. We know from past experience that the message 'a_message_i_dont_understand ' will crash the link_ex server. The other

addition is at code annotation #2 where we start the original link_ex server in our initialization phase. Place this module into a file called link_ex2.erl and compile it. Make sure it's in the same directory as the original link_ex.

With that done lets get down to business. link_ex2 does everything we need it to do with link_ex so we only have to interact with link_ex2. We want to do one additional piece of setup though. To see the processes as they start and fail lets launch pman. Its Erlang's built in process manager and is a really useful tool. Listing 5.4 has all commands that we need to execute to get everything going.

5.4 Running link_ex2 and pman

```
Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]

Eshell V5.5.5  (abort with ^G)
1> pman:start().
2> link_ex2:start_link().
{ok,<0.38.0>}
3>
```

If you look in pman's process list you should see the named servers link_ex and link_ex2. Both of them are running very happily. In Listing 5.5 we run some of the commands and see what the output looks like.

5.5 Running the functions

```
3> link_ex2:ping().
okGot It!
4> link_ex2:ping_error().
** exception exit: function_clause
     in function  link_ex:handle_info/2
        called as link_ex:handle_info(a_message_i_dont_understand,{state})
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p_do_apply/3
4>
=ERROR REPORT==== 7-Sep-2009::14:58:21 ===
** Generic server link_ex terminating
** Last message in was a_message_i_dont_understand
** When Server state == {state}
** Reason for termination ==
** {function_clause,[{link_ex,handle_info,
                            [a_message_i_dont_understand,{state}]},
                    {gen_server,handle_msg,5},
                    {proc_lib,init_p_do_apply,3}]}
```

We see that link_ex exited because it couldn't handle the message. In fact, if you look in pman's process list right now you will see that link_ex and link_ex2 no longer exist! That's because link_ex2 was linked to link_ex and when link_ex died link_ex2 died with him. Just to make sure lets run the ping function again.

5.6 Pinging the Dead Server

```
3> link_ex2:ping().
ok
```

Alas, Alas we get nothing! That's because both processes are dead and we are casting our messages into the void left by their departure. This is very useful behavior in many situations. When process depend on one another it, many times, makes sense for them to die together. In this case, we want to do something a little more interesting. Remember when we where talking earlier about process electing to receive death messages instead of simply dieing themselves? Well lets do that here. It's a relatively simple change.

Both the init_function and the handle_info function need to change slightly. I will show you the changes below.

5.7 Changing link_ex2 To Handle Thread Death

```
init([]) ->
    process_flag(trap_exit, true),
     link_ex:start_link(),
     {ok, #state{}}.

 handle_info({'EXIT', Pid, Reason}, State) ->
     io:format("Pid ~p died with Reason ~p~n", [Pid, Reason]),
     {noreply, State}.
```

Fundamentally we have made two changes here. The first significant change is that we added the 'trap_exit' process flag to the process. This indicates to the system that we want to trap death messages instead of just die with them. The second change is related to that change. We change handle_info (where we get out of band messages) to handle the messages we will now get about thread death which are of the format {'EXIT', Pid, Reason}. Figure 5.3 demonstrates the process configuration for link_ex2.

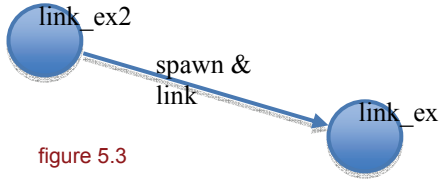


figure 5.3

Go ahead and make the relevant changes to link_ex2 and get it compiled in the usual way. Then lets run our breaking commands! they are detailed in Listing 5.8.

5.8 Running the Commands With Our Changes

```
Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]

Eshell V5.5.5  (abort with ^G)
1> pman:start().
2> link_ex2:start_link().
{ok,<0.40.0>}
2> link_ex2:ping_error().
Pid <0.41.0> died with Reason {function_clause,
                                  [{link_ex,handle_info,

[a_message_i_dont_understand,{state}]},
                                   {gen_server,handle_msg,5},
                                   {proc_lib,init_p_do_apply,3}]}
ok
3>
=ERROR REPORT==== 7-Sep-2009::15:06:58 ===
** Generic server link_ex terminating
** Last message in was a_message_i_dont_understand
** When Server state == {state}
** Reason for termination ==
** {function_clause,[{link_ex,handle_info,
                             [a_message_i_dont_understand,{state}]},
                     {gen_server,handle_msg,5},
                     {proc_lib,init_p_do_apply,3}]}4>
```

Well we still got the same error. However, if you look in the pman's process list you will see that link_ex2 still exists! That's useful. If you look at the output in the console you will see that we received a potentially useful error message as well. We got an exit message that gives us the Pid and the reason for that Pid's failure. That also has the potential to be useful even if we don't know quite what to do with it yet.

Lets make one last interesting change. Instead of just printing out the death message that we are getting from link_ex lets actually restart link_ex when we get a death message. As you can see in Listing 5.9, the relevant change is trivial.

5.9 Changing link_ex2 to Restart link_ex on death

```
handle_info({'EXIT', _Pid, _Reason}, State) ->
    io:format("Restarting link_ex~n"),
    link_ex:start_link(),
    {noreply, State}.
```

We are still going to print out a little message informing us of whats going on. However, instead of doing just that we restart link_ex. Figure 5.4 demonstrates how this will work.

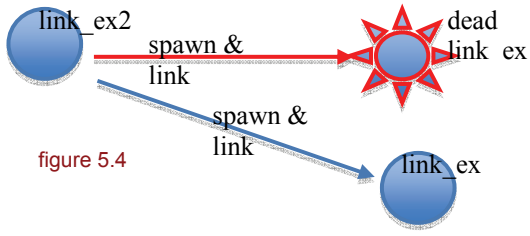


figure 5.4

This is starting to get fun, get this compiled again and in Listing 5.10 get will down to business.

5.10 Running the Revised Code

```
Erlang (BEAM) emulator version 5.5.5 [source] [async-threads:0] [hipe]

Eshell V5.5.5  (abort with ^G)
1> pman:start().
2> link_ex2:start_link().
{ok,<0.40.0>}
3> link_ex2:ping_error().
Restarting link_ex
ok
4>
=ERROR REPORT==== 10-Sep-2009::18:35:35 ===
** Generic server link_ex terminating
** Last message in was a_message_i_dont_understand
** When Server state == {state}
** Reason for termination ==
** {function_clause,[{link_ex,handle_info,
                              [a_message_i_dont_understand,{state}]},
                    {gen_server,handle_msg,5},
                    {proc_lib,init_p_do_apply,3}]}

4> link_ex:ping().
Got It!
ok
5>
```

Kick off pman and start the processes. Now watch pman's process list as we execute the ping_error function. What happens? link_ex disappears for a moment and then reappears! What happens when we execute ping? It outputs as we expect it too! Congratulations you have just written a custom supervisor. This is not to say that you ever want to do such a thing in good production code. Supervisors have much more functionality than what we just put together, are time tested and handle all the edge cases that we did not. In fact, if I ever

see this example in production code, refactor, immediately. That said it does a good job of illustrating the basics of process linking, process isolation and how supervisors are implemented. Linking and exit signal handling are not only useful as a way to understand supervision, it can be quite a useful technique in other respects. We will get into some real examples later in the book. An example to keep in mind for now are resource pools. In Erlang processes are cheap and plentiful but resources like a database connections or external ports are not. In these cases its useful to have one named process fronting a number of unnamed dynamic processes that front or contain the resource. You can probably figure out where I am going here. Its very useful for the named process to be linked to the resource processes so that if they die it can remove their pid from its internal store. There are a lot of other applications as well.

At this point you understand that it is common to have a large number of processes running on any single Erlang node. Its not unheard of for there to be hundreds of thousands of processes running at the same time and we have even played with running millions of processes on our desktops. We have demonstrated in this section the power of process isolation and linking and even shown an example demonstrating how this can be used to create complex and powerful behavour. Processes ultimately go from theory to actual running program code within the Erlang virtual machine, more commonly referred to as the Erlang Rumtime System or ERTS. How does ERTS handle all of those processes efficiently, how does it handle IO, scheduling, and garbage collection? We are going to talk a bit about the important features of ERTS VM now.

5.3 Scaling the VM to Large Numbers of Processes

There are many interesting features of the Erlang VM that you just don't know about unless you really go digging or spend a lot of time on the erlang-questions mailing list. These features are really at the core of what enables the VM to handle such a large number of processes at the same time. The pragmatic approach the implementers have taken has allowed them to get an extraordinarily efficient, production oriented and stable system out there and available to us. These features are among the many things that make Erlang unique. They are things that most concurrent systems should seriously consider implementing but so far haven't. In this section we are going to cover three important aspects of the Erlang VM that contribute to its power and effectiveness. These are:

1. The Scheduler
2. The IO Model
3. The Garbage Collector

First up, the scheduler.

5.3.1 The Scheduler

This will be obvious to some of you with experience in concurrent systems and not so obvious to those that don't have that experience. In either case, its worth going over again. The Erlang VM has it's own process scheduler, and with the SMP, many. It does not rely on the OS's process scheduler at all. This makes all Erlang process the equivalent of 'Green Threads'. This is how Java's threads where in the very beginning. In the case of Java, the world clamored for the ability to have true concurrency instead of simple time-slicing. The Java implementers gave the world want it wanted and very quickly moved from Green Threads to Native Threads scheduled by the OS. Erlang has always approached concurrency more from a paradigm prospective then as a simple optimization. What I mean by that is that in Erlang processes are a unit of organization, a major design feature, rather then something you only use when you absolutely must have concurrency. With that in mind time slicing on a single OS process worked for many, many years with little or no complaint from the Erlang community. This continued use of Green Threads has recently allowed them to make a significant and important change to the system though. They added the ability to schedule Erlang threads across multiple OS threads, transparently to the user. Doesn't sound like a big deal does it? Well think about this. The Erlang VM schedules one process for each available CPU and schedules hundreds or thousands of Erlang Process in each OS process. So what happens when you run on a one CPU system and want to take advantage of a four CPU system? In Erlang you just move your project to the new system and you are golden. How about sixteen CPU's, twenty? Its all the same to your Erlang code. That is a feature that no other platform can readily offer at the time of this writing.

There are caveats though.

1. You don't quite get a linear increase in performance with each additional CPU.

2. New Erlang coders who develop on a single CPU system have a tendency to rely an synchronous behavior that is a result of time-slicing in a single process. When they move to a multiple CPU system with real concurrency things may break. So testing is always in order.

All that being said, the ability to relatively easily take advantage of multiple cores is very useful and important and something that is very attainable with Erlang.

5.3.2 IO and the Scheduler

One of the things that so many concurrent languages out there get wrong is that they don't think about IO. With few exceptions they implement a concurrent language and then have the entire system, or a large subset of the system block when any process is doing IO. This annoys us greatly when Erlang has had this problem solved almost from the get go. We talked about Erlang having its own scheduler in the last sections. Well one of the other things

this allows the VM to do is elegantly handle IO. At the lowest levels of the system Erlang does all IO as event based IO. Event based is simply an IO system that allows a program to handle each bit of IO as it comes into the system in a non-blocking manner. It generally provides identifiers for where the IO is coming from and how much data is available and lets the system handle the rest. This is important because it removes the need to setup and tear down connections, it removes the need for OS based locking and context switching. Its basically a much more efficient method of handling IO. Unfortunately, it also seems to be a lot harder for programmers to reason about and understand. That's why we only generally see these types of systems when there is an explicit need for highly available, low latency systems. Dan Kegel wrote about this problem in his 'The C10K Problem' paper back in 2001. It's a bit out of date in some respects now but it is still generally very relevant and worth reading. It should give you a nice overview of the problem and approaches available to solve it. All of these approaches are complex and painful to implement though. That just seems to be the nature of event based IO in the current age. Well everywhere except in Erlang that is. As we said before Erlang does the 'event' part of event based IO for you. It completely integrates the event based IO system into its custom thread scheduler. In effect you get all the benefits of event based IO with none of the hassle. This makes it much easier to build highly available systems when using Erlang/OTP. The third aspect of the ERTS VM is up next, Gargabe Collection.

5.3.3 Process Isolation and the Garbage Collector

As you know at this point, Erlang is a garbage collected system. It uses a fairly straight forward generational garbage collector. It's interesting and innovative but its nowhere near as sophisticated as say Java's garbage collection. Even with this relative simplicity programs in Erlang generally don't suffer from GC stops like systems written in other languages. Why is that? Well it has everything to do with process isolation. Processes in Erlang each have their own small heap that comes into existence when the process is created and goes out of existence when the process dies. That may not sound very important but it is. Think about this, processes only allocate and deallocate memory if they have been scheduled. The Erlang VM, because of its custom scheduler, always knows when a process has been run. This means that it only has to garbage collect the heaps for the process that have been run since the last garbage collection. This alone vastly reduces the amount of memory the garbage collector needs to traverse in any single pass. It almost doesn't matter how much memory the system is currently using Erlang only ever has to traverse a small subset of it.

If it's true that the VM only has to traverse the heap for those processes that have run since the last garbage collection what does that mean for those processes that where created and died between the two runs? This means that they never have to be garbage collected at all! Those heaps lived and died with the processes themselves. All of this together means that the VMs GC is doing much, much less work relative to GCs in other languages/platforms. You reap the benefits of this implementation.

We have just seen that the Erlang VM allows you to schedule many, many processes. It provides for asynchronous IO out of the box and it is implemented with a very efficient garbage collection scheme. We have deepened our knowledge of the platform that allows us to create processes and forge links between them in order to create complex behaviour as shown in our link_ex example. This understanding of the platform is going to give us much greater insight into the runtime behaviour of what we write.

5.5 Summary

So we have spent quite a bit of time going over several interrelated topics. We talked about

1. Approaches to handling concurrency and the attendant issue of ”to share, or not to share state”.
2. The Erlang approach to concurrency using processes.
3. Erlangs uniquely useful notion of process linking.
4. The platform and implementation.

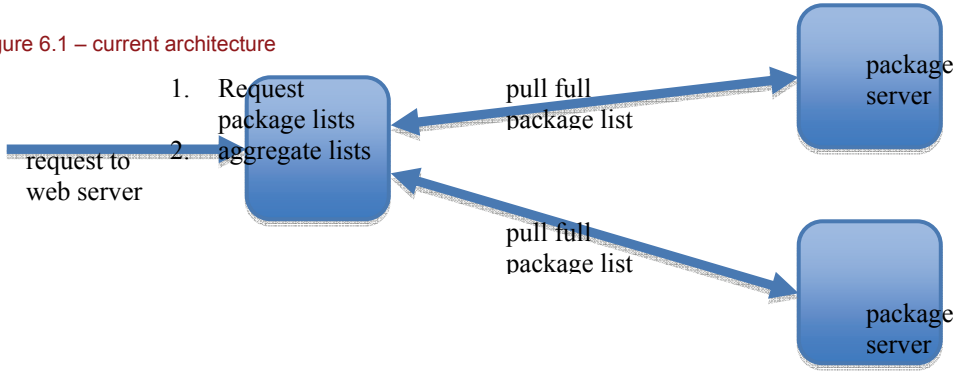
We now have background on the approaches to concurrency. Having a background on the different approaches to concurrency allows us to compare and contrast them with the approach Erlang/OTP has taken. We covered linking and now can make use of it in our code. Knowing about linking gives us the ability to understand of how the facility is used in critical OTP library code like supervisors. We also now have more of an understanding of the VM. Knowing these details about the VM will not directly improve your code, it may, but it certainly can lend an appreciation for the platform and what it can do and more importantly it provides you with a better understanding of the tool that is Erlang/OTP so that you can make informed decisions about when to use it vs other tools you have at your disposal. In the next section of the book we are really going to start digging deep into OTP and how to use it to build production fault tolerant applications. We are actually going to build a distributed, and even partially replicated, cache. No trivial task but the knowledge that we have gained here is going to go a long way towards helping us understand many of the libraries and applications that we are going to be talking about soon.

6

Implementing a Caching System

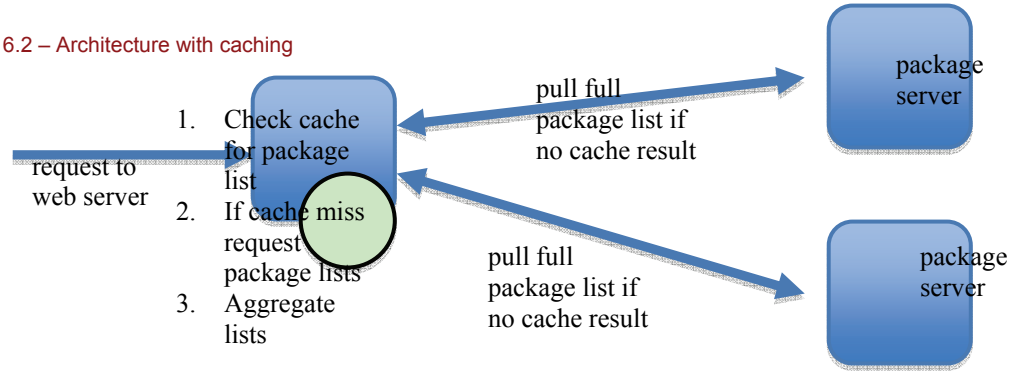
We have talked a lot about some of the interesting and useful features of Erlang and OTP. From a language introduction to an intro into behaviours and a look at the packaging model, we have covered a lot of the things that make Erlang/OTP interesting. Now it's time to start putting that knowledge to use while learning even more. To do that we are going to spend the rest of this book building up a useful, distributed application in Erlang. We are couching this application in a real world narrative that, we think, should help you understand what we are doing and why. This narrative will provide some useful features for a project called Erlware that is facing some fictional, but realistic, challenges that we will solve. Erlware is an open source project focused on making Erlang applications easily accessible so that others may run them as end user applications, or use them as dependencies in their own projects. Erlang is growing quickly so the demand for easy access to open source applications and build tools has grown quickly as well. This expanded interest in Erlang tools has caused quite a bit of growth in the Erlware project. Due to this growth the project administrators are starting to find that users are complaining about the responsiveness of the erlware.org website. Most of the complaints are related to how quickly pages can be served up on the website. The Erlware team is concerned that this is affecting users, not to mention that Google is rumored to punish sites that don't serve up pages lickedy-split. The page on the Erlware website that garnered the most complaints was the package search page. This page allows users to search for a particular piece of Erlang/OTP software among the many that are in the Erlware repository. To serve this page the web server must go out to a disjoint set of package servers and gather a list of packages from each server. It must do this because the package servers are independent and is no central database for all packages across all the different repo type. This was fine when Erlware was small and had one package server, it was even find when it grew a bit and had three, but now with many it is not working too well. You can see a representation of this in Figure 6.1

Figure 6.1 – current architecture



The core team at Erlware sat down and decided that one piece of the plan to speed things up would be augmenting the web servers with local caching. This way when a package listing was performed the list of packages could be cached and keyed off the URL provided so the next time the same package listing URL was referenced the list of packages could be pulled quickly from the cache and used to render the page. This caching architecture is illustrated in Figure 6.2.

6.2 – Architecture with caching



To put this functionality in place we need to get to work on this simple cache so that we can meet our goal of improving the user experience on the Erlware.org website. This chapter is centered on implementing the fundamental functionality our simple cache will offer. This comes down to implementing 4 basic functions:

1. start – start the cache
2. insert – add a key value pair to the cache
3. lookup – pull a value out for a given key
4. delete – delete a key value pair

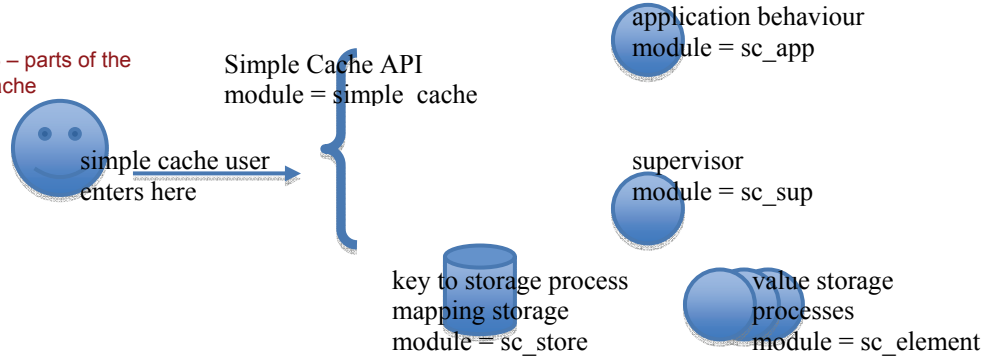
Once these functions are implemented we essentially have a working cache in its most primitive form. The initial version of the cache we construct in this chapter will not leverage features our design and Erlang/OTP makes possible like ready distribution and scalability. This cache will be a standalone cache for the purpose of speeding up single webservers and for use on a single machine easily accessible via Erlang function. More advanced features like distribution will be added in subsequent chapters. Before we get into the nitty-gritty parts of implementing our basic cache it is a good idea to spend some time covering the proposed design.

6.1 – Our Cache Design

Our simple cache will store key value pairs where keys are unique. Each unique key will reference exactly one value. The core principle behind the design of our simple cache is that we will use a single process to store each value we insert into it. We will map the processes that hold these values to the keys that they are associated with. You may consider it somewhat strange, or even unbelievable, that we use a process for each value in the element. For something like a cache this makes sense as values may have lifecycles behaviors all their own. Erlang's support for very large numbers of lightweight processes makes this approach possible.

In building this cache we will need to create some basic sub systems and artifacts. Each of these will be housed in a separate module. Figure 6.3 visually illustrates all of the different pieces of the simple cache and which module will house each.

figure 6.3 – parts of the simple cache

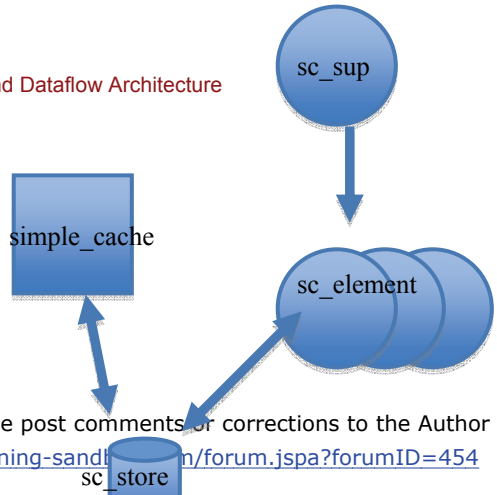


As you can see from figure 6.3 we will be creating 5 modules.

1. simple_cache.erl – The API. Users of the cache will use this to interact.
2. sc_app.erl – The application behaviour file.
3. sc_sup.erl – The main supervisor which will start the element/value storage processes.
4. sc_store.erl – A module to encapsulate all our datastore code.
5. sc_element.erl – Code for the processes that store elements/values in the cache.

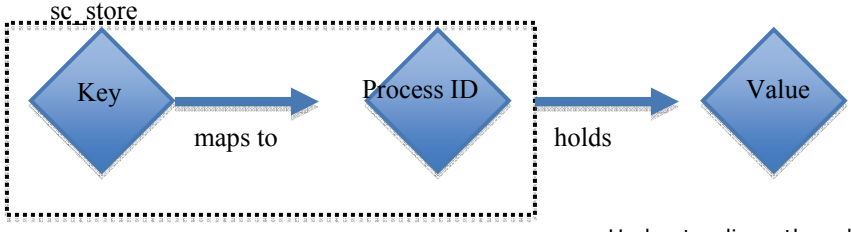
Generally the way all these modules and processes will fit together is that users of the cache will interact with the API. The API will drive data through the rest of the system. When a cache API function is called the API will interact directly with the sc_store in order to determine or create a key to process mapping so that it can insert, replace, lookup or delete data that resides in one of the value store processes (sc_element) that are supervised by the sc_sup supervisor. Figure 6.4 illustrates this architecture from a process and data flow perspective.

Figure 6.4 Process and Dataflow Architecture



Getting into more detail about the workings of the cache, the Application itself is started via sc_app, which starts the sc_sup supervisor. The supervisor is capable of spawning supervised sc_element processes on demand at runtime. The supervisor can do this because it is configured to be of the type simple_one_for_one. Simple one for one supervision will be explained in detail later in section 6.2.3. These sc_element processes are spawned by the supervisor upon the insertion of a key value pair via the API. After an sc_element process is spawned for a particular value its pid is stored along side that values key in the sc_store. This creates an indirect mapping from key to value. When a request for a lookup comes in via the simple_cache API the supplied key is used to lookup the pid of the particular sc_element process containing the value that is being requested. Figure 6.5 illustrates the indirect key to value mapping.

Figure 6.5 – Key to Value relationship



Understanding the design of the cache is half the fun, the other half is coding it up. We start that process in the next section with creating the Application infrastructure for the simple cache.

6.2 – Defining Basic OTP Application Code and Directory Structure

We have an important part to play our part in making the Erlware.org site faster. We have decided to build a cache and we have a pretty interesting initial design for it. The starting point for any good Erlang project is Application creation. Many times a project breaks down into more than a single Application. For now our simple cache consists of only one. In creating the Application framework for an Active Application we will create the following things:

1. Proper Application directory structure on the file system
2. Create our application behaviour implementation module; sc_app
3. Create our top level supervisor implementation; sc_sup
4. Create our *.app file

On to step number one, creating our directory structure.

6.2.1 – Defining Our Application directory structure

We start by creating a directory called simple_cache; this houses our Application. From there we create two more directories, the first being the src directory which will hold all our our *.erl source files. Next we create the ebin directory which will house our *.app file and our compiled Erlang code in the form of *.beam files. The directory tree should look as follows:

```
simple_cache
    | - src
    | - ebin
```

As you may recall from chapter 4 table 4.1 there are number of other directories that applications can contain. We will add in such directories as "include" or "priv" on an as needed basis. ebin and src are clearly the two that are indispensible and for now all that we need.

Remember, that there are two types of Applications in Erlang Active and Library. Active Applications have a lifecycle and Library applications do not. We are creating an Active Application so we will need to create an application behaviour implementation and a supervisor implmentation. Employing the canonical naming scheme we have sc_app and sc_sup respectively. Next we flesh out the Application behaviour implementation.

6.2.2 – Defining sc_app

sc_app.erl will house the module that implements the application behaviour. Remember that it is the start/2 function specified by the application behaviour and referred to by the mod tuple in our .app file that starts an Active Application. Below in listing 6.1 is the code contained in sc_app.erl in its entirety.

Listing 6.1 - sc_app.erl

```
-module(sc_app).

-behaviour(application).

-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
   case sc_sup:start_link() of      #1
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.

stop(_State) ->
    ok.
```

First we define our module name, indicate that this is an application behaviour, export our two major callback functions, and finally define the functions themselves. We don't actually do anything in 'stop' in this module as we do not have any shutdown semantics.

The start/2 callback function, as part of the application behavior, takes two arguments. Both of these arguments prefaced with the _ in this case indicating that we are not going to be making use of them. A more detailed treatment of the arguments to the start callback can be found in the official Erlang/OTP documentation. It is with this start function that our application will begin its life. The most important part about this function is at code annotation #1 reading sc_sup:start_link(), this is where we start our top level supervisor. If that startup is a success, we get the supervisor process id back from the call to start our sc_sup, then our application startup is considered complete and we know we have our full supervision structure in place. This leads us nicely into a discussion of what our supervision structure for the Simple Cache looks like.

6.2.2 – Defining the Supervisor and Simple One For One Supervision

sc_app.erl houses the module that starts our application, that is done primarily by firing up the top level supervisor which spawns the process tree that does the real work in the simple cache application. It is the module sc_sup.erl that will house the module that contains the supervisor behaviour implementation. This supervisor is a bit different from the supervisor that we created in chapter 4. This supervisor is set up for simple_one_for_one supervision. With the basic supervision structures a supervisor manages some number of children, these children can be the same of different, but they all start when the supervisor starts and are often times long running. The simple_on_for_one supervisor is different. A simple_one_for_one supervisor can only start one type of child but it can do it dynamically at runtime. Only a single child spec is returned to the supervisor container from the init/1 callback at startup. When asked the simple_one_for_one supervisor can start that type of child on an as needed basis. Our sc_sup supervisor can start sc_element processes at runtime when required to do so. Below in listing 6.2 you can see the code for how this is accomplished.

Listing 6.2 - sc_sup.erl

```
-module(sc_sup).

-behaviour(supervisor).

-export([
         start_link/0,
         start_child/2
        ]).

-export([init/1]).

 -define(SERVER, ?MODULE).    #1
```

```
start_link() -> #2
    supervisor:start_link({local, ?SERVER}, ?MODULE, []).

start_child(Key, Value) -> #3
    supervisor:start_child(?SERVER, [Key, Value]).

init([]) ->
    RestartStrategy = simple_one_for_one,
    MaxRestarts = 0,
    MaxSecondsBetweenRestarts = 1,

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

    Restart = temporary,
    Shutdown = brutal_kill,
    Type = worker,

    AChild = {sc_element, {sc_element, start_link, []},
              Restart, Shutdown, Type, [sc_element]},

    {ok, {SupFlags, [AChild]}}.
```

The header code will be self-explanatory at this point and so we are not going to cover it too deeply. The one part I will comment on is the use of the –define macro definition (annotation #1) declaration. Our use of this in this module is to alias the built in macro MODULE with our macro SERVER. We do this for clarities sake. This supervisor starts with a registered name and instead of referring to that registered name as ?MODULE it is more intentional to refer to it as ?SERVER.

We have just two API functions in the sc_sup module, the first being start_link/0 (annotation #2). start_link/0 starts the supervisor container registering it under the name in ?SERVER and indicating that the callback module (the implementation) associated with it is ?MODULE which of course is sc_sup itself. When supervisor:start_link is executed by our start_link/0 function the supervisor container then invokes sc_sup:init(annotation #3) callback function. This function as you can see differs from the init/1 function we wrote for our last supervisor behaviour implementation in chapter 4. The reason for this is the simple_one_for_one supervision. Notice the first two lines:

```
    RestartStrategy = simple_one_for_one,
    MaxRestarts = 0,
```

The restart strategy is set to simple_one_for_one and then that is followed up by a MaxRestarts of 0. Why is this? This is because in our supervisor all the children are temporary, meaning that if they die, well, that is that, they are dead. Our supervisor will not restart them. This supervisor is in many ways just a factory for sc_element servers. The three lines in the next code snippet specify that each child started by sc_sup is going to be a temporary child, shutdown brutally, and of type worker.

```
Restart = temporary,
Shutdown = brutal_kill,
Type = worker,
```

Temporary means that if the child dies it is not to be restarted under any circumstance. Shutdown is brutal kill, which means that if this supervisor shuts down it does not wait for it's children to shutdown cleanly but instead just kills them forcefully. Finally we have type specification indicating worker which indicates that all the children of this supervisor are worker processes as opposed to other supervisors.

The next two lines, the last in the init function, bring everything together and fully specify the child type this supervisor is capable of producing dynamically.

```
AChild = {sc_element, {sc_element, start_link, []},
          Restart, Shutdown, Type, [sc_element]},

{ok, {SupFlags, [AChild]}}.
```

AChild is a single child specification for the sc_element process. It is exactly the same as the child specs we have seen earlier in the text with the one difference being not in the structure of the child specification but in that the supervisor container will not start it automatically but will instead wait for an explicit request to start the child. The OTP supervisor container contains the function supervisor:start_child/2. This function as it is invoked from our sc_sup:start_child/2 API function is below.

```
supervisor:start_child(?SERVER, [Key, Value])
```

We wrap this function in our own API start_child/2 function for the same reason as we wrap other protocols and implementations. The fact that this child is started in the manner it is, with the argument order it implies, with the registered supervisor container process ?SERVER and so on is the business of the sc_sup module and should not leak out of it. This is another example of basic Erlang/OTP encapsulation. A call to this start_child/2 function results in a message being sent to the supervisor identified by the macro ?SERVER which we have defined in our header. This call results in the arguments Key and Value being sent to the start_link function of sc_element. This connection can be seen in the child spec above, we have the tuple

```
{sc_element, start_link, []}
```

which indicates that we will start the child of this supervisor by calling sc_element:start_link appending [Key, Value] onto the default argument list supplied in the child spec. The following snippets show the relationship between supervisor:start_child and the manner in which it will cause an actual child to be started.

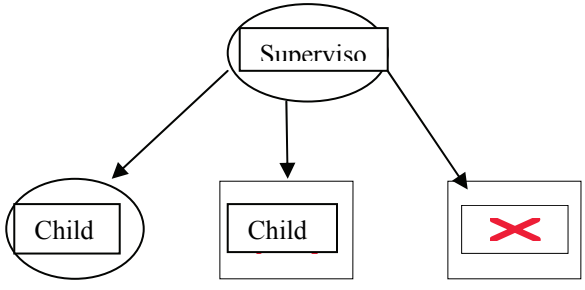
```
supervisor:start_child(?SERVER, [Key, Value])
```

yields in sc_sup

```
sc_element:start_link([] ++ [Key, Value]).
```

Each time we call sc_sup:start_child/2 which in turn calls supervisor:start_child/2 a new sc_element server will be started. This process of calling start child is what yields a runtime generated supervisor tree consisting of many sc_element processes(remember those in chapter 4?). You can see a visualization of that in the Figure 6.6.

Figure 6.6 Supervisor Hierarchy



At this point we have an application behaviour and a supervisor behaviour all set up. The very last thing we need to do now is create our *.app file which will contain meta data about what we just created.

6.2.3 – Creating the Application Meta Data *.app File.

OTP needs a little metadata about your Applications to be able to do things like start the Application or hot swap its code at runtime. One of the things it needs to know is all of the modules that are included in the application. So before we move on to coding the meat of our application there is one little thing we can't forget to do, we need to create a .app file which resides in the ebin directory of our application as "simple_cache.app". Below is what your app file should look like thus far.

```
{application, simple_cache,
 [{description, "A simple caching system"},
  {vsn, "0.1.0"},
  {modules, [
             sc_app,
             sc_sup
            ]},
  {registered,[sc_sup]},
  {applications, [kernel, sasl, stdlib]},
  {mod, {sc_app,[]}},
```

```
    {start_phases, []}]}.
```

This app file is another example in the same vein as the app files we described in chapter 4. If you would like to get a refresher on those topics feel free to head back to chapter 4 and reread section 4.1.1 on applications and *.app files.

With the addition of the *.app file we now have a working application, we can actually go and start this and see it run within the Erlang shell. Of course once you get it running you can't do anything with it because our application has no real functionality and furthermore no interface exposing what functionality it may have. We have reached a milestone though; we have a basic OTP application skeleton ready for us to fill out. The next step is adding functionality to this application.

To sum things up a bit Every time you write an Active Erlang Application you will start with these steps to create a basic OTP application skeleton.

1. Create the Application directory structure
2. Create the application behaviour implementation module; <prefix>_app
3. Create the top level supervisor implementation module; <prefix>_sup
4. Create the *.app file

As of right now we run sc_app:start/2 it will in turn start the supervisor we have defined in sc_sup. When the supervisor starts it will not start any child processes because it is simple_one_for_one. It is configured to start sc_element processes which will store our cache values. As of now we have not implemented the sc_element module. We will devote a large part of the rest of this chapter to defining that module/process and all the other code needed to support a fully functioning simple_cache. We start on the next section

6.3 – Going from the Application Skeleton to a Working Cache

Our simple_cache API module defines what our system is capable of to the users of the simple_cache; it is the front door to our application. The API will define the following functions:

1. insert/2 – place a key and a value into the cache
2. lookup/1 – using a key retrieve a value
3. delete/1 - using a key delete a key value pair

By the end of this chapter we will have all of these functions fully implemented. We will knock out each of the modules required to fully implement the cache one by one. Take a

look back to figure 6.3 for a visual of what the working parts and modules of the cache are. We list them here again for your convenience:

1. simple_cache.erl – contains the API
2. sc_app.erl – contains the application behaviour implementation
3. sc_sup.erl – contains the supervisor behaviour implementation
4. sc_element.erl – contains the code for the process that stores cached values
5. sc_store.erl – contains the code used to interact with the key to process id storage

We have implemented the sc_app and sc_sup modules already as part of our Active Application framework in the first part of this chapter. We will now move on to implementing the sc_element module so that our top level supervisor will have something to start.

6.3.1 – Coding sc_element

First things first we need to create the sc_element.erl file which will contain our sc_element module. sc_element is the process that will be spawned each time new data is entered into the cache. sc_element will hold the values that are associated with each key we enter and it will do so within gen_server state; which means that the sc_element module will be a gen_server implementation. If you need any refresher on what a gen_server is and how it functions take a quick scan of chapter 3 to refresh your memory. We define our gen_server implementations following the same file layout template every time and so you should recognize it's parts very easily, we will go through them in order.

1. module head
2. API
3. callback functions and associated internal functions

The module head below in listing 6.5 should be very familiar, the names have been changed but the roles remain the same.

listing 6.5 – sc_element module for storing values

```
-module(sc_element).      #1

-behaviour(gen_server).

-export([                 #2
        start_link/1,
        start_link/2,
        create/1,
        fetch/1,
        replace/2,
        delete/1
```

```
    ]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
        terminate/2, code_change/3]).       #3

-define(SERVER, ?MODULE).          #4
-define(DEFAULT_LEASE_TIME, 60 * 60 * 24). % 1 day default (in seconds)

-record(state, {value, lease_time, start_time}).     #5
```

First we define the module at annotation number one. Next we indicate that it is a gen_server bahaviour implementation. At annotation number 2 export our API functions followed by the export our callback functions at annotation #3. We alias ?MODULE to be SERVER for intentionality and readability at annotation #4 the next thing we do is create the DEFAULT_LEASE_TIME macro. This determines the longest time that a key value pair can live in the cache without being evicted. The default time is a single day. The lease time is measured in seconds. Essentially the lease time says then that after a single day any value inserted into the cache will be evicted. This is so that the cache stays fresh; after all it is a cache, not a database. This value can be overridden in the API when creating a new sc_element process. The last thing done in our header is to define a record to represent our gen_server state structure at annotation #5. If any part of that is unfamiliar to you please don't hesitate to refer back to chapter three for the full explanation. The API follows the header and so next we will get into our first API function; replace/2/1.

SC_ELEMENT:CREATE/2/1

Each time a value is entered into the cache we store that value in an sc_element process. That means that each time a new key value pair is inserted into the cache we have to start an sc_element process which leads us to implement the first of our sc_element API functions; sc_element:create/2/1. When we want to create a new element storage space and store an element we don't call start_link but instead create/1 or create 2. The create/2/1 functions delegates process startup to the sc_sup:start_child/2 function as you can see in the next snippet.

```
create(Value, LeaseTime) ->
    sc_sup:start_child(Value, LeaseTime).

create(Value) ->
    create(Value, ?DEFAULT_LEASE_TIME).
```

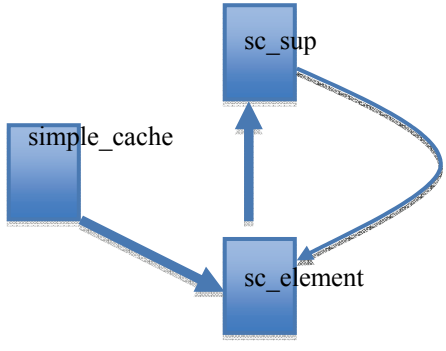
Hiding how sc_element storage creation is actually accomplished is another example of basic Erlang encapsulation. The fact that an sc_element is a process and that relies on the sc_sup supervisor for process startup is an implementation detail that need not leak out of the sc_element module. If later on down the line we want to make this whole thing database

driven our interface need not change. sc_sup:start_child/2 is the function used to tell our simple_one_for_one supervisor that it needs to dynamically start a new child by calling the more standard start_link/2 API function from sc_element which is shown in the next snippet.

```
start_link(Value, LeaseTime) ->
    gen_server:start_link(?MODULE, [Value, LeaseTime], []).
```

To make sure this is clear let's go over it once again. When a new element is to be inserted into the system the storage gets created and the element stored by calling into the sc_element:create/2/1 functions. This function calls the sc_sup:start_child/2 function which in turn calls the standard start_link/2 function on the sc_element module. We could just call directly to start_child when a new elment needs to be inserted into the cache but that is less intuitive. Why would a call be made into the supervisor to store an element in the cache? The sc_element module is responsible for managing element storage and therefore the function to create that storage and store the element lies there. The call flow is again illustrated in figure 6.6.

Figure 6.6 – Insertion of data



The next thing that happens after sc_sup calls sc_element:start_link/2 is that the gen_server callback function init/1 is called to initialize the sc_element gen server. The init/1 function will be called and return a fully initialized state record to the gen_server container prior to the start_link function unblocking and returning the pid of our newly created process for usage.

```
init([Value, LeaseTime]) ->
    StartTime =
        calendar:datetime_to_gregorian_seconds(calendar:local_time()),    #1
    {ok,
     #state{value = Value, lease_time = LeaseTime, start_time = StartTime},
     time_left(StartTime, LeaseTime)}.
```

At annotation number 1 we capture the time at which this sc_element process is starting in seconds. We do this with functions located in calendar module located in the stdlib. Once we have this accomplished we return from init setting our state up appropriately. Init initializes the #state record with the Value we care to store and the LeaseTime that was set upon calling create as well as with the StartTime we just captured. Notice that it also make suse of the third position in the return tuple. This as you know from earlier experience is the timeout position. This is how we manage lease times. We just use the gen_server timeout. Our architecture of using a process per every value stored makes lease time management trivial. In most cache implementations you would need something to periodically scrape the storage system for any elements that have not been accessed for the period specified by the lease time. In our system we can abstract that away nicely using the gen_server timeout. Our method for actually arriving at the amount of time a Value has left to leave is fairly straightforward and contained in the next code snippet.

```
time_left(_StartTime, infinity) ->      #1
    infinity;
time_left(StartTime, LeaseTime) ->
    CurrentTime =
       calendar:datetime_to_gregorian_seconds(calendar:local_time()),
    TimeElapsed = CurrentTime - StartTime,
    case LeaseTime - TimeElapsed of
        Time when Time =< 0 -> 0;
        Time                -> Time * 1000
    end.
```

Code annotation number 1 handles the inifinity case. If the lease time is set to the atom 'infinity' then we simply return that which ensure that our value lives forever. Atoms have the convenient property of being greater than any number when you compare them with comparison operators like > or <. The next clause in the function captures the current time in the same way we did in init/1. The StartTime is subtracted from the CurrentTime to tell us how much time has elapsed. If the time elapsed is equal to or greater than the lease time then we return a timeout value of 0 which will effectively kill our sc_element process by sending it into the timeout clause of handle_info where we have the death blow waiting. If the time that has elapsed is less than the lease time we then return the time remaining.

This finishes off the 'create' and 'start_link' functionality. Next we will start looking at the sc_element:replace/2 function makes sense because it is also dealing with setting values in the cache just as create was.

SC_ELEMENT:REPLACE/2

The sc_element:replace function is an API function that we expose to our callers to replace an already existing value with another. It wraps a gen_server messaging function. With that in mind there are two steps to creating it. The first is the function that clients will

call and the second is the gen_server infrastructure that will handle that call. Let's take a look at the client API.

```
replace(Pid, Value) ->
    gen_server:cast(Pid, {replace, Value}).
```

This API function sends an asynchronous cast message to the Pid we pass. That Pid is, of course, the one we pulled from the sc_store which was inserted at the end of our init/1 function after calling sc_element:start_link. Notice that the message we are sending {replace, Value} is tagged with the same name as the function that sends the message. This simple idiom will save you lots of confusion later when you're the number of messages your process accepts grows greatly. This message is sent from the caller of the replace API function and handled in the gen_server sc_element itself via the handle_cast function, seficially the replace clause of the handle_cast function. All gen_server:cast messages get sent to ?MODULE:handle_cast, simple eh.

```
handle_cast({replace, Value}, State) ->
   #state{lease_time = LeaseTime,
       start_time = StartTime} = State,      #1
     TimeLeft = time_left(StartTime, LeaseTime),
     {noreply, State#state{value = Value}, TimeLeft};
```

handle_cast receives the new value and replaces the old value contained in state with the new. This takes place at the last line of the function. Present in this return tuple is the #state tuple which has been updated to reflect the new value, effectively replacing it. Also included in our return value is the TimeLeft as calculated by our time_left/2 function. This resets the gen_server timer and effectively resets our lease time. You may also notice the syntax at code annotation 1. This only pulls values out of the State record. This is done on the second line of the function so as to save space and not wrap the top line. We can move right along now and add an API function that allows the user of the sc_element module to actually fetch the values stored by create and replace.

SC_ELEMENT:FETCH/1

We don't suppose that what fetch/1 does needs too much general explanation.

```
fetch(Pid) ->
    gen_server:call(Pid, fetch).
```

Fetch takes the pid of an sc_element server and uses it to send the message fetch to that server. The server then returns the value that is contained within it. Notice again that fetch, the message sent, is the same as the function name. For this call return, synchronous action gen_server:call must be used. All calls are handled with a handle_call callback function written into the gen_server callback module, which of course is sc_element.

```
handle_call(fetch, _From,  State) ->
    #state{value = Value,
           lease_time = LeaseTime,
           start_time = StartTime} = State,
    TimeLeft = time_left(StartTime, LeaseTime),
   {reply, {ok, Value}, State, TimeLeft}.
```

Another nice and simple callback function; we pull Value from state and simply return it. The atom reply signifies that we are returning a value. There are other possible return values that the gen_server will understand however, we wont go into them here. It is worth reading gen_server documentation at erlang.org to get a full sense of what all is available to you. The next part of the return tuple, {ok, Value}, is exactly what will be returned by gen_server:call. Next the gen_server state is returned to the container and finally the timeout value is reset to the time that is left.

Setting Timeouts

Remember that if we ever forget to set the timeout value in a handle function the timeout will revert to infinity by default. When you have a timeout it is very important to be mindful of setting them in each and every access function.

That was quick, fetch is done which leaves us free to move on to delete.

SC_ELEMENT:DELETE/1

Delete takes a key and removes all the state associated with that key, deletes the key and its value essentially. Starting as usual with the API function:

```
delete(Pid) ->
    gen_server:cast(Pid, delete).
```

Notice that we use an asynchronous cast here, we do not expect nor care about a response from the delete function. The delete message sent from the delete function is caught by the handle_cast callback as always.

```
handle_cast(delete, State) ->
    {stop, normal, State}.
```

Here notice the difference between the return of delete and that of replace. The replace handle cast returned noreply where as this returns stop. Stop will cause the gen_server to shut itself down. The reason for the stoppage is included next and it is 'normal'. This is a special value that tells the system that the server shutdown normally. If this was configured to be transient process for the supervisor it would let it shutdown and not try to restart it

whereas any other return value here could be considered an abnormal close and the supervisor would try to restart it. Again, that only applies in the case that this process were transient. As it is in our supervisor we set this process up as temporary and so the supervisor will never try to restart it no matter what it may return. Why use normal then? The other reason is that SASL will automatically log the shutdown of any behaviour that shuts down with anything but a normal reason. So we are done right? No! There is one more callback that is called prior to shutting down a gen_server and that is ?MODULE:terminate/2. Terminate is called as a kind of catch all cleanup function. In this case we are cleaning up all traces of the value associated with with process by terminating the process. The process going away removes the server state which takes care of the value but does not handle the key to pid mapping. The terminate function will take care of the Key and Pid mapping we stored when we created this process. The terminate function is shown in the next snippet.

```
terminate(_Reason, _State) ->
    sc_store:delete(self()),
    ok.
```

The terminate function takes the reason for the shutdown so that it is possible to implement conditional logic depending on what caused the shutdown and it takes State so logic for termination can be based on the current state of the process. In this case neither are used, hence the _ preceding each of the variables. The only thing we do here is remove the mapping that exists in our key and pid store which we are calling sc_store. To accomplish this cleaning of the store we use the function sc_store:delete/1 which takes the process id of the sc_element server itself. This is a nice segue as well. We have just finished implementing sc_element and its API functions:

1. create/2/1
2. start_link/2
3. replace/2
4. fetch/1
5. delete/1

Together these functions allow us to store data in an sc_element process, replace it, fetch it, and finally delete it. In doing the delete we see our first interaction with the key to pid mapping storage system in our cache which is what we will be implementing next.

6.3.2 – Implementing the sc_store Module

We are now at the point where we implement the storage system for our key to pid mappings. These mappings allow us to take a key and map it to the sc_element process that is holding the value associated with it. To implement this store we are going to use a very useful Erlang storage system called ETS. We will be spending time not only directly

implementing the sc_store module and its API but also in explaining this very powerful storage system called ETS.

The fact that our storage is implemented with ETS however, as you may be able to guess, should not be known to the whole cache system. The sc_store module serves as an abstraction for whatever storage mechanism we are using to store the key to PID mapping data. Instead of using ETS this storage mechanism could have been a gen_server process that keeps this mapping data in it's state, it could have been a flat file that we write out to on each insert and delete, or it could have been a relational DB. Whatever the implementation is abstracting it away serves many functions, among them being that it decouples your application from the choice of storage system allowing you to change gears later on and use something else. We could even do something like move from a single machine store implemented with ETS to perhaps even a replicated solution with no change to application code…

For right now we are going to use the very powerful ETS system as our storage mechanism which means that we need to gain a little bit of an understanding of what it is as we implement our storage system with it. For starters ETS is Erlang Term/Tuple Storage, a very fast, in-memory storage space for Erlang data. ETS is as fast as it is because it is implemented as a set of built in functions, meaning it is a part of the Erlang Runtime System (ERTS), and implemented directly in the VM in C code. This on top of the fact that it runs entirely in memory and was implemented efficiently by smart people makes it a nice choice for data that:

1. Needs to be persistent as long as the VM is alive.
2. Single VM; non replicated data.
3. Data which needs to be fetched in low millisecond time ranges.
4. Needs to be accessed by a number of different processes.
5. Flat data without foreign key relationships.

Our storage requirements match up well with these criteria for ETS usage. Lets go through thise one by one. First, key to pid mapping data needs to persist across process starts and restarts for as long as the cache is running, because as long as the cache is running new data can be inserted and old deleted, so we need a store that will be around for the duration of the caches' life. Second, the data does not indeed need to be replicated; our cache is standalone. Third, the data needs to be fetched quickly so as not to slow down a critical lookup operation of our cache. Fourth, this mapping data needs to be accessed by different processes, such as each sc_element process. Finally, the data is flat and has no foreign key relationships, it is just a single table of key to process ID mappings.

We are going to start in a familiar place, with the module header, which is detailed in the next snippet. We have 4 exports that handle the basic CRUD (Create Read Update Delete) operations. Insert handled create and update, lookup is

responsible for read, and delete… The first of the exported functions has nothing to do with CRUD. init/1 is responsible for initializing our storage system.

```
-module(sc_store).

-export([
         init/0,
         insert/2,
         delete/1,
         lookup/1
        ]).

-define(TABLE_ID, ?MODULE).
```

In the next section we will dig into the init function and how exactly the sc_store is initialized.

SC_STORE:INIT/0, INITIALIZING THE STORE

The initilization we need to do here, given our choice of using ETS for our sc_store, is to create the actual ETS table that will store key to pid mappings. As you can see below all we do is call ets:new/2 and then return from our init/0 function.

```
init() ->
    ets:new(?TABLE_ID, [public, named_table]),
    ok.
```

The first thing we are going to note here is a point of style; this function should be placed first in our API function section ahead of all the others because it will be called before the others, it is a startup function. Similarly in other modules functions like start_link or start typically appear at the top of the list of API functions, this is done for ease of reading; first things first.

The sc_store:init/0 function makes use of the ets:new/2 function in the ETS module that is a part of the stdlib library application. The first argument is the ?TABLE_ID, a macro that is defined in the sc_store module header. ETS requires that tables be given IDs but these are not always used after table creation. There are two ways that an ETS table can be accessed. The first and most common is using the table handle that the ets:new/2 function returns. This table handle uniquely identifies a table in kind of the same way a process ID identifies a process. When accessing tables in this manner the table id need not ever be referenced. The other way is to register a name for the table. This works much the same way as registering a name for a process. Once registered a process can be accessed by the well known name without needing the process id. Once the table id is registered for a table the table can be accessed by that name and the table handle is not needed. The reason we opt to register a name for table here is that we don't want to require the user of our library to hang onto the table handle. This would require us to pass the table handle on to all processes that care to use sc_store. It would also require that the table handle to be passed into every sc_store API

call. Passing the table handle in through the sc_store API would make our storage system just a little less generic. The second argument to the ets:new/2 function is an options list. It is by passing the option 'named_table' in through this list that we register ?TABLE_ID as the tables name. We also pass in the option 'public'. This option allows our table to be accessed by any process, not just the one that created it, which is important if all sc_element processes among others are going to use the sc_store.

Here is a question for you; from where should we call the sc_store:init/0 function? We have essentially two options. One is in our application behaviour module, the other is in our top level supervisor. As has been explained earlier it is a design principle to limit application code in supervisors so as to keep them reliable. There is no need for a supervisor of the top level supervisor because supervisors are so simple and well tested that they can be considered an invariant. Putting your own code into it negates this property. Inserting code into the init/1 function of a top level supervisor is somewhat forgivable because if it breaks the application simply can't start. We still don't much like that practice on principle and, while we may do it on occasion, we tend to prefer placing this sort of initialization code in the top application behaviour file. You can see the placement of the sc_store:init/1 function in the next snippet.

```
start(_StartType, _StartArgs) ->
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.
```

With this code in place our sc_store will be initialized first thing when the simple_cache application starts up. If we were to initialize later, say after the call to start the top level supervisor, we would run the risk that the simple_cache would look to call into an ETS table that did not yet exist. What's next? How about insert?

SC_STORE:INSERT/2

To be able to resolve key to pid mappings we need to be able to store the the mapping first. The sc_store:insert/2 function does this. The code is in the next snippet.

```
insert(Key, Pid) ->
    ets:insert(?TABLE_ID, {Key, Pid}).
```

Notice that this function does not check the data types, it does not seek to ensure that what is being inserted is a pid. This is because this is internal code, it is trusted code, nothing other than a pid should be inserted here because the code that will eventually do the insert will be code we write. We will check for sanity at the borders of our system but not afterwards. If we have a problem in the end we will discover that through testing. Checking only the borders and not internal code like sc_store keeps our code clean and makes it easier

to maintain. The actual inserting of the data into the ets table is accomplished in this case by ets:insert/2. The function insert in the ETS module takes two arguments, the first is the table id for the table, this can be a table handle or or table name if the table has registered a name. The second argument is a tuple and this must be a tuple, because ETS only takes tuples. Perhaps ETS should be referred to as Erlang Tuple Storage. Anyhow, we insert a tuple containing first the key, and then the pid it maps to. The first element in an ETS record tuple is by default the key, the rest the body. So there can only be one record in a standard ETS table that contains a specific key. So if our key is 'jello' then there can be only one record containing the key 'jello' and upon inserting another record where the first element was 'jello' we would have overwritten the first. Next we will implement the code that allows to retrieve the information that insert/2 allows us to store.

SC_STORE:LOOKUP/1

A lookup on a key in ETS is a constant time operation, i.e very fast. If you want to look up something by matching the body of a tuple stored in ETS the full table must be scanned and so speed of the lookup is dependent on the size of the table. In our case we are going to do our looking up by the key. The code for this is shown in the next snippet.

```
lookup(Key) ->
    case ets:lookup(?TABLE_ID, Key) of
        [{Key, Pid}] -> {ok, Pid};
        []           -> {error, not_found}
    end.
```

We will be using this lookup function to find a process id for an sc_element process that is storing the value associated with key we pass in to it. The ets:lookup/2 function, provided there is a valid table associated with ?TABLE_ID, will return either a list of elements matching our lookup criteria or an empty list if it finds nothing. Since we are doing a lookup on a key it can either find one element or it can find none.

OTHER TYPES OF ETS TABLE SEMANTICS

You may be wondering why the lookup function returns a list of tuples if there can only be a single element in a table with a given key. The reason is that this property is the default for an ETS table but it is quite possible to set up an erlang table such that you can have more than one record with the same key. These table types are known as a "bag" or a "duplicate_bag". When a new table is created with ets:new/2 bag semantics can be set by passing in either the atom bag or duplicate bag. Further documentation on ETS tables can be found at erlang.org.

If a pid is found for the key we transform the ETS return value into a more consumable {ok, Pid}. Altering the return values is another example of encapsulation. There is no reason to return the unwieldy ETS return value when the fact that our implementation uses ets is

quite incidental and unimportant to the actual client of this function. If we find no match for our key in the table we return {error, not_found}. With lookup/1 now complete we can move on to our final function, delete/1.

SC_STORE:DELETE/1

Remember how we stated that a lookup operation on a key within ETS is a fast constant time operation and that a lookup on the body of an element in an ETS table is slower? This property applies not just to lookups but to any operation on body data within ETS. Our delete operation will be just such an operation. It deletes by the pid part of the key pid mapping. The reason this is a more time consuming operation is that body elements can be duplicated in a table. This means that ETS must scan the whole table before returning anything so as to make sure that it will return a complete result. The next snippet contains the code for delete/1.

```
delete(Pid) ->
    ets:match_delete(?TABLE_ID, {'_', Pid}).
```

The delete operation will generally not be performed nearly as often as the insert operation and therefore this implementation should serve well enough. We use the ets:match_delete function here to accomplish our body matching delete operation. We will only cover a little bit about matching here. A full explanation of matching functions in ets is beyond the scope of this book and for more information we recommend you read the standard Erlang/OTP documentation online. The match pattern is the second argument oo the ets:match_delete function. These patterns can consist of three things:

1. Erlang terms and bound variables to match against
2. '_' the match anything symbol (as an atom). Means the same as _ in a regular pattern match.
3. Pattern variables '$<integer>' ('$1', '$2', '$3' …)

ETS patterns work much like the standard patterns you have become so used to. Given a stored tuple of the form {erlang, number, 1} a pattern like {erlang, '_', 1} will match against it. The '_' atom indicates that we don't care what is in this position. This pattern asks that any tuple with the atom erlang in first position and the number on in last be returned. We can also get fancy with data retrieval type functions on an ETS table by using pattern variables. When doing a using a match to retrieve a value from ETS pattern variables can be used to select out only certain parts of a matched tuple. A match pattern like {'$2', '$1', 1} would return [number, erlang] because basically {erlang = '$2', number = '$1', 1} and pattern variables always return in order, '$1', '$2' etc…

Our delete operation uses the simple pattern {'_', Pid} pattern. This will match against any tuple that contains, in second position, the process id contained in the bound variable Pid. Because this pattern is supplied to the ets:match_delete/2 function the match will result

in a delete. This operation will only ever match one tuple at a time in our ETS table because it is not possible to associate more than one key with a particular process id. The completion of delete means that sc_store is itself complete. We have covered the initialization of our storage and all required CRUD operations on our storage system. We have also effectively abstracted away our storage implmentation from the rest of our cache application. Given that our application framework has been setup complete with application behaviour implementation, supervisor implementation, and *.app file and we now have both sc_element and sc_store completed there is only one thing left to do. Create the overall API for the simple cache system.

6.3.3 – Pulling everything together with the simple_cache API module

The convention we use for application level APIs is to name them the same thing as the application itself. In this case then we are creating the simple_cache.erl to house our simple_cache module. This module will contain the three functions used by those who wish to use the Simple Cache:

1. insert/2 – place a key and a value into the cache
2. lookup/1 – using a key retrieve a value
3. delete/1 - using a key delete a key value pair

These functions will make use of all the functionality we have created in this chapter. We will get started with insert/2.

SIMPLE_CACHE:INSERT/2

simple_cache:insert/2 takes a key and a value and inserts them into the cache. The code is contained in the next snippet.

```
insert(Key, Value) ->
   case sc_store:lookup(Key) of        #1
        {ok, Pid} ->
            sc_element:replace(Pid, Value);
        {error, _Reason} ->
            {ok, Pid} = sc_element:create(Value),
            sc_store:insert(Key, Pid)
    end.
```

At code annotation number one the sc_store:lookup/1 function is used to determine if we already have an entry for the key supplied to insert/2. If so then the sc_element:replace/2 function is used to replace the value with the one supplied. If there is no entry for this key then a new sc_element is created and a key to pid mapping inserted into the sc_store. Notice that we are not exposing the lease time functionality in our API. We are relying on the default lease time of one day for now. If ever we wanted the ability to change that from the API it would be as simple as just exposing it. Moving on to lookup.

SIMPLE_CACHE:LOOKUP/1

Lookup is very straight forward basically using sc_store:lookup to resolve a key to pid mapping for the supplied key. If the key indeed maps to a pid then the sc_element process is queried with that pid and the value returned. In any other case, thanks to the use of exception handling, a simple error is returned. These other cases can be the obvious one such as no mapping being present but could also be the very subtle case when a pid is returned and before the sc_element process is queried the process referenced by the pid is killed or dies.

```
lookup(Key) ->
    try
        {ok, Pid} = sc_store:lookup(Key),
        {ok, Value} = sc_element:fetch(Pid),
        {ok, Value}
    catch
        _Class:_Exception ->
            {error, not_found}
    end.
```

Moving on to delete.

SIMPLE_CACHE:DELETE/1

The code is present in the next snippet.

```
delete(Key) ->
    case sc_store:lookup(Key) of
        {ok, Pid} ->
            sc_element:delete(Pid);
        {error, _Reason} ->
            ok
    end.
```

The sc_store is queried to find out if there is a mapping. If not we simply don't do anything but return ok. If there is a mapping found the actual deletion operation is entirely delegated to sc_element:delete/1. We don't do the sc_store:delete here because sc_element handles that for us. The reason for that is so that in the case of an accidental sc_element process death the use of terminate in our sc_element gen_server implementation will take care of cleanup then too.

That is it, quite simple. We left the header out because it consists of only a module attribute and an export attribute. That means our simple cache is ready and functioning. To give it a try, and we recommend that you do, compile all the modules in the src directory placing all the beams in the ebin directory. Then run:

```
erl -pa simple_cache/ebin
```

and try out the functions we just created in the simple_cache API.

6.4 – Summary

We have a cache, and how easy was that? Really easy; our cache can take values, insert them into the cache, it can look them up, and it can get rid of them without leaving a trace of state hanging around. We put together quite a bit in this chapter. This has really put to work all that you learned in the first part of this book. We have a working application. We started by setting up the application framework with the application behaviour module that led us naturally to our top level supervisor sc_sup. sc_sup was a different take on supervision. It did not spawn any of its own children but instead acted as more of a factory for other processes. This was accomplished by setting it up as a simple_one_for_one supervisor. With our application skeleton created we set about creating the sc_element module to store values, the sc_store value to abstract storage and handle our key to pid mappings, and finally we created the API in the simple_cache module that tied it all together.

You should not have a nice ability to create an app and we are certainly getting a pretty good idea of how to encapsulate protocols, manage state, layout clean modues, and in general write a nice clean OTP style application. From here we are going to delve down deeper into some more Erlang technology and augment our cache because as nice as it is, the core developers have raised the issue that it is not quite suitable to the job of augmenting the Erlware website just yet. Fundamentally our cache just is not up to production standards just yet. If we have a problem in production we are not going to know a thing about it, we don't even have the most basic form of logs. Further more monitoring and eventing has not been considered at all and so as this cache runs in production we will have no idea whether our cache lives or dies unless we explicitly go and look. These issues will be addressed in the next chapter.

7

Logging and Eventing the Erlang/OTP Way

So we have this wonderful cache now. It can take information in the form of key value pairs and store it away. It also allows you to grab that information in a hurry. It does this cleanly with various processes, supervisors and applications. As clean as the cache is there is still a lot going on within it. There are applications that start, there are processes that start, and die, there is data being stored, retrieved, and expiring. All these events are taking place without us having much of awareness or even capability for having awareness of them. So if we were to ask how many inserts took place over the last hour we would have no way of knowing. What's more, if something were to go wrong we would have no way of knowing about that either. In this chapter we are going to introduce you to the concept of Eventing. Events transpire in the system all the time and fortunately Erlang/OTP provides a mechanism by which it is possible to hook into an event stream with event handlers and act upon those events. In this chapter we will show you how the event system is used to support a robust standard OTP logging system. We will show you how to modify that logging system by extending the event system with custom event handlers. We will also show you how to generate your own application level events so that you will ultimately provide users of your systems a way to hook into an application level event stream to provide custom functionality. Just to sum that up here is what we will be covering:

1. The logging system
2. Hooking into to the logging system with custom handler
3. Creating and handling custom events

So let's now entertain the question; "what something goes wrong during all this storing and retrieving of information from the Simple Cache?" and use that as a nice segue into talking about an answer to that question in our discussion of the OTP logging system.

7.1 – Logging in Erlang/OTP

"what something goes wrong during all this storing and retrieving of information from the Simple Cache?" Well right now we probably wouldn't even know about it unless the entire service crashed. We need to put an end to the silent treatment the simple cache is currently giving us. In this chapter that's exactly what we are going to do. Erlang/OTP has pretty good facilities for doing so. They are the logging application, called SASL (system architecture support libraries), and the Eventing infrastructure, called gen_event. Together they give us powerful tools for communicating certain types of information to the world at large. In fact, the gen_event system even gives us a useful method to allow other applications to hook into our system. We will dig down into both of these before we are done. Right now we want to provide you with a bit of an overview of how logging is done in general.

7.1.1 – Logging in general

You have most likely used logging systems like log4j(log4c, log4r, etc…) or similar systems in the past. Every language has a flavor that becomes the basic defacto standard because logging is such an essential part of the software development landscape. Usually, there are several levels to any logging system. These 'log levels' indicate the importance of the information being logged. One commone scheme is to have four 'log levels', critical (or severe), warn, info, and debug. For the most part these are self explanatory. For those of you that are already familiar with how logging levels should be defined and the semantics around them you can jump ahead to section 7.2.

The critical, or severe, log level is for *bad* stuff. Things for which action should be taken immediately because the system will fail catastrophically or be unusable by its customers. You shouldn't use this often. It should be reserved for only those types of events. The 'warn' log level is used to warn the operator of the system that something bad, but not fatal happened. You use this when an unexpected event occurs that you can work around but probably shouldn't happen. Once again this shouldn't be used that often. The next level of severity is 'info' and it represents just that. You use that when you want to inform the operator that something happened. Its information which means the event that took place may be good or it may not. You can use this as much as you would like, but don't go crazy with it. That's what the next level is for. The 'debug' level is used for *everything*. This level

is mostly for you, the developer, it will help you debug problems in the running system. So when you are setting up logging you will probably log every message, state change, etc out as a debug level message.

You can think about these log levels as a hierarchical system with 'debug' at the lowest level and 'critical' at the highest. Most logging systems allow the operator to set the level of messages he is interested in. So if they want to see everything, they set the level to 'debug'. With that they will get all four message types. If they want to see most everything, but debug just too verbose they would set the level to 'info'. This will allow them to see the top three levels, critical, warn and info in the logs. If they only want to know if there are problems then they would set it to 'warn' and just get the critical and warning messages.

Logging system tends to do other things, like provide adjustable formats, timestamp logging messages etc. However, we are going to skip over that for now and go into what Erlang provides you for logging.

7.1.2 - Erlang/OTP built in logging facilities

Logging is a common enough requirement that there is a facility for accomplishing it included in the base Erlang/OTP distro. That facility is built into the SASL application. So what does SASL give you in the way of logging?

> **SASL and SASL**
>
> Now, as some of you may know SASL is also the name of a pretty common network management protocol. However, it has absolutely nothing to do with that. When I first started using Erlang I spent way to many cycles trying to figure out what Erlang's SASL library had to do with the SASL protocol. The answer is absolutely nothing, nothing at all. So divorce the two in your mind. In this case the divorce will be a happy one.

Quite a bit actually. Not only does it give you direct facility for logging it also gives you some quite interesting ways to do log pulling and reviews. It also gives you a method to plug into the logging system so that you can log in your own format, nifty right? Let's get started by just looking at the logging API itself.

7.1.3 – The Logging API

The API is pretty straightforward but only provides you the ability to log at three of our four levels. For some reason it does away with the debug level. That's ok though. We will just use the info level for debug. So further adieu let's look at the API.

```
error_msg(Format) -> ok.
error_msg(Format, Data) -> ok.

warning_msg(Format) -> ok
warning_msg(Format, Data) -> ok.

info_msg(Format) -> ok.
info_msg(Format, Data) -> ok.
```

So you are probably able to work out the basics of these calls on your own. Each call is prefixed with it the type of message it is. So 'error_msg' sends error messages, 'warning_msg' sends warning messages and 'info_msg' sends info messages. Let's play with them a bit. Start up an Erlang shell and manually fire up the sasl application as seen in the next snippit.

```
Erlang R13A (erts-5.7) [source] [64-bit] [smp:2:2] [rq:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.7  (abort with ^G)
1> application:start(sasl).
```

You will see a lot of stuff flowing past your screen. That's ok; it's just the SASL logger getting started on its business. The SASL application provides way more than just logging. What you are seeing scrolling past is start up information various processes that are starting. This is start of its progress report functionality. Basically as each process starts up important information about whets going on is printed to the log (in this context the screen). It also gives quite useful information when processes crash or shutdown. We will get into that in a bit. Let's concentrate on the logging methods for now and send some messages.

```
3> error_logger:info_msg("This is a message~n").

=INFO REPORT==== 4-Apr-2009::14:35:47 ===
This is a message
Ok
```

What we did here is send an info message to the logger. SASL tags it with some information (timestamps and what not) and prints it to the log. That 'ok' you see is what's returned by the 'error_logger:info_msg' function itself which happens to be printed to standard out the same way that the log message is by the shell. What if we want to print some data? Well that's where the two argument versions of the function come in. Erlang has a format syntax that is somewhat like C's printf. It's exposed in the 'io_lib' module and is

used in quite a few applications. It's well documented on the Erlang site so we don't have to go into quite so much detail here. However, we will play around with writing out some data.

```
4> error_logger:info_msg("This is an ~s message~n", ["info"]).

=INFO REPORT==== 4-Apr-2009::14:39:23 ===
This is an info message
ok
5>
```

So we have moved to using the two argument format and we have embedded a string into the format message. There are a couple of things you need to pay attention to here. First and foremost is that the second argument must always be enclosed in a list. You can put as many elements in the list as are format specifies in the format string.

```
5> error_logger:info_msg("This is an ~s message~n", ["info",
this_is_an_unused_atom]).

=INFO REPORT==== 4-Apr-2009::14:42:37 ===
ERROR: "This is an ~s message~n" - ["info", this_is_an_unused_atom]
ok
```

As you can see, you may put in more or less you will not get an error. The SASL system will happily print out both the format string and the arguments without merging them. This is mostly so that you will always get what may be critical information even if you screw up the coding a bit. It may not seem like much but it's a pretty awesome feature of the system.

Let's put in a couple of real arguments now.

```
9> error_logger:info_msg("Sent invalid reference ~p from ~s ~n",
[make_ref(), "stockholm"]).

=INFO REPORT==== 4-Apr-2009::14:53:06 ===
Sent invalid reference #Ref<0.0.0.70> from stockholm
ok
10>
```

What we have done here is sent a useful informational message using data that we might have been passed in. Take note of the ~p. This is a very useful format specifier. Basically, it prints out the referenced term using the Erlang pretty printing methods. In this case it we can see the pretty printed output of an Erlang reference.

Refs

> Refs aren't really relevant here but we wanted to explain what they are. They are basically arbitrary values that have a very high likelihood of being unique. They are really useful to use as identifiers and may be pattern matched like any other Erlang term. You should be aware that they are not guaranteed to be unique but they do have a very high probability of being unique. Basically, they re-occur after approximately 2^82 calls. That is unique enough for most practical purposes.

The other types of messages work exactly the same way. You may rerun the above examples replacing 'info' with 'warn' or 'error' and get exactly the same result. With the API to the error_logger well understood we will next cover a special class of logging; startup and shutdown reports.

7.1.4 - Startup and Shutdown Reports

As you saw when we first started SASL the application has the ability to give you some useful information about startup and shutdown. When you use OTP behaviours you get this information for free. Let's create a little module whose only job is to come up for a little while.

Code Listing 7.1 - Error Example Module

```
-module(die_please).

-behaviour(gen_server).

-export([start_link/0]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
      terminate/2, code_change/3]).

-define(SERVER, ?MODULE).
-define(SLEEP_TIME, 2*1000).

-record(state, {}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->                 #1
    {ok, #state{}, ?SLEEP_TIME}.

handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(timeout, State) ->          #2
    i_want_to_die = right_now,
```

```
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

This is a pretty straightforward gen_server. You have created more than a few of these by this point. So this should be all familiar. The two places you want to take note of here are the init/1 function and the handle_info/2 function, which can be seen at code annotations #1 and #2 respectively. In the init/1 function we return three values instead of the usual two. The third value is a timeout in milliseconds. It indicates that we want to get a 'timeout' message in the number of milliseconds specified. That message will come in the handle_info/2 function. In the handle_info function we do something that you wouldn't do in a normal application. We are intentionally including code that we know will cause an exception and result in the process dying. Since our purpose here is to play with the SASL logging functionality that works just fine. Let's compile it in the next code snippet.

```
1> c("die_please.erl").
./die_please.erl:101: Warning: no clause will ever match
{ok,die_please}
2>
```

Be sure to kill your shell and restart it for this example. We want to start fresh for this. We cd to the directory where our nice little file is and we compile it with the 'c' shell command. The Erlang compiler is reasonably smart and recognizes that we did something stupid. Fortunately for us it still compiles it because what we have done is not invalid syntax.

Once everything is compiled go ahead and start it up using the start_link/0 function as is done in code listing 7.2.

Code Listing 7.2 - Calling the Example

```
2> die_please:start_link().
{ok,<0.40.0>}
3>
=ERROR REPORT==== 4-Apr-2009::15:18:25 ===
** Generic server die_please terminating
** Last message in was timeout
** When Server state == {state}
** Reason for termination ==
** {{badmatch,right_now},
    [{die_please,handle_info,2},
     {gen_server,handle_msg,5},
     {proc_lib,init_p_do_apply,3}]}
** exception error: no match of right hand side value right_now
```

```
      in function  die_please:handle_info/2
      in call from gen_server:handle_msg/5
      in call from proc_lib:init_p_do_apply/3
3>
```

Our nice little process comes up just fine and two seconds later terminates with some reasonable useful error information. You should see something similar to what above in listing 7.2 when you run this code. Next in code listing 7.3 we start SASL again and see how things change.

Code Listing 7.3 - Calling the Example with SASL

```
3> application:start(sasl).
ok
4> die_please:start_link().
{ok,<0.53.0>}
5>
=ERROR REPORT==== 4-Apr-2009::15:21:37 ===
** Generic server die_please terminating
** Last message in was timeout
** When Server state == {state}
** Reason for termination ==
** {{badmatch,right_now},
    [{die_please,handle_info,2},
     {gen_server,handle_msg,5},
     {proc_lib,init_p_do_apply,3}]}
5>
=CRASH REPORT==== 4-Apr-2009::15:21:37 ===
  crasher:
    initial call: die_please:init/1
    pid: <0.53.0>
    registered_name: die_please
    exception exit: {{badmatch,right_now},
                     [{die_please,handle_info,2},
                      {gen_server,handle_msg,5},
                      {proc_lib,init_p_do_apply,3}]}
      in function  gen_server:terminate/6
    ancestors: [<0.42.0>]
    messages: []
    links: [<0.42.0>]
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 377
    stack_size: 24
    reductions: 132
  neighbours:
    neighbour: [{pid,<0.42.0>},
                {registered_name,[]},
                {initial_call,{erlang,apply,2}},
                {current_function,{shell,eval_loop,3}},
                {ancestors,[]},
                {messages,[]},
```

```
                    {links,[<0.27.0>,<0.53.0>]},
                    {dictionary,[]},
                    {trap_exit,false},
                    {status,waiting},
                    {heap_size,1597},
                    {stack_size,6},
                    {reductions,3347}]
** exception error: no match of right hand side value right_now
     in function  die_please:handle_info/2
     in call from gen_server:handle_msg/5
     in call from proc_lib:init_p_do_apply/3
5>
```

We still get our usual information. However, we also get a lot more information about the process that crashed why it crashed and how it crashed. This is the kind of information that comes in useful when you are debugging a problem with a process crash.

Another example is in order. Let's create a really simple module that doesn't use gen_server and see what happens. the code for this can be seen in listing 7.4. This one really has the same purpose as our other module. It's just a bit bare. It's also going to startup and after two seconds die. Go ahead and compile it.

Code Listing 7.4 - A New Non-OTP Example

```
-module(die_please2).

-export([go/0]).

-define(SLEEP_TIME, 2000).

go() ->
    timer:sleep(?SLEEP_TIME),
    i_really_want_to_die = right_now.
```

Let's run it and see what happens.

```
14> spawn(fun die_please2:go/0).
<0.79.0>
15>
```

It should startup and then die with a bad match after a few seconds. You may notice that the information is much less copious then in our previous example, even with SASL started. Why is that? Well it's actually pretty straightforward. SASL needs a little setup work to actually do anything. This setup work is done for you when you use behaviours like gen_server. However, when you roll your own processes you don't get that. Well, that's not entirely true. If we change our semantics just a bit we can get the behaviour we expect. Let's start our process just a bit differently as seen in code listing 7.5

Code Listing 7.5 - Enabling SASL

```
16> proc_lib:spawn(fun die_please2:go/0).
<0.83.0>
17>
=CRASH REPORT==== 4-Apr-2009::15:34:45 ===
  crasher:
    initial call: die_please2:go/0
    pid: <0.83.0>
    registered_name: []
    exception error: no match of right hand side value right_now
      in function  die_please2:go/0
    ancestors: [<0.77.0>]
    messages: []
    links: []
    dictionary: []
    trap_exit: false
    status: running
    heap_size: 233
    stack_size: 24
    reductions: 72
  neighbours:
```

Ah we got a crash report more like we expected didn't we. The module that we are using here is proc_lib. It has all the normal spawn functions, but it also wraps up the started process in the code that we need to get interesting error information. In the unlikely event (for now) that you write non behaviour driven processes you should typically start them with proc_lib. You will be doing yourself a favor in the long run. In the next section we will show you how to control your systems logging ability ever more tightly and start to gain a little more of an understanding of how the eventing system works and how we can hook into it with custom event handlers.

7.2 - Hooking Into the System with a Custom Event Handler

Let's say we don't like the format of the logs outputted by the error logging system. They are pretty different from what the rest of the world uses. It may also be that we work at a company that already has a wealth of tools written around its own log format and our format just doesn't fit in what can we do about it? SASL gives you the ability to plug into the Error Logging system and write out your own error information.

This facility is based on Erlang's event handling framework and the gen_event behaviour. This behaviour basically wraps up a lot of the requirements of an Eventing system in a nice easy to use interface. That's one of the reasons they based the pluggable logging features on top of it. To plug into the logging infrastructure we need to implement a new gen_event call back module. Fortunately, that's pretty simple. The gen_event behaviour interface is very similar to that of the gen_server behaviour that we have already gone over. In the case of

gen_event the functions that we implement are slightly different. Both gen_event and gen_server interface specifications require the init, code_change and terminate functions that we have already seen. 'gen_event' even requires the handle_call and handle_info functions the same way gen_server does. However, gen_event adds a requirement for a 'handle_event' call. As you may be able to guess that's where we are going to get our events. In the case of our custom error logger that's where we will get the log events that we want to write out. Let's go ahead and start implementing our log event handler in code listing 7.6.

Code Listing 7.6 - Custom Logging Plug-in

```
-module(custom_error_report).

-behaviour(gen_event).

-export([start_link/0, add_handler/0]).

-export([init/1, handle_event/2, handle_call/2,
      handle_info/2, terminate/2, code_change/3]).

-record(state, {}).


start_link() ->
    gen_event:start_link({local, ?SERVER}).

register_with_logger() ->
    error_logger:add_report_handler(?MODULE).          #1

init([]) ->
    {ok, #state{}}.

handle_event(Event, State) ->
    {ok, State}.

handle_call(_Request, State) ->
    Reply = ok,
    {ok, Reply, State}.


handle_info(_Info, State) ->
    {ok, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Here we do a very bare bones implementation of the gen_event behaviour for the error_logger. It doesn't do really anything yet except receive events. We want to do something special with the events that we receive though. In this case we want to write them out to the screen. The error_logger generates a very specific set of events. We can leverage that knowledge without a problem.The table 7.1 below shows the events that are generated by the error logger.

table 7.1 – error logger events

Event	Description
{error, Gleader, {Pid, Format, Data}}	Generated when error_msg/1, 2 or format is called.
{error_report, Gleader, {Pid, std_error, Report}}	Generated when error_report/1 is called.
{error_report, Gleader, {Pid, Type, Report}}	Generated when error_report/2 is called.
{warning_msg, Gleader, {Pid, Format, Data}}	Generated when warning_msg/1, 2 is called, provided that warnings are set to be tagged as warnings.
{warning_report, Gleader, {Pid, std_warning, Report}}	Generated when warning_report/1 is called, provided that warnings are set to be tagged as warnings.
{warning_report, Gleader, {Pid, Type, Report}}	Generated when warning_report/2 is called, provided that warnings are set to be tagged as warnings.
{info_msg, Gleader, {Pid, Format, Data}}	Generated when info_msg/1, 2 is called.
{info_report, Gleader, {Pid, std_info, Report}}	Generated when info_report/1 is called
{info_report, Gleader, {Pid, Type, Report}}	Generated when info_report/2 is called.

Code listing 7.7 shows how we change our handle_event callback function to make use of this new knowledge.

Code Listing 7.7 - Handling error_logger events

```
handle_event({error, _Gleader, {Pid, Format, Data}}, State) ->
    io_lib:fwrite("ERROR <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({error_report, _Gleader, {Pid, std_error, Report}}, State) ->
    io_lib:fwrite("ERROR <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({error_report, _Gleader, {Pid, Type, Report}}, State) ->
    io_lib:fwrite("ERROR <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};
handle_event({warning_msg, _Gleader, {Pid, Format, Data}}, State) ->
    io_lib:fwrite("WARNING <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({warning_report, _Gleader, {Pid, std_warning, Report}}, State)
```

```
->
    io_lib:fwrite("WARNING <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({warning_report, _Gleader, {Pid, Type, Report}}, State) ->
    io_lib:fwrite("WARNING <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};
handle_event({info_msg, _Gleader, {Pid, Format, Data}}, State) ->
    io_lib:fwrite("INFO <~p> ~s", [Pid, io_lib:format(Format, Data)]),
    {ok, State};
handle_event({info_report, _Gleader, {Pid, std_info, Report}}, State) ->
    io_lib:fwrite("INFO <~p> ~p", [Pid, Report]),
    {ok, State};
handle_event({info_report, _Gleader, {Pid, Type, Report}}, State) ->
    io_lib:fwrite("INFO <~p> ~p ~p", [Pid, Type, Report]),
    {ok, State};
handle_event(_Event, State) ->
  {ok, State}.
```

So this is a bit contrived. All it really does is turn around and write out the event coming in directly to standard out in slightly different format then the standard error logger. However, it gives you what you need to understand the writing of your own custom plug-ins. One thing you should take note of is that we may receive events that don't fit this list of formats. These are generally system messages that we can safely ignore. With that in mind we need to always have that last 'catch all' function clause. We have covered the three major and most common aspects of the logging infrastructure in Erlang at this point. We know understand:

1. The Error Logger API
2. The SASL startup and shutdown/crash reports
3. How to extend and customize the Error Logger

This means we are ready to take this knowledge and apply it to the Simple Cache which we will do in the next section when we learn how to create our own custom application level event stream.

7.3 - Bringing It Back to the Cache (Custom Event Streams)

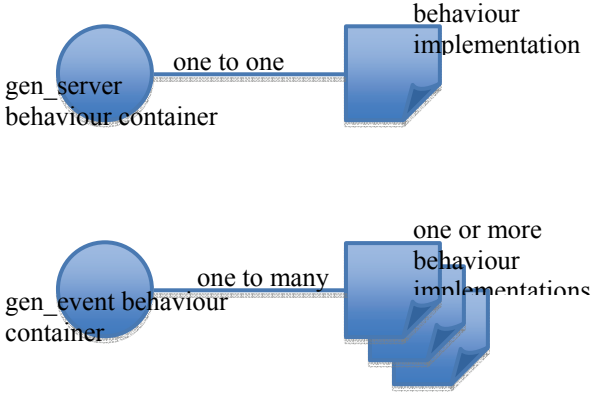
In the introduction to this chapter we indicated that there is a lot going on in the system. We just saw how we can create logging events with the error_logger API and how those events are handled by event handlers. What if we wanted to create our own application specific event stream outside of the logging stream? Our system in particular has a number of very cache specific events that we might want to create. These might be of the form of an insert event, a delete event, and a lookup event among potentially others. Once we had such events other users of the system could easily code event handlers to hook into our

event stream as we did when we hooked into the error_logger event stream. These custom handlers could be coded to answer questions like, how many lookups did the cache have in the last hour, or how many inserts, or what was the frequency of deletes for a particular key relative to another. I am sure there are many other things we could imagine creating off the back of this event stream and many we can't imagine right now as well. That is kind of the point though. By creating this event stream on top of the gen_event behavour we provide a way for others to plug into what the cache is doing so as to implement custom behavour. In this section we will dig deeper into the gen_event behaviour and we will use it to create our custom event stream. Then we will create an event handler to intercept those events and funnel them to the error logger system we are already so familiar with. We could just as easily hook into the stream with a custom handler and funnel the data into a statistics sub system, or a remote monitoring engine. We are familiar with error logger so we will choose that for this exercise. Through this implementation you will learn the essentials of creating and hooking into event streams. First things first, lets deepen our understanding of the gen_event behaviour.

7.3.1 – A Little on the gen_event Behaviour

gen_event is a little bit different than other behaviours like gen_server. The main difference between the gen_event behaviour and the others is that the behaviour container can support many behaviour implementation modules within a single behaviour container. This means that the container is started once and and subsequently behaviour implentations are added to it. Figure 7.1 illustrates the difference between gen event and gen_server in this respect.

figure 7.1 – gen_server vs gen_event



When a message (event) is sent into an event manager all the registered behaviour implementations within that container receive the event.

Because of this one to many relationship you will not find a start_link function in many modules that implement the gen_event behaviour. If you do you will probably see that it checks to see if the behaviour container is already started before trying to start one anew. Most times however the start *link function will not appear in a gen*_event behaviour implementation. One methond for starting the gen_event container or event manager as it is frequently referred is to start it directly as a child spec in a supervisor. This is demonstrated in the next snippet.

```
{my_manager,
   {gen_event, start_link, [{local, my_manager}]},
     permanent,
     1000,
     worker,
   [gen_event]}
```

The manager is given the name my_manager in the tuple contained in the startup arguments. Once stated the manager above can be referenced by that name in order to add behaviour implmentations (callback modules) into it. We will not be starting the event manager this way however. In our case we will use an entirely separate module to house the startup code because we have an entire eventing API to contend with as well and the location of the API just so happens to make a really nice place to house the event manager startup function. Lets get into developing this API.

7.3.2 – Our Eventing API

The API module will not export the behaviour attribute for gen_event nor will it be a gen_event implmentation. This module will serve to house the API for our eventing system. Events that get sent into the system are of course wrapped with functions to hide the actual eventing protocol from our users. This API module will serve to encapsulate those protocols and start our event manager. Code listing 7.8 contains the first portion of the sc_event module that houses within it our event manager startup code. The rest of the API will be explained in further code listings as we move on.

Code Listing 7.8 - Simple Cache Eventing API

```
-module(sc_event).

-export([start_link/0,
         lookup/1,
         create/2,
```

```
          replace/2,
          delete/1,
          add_handler/2]).

-define(SERVER, ?MODULE).

start_link() ->
    gen_event:start_link({local, ?SERVER}).
```

This is the initial definition of our Eventing API module. As you can see it doesn't implement any callback functions. It does, however, provide a start link function very similar to what we are used to. In this case we just call the gen_event:start_link function. That function returns all that's required to run under the supervisor.

We want to make it easy to add event handlers to the eventing system. To that end we will provide a wrapper for the standard event registration function which can be seen in the next snippet. This function knows what event manager to register the handlers passed into it. It hides the fact that we registered the event manager as ?SERVER. It is a pretty simple function but it makes it nice for people implementing their own event consumers.

```
add_handler(Handler, Args) ->
    gen_event:add_handler(?SERVER, Handler, Args).
```

What follows here in code listing 7.9 is the eventing API as implemented within this module. The gen_event module provides the 'notify' function for Eventing. Our Eventing API is going to use these functions and define the event protocol that our system understands. For this system we will have to break encapsulation a little bit strictly speaking. The protocol defined in this module will need to be understood by each of the behaviour implementations we create and add to our manager. This breaking of encapsulation is not too bad though because all these modules are really part of the same eventing subsystem. This protocol should be exported no further than this however or things will start to get ugly. In listing 7.9 we create a function per event and use gen_event:notify to send these events to the event manager. These functions also serve to encapsulate the naming of our manager and render it such that the users of this API need not know what the name of the event manager is.

Code Listing 7.9 - The Eventing API

```
create(Key, Value) ->
    gen_event:notify(?SERVER, {create, {Key, Value}}).

lookup(Key) ->
    gen_event:notify(?SERVER, {lookup, Key}).

delete(Key) ->
    gen_event:notify(?SERVER, {delete, Key}).
```

```
replace(Key, Value) ->
    gen_event:notify(?SERVER, {replace, {Key, Value}}).
```

So in our code, when we want to do something like send a 'lookup' event, all we need to is call sc_event:lookup(Key) instead of the much less consumable gen_event:notify(sc_event, {lookup, Key}). We also have the benefit that if your event format changes we don't need to go back and change the Eventing code in every place the event is sent throughout our code base.

7.3.3 - Integrating into Simple Cache

Remember, that gen_event is a gen_server that needs to be managed by a supervisor. To that end we are going to put it under our root supervisor. This is another area where wrapping our code comes in handy. Putting it under supervision becomes a simple matter of adding sc_event to our supervisors child spec.

Code Listing 7.10 - Simple Cache Supervisor

```
init([]) ->
    RestartStrategy = simple_one_for_one,
    MaxRestarts = 0,
    MaxSecondsBetweenRestarts = 1,

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

    Restart = temporary,
    Shutdown = brutal_kill,
    Type = worker,

    Element = {sc_element, {sc_element, start_link, []},
           Restart, Shutdown, Type, [sc_element]},

    Event = {sc_event, {sc_event, start_link, []},
           Restart, Shutdown, Type, [sc_event]},

    {ok, {SupFlags, [Element, Event]}}.
```

Easy peasy. We just create a new child spec for sc_event and return it with our list of children to start. The supervisor starts and manages our event manager just like any other gen_server. Now all we have to do is instrument our code. For the most part this involves going back and modifying the simple_cache module that serves as the API for simple_cache, to fire events when Eventing happens.

We don't want to include the entire simple_cache module here so let's just look at one function and how it changes. If you want to see the module as a whole, check the included code for this chapter.

Code Listing 7.11 - Code without Instrumentation

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_element:replace(Pid, Value);
     {error, _Reason} ->
         sc_element:create(Key, Value)
    end.
```

This is our code as it exists in the system currently. We want to add the Eventing around this code so that when things change we can fire events about the change.

Code Listing 7.12 - Code with Instrumentation

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_event:replace(Key),
         sc_element:replace(Pid, Value);
     {error, _Reason} ->
         sc_event:create(Key),
         sc_element:create(Key, Value)
    end.
```

Our code changes just slightly. Along with actually doing the thing we are Eventing about we fire the event that informs event consumers what's happening with the system. Now remember that we spent a lot of time talking about logging in this chapter. Logging is critical and now we get to see how our custom eventing system and logging play together.

7.3.4 - Using Event Handling to Log Events

Now that we have an eventing system and API we want to be able to take advantage of that system. To that end we are going to create an event handler for the event system we just got done creating and attach it to our event manager. Amazingly enough it's very similar to the custom_error_report module that we created for the error_logger system as seen in code listing 7.7. That's because they are both gen_event behaviour implementations. One consumes logging events while the other consumes simple cache events. Figure 7.13 presents the beginning of our logging handler for our cache events.

Code Listing 7.13 - Event Consumer

```
-module(sc_event_logger).
```

```
-behaviour(gen_event).

-export([add_handler/0, delete_handler/0]).


-export([init/1,
         handle_event/2,
         handle_call/2,
         handle_info/2,
         code_change/3,
         terminate/2]).

add_handler() ->                #1
     sc_event:add_handler(?MODULE, []).

delete_handler() ->             #2
     sc_event:delete_handler(?MODULE, []).
```

Notice of course that we use the –behaviour attribute to specify that this is a gen_event behaviour implementation module. We have our two export attributes one for our API and one for the gen_event callback functions we are implementing. Code annotations #1 and #2 highlight the API which handles two things; how it is we start and stop this logging functionality. The add_handler/0 and delete_handler/0 functions rely on the convenience functions that we added to the sc_event module which contains our eventing API. When sc_event_logger:add_hander/0 is called the sc_event_logger module is added to the sc_event manager behaviour container and logging of simple cache events commences because each event sent to the sc_event manager will subsequently get passed to this hander. The delete_handler/0 function does the opposite of course. Code listing 7.15 contains the code that actually does this logging.

7.15 Event Handling Functions

```
handle_event({create, {Key, Value}}, State) ->
    error_logger:info_msg("create(~w, ~w)", [Key, Value]),
    {ok, State};
handle_event({lookup, Key}, State) ->
    error_logger:info_msg("lookup(~w)", [Key]),
    {ok, State};
handle_event({delete, Key}, State) ->
    error_logger:info_msg("delete(~w)", [Key]),
    {ok, State};
handle_event({replace, {Key, Value}}, State) ->
    error_logger:info_msg("replace(~w, ~w)", [Key, Value]),
    {ok, State}.
```

You may recognize these events from the API we just created with sc_event in code listing 7.9. Here is where we consuming the events that our API creates and sends to the event manager that is started by its start_link function. In this case we are just turning around and logging them out via the error logger.

Conclusion

So we have learned all about the concept of eventing within Erlang/OTP. We have learned how to use the Erlang/OTP logging system. We have learned a bit about how the logging system uses the event system and how to exploit that fact by plugging in event handlers to allow arbitrary report output. Finally we took all that knowledge and used it to create an application level event stream and a custom handler that funneled those application level events into the logging system. This chapter was pretty exciting. If you are anything like us you probably need to take a few moments and get your breath. You will need it because the next chapter is going to knock your socks off. We are going to be talking about Erlang's distribution mechanisms and how to make use of them and being as this is one of the most exciting aspects of Erlang/OTP we know that you have been looking forward to learning about it.

8

Introducing Distributed Erlang/OTP

In this chapter we are going to take a slight turn away from directly implementing any more functionality on our cache and we are instead going to do a little dive into Erlang distributed programming itself and explore its capabilities on their own. We will return to the cache in the next chapter. In this chapter we are going to lay the groundwork in education and understanding to give you what you need to understand distribution in the next chapter. Though Erlang and OTP make distribution much simpler then it is in other languages it is still a complex topic that will take us a bit to work our way through. With that in mind let's get started with understanding Erlang's approach to distribution.

8.1 – The Fundamentals of Erlang Distribution

Lets say that we have one instance of our cache running on machine A and another instance of our cache running on machine B. If would be really great if, when I insert a key value pair into the cache on machine A it was also available on machine B. Machine A would somehow have to communicate that information to machine B. There are a large number of ways we could go about propagating this information, some easier and more straightforward then others, but they are all distribution. Erlang makes some types of this distributed programming easy; in just a few minutes it is possible to have many machines across a network all happily chatting with one another. Two things fundamentally make this possible.

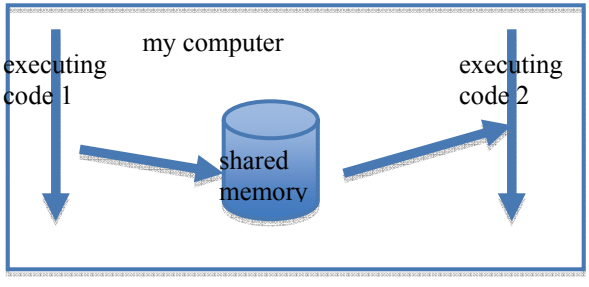
1. Share nothing copy everything; processes
2. Location transparent (mostly) syntax

We are going to take a look at both of these attributes of Erlang and talk about how they make distribution possible. Let's get into the first one 'Share nothing copy everything' immediately.

8.1.1 - Share nothing copy everything; processes

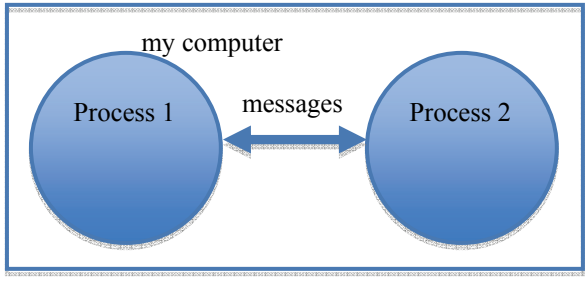
The most widespread models for communicating between two concurrently executing bits of code involve data sharing using the memory space found on the machine they are executing on. We talked about this approach to concurrency in Chapter 5 and there is a lot more information there if you are interested. For now, the figure below illustrates traditional communication between concurrently executing bits of code.

Figure 8.1 In this figure we see two concurrent bits of code communicating via shared memory on the same box.



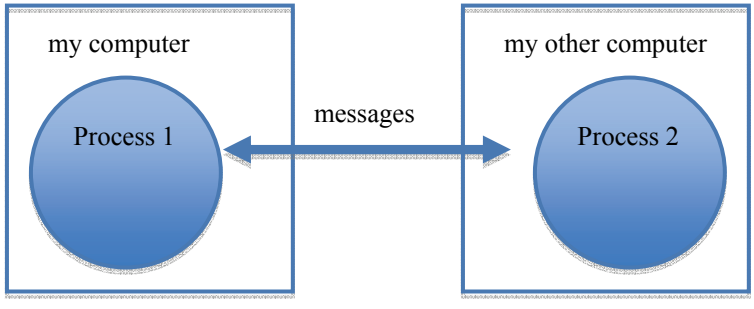
As we said in Chapter 5 the shared memory model has a number of problems, this is especially true when you take that module and try to share information between two different machines. In Erlang when you message another process you copy the data you have to that other process, you don't share it. This is a model that works quite well between processes and even works to a large extent when those processes are not local. The idea of messaging a process is with non-mutable messages works quite well wherever the process is located. The following diagrams contrast the Erlang cases for process communication within the same node and between processes that exist within different nodes on different machines on a network. The first illustrates communication on the same machine same node.

Figure 8.2 In this figure we see two bits of code communicating via messages.



As you can see, the difference between figure 8.2 and 8.3 is pretty minor. The major difference is simply the fact that these concurrent communicating processes are now on different physical pieces of hardware instead of being on the same piece of hardware. The way you go about message passing doesn't change, though there are many additional things you need to be aware of when communicating over the network.

Figure 8.3 Message Passing on Multiple Nodes



Notice that the only thing that changes are what machines the processes are on, nothing about the communication model changes. We don't have to switch from sharing to messaging to accommodate the network, in Erlang we are always messaging and never sharing and so the distributed case is, for the programmer, pretty much identical to the single node case, at least in the APIs and approaches we are using. Of course, there are plenty of possible non-deterministic problems that can occur when you stick a network between two Erlang processes. In any case, with share nothing, copy everything and its benefits covered we take a look at the significance of location transparent syntax, which is a basic facet of Erlangs built in distribution.

8.1.2 - Location Transparent Syntax

The next thing to understand about Erlang distribution is the notion of location transparent syntax. Since Erlang is doing the same sort of thing in order to communicate with processes locally in the same node as it does to communicate with them remotely in a different node the syntax too can be the same. Below is the syntax for communicating a message to a process on the same node followed by the syntax for communicating with a process on a remote node. Starting with the local node communications:

```
Pid ! "my message"
```

and now for case of communications between processes residing on two different nodes on two different machines.

```
Pid ! "my message"
```

I could not resist that one! Yes, exactly the same, that is location transparent syntax in action. Pid can be anywhere, we don't care because we send a message just the same. Erlang guarantees that the Pid is unique on the network so that a conflict of confusion within a cluster of nodes and processes never occurs. It does that in much the same way that machine and port combinations together are unique. Give the property of location transparency it is possible to code systems in Erlang that don't change at all as you scale them from one node to 100; and even back again (something you may wish to do for testing purposes).

These two properties make Erlang distribution a real pleasure when you know how to do it. In this chapter we will dig into how to do it. Currently our cache is totally network unaware. It has no clue that other caches exist. The first thing that would need to change in order to render our cache distributed is that fact. Our Erlang nodes need to become aware of one another.

8.2 – Inter-node Awareness; Forming the Erlang Cloud

Erlang nodes are said to form a cloud when they are aware of one another. You can see several erlang nodes that are unaware of each other and disconnected in figure 8.4.

Figure 8.4 Erlang Nodes Unaware of Each Other on the Network

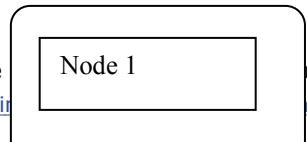
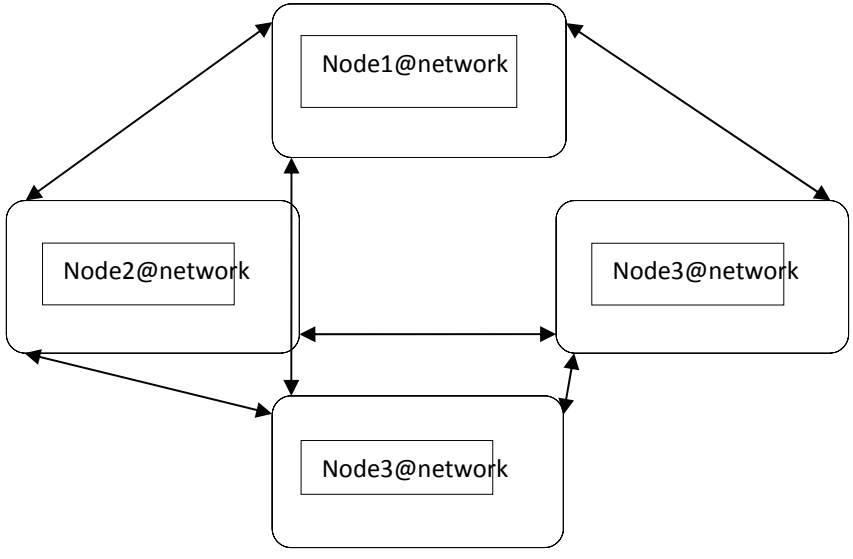


Figure 8.5 shows the same nodes all connected. The main difference here is that each node is aware of every other node and can communicate with it. The differences between the connected figure (8.5) vs the disconnected (8.4) examples quite obvious.

-

Figure 8.5 Erlang Nodes on the Network as Part of a Cluster



So in the last two figures we saw how distributed and non-distributed erlang nodes work. Lets start looking at how to turn non-distributed Erlang nodes into distributed Erlang Nodes. To start an Erlang node in distributed mode you need only give it a name like so

```
erl -name simple_cache
```

On some machines, notably those running SunOS or those that have not set up a fully qualified host name it may be necessary to run your node in short name mode, this can be accomplished by using the flag –sname instead of –name. In either case these commands spawn a shell that is named and ready for distribution. The full name of our node is printed in the shell itself.

```
Erlang (BEAM) emulator version 5.6.2 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.2  (abort with ^G)
(simple_cache@mybox.home.net)1>
```

The name of this node, fully qualified is: simple_cache@mybox.home.net. When you take a look at the processes running on your system and try to find the one called EPMD. I am working with a unix like OS so I am going to run ps and show you what I mean.

```
Mybox.home.net:~ martinjlogan$ ps ax | grep -i epmd
  758   ??  S      0:00.00 /usr/local/lib/erlang/erts-5.6.2/bin/epmd -
daemon
```

Whether you start a single distributed node on a machine or twenty there will always be a single instance of EPMD running. You don't need to know much about EPMD except it's basic function. It is basically DNS for your distributed Erlang VMs. It resolves your VM name to your actual VM so that distributed communication can take place. In doing so it keeps track of all the nodes that are running on a local box which is important for creating a fully connected cloud. Which brings us to our next topic which is connecting the cloud.

To create a cloud of Erlang nodes you need to have at least two. Below I will start nodes with the name "a" and "b" and connect them as a cloud. You can start these nodes on the same machine or different machines, it really makes no difference.

```
> erl -name a
Erlang (BEAM) emulator version 5.6.2 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.2  (abort with ^G)
(a@mybox.home.net)1>
```

and

```
> erl -name b
Erlang (BEAM) emulator version 5.6.2 [source] [smp:2] [async-threads:0]
[kernel-poll:false]
```

```
Eshell V5.6.2  (abort with ^G)
(b@smallserver.home.net)1>
```

the next step is to connect them, this can be most easily accomplished through the ping command in the net_adm module which is included in the kernel application shipped with Erlang itself. We use it as follows to connect two nodes:

```
(a@mybox.home.net)1> net_adm:ping('b@smallserver.home.net')
pong
```

This command will either return "pong" in the case of a successful ping or "pang" in the case of a failure. One of the most common causes of failure is an incorrectly set net cookie.

THE NET COOKIE

Once you have a named node up and running take a look in your home directory. There will be a file there named .erlang.cookie. Open this file with your favorite text editor and you will see a long string. This string of characters is your net cookie. You can verify this by running the following command from the shell on an Erlang node running in distributed mode:

```
(b@smallserver.home.net)1> auth:get_cookie().
'CUYHQMJEJEZLUETUOWFH'
```

Notice that the string returned here is the same one as you see in the .erlang.cookie file. For distributed Erlang nodes to join the same cloud this cookie must be equal for all nodes in the cloud. Two nodes with different cookies can't communicate. This is so that network topology can be managed effectively. There are times when it is desirable to have separate clouds of nodes. For example if you have multiple collocation facilities connected by lower bandwidth cable it may be desirable to have clouds on each network not join one another. The cookie is how you can guarantee they won't accidentally be 'pinged' together. The two ways of having the cookies equal are to programmatically set the cookie with the set_cookie/1 function in the auth module or have it already present on each machine set to the proper value. This second option is quite a bit more common. Typically you don't want to have your code setting the cookie because with well written Erlang code the topology should not matter a bit to the code itself.

If the cookies are set properly and failure still occurs kill your nodes and try starting them with the –sname flag. If success is had with this it can indicate that you need to configure your hostname settings or that you may be having DNS configuration issues. Once you get a successful pong returned your nodes are connected. This can be easily verified by running the nodes() function on either node.

```
(b@smallserver.home.net)2> nodes().
[a@mybox.home.net]
```

Running nodes() on node b reports that it knows about node a. Vice versa will be the case upon running the same command on node a. Erlang maintains what is called a fully connected network, meaning it is transitively closed. Put simply if a connection occurs between node "a" and node "c" and between node "a" and node "b" then there will also exist a connection between node "a" and node "c". All nodes in a cloud know about all others. This can be easily demonstrated by introducing a third node into our cloud named "c" and then pinging it into the cloud and running nodes.

```
> erl -name c
Erlang (BEAM) emulator version 5.6.2 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.2  (abort with ^G)
(c@mybox.home.net)1> net_adm:ping('a@mybox.home.net').
pong
(c@mybox.home.net)1> nodes().
[a@mybox.home.net, b@smallserver.home.net]
```

Notice how by simply pinging node "a" node "c" is made aware of node "b" as well. The same would hold true if we introduced another node "d" and pinged one of our existing nodes, this new node would know about all others from that single ping thus forming a fully connected cloud. I think we need to play just a little bit to demonstrate the ease at which networking can happen in Erlang. Our three nodes are now obviously aware of each other and so keeping them running let's make them chat just a bit to demonstrate the ease of passing messages between them. This may seem somewhat frivolous but this will really help get your mind around the ease of distribution in Erlang.

8.2.1 – Sending Messages Between Connected Nodes

A process running on node "a" communicates with another process running on node "a" by copying it's data over to the second process via a message.

The syntax for the sender looks like this:

```
Pid ! Msg.
```

or as is the case almost always for us it is

```
gen_server:cast/call…
```

on the receiving end at it's most primitive level a receive clause is used.

```
receive
     Msg -> …
end
```

using this knowledge lets get our nodes chatting just a bit. I enter the following on my Erlang shell for node "c"

```
(c@Mybox.home.net)2> register(shell, self()).
true
(c@Mybox.home.net)3> receive
(c@Mybox.home.net)3>    {From, Msg} ->
(c@Mybox.home.net)3>        io:format("Msg ~p~n", [Msg]),
(c@Mybox.home.net)3>        From ! {response, "thanks I printed it"}
(c@Mybox.home.net)3> end.
```

The first line seen above, the one labeled 2, I am using the register command. Recall chapter 3 where we spawned a registered gen server. Registration makes it possible to talk to a process by name instead of by process id. Register caches the pid() supplied and keys it off of a name so that a translation can be done when a message is sent to a particular name. The names must be atoms.

WARNING ABOUT ATOMS – AGAIN

Remember that atoms are not garbage collected. This means that you should not register infinite sets of names. It would be a really bad idea to create atoms based off of something you read in off the network. You would grow your atom table at some non-deterministic rate and it would never shrink. In fact, this is a great way to add a vulnerability to your Erlang application that is really easy to exploit. Just don't do it.

On line 2 the process running the shell gets aliased to the atom 'shell'. This means that the shell can receive messages that are sent to the name 'shell' as in:

```
shell ! hello
```

The register function on line 2 returns true and so I know that it was a success. On the lines labeled 3 I have entered a number of expressions. They should be easily recognizable at this point. First we put the shell in a receive state where it will block until it receives a message of the pattern stipulated on the next line of the clause. {From, Msg}. From is going to be used to capture the pid of the sending process and Msg will be bound to a message sent over from the sending process. Upon receiving this message it will be printed via the io:format function. The ~p marker tells io:format that it is to pretty print the term it is fed. Lastly the message {response, "thanks I printed it"} is sent back to the calling process and

the receive clause is closed out with an end. Once these expressions are entered into the shell we drop into the receive blocking until a message is received.

Next pick one of the other running shells "a" or "b" it is not important which. Run the same functions we ran above there; register the shell and then drop into receive. Finally lets hop onto the shell for the last remaining node and send some messages to the other VMs.

```
(a@mybox.home.net)7> lists:foreach(fun(Node) ->
(a@mybox.home.net)7>                 {shell, Node} ! {self(), "hello there"
(a@mybox.home.net)7>               end,
(a@mybox.home.net)7>               nodes()).
```

Above we are using the foreach higher order function to loop through the list of known nodes, "b" and "c" in this case, and send the message {self(), "hello there"} to each of them. Notice the manner in which we send the message, we are using the familiar ! (bang) operator but the left hand side is a construct that we have not covered yet. It is possible to send messages to processes identified in the following three ways:

1. pid()
2. atom() where the atom is a registered name on the same node
3. {atom(), atom()} or {Name, Node} to be more clear.

We are using number 3 in that list. The first atom represents a registered name, and the second is telling the system which node that registered name is located on. In this way we can send a message to a registered process on a remote node. In our case the name will remain the same throughout our iterations in foreach but the Node will be first "b" and then "c". Upon executing this a message will be sent to each of our other nodes which are patiently waiting in a receive. Those nodes will receive the message, and send us back a message as we instructed them to do a minute ago.

```
(c@mybox.home.net)3> receive
(c@mybox.home.net)3>     {From, Msg} ->
(c@mybox.home.net)3>         io:format("Msg ~p~n", [Msg]),
(c@mybox.home.net)3>         From ! {response, "thanks I printed it"}
(c@mybox.home.net)3> end.
{response, "thanks I printed it"}
(c@mybox.home.net)4>
```

Both nodes should look exactly the same, both having dropped out of receive. Note that in our receive block we have bound the variable From to the sending pid(). On the shell if you type

```
(c@mybox.home.net)4> From.
<5135.37.0>
```

You should see the text representation of the sending process id. In the code we entered on each of the receiving nodes we use From to send the message {response, "thanks I printed it"} back to the sender; which means we should have something waiting for us on the process mailboxes. Lets scoop it out.

```
(a@mybox.home.net)8> receive Reply -> Reply end.
{response, "thanks I printed it"}
```

There it is, Erlang communication at it's simplest. To be honest it does not get much more complex from here. This is the heart and sole of it and the rest is just a little bit of sugar. You now have the basis for working with distributed Erlang because we have covered:

1. Starting distributed nodes
2. Ensuring compatible network cookies
3. Connecting nodes to create a cloud
4. Sending and receiving messages between nodes

8.2.2 – Using the Distributed Shell and the ctrl g Special Shell Mode

Erlang's distributed properties are highlighted nicely by its rather remarkable ability to distribute it's shell. When you start a shell you are just a process talking to the VM through a set of functions that ultimately send messages to other processes on that VM. As a consequence of these Erlang's semantics a shell process running on one machine is as good as one running on the next, they have equal capability to access any particular VM's resources. This means then, that you can "become another node" if you know the deep magic. To demonstrate this, start two nodes, "a" and "b", as before. The nodes can be on the same machine, or different machines, as you should expect by now it does not matter even a little bit.

```
Macintosh:~ martinjlogan$ erl -name b
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.5  (abort with ^G)
(b@Macintosh.local)1>
```

and

```
Macintosh:~ martinjlogan$ erl -name a
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.6.5  (abort with ^G)
(a@Macintosh.local)1>
```

Pick one of the nodes, this node will be the one that assumes control of the other. I am going to pick "a" and use it to essentially become "b". Type ctrl g on the node you have chose to start from. As below you drop into a special shell mode.

```
(a@Macintosh.local)1>
User switch command
 -->
```

Entering a "?" and pressing enter will result in a list of commands that can be run from this mode.

```
User switch command
 --> ?
  c [nn]            - connect to job
  i [nn]            - interrupt job
  k [nn]            - kill job
  j                 - list all jobs
  s [shell]         - start local shell
  r [node [shell]]  - start remote shell
  q        - quit erlang
  ? | h             - this message
 -->
```

The command we are initially interested in is r, which is start remote shell. This command is going to spawn a shell on a remote node and this shell is going to be a proxy through to it, just as if we were on the remote node (a fact that really has nothing to do with anything as location is, again, just not important). The r command takes a node name as an argument and sets up the remote shell there on that node. I will run the "r" command below.

```
--> r 'b@Macintosh.local'
-->
```

Be sure to place the node name argument in single quotes because node names are atom()s. The command does not respond with any type of value, so one might be tempted to think it failed. It did not. Running the "j" command demonstrates this by printing out the jobs currently running. Job 1 is our normal shell, but now notice job 2, which is the shell identifier for a shell running on node "b".

```
--> j
   1  {shell,start,[init]}
   2* {'b@Macintosh.local',shell,start,[]}
-->
```

The next step is to go ahead and connect to that job, thus assuming control of the shell, by using the "c" command for making connections to jobs.

```
--> c 2
Eshell V5.6.5  (abort with ^G)
(b@Macintosh.local)1>
```

Now look at that, we are the shell for "b@Macintosh.local". This means we can execute any command that we could execute on a shell we had started the standard way from the commandline. This includes l(<module-name>) to code load a binary, or commands to kill processes, or take a look at all registered processes, or, well anything. Pretty powerful stuff, but of course with great power comes great responsibility! If you find yourself or your organization making a habit of using this technique in production you may want to rethink things a bit!

QUIT WITH CARE WHEN ASSUMING REMOTE SHELLS

When you are done with your remote shell session and ready to quit your fingers might find themselves using the standard q() shell shortcut to quit. Think! That command shuts down the nde that the current shell is attached to, which VM is that? The node you started from the commandline initially, or the remote VM you connected to? It is of course the remote. I think every Erlang programmer has been burned on this at one time or another. Connect into a node you really don't want to come down and accidentally run q() in a remote shell to the node and bring it to a sudden and early demise. The way to get out of a remote shell quickly is ctrl c. You may also drop back into ctrl g mode and connect back to your initial shell session and kill the remote shell job. Either will work, just don't use q() or init:stop() or the like!

We have learned a lot about Erlang's distributed and we have had the opportunity to learn about and interact with the distributed shell. That's going to serve us in good stead asd we move forward into actually using these distributed features. Now with that in mind, lets build something interesting to drive all these points home. Lets say that we have a bunch of different services on the network, how does one service communicate with other services on the network, in the cloud, to find the services that it would like to consume, or communicate with at runtime? One solution is to hard code the node names of the services needed. That is not very appealing, particularly when we have in our tool box the effortless distribution we just learned about. How about we implement a resource discovery system? What you say, resource discovery is a hard problem that will take us at least 10,000 lines of code sprinkled with weird things like AF_INET and SOCK_DGRAM! Not true, we are learning Erlang here and I do believe we can put something reasonably powerful together without stretching us too far.

8.3 – Implementing Nuts and Bolts; Resource Discovery

Resource discovery means that producers and consumers can find one another without having to be hardcoded or known about prior to discovery time. Resource discovery is kind of like the yellow pages for processes. Resource discovery contains information about where resources are located and makes those resources available to the calling code so that services can easily use them when they need to. A resource comes online and advertises itself so that any other resources that care to consume it can become aware of its existence. At that same time this advertisement tells the rest of the network that this resource would like to be made aware of the resources it needs to consume in order to function; these resources are then cached. This distributed, dynamic approach to resource discovery, makes life very dynamic and is very powerful for a number of reasons.

1. No hardcoded network topology – add resources where and when you need to
2. Scaling becomes much easier – only add resources where and when you need to
3. Testing is simplified – location independence means the whole stack can run in a single vm
4. Upgrading is easier – bring down old services and up new ones dynamically

I am sure that if pressed you can come up with others. In practice dynamically discoverable resources makes life easier from development to production – especially in production. This is because when you have hard coding of where things are located the connections between resources and their consumers becomes brittle. Dynamically creating and registering resources in a consistent way makes your applications much more flexible. However, Before we dive into the implementation we must get some terms and types well understood.

8.3.1 –Terms and Taxonomy

Resource discovery is concerned with producers and consumers, or at least enabling that relationship, to do that it needs to track resources, things that produce, and the desire of consumers to discover those resources. It tracks in a sense, "I have" and "I want". The "I have" must be a fully fledged resource that can be communicated with and the "I want" must accurately identify a particular type of fully fledged resource. This system breaks it down like this:

Resource Terms and Definitions

Term	Definition
Resource Type	A resource type identifies a type of resource.

Resource Instance	A resource instance uniquely identifies, or is, a unique consumable resource.
Resource	A resource is the combination of a resource type, with a resource instance.

RESOURCE TYPE

A resource type identifies a type of resource. Our cache for instance could identify itself on the network as being of resource type 'simple_cache'. There may be many instances of the resource type 'simple_cache' in an Erlang cloud. A consumer that identifies that it is interested, that "it wants" resources of the type 'simple_cache' would discover all instances of the simple cache that exist in its Erlang cloud. This brings us to the definition of an instance.

RESOURCE INSTANCE

A resource instance uniquely identifies, or is, a unique consumable resource. A resource instance most simply is a pid(). It could be a node() or anything else that is unique and consumable by something that cares to consume it. It is this resource instance, that combined with a type we will call a fully qualified resource within the resource discovery system.

RESOURCE

A resource is the combination of a resource type, with a resource instance. Using the resource discovery system we can express "I want" resources of type X and we can find all the resources instances of that resource type that some other network service has indicated "I have".

Now that we have that all straight lets get into the implementation. This is not going to be trivial, no distributed application is, but do pay close attention and be prepared to put a little thought in here and it should be approachable. This is going to help open up all kinds of possibilities for your programming that you did not imagine prior. When producers and consumers can find each other at runtime deployment changes, so does notions of scaling, so does upgrading versions, the ease of slow rolling servers, and a host of other wonderful things. Let's dig in now to implementing our bare bones, ready for you to take to the moon afterwards, resource discovery system.

8.3.2 – Implementing the Resource Discover OTP Applicatino

This resource discovery system is going to be implemented in a single module. Later on I will point you to a version of resource discovery that is much more fully featured and spread across multiple modules, but for the purposes of this chapter we are going to be just fine with a single module. I am sure it comes as no surprise that this module will implement the gen_server behaviour. The header, I am almost tempted not to show it because by this point it should be very predictable stuff for you, is none the less where we will start.

```
-module(resource_discovery).

-behaviour(gen_server).

-export([
         start_link/0
        ]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).

-define(SERVER, ?MODULE).
```

-record(state, {local_resources, target_resource_types, resources}). #1

The first thing to notice is our state record (annotation #1). Defined within are three fields; local_resource, target_resource_types, and resources. local_resources defines the "I have" bit we spoke about earlier. Here is where we put fully qualified resources that exist on the local node. A good example would be 'simple_cache' and 'a@Machintosh'. target_resource_types satisfies the "I want" bit. This is a list of resource types for which we want to discover and cache running processes that provide those resources. Finally 'resources' is the place where we will cache resource instances for the types we have indicated we want to cache in our target_resource_types list.

To build out the rest of this module we will drive the development through our API. As you can see in the header we have thus far defined one function, start_link/0. Let's thread our way through to getting it fully fleshed out.

```
start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).
```

As you can see we locally register this module as ?SERVER and we define the callback module to be the current module with ?MODULE. This leads us to where? What is the next function in the chain, what callback function corresponds to gen_server:start_link? Yes, ?MODULE:init/1. Let's see it.

```
init([]) ->
    {ok, #state{resources             = dict:new(),
```

```
                target_resource_types = [],
                local_resources       = dict:new()}}.
```

Our init function defines our initial server state. Our target_resource_types field is initialized to be an empty list. We set resources and local resources to be blank dictionaries.

Start link is fully implemented which means we can start our server. Let's hop back up to the API and implement the next function, or two functions as in this case. These functions operate in much the same way so we can implement them at the same time. Both are asynchronous casts and both simply store some new data in our state and not much more than that.

```
add_local_resource(Type, Instance) ->
    gen_server:cast(?SERVER, {add_local_resource, {Type, Instance}}).

add_target_resource_type(Type) ->
    gen_server:cast(?SERVER, {add_target_resource_type, Type}).
```

add_local_resource/2 and add_target_resource_type/1 add to our state the "I have" and the "I want" respectively. As you can see add_local_resource adds a fully qualified resource to the system. It adds a resource type and a resource instance which is present on the local node to our state for broadcast to the rest of the erlang cloud. This data will be stored in our state record field called local_resources. add_target_resource_type/1 stores a resource type in the list that identifies which sorts of resource instances we are interested in caching and consuming. As you can see they both use the gen_server:cast/2 function and so we can jump right to ?MODULE:handle_cast/2 to implement them.

```
handle_cast({add_local_resource, {Type, Instance}}, State) #1 ->
  #state{local_resources = LocalResources} = State, #2
  NewLocalResources = add_resource(Type, Instance, LocalResources), #3
    {noreply, State#state{local_resources = NewLocalResources}};
handle_cast({add_target_resource_type, Type}, State) #1 ->
    #state{target_resource_types = TargetTypes} = State,
    NewTargetTypes = [Type|lists:delete(Type, TargetTypes)],
    {noreply, State#state{target_resource_types = NewTargetTypes}};
```

What do we have here? The first line is our handle cast, notice that the form of the data we send from our cast and match on here is {Tag, Value} even if our value is compound as it is here {Type, Instance} we send tag value as in {add_local_resource, {Type, Instance}} (annotation #1). This is a point of good style and one you should try to use for consistencies sake. The next line we break apart our state record (annotation #2). This can be done in the function head but to keep the line narrow I have broken it out. The next line is where we take the resource type identifier and the resource instance we have been provided through the API and we add them to out local resources cache, in this case dict (annotation #3). To do this an internal function is used called add_resources.

```
add_resource(Type, Identifier, Dict) ->
    case dict:find(Type, Dict) of
        {ok, ResourceList} ->
            NewList = [Resource|lists:delete(Identifier, ResourceList)]
            dict:store(Type, NewList, Dict);
        error ->
            dict:store(Type, [Identifier], Dict)
    end.
```

Add resources takes a resource type and a resource identifier and adds them to the dict that we provide. The structure of this dict is that a key, the resource type, relates to a list of resource identfiers. This function assures that if other resource identifiers of this resource type are already present that we append to the existing list. In this function we use lists delete to ensure that each target resource instance is only added to the list at most once. lists:delete/1 returns a new list of resources whether it actually performs a delete or not. If no such resource is present at all then we create the first entry in our dictionary creating a list consisting of a single resource.

Bouncing back up to handle call we then store the new local resources returned by our add resource function as an updated dict structure into our state record and return. The add_target_resource_type handle_cast is much the same. We make the same use of lists:delete/2 to ensure our list of target_resource_types contains unique elements. Once we update our list we store it in state and we know have a new target resource type; a new "I want".

Quickly now, back up to our API and lets implement the next function fetch_resources/1.

```
fetch_resources(Type) ->
    gen_server:call(?SERVER, {fetch_resources, Type}).
```

This function is a synchronous gen_server:call/2 to send the tag and value {fetch_resources, Type}. It will return for us a list of all the resource instances we have cached for a particular resource type. The function returns ok and a list of resource instances or the atom error if there are none available. Jumping into the corresponding handle call…

```
handle_call({fetch_resources, Type}, _From, State) ->
    {reply, dict:find(Type, State#state.resources), State}.
```

Using the alternate notation for fetching a value from a record we pull our resources struct and pass it directly into dict find. This function returns either {ok, Value} or error, which conveniently corresponds directlhy to our stated return value in our API. The value will or course be the list of resource instances we will cache here when we talk to our Erlang cluster to do the real heavy lifting of resource discovery, a function we will cover next.

```
trade_resources() ->
    gen_server:cast(?SERVER, trade_resources).
```

A simple gen_server:cast/2 which sends the tag 'trade_resources'. This next bit is where you need to pay close attention to understand the core functionality of resource discovery as we have implemented it here. This function sends a message to the local resource discovery process. That process then sends a message to all the registered 'resource_discovery' processes on all the nodes in the entire Erlang cluster, including the local node for a nice recursive symmetry. The message it sends is a tag value combination where the tag is again trade_resources and the value is first the sending node, which can be fetched from the command node(), and second the structure containing our local resources. The remote node upon receiving this checks to see which of the sending nodes resources are on its list of target resource types, its "I want" list and caches the ones it cares about. The remote node then sends the originator of the message, the registered 'resource_discovery' process on the node identified by the node name that was sent along with the original message, the same message containing its list of local types. The difference is that instead of sending the originating resource discovery process its node name for a response it sends back the atom 'noreply' indicating to the originating process that no reply is needed. The originating process checks to see which of the remote resource discovery processes resources it has on its target resource type list and caches them accordingly thus ending the transaction with all resources shared. As you can imagine there are a number of things that can go wrong here. One of the biggest is that nodes may just not be addressable. You send the message out into the either and it never ends up at its target node. There are a few ways to mitigate the problem, though not remove it entirely. One of the easiest that covers a lot of the common issues is simple heart beating. For simplicity and clarities sake we are not going to go into it here, but we wanted to call out some of the issues and how to address them.

This approach is actually fairly straightforward once you get your mind around it. In the simple case this algorithm covers quite a bit of resource discovery and will guarantee a fully connected network of asynchronously shared resources. This can be made more robust at the application level with reregistration accomplished by placing trade_resources in strategic places within the supervision hierarchy and coding good crash only software – something I hope you are learning. I will get into the reregistration bit more in the next chapter. Without further adieu we dive into the code. Note that I have removed the logging in this example to make the code more clear. Add it back in when you implement this – logging is critical!

```
handle_cast(trade_resources, State) ->
    #state{local_resources = LocalResources} = State
    lists:foreach(
       fun(Node) ->
           gen_server:cast({?SERVER, Node},
                           {trade_resources, {node(), LocalResources}})
```

```
        end,
        [node()|nodes()]),
    {noreply, State};
handle_cast({trade_resources, {ReplyToken, RemoteResources}}, State) ->
    #state{local_resources       = LocalResources,
           target_resource_types = TargetTypes,
           resources             = Resources} = State,
    ResourceList = return_resources_for_types(TargetTypes,RemoteResources),
    NewResources = add_resources(ResourceList, Resources),
    case ReplyToken of
        noreply ->
            ok;
        ReplyToken ->
            gen_server:cast({?SERVER, ReplyToken},
                            {trade_resources, {noreply, LocalResources}})
    end,
    {noreply, State#state{resources = NewResources}};
```

The explanation above should be enough to follow this code or at least get you a good start.

There is one last thing that I recommend you do before using the resource discovery system, and I am leaving this as an exercise for the reader. This little excersize very, very simple and we have already covered everything you need to know to do it yourself. That would be to implement another process, or some other mechanism to periodically ping the contact nodes in the system as well as reregister with the cloud in order to overcome contact node failures or other such hard to predict network or application failures. Once that is done you have a reasonable guarantee of a healthy cluster. When you design your system on top or resource discovery it is important to understand what reasonable guarantee means. Here, for this case it is largely based on the periods of time for which you can have a disconnected network which will largely depend on the frequency of re-registration and the length of time network or other such outages are allowed to persist.

That said, using the skills we have picked up here with regard to distributed communication we have built a system that uses that communication to automatically discover all the services it needs to run. With this the requirement for foreknowledge of network topology and the location of producers and consumers on our network goes away. Through smart use of distributed communication within an Erlang cloud we have opened up the door to writing systems that are far more dynamic and easily scalable than they would have been without this technology. All this makes us very ready to go about distributing our cache in the next chapter essentially taking it from a standalone entity that sits on one machine to a cloud caches that can work together to help the Erlware folks provide an even better experience for their users. In the next chapter we will see exactly how they aim to do that.

8.4 – Summary

We have really covered more than a little here.

1. Location transparency
2. Bringing nodes together into a cloud
3. Cluster control with the cookie
4. Messaging between nodes
5. Deep magic with ctrl g
6. Applied our skills to resource discovery

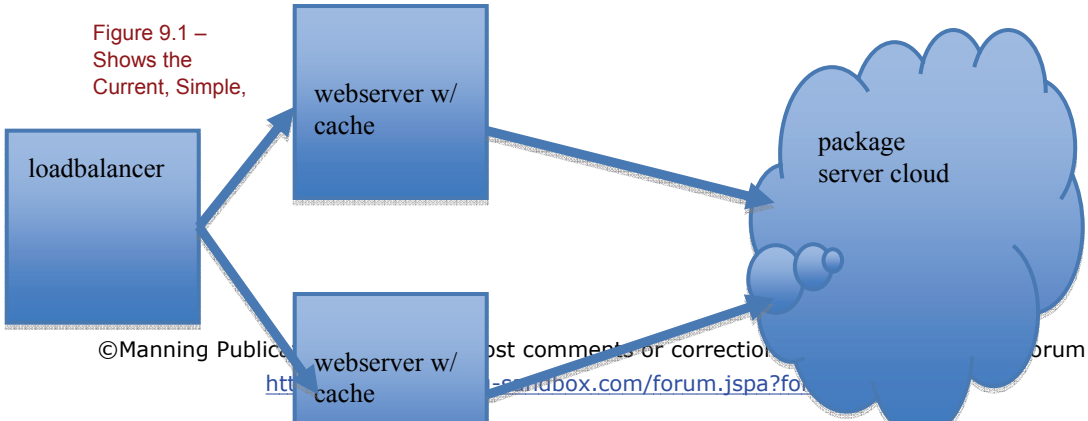
I hope this intro has sparked some real creativity by opening up your eyes to what can be accomplished in very short order with Erlang distributed communication. In the next chapter we will take what we have learned, and indeed what we have built, and apply it to the task of making our cache a suitable session store.

9

Converting the Cache into a Distributed Application

Our cache is now operationally sound as of chapter 7 and you have learned more than a bit about Erlang/OTP distribution in chapter 8, which is a good thing because now we are going to need it. During the process of helping the Erlware guys make their users happy by serving up pages at a rapid pace with our simple cache, members of the core team at Erlware have been adding login and sessions to the site. This will allow package authors to log and update packages, tweak documentation or change availability of their packages. They have asked us to augment our cache to make it suitable for session storage as well. Sessions have the property that they need to be available to any server that participates in serving the web application. Since our application is fronted by a stateless load balancer and any of the available servers could be called upon to load a page, this becomes a problem. The problem is that currently our cache is just a local cache, it does not know anything about other caches that may exist on other servers serving the same website. A simplified view of the current architecture is pictured below in Figure 9.1.

Figure 9.1 – Shows the Current, Simple,



As you can see in figure 9.1 our cache is current unaware of any other nodes and has little or no ability to

9.1 Distributing the Cache

In the current implementation of the cache, if a user logs in and establishes a session on webserver one and, as part of a subsequent request, is sent over to webserver two she will all of a sudden be completely without a session and have to log in again; not the best user experience. What is being asked of the cache is that it be able to store the session in such a way as to allow any cache supplied with the correct session key to return the actual session state. It is no longer good enough that our cache knows only about itself and stores values specific to the webserver it serves. We now need to distribute our cache and make each instance aware of the other instances.

9.1.1 – How should our distributed cache communicate?

When distributing our cache there are two flavors of messaging worth talking about; synchronous and asynchronous. Asynchronous messaging is when a message is sent and the sender immediately returns from that send ready to execute further steps along its thread of execution. Synchronous messaging is when a message is sent and the sender suspends any further action until a reply to the message is received. The messaging semantics themselves are not entirely interesting on their own, it is the type of systems that can be built with each that makes understanding these two communication types worthwhile.

SYNCHRONOUS/ASYNCHRONOUS CONFUSION

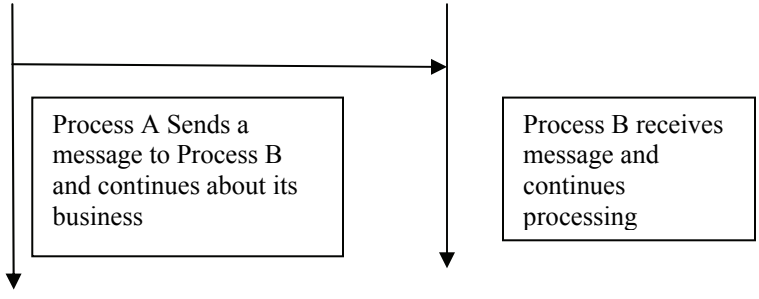
It is important not to confuse "Synchronous/Asynchronous messaging" with a "Synchronous/Asynchronous distribution model". They are quite different.

The next couple of sections take us a little deeper into what characterize these communication methods, when to use each, and the type of systems each are used in conjunction with.

ASYNCHRONOUS MESSAGING

The definition of asynchronous messaging is simply a relay of a message where the sending side does not sit around waiting for a reply. Asynchronous messaging by this definition is sometimes referred to as "fire and forget" communication. Basically a message is sent out, and the sender is free to do something else the instant the message sending operation completes. You can see how this works out in figure 9.2.

Figure 9.2 Asynchronous, Fire and Forget Messaging

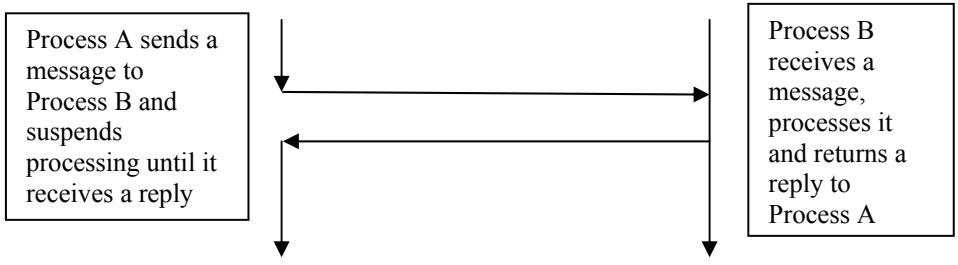


This is an ideal form of communication for computer systems because it implies very little in terms of overhead. There is nothing to check, scan, verify, time, or otherwise care about at all. This means asynchronous messaging is fast, and it generally lends itself to the creation of simple intuitive systems. I strongly recommend using this type communication, except when you can't. Perhaps an example will help make clear the concept of asynchronous messaging and consequently where it makes sense to use it. The postal service is an example of asynchronous messaging. A sender, me for example, sends a letter to my grandmother. I write it, I put it in an envelope, I put a few stamps on it, and then I drop it in the mail box and then I am done, that's it. I go on about my day the second the letter leaves my hand. I am fairly sure the message will get to her, but it may not, either way it is not going to impede my activities for the day. This is really nice because if my grandmother were to not get the message, or she were to read it much later and respond to me only next month, I was not sitting around pacing and waiting that whole time, I was still out being a productive citizen even in the face of all that may have gone wrong with the delivery of my message. In other words this system is efficient, gets work done, even in the face of faults.

SYNCHRONOUS MESSAGING

Synchronous messaging is when the sender sits and waits for a response from the receiver. The sender "blocks" waiting for a reply, meaning he is not doing anything else while the reply is outstanding. You can see an example of this synchronous messaging in the next figure, figure 9.3.

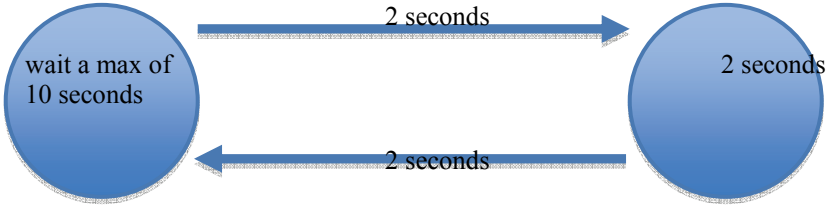
Figure 9.3 Synchronous, Send and Wait, Messaging



Relative to the last sections example and benefits this may sound less than ideal because the caller is not able to perform work while waiting on the reply. The clear benefit of this semantic is that systems can be synchronized around an activity easily. An example may clarify this, let's say a very busy businessman walks into a government office because he has a violation on his car such that if he does not get it cleared then his car could be towed and impounded the second he parks it outside of the government parking lot where he is currently parked. He walks into this office and goes to the desk and asks the clerk to please clear the violation and he hands her a check (the message is delivered). Now, as in the case of the earlier mail box example our fellow is not done, he can't just leave and do other productive work, he must block/wait there in the office for the desk clerk to process the payment and tell him the violation is clear – only then is he safe to drive his car without fear of it being towed the next time he parks it. The system is synchronized to an extent when the reply that the violation is clear. The businessman can leave the office knowing that the part of the system he interacted with is in a known state before he continues doing work.

There is one more thing to understand about synchronous messaging, and that is the use of timeouts. Our fellow in the government traffic authority office needs a maximum waiting time built into him such that he does not sit in the office until he dies of thirst – at some point he must abandon the waiting process consider the operation a failure and potentially return another day or try some other exceptional behavior to accomplish the job he has set out to do. Figure 9.2 illustrates a call within the bounds of a timeout.

Figure 9.2 – synchronous communication



Why bother talking about these different types of messaging? The reason is because we need to know what kind of system our cache is; do we need to guarantee certain states at certain moments? Do we need to know that when a piece of data is inserted or deleted from the system that all instances of the cache reflect the operation as soon as the call to delete or insert returns? It could be the case that we don't strictly need to know, that our system does not need to be synchronized in that way, perhaps it can be more asynchronous such that setting a delete operation in motion and returning before it may be complete is just fine. This will have big implications on how we code the final solution. Next, before we actually get into distributing the cache, we need to spend a little time taking a look at our options with regard to how we communicate.

9.1.2 – Synchronous vs Asynchronous cache

As we talked about in the last section there are two fundamental approaches to distributing the cache. Both approaches have some benefits and drawbacks. That is, of course, going to be true of any and every decision we make when writing software. In this case, the choice between the Asynchronous cache and the Synchronous cache is going to profoundly affect the way we distribute the cache. With that in mind let's take a couple of minutes to go over both approaches to this distribution problem.

ASYNCHRONOUS CACHE

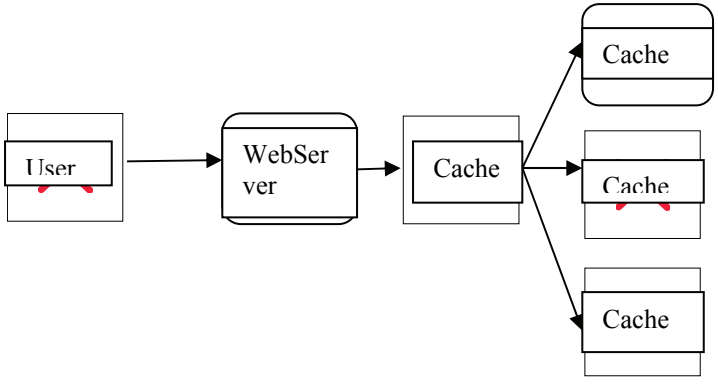
If it would be ok if a person were to log into the site, then make another request through our server infrastructure and find that there was no record of them having been logged in, then we could potentially use non-blocking communication on inserts into the cache. Don't get me wrong, just because we don't guarantee the state of the system upon insert returning, does not mean that insert will fail frequently or be slow, it just means we are not

providing, or holding up the client for an iron clad guarantee. Basically, there would be some small probability that the system as a whole might be in some inconsistent state. The reality is that much of the time the asynchronous messaging based system gets the job done faster than it's blocking, synchronous. This speed could mean that it would be unlikely that someone could log into the site and then it appear to them on a subsequent operation that they were not logged in because the load-balancer directed them to a different web server, one whose cache had not yet picked up knowledge of the previously authenticated session.

In Erlang, an asynchronous messaging based design would be very simple to implement. Remember how we saw in chapter 8 that we can put Erlang nodes in a cloud? Well the asynchronous design would first depend on the fact that all cache nodes were in the same cloud. Once they were it would be easy to broadcast a message to each and every cache in the cloud. This means that we could simply send the insert message to all the cache instances in the cloud upon receiving an insert on any one of the caches. Picture the login sequence.

Figure 9.4 Asynchronous Cache Interaction

1. User logs into the site
2. A session is created on the webserver
3. Webserver calls simple_cache:insert
4. The insert functions sends the insert message asynchronously to all nodes received from calling the nodes() built in function and immediately returns.
5. System informs the user that she is logged in.



How easy is that. Assuming the cache sends the messages faster than the user uses the logged in session to make another request and for that request to subsequently be processed all the way down to the cache hit level, everything will work seamlessly. The big advantage

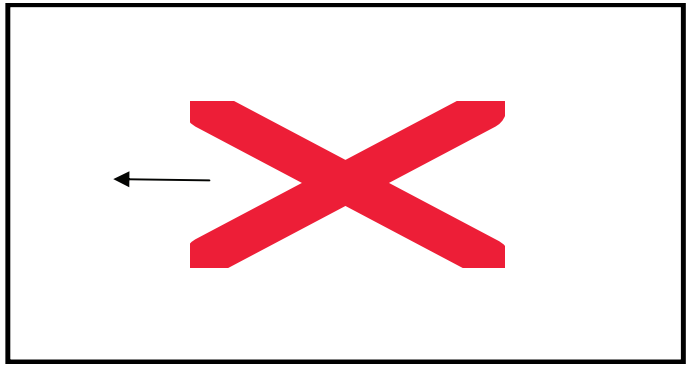
here is the code and operation of this version of the cache will be very simple. Unfortunately for us implementing the cache in thinking about this the Erlware guys feel that it would be unacceptable for even one user of the site to get a "not logged in" message following a successful login weather site load was heavy or not. This means that a synchronous messaging based system is what we need. Ok then, if they want to be that way, let's talk through some synchronous design paths.

SYNCHRONOUS CACHE

Synchronous messaging of our simple cache means that we want verification of the fact that messages were received and data inserted by all instances of the cache before we tell the user that she is logged in.

Figure 9.5 Synchronous Cache Interaction

1. User logs into the site
2. A session is created on the webserver
3. Webserver calls simple_cache:insert
4. The user waits while all cache instances fully insert the data before the high level insert function returns.
5. The system informs the user that she is logged in.

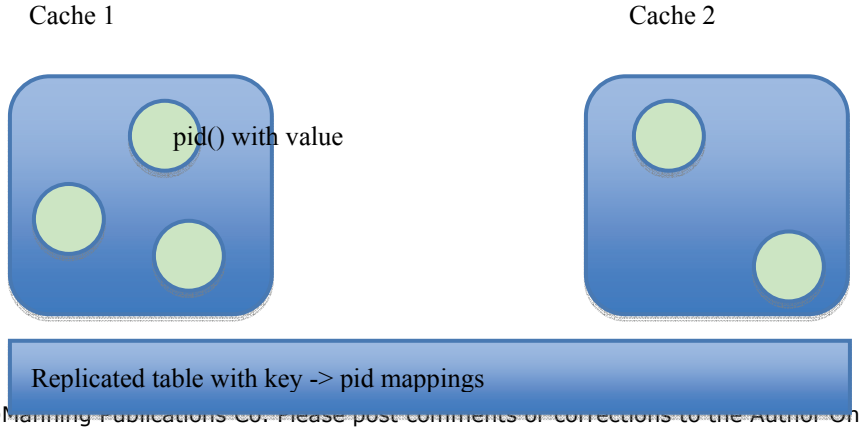


The semantics there are quite different from the asynchronous case. In this flow we can see that the user will not be told that she is logged in until all caches are made aware of the

session data. In this case the insert function can't use fire and forget, it must receive verification from the all of the caches it forwards data onto that they received the data correctly before any timeout occurs at the login screen.

Two different methods occur to me right away in terms of how we could go about implementing this behavior. The first is much the way we would implement the a version of the cache that instead of asynchronously sending out the insert message to all nodes uses a gen_server:call/2 function to do so synchronously. We would loop through all nodes doing synchronously messaged inserts only returning at the top level once all the inserts were completed on all nodes. The second way will require me to show you a new bit of Erlang technology called Mnesia; that will come in the next section, for now all you need to know is it is a distributed database capable of performing atomic operations across all copies of a given distributed table. What I am picturing in my head is a cache that would share only the key -> pid mapping we use to keep track of the processes that hold values and how they relate to particular keys. In this case the values would still only be stored on a single machine and the key to pid mapping would go into a distributed table that would operate synchronously underneath. Figure 9.3 illustrates where the data store will sit in relation to our simple cache instances.

Figure 9.3 – 2 cache instances sharing a replicated table



When the call to insert into the table returned we would have verification that the key and pid mapping was sent and stored across all known nodes. The beauty of location transparency is then that once we do a lookup on this table locally accessing a given pid for a key, it does not matter when the process associated with the pid sits, we simply query it for its value data and the message is delivered whether it is local or on another node. To make this scheme even more interesting we are going to use the resource discovery system we constructed in the last chapter to keep track of the cache nodes and differentiate them from any other service types we may want to have running in our cluster. If that sounds like a lot, you are right, it is a lot. We need to get moving or we will never get this all done and the first step in getting this all moving is an explanation of Mnesia and how it works.

So, for various reasons, we have decided on a synchronous approach using an Erlang technology called Mnesia. We have a rudimentary design sketched out and can move into making that design a reality. In the process, we get to teach you quite a bit about Mnesia and how to use it for purposes such as this. We took our new design and ran it by the Erlware guys. The guys loved the design and told us to go ahead, we of course would not just want to go on and implement something without stakeholder approval.

9.2 Distributed Data Storage with Mnesia

How Mnesia Got Its Name

Mnesia, what an odd sounding name, When it was first created the developer in charge decided that he would like to call it Amnesia! Management quickly informed him that there would be no database named Amnesia in their shop. So the developer kindly chopped off the pesky "a" and to this day we have Mnesia.

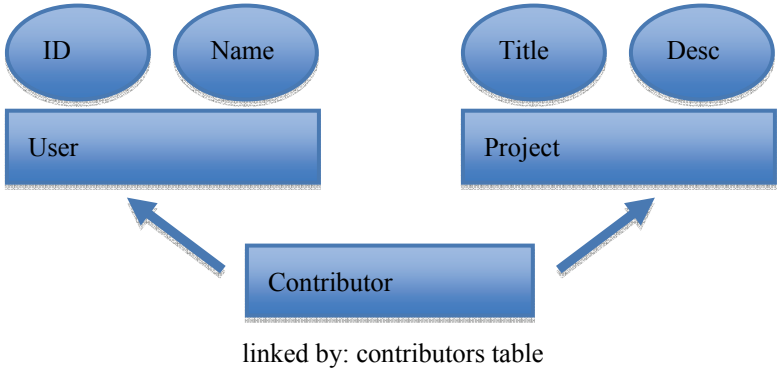
Mnesia is a lightweight, soft real time, distributed, replicated, data store. It is fantastic at storing discrete chunks of data at runtime. It is not meant to hold and persist 675 gigabytes over 200 instances of itself. There are stories of this being done, but I recommend against you trying yourself. Mnesia is fantastic at lower numbers of instances and smaller quickly changing units of data. Mnesia is best used for runtime data that you want to make fault tolerant by distributing. Key to pid mappings are a great example of this. The ability to quickly, and I mean minutes here, bring up a fault tolerant, replicated data store is something that will shock you at first. One of the best ways to learn is by example, so we are going to create a real world working Mnesia database in the next section. By creating

this database you will become familiar with Mnesia. This will allow us to move on and actually distribute our key -> pid store with Mnesia itself.

9.2.1 – Creating a Project Database

In this section we are going to create a database that could be used to store information about projects on an Erlware project repository server. This basic version will house information about users and which projects they own. The information will be stored in tables, you know, the old row/column thingy. Figure 9.2 illustrates the relationships between the data we wish to model. As you can see from the figure the User and Project tables which have two columns a piece are joined by the contributor table.

Figure 9.4 – Data model



To model these tables we will use records to define the structure of Mnesia tables. Listing 9.1 defines the records that underpin our tables.

9.1 – Project database record definitions

```
-record(user, {
               id,
               name
              }).

-record(project, {
                  title,
                  description
                 }).

-record(contributor, {
```

```
    user_id,
    title
  }).
```

These record definitions will be used as is when we create our tables further on in this section. Before we get to creating our tables there is a bit of work we have to do. During the course of creating this database we will cover:

1. Initialize mnesia
    a. start our node
    b. create a schema
    c. start mnesia itself
2. Create database tables
3. Populate the tables we have created
4. Do some basic queries on our data

So let's get started with initializing Mnesia.

9.2.2 – Initializing the Mnesia Database

Before getting into the business of creating our application specific structure within Mnesia we need to do some rather more generic initialization of the database. That starts with bringing up an Erlang node that is setup to write Mnesia information to the file system.

STARTING THE NODE

The first thing that needs to be done when using Mnesia is to start the Erlang node like this:

```
erl -mnesia dir '"/tmp/mnesia_store"' -name mynode
```

This tells Mnesia to store its data in the directory supplied, using the application config command line syntax. It also tells Erlang that we are starting in distributed mode. Once the node is up we need to create an empty schema on the nodes we care to replicate with at startup.

CREATE THE SCHEMA

Mnesia requires this thing called a schema. For the most part you can ignore this, its just what Mnesia uses to keep track of its data. This schema will get placed into the directory we specified at node startup. This operation will fail if any of the nodes we are trying to create a schema on is not running, has Mnesia already running or has a previously existing schema. In the case of a previously existing schema the mnesia:delete_schema/1 function can be used to purge the old schema. In this simple example we are only going to configure one node for schema creation at startup; our own local node.

```
mnesia:create_schema([node()]).
```

after running that command, if successful, we have a blank schema created on our node. From here all we need to do is start Mnesia itself.

STARTING MNESIA

To run Mnesia and to run any of the previous commands Mnesia must be available in your release. Its available, by default, on the VM the is started as part of the standard Erlang shell. If you are creating and testing as part of a custom release package then this means that Mnesia has to be a dependency of your release. To start Mnesia manually you run the mnesia:start/0 function. Once Mnesia is started you can run mnesia:info/0 to discover information about the running system such as how many tables it contains and how many nodes it is connected to, etc… Listing 9.2 is the output of Mnesia info/0 on our shell.

Listing 9.2 – Mnesia info output

```
(mynode@erlware.org)3> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---
---> Uncertain transactions <---
---> Active tables <---
schema         : with 1        records occupying 422      words of mem
===> System info in version "4.4.8", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [mynode@erlware.org]
stopped db nodes   = []
master node tables = []
remote             = []
ram_copies         = []
disc_copies        = [schema]
disc_only_copies   = []
[{mynode@erlware.org,disc_copies}] = [schema]
2 transactions committed, 0 aborted, 0 restarted, 0 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
```

```
ok
```

As you can see it gives you quite a lot of information about the state of the running system. This will come in handy as you are working to ensure that you have proper connectedness and configuration applied to your system. With the db initialized we can get into our application specific code, starting with creating our table structure within the empty schema.

9.2.3 – Creating the Tables

In this section we create a short module to initialize our database tables for us. Doing this on the shell would be a bit ugly. The module is listed below in listing 9.3.

Listing 9.3 – Mnesia table creation module

```
-module(create_tables).

-export([init_tables/0]).

-record(user, {
          id,
          name
         }).

-record(project, {
          title,
          description
         }).

-record(contributor, {
          user_id,
          project_title
         }).

init_tables() ->
    mnesia:create_table(user, [{attributes, record_info(fields, user)}]),
    mnesia:create_table(project,
       [{attributes, record_info(fields, project)}]),
    mnesia:create_table(contributor,
         [{type, bag}, {attributes, record_info(fields, contributor)}]).
```

I am going to harp on this again, because that is what I do. I have not included inline documentation for the init_tables API function here in order to save space. I would not actually release code like this and neither should you! Ok, back to our regularly scheduled explanation… What follows is the shell interaction to get this compiled and run, which is then followed by the Mnesia info function output to check to ensure our init_tables/0 function does what we want it to. Listing 9.4 manually runs through running the code we just wrote.

Listing 9.4 – Creating Our Tables

```
(mynode@erlware.org)7> c(create_tables).
{ok,create_tables}
(mynode@erlware.org)8> create_tables:init_tables().
{atomic,ok}
(mynode@erlware.org)9> mnesia:info().
---> Processes holding locks <---
---> Processes waiting for locks <---
---> Participant transactions <---
---> Coordinator transactions <---
---> Uncertain transactions <---
 ---> Active tables <--- #1
contributor    : with 0        records occupying 312      words of mem
project        : with 0        records occupying 312      words of mem
user           : with 0        records occupying 312      words of mem
schema         : with 4        records occupying 752      words of mem
===> System info in version "4.4.8", debug level = none <===
opt_disc. Directory "/tmp/mnesia" is used.
use fallback at restart = false
running db nodes   = [mynode@erlware.org]
stopped db nodes   = []
master node tables = []
remote             = []
 ram_copies       = [contributor,project,user] #2
disc_copies        = [schema]
disc_only_copies   = []
[{mynode@erlware.org,disc_copies}] = [schema]
[{mynode@erlware.org,ram_copies}] = [user,project,contributor]
5 transactions committed, 3 aborted, 0 restarted, 3 logged to disc
0 held locks, 0 in queue; 0 local transactions, 0 remote
0 transactions waits for other nodes: []
ok
```

Code compiled, init_tables/0 ran without error and mnesia:info/0 shows us what we have. #1 we can see we have 4 tables created and active; the schema itself as well as contributor, project and project. #2 shows us that our application tables are or the default type, ram copies. This means that they are only stored in memory. This type offers the highest performance. The line below indicates that our schema itself is a disc copy and consequently can survive a restart of our node with data intact something that our ram copy application level tables can't do. Tables can be configured for different storage types upon creation and can also be altered at runtime. I think that last sentence offers a nice segue into a discussion of the mnesia:create_table/2 function which we used to create these very tables.

The create_table/2 function takes two arguments Name and TableDefinition and it creates a table called Name from the information specified in TableDefinition. Tables in Mnesia can

come in a number of forms. The most vanilla, the default, is what we created when we created the user and the project tables.

```
mnesia:create_table(user, [{attributes, record_info(fields, user)}])
```

The line above is that which we used to create the user table. It only specifies the attributes option in its table definition. Within the attributes option we use the record_info/2 function to extract the names of the fields from the record it is supplied in order to use them as the names of the fields in our actual table. This process can be hardcoded but it lacks elegance. Only specifying the attributes option means that our table inherits the default properties for all other aspects of the table. It is:

- readable and writeable
- it is stored in ram; type ram_copy
- it has a load priority of 0 (the lowest)
- only records of the same name as the table name can be stored within
- it is of type "set" meaning keys must be unique
- local_content is set to false

I want to give a little bit more attention to a few of these options.

STORAGE TYPE

The storage type may be, ram_copy, disc_copy, or disk_only_copy. Our tables all default to ram_copy. This means that they are stored in ram and as speedy as Mnesia tables get. They, of course, will not survive a node crash. If it is desirable to survive a node crash a set the table to be a disc_copy (disk and ram) or a disc_only_copy within the table definition. The disc_copy means that though the data is stored in ram it is persisted to disc. The disc_only_copy means that the data only exists no disc. This has a tendency to be rather slow, but may be required for large data sets. The second attribute that bears further attention is the table type.

TABLE TYPE

The type option can either be set, ordered_set, or bag. 'set' means that keys, by default the first field in the table definition record, must be unique. If a key is inserted into the table with key 'x' subsequent inserts of the same key will overwrite those inserted previously. The ordered_set option stores all the elements within the tables ordered on their keys. This can be useful in a few situations but is not in practiced used all too often. Furthermore it is not supported for disc_only_nodes. The bag option allows the table to store records for which the keys are not unique. The records themselves must be different in some way but the

keys can be repeated. Records that are complete clones of other records in the tables will only be stored once. Notice that the table contributor uses the bag semantics:

```
mnesia:create_table(contributor,
    [{type, bag}, {attributes, record_info(fields, contributor)}]).
```

This allows this table to hold many records for a particular user that happens to be a contributor on many projects.

With our tables created and containing the proper attributes the next logical step is to get data inserted into each of them. This will be completed with a small amount of code in the next section.

9.2.4 – Populating Our Tables

We are not going to allow the insertion of users into the database if they are not contributors to at least one project. A user may of course be a contributor to many projects, but never zero. The code for accomplishing the insertion of users and projects is listed in listing 9.5.

Listing 9.5 – Data Insertion Functions

```
insert_user(Id, Name, ProjTitles) ->
    User = #user{id = Id, name = Name},
    Fun = fun() ->
            mnesia:write(User),
            lists:foreach(
              fun(Title) ->
                [#project{title = Title}] = mnesia:read(project, Title),
                  mnesia:write(
                    #contributor{user_id = Id, project_title = Title})
              end,
              ProjTitles)
          end,
    mnesia:transaction(Fun).

insert_project(Title, Description) ->
    mnesia:dirty_write(#project{title = Title, description = Description}).
```

What we have above will get data into our database. The first function insert_user/3 takes three arguments, the unique user id of the user we are creating, the name of the user we are creating, and a list of all the project titles that this user contributes to. This function creates an anonymous function, lambda function, or as it is commonly referred to in Erlang a

fun. This fun executes within the context of a Mnesia transaction. Mnesia transactions are important in two ways;

1. it is executed as a unit, all operations either succeed or the whole transaction fails.
2. It is capable of locking information in the db so that concurrent operations on the db don't negatively impact one another.

These transactions are critical for ensuring the integrity of the database across complex operations. The transaction here first writes in the user record to the user table and then users the foreach higher order function to check to ensure that contributor records for each project in the ProjList are written if and only if a corresponding project has already been entered into the project table. If any of these operations fails the whole transaction fails and the Mnesia is left in the same state as it was prior to executing the insert_user/3 function. Because project records must be in the database prior to adding users we need a way to insert projects. This is done with the insert_project/2 function. Notice the user of the mnesia:dirty_write function. Any Mnesia function with the prefix dirty is what is called a dirty operation. Dirty operations do not respect transactions or locks on the database. This means they must be used with care. They are generally quite fast because they dispense with transaction overhead and this is why they are used when they are. Use with care. Running the following on the shell will ensure that we have a single project, user, and contributor record each in the database.

```
(martin@erlware.org)4> create_tables:insert_project(simple_cache, "a simple
cache").
(martin@erlware.org)5> create_tables:insert_user(1,martin,[simple_cache]).
```

The records we added will look like this:

Project

Title	description
simple_cache	"a simple cache"

User

Id	name
1	martin

Contributor

user_id	project_title
Martin	simple_cache

With our data now in the database let's look at some ways that we can extract and view the data in order to prove it is in fact there.

9.2.5 – Do Some Basic Queries on Our Data

The create_tables:insert_user/3 function already uses Mnesia read within a transaction to pull data from the project table. Dirty operations can also be used to read the database outside of a transaction as in the following excerpt from a shell session.

```
(martin@erlware.org)5> create_tables:dirty_read(contributor, 1).
[{contributor, 1, simple_cache}]
```

The function returns a list of contributor records. Remember records are really just tuples with the first element being the record name. If we had added more than one project to user 1, martin, we would see the list returned come back with more than one record. Remember that this is possible on the contributor table because it is a "bag". Read operations, like dirty_read select via the key of the table. There are other operations like select that offer a bit more flexibility such as the select operation. Below is a select capable of pulling potentially many records from the user table.

```
mnesia:transaction(
  fun() ->
    mnesia:select(user, [{#user{id = '$1', name = martin}, [], [$1]}])
  end)
```

which will yield, upon success, the tuple keyed by the atom atomic and followed by a list of the contents of the field that we indicated via the atom $1.

```
{atomic,  [martin]}
```

As you have probably guessed it is quite possible to add '$2' up to '$n' to pull other fields and have them returned as well. There is another, perhaps more powerful, at least more expressive way to query Mnesia and that is through the use of QLC. QLC queries via list comprehensions. The mnesia:table/1 helper function is used to represent the contents of the table to the comprehension and then standard list comprehensions filtering and aggregation syntax is used. To accomplish the same thing as we accomplished with our select function above using QLC we would write:

```
mnesia:transaction(
  fun() ->
    qlc:q([U#user.id || U <- mnesia:table(user), U#user.name == martin])
  end)
```

Getting comfortable with list comprehensions is worth while on its own but this is certainly another case for doing so. List comprehensions are a considerably more elegant way to query the db than are Mnesia selects in my humble opinion. Reading this comprehension is very easy, basically it states that we must create a list containing the contents of U#user.id where U is a record from the table user and U#user.name is equal to martin. These QLC type queries can also be used in transactions where reading is not the only thing going on, they can be mixed in with any other type of Mnesia function that belongs in a transaction.

What we have covered here is by no means exhaustive coverage of the Mnesia soft real time store. For that we need another book. What this does do is give us enough of a foundation that we can move forward distributing our cache based upon the Mnesia store and that is exactly what we will do next.

9.3 – Distributing the Cache with Mnesia

Now you have a solid understanding of the basics of Mnesia and we have a descent high-level design for our cache. We have what we need to dive into the implementation of our distributed cache. For this cache to work properly we need to address the following points:

1. Convert from ETS to Mnesia
2. Make the cache aware of the other nodes it must communicate with
3. Bring the Mnesia Tables into dynamic replication

We will address number one right away in the next section so that by the end of it, we are not using ETS for our cache table anymore but Mnesia.

9.3.1 – Conversion from ETS to Mnesia

Remember the sc_store module from the simple_cache we implemented in chapter 6? This was the module that we used to encapsulate all of our storage functions. This module hid the implementation of our store from the rest of our codebase. Now this encapsulation comes in handy. The rest of our codebase will not have to change as a result of this complete storage conversion. The sc_store module contained 4 key functions:

1. init/0
2. insert/2
3. lookup/1

4. delete/1

Init is going to be used to setup our Mnesia table, just like it does now for ETS. We will then modify init even further to encapsulate the logic required to bring distributed cache nodes into replication. For now though, it is just going to create our table.

REWRITING INIT/0

init/0 before:

```
init() ->
    ets:new(?TABLE_ID, [public, named_table]),
    ok.
```

and now, init/0 after its Mnesia makeover:

```
init() ->
     mnesia:create_schema([node()]),
     mnesia:start(),
     mnesia:create_table(key_to_pid,
        [{index, [pid]}, {attributes, record_info(fields, key_to_pid)}]).
```

Notice the use of record_info/1 on the 'key_to_pid' record… this of course means that we need to have a key to pid record. Since this file is to encapsulate all storage concerns for our cache this record is defined here as:

```
-record(key_to_pid, {key, pid}).
```

Secondly now notice the {index [pid]} option on the table definition. Indexes are extra tables that allow speedy operations on non key fields within a table. Remember that when creating an index an extra table is created, and that table consumes extra space. Furthermore that table must be populated and kept up to date with each insert into the primary table, which means that insert operations become just that much slower. It is important to be aware of these tradeoffs. With that done and said lets go on down the line and rewrite insert.

REWRITING INSERT/2

Just like the last section on rewriting init/0 let's start with a before picture of the insert/2 function.

```
insert(Key, Pid) when is_pid(Pid) ->
    ets:insert(?TABLE_ID, {Key, Pid}).
```

And now for the reveal:

```
insert(Key, Pid) when is_pid(Pid) ->
    mnesia:write(#key_to_pid{key = Key, pid = Pid}).
```

Pretty straight forward all in all. No transactions needed in this case, just a straight write to our table. Delete and lookup are left to convert. We will handle them together in the next section.

REWRITING LOOKUP/1 AND DELETE/1

Before:

```
lookup(Key) ->
    case ets:lookup(?TABLE_ID, Key) of
     [{Key, Pid}] -> {ok, Pid};
        []        -> {error, not_found}
    end.


delete(Pid) ->
    ets:delete_object(?TABLE_ID, {'_', Pid}).
```

and after:

```
lookup(Key) ->
    Fun = fun() ->
                  [{key_to_pid, Key, Pid}] = mnesia:read(key_to_pid, Key),
                  Pid
          end,
    case mnesia:transaction(Fun) of
        {atomic, Pid}      -> {ok, Pid};
        {aborted, _Reason} -> {error, not_found}
    end.

delete(Pid) ->
    try
        [#key_to_pid{} = Record] =
      mnesia:dirty_index_read(key_to_pid, Pid, #key_to_pid.pid), #1
        mnesia:dirty_delete_object(Record)
    catch
        _C:_E -> ok
    end.
```

Take a look at code annotation #1. This is how an index is used. Special index aware functions, this being the dirty variety, are employed to delve into indexes. dirty_index_read/3 takes as its first argument the table name to look into, then it takes the index key, and finally it takes an indication of which index on the table it needs to look into.

This is defined as one of the columns on the table which in our case is the pid column. Secondly, notice that this operation is prefixed with dirty_. This means that the operation is fast, but it also means that it does not respect transactional boundaries. Use these sorts of operations with care.

That really was not all too bad was it? We now have a fully Mnesia enabled cache and we did not have to change a single like outside the sc_store module to accomplish it. Lets see if we can stay on the track of keeping things clean and simple as we tackle the steps below in the next three sections.

1. Make the cache aware of the other nodes, make it aware of a cloud
2. Implement Resource Discovery for the cache
3. Implement the logic required to bring the tables into replication

9.3.2 – Make the Cache Aware of Other Nodes

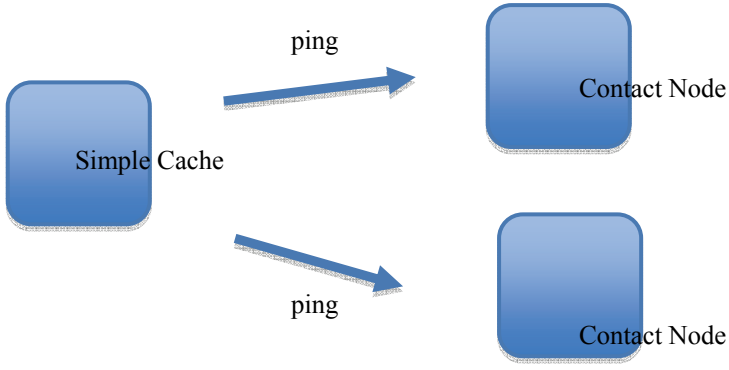
In the following sections we are going to make the cache aware of all the other cache instances within its cloud of nodes. Doing this lays the ground work for synching up with those instances so that we can share data between them. So first of all that it needs to get into a cloud of nodes and then use resource discovery to find all other simple_cache resources in that cloud. In this section I am going to explain a simple way to manage that. There are more advanced ways to do this but they are beyond the scope of this book – and this works quite well anyhow.

This simple method for bringing the cache into a cloud of nodes is to always have two blank Erlang nodes up at startup. Basically nodes started without any user defined code running on them, basically just start them vanilla as in

```
erl -name <some-node-name> -setcookie <cloud-cookie>
```

These two nodes must always be up. Each cache node we bring up will be configured to ping both of these nodes. If one of the net_adm:ping/1 calls succeeds then startup is allowed to progress, if not, then startup is considered a failure because the node could not join its intended cloud; the node fails with a crash dump. It should not be hard to decide where to add the code for this – we covered this already – think about where we put our storage initialization function call… We will use the same place, the the start/2 function in the sc_app module. The function that I will include below in a minute is pretty simple. All we are going to do is pull two nodes out of our configuration, and we are going to ping them. The figure illustrates the overall architecture here.

Figure 9.5 – contact nodes and the simple cache



Starting with the configuration we will add the following entry for our application:

```
{simple_cache, [
                {contact_nodes, ['contact1@erlware.org',
                                 'contact2@erlware.org']
                }]}
```

Next up, in listing 9.6, is the actual function we will add to the sc_app module.

Listing 9.6 – Making Contact in sc_app

```
ensure_contact() ->  #1
    case get_env(simple_cache, contact_nodes, []) of
        {ok, []} ->
            error_logger:info_msg("no contact nodes~n");
        {ok, ContactNodes} ->
            ok = ensure_contact(ContactNodes),
            {ok, WaitTime} = get_env(simple_cache, wait_time, 6000),
            wait_for_nodes(length(ContactNodes), WaitTime)
    end.

ensure_contact([Node|T]) ->
    case net_adm:ping(Node) of
        pong ->
```

```
            lists:foreach(fun(N) -> net_adm:ping(N) end, T);
        pang ->
            ensure_contact(T)
    end;
ensure_contact([]) ->
    {error, no_contact_nodes_reachable}.

wait_for_nodes(ContactNodes, WaitTime) ->
    wait_for_nodes(ContactNodes, round(WaitTime / 3), 3).

wait_for_nodes(_, _, 0) ->
    ok;
wait_for_nodes(ContactNodes, WaitSlice, Iterations) ->
    case length(nodes()) > length(ContactNodes) of
        true ->
            ok;
        false ->
            timer:sleep(WaitSlice),
            wait_for_nodes(ContactNodes, WaitSlice, Iterations - 1)
    end.

get_env(AppName, Key, Default) ->
    case application:get_env(AppName, Key) of
        undefined -> {ok, Default};
        Found     -> Found
    end.
```

ensure_contact/0 (annotation #1) pulls config using the application module. The syntax for this call is first <application-name> in our case simple cache. The second argument is the config key to pull and finally we have passed in a default value in case nothing was specified in configuration, in this case the empty list.

CARE WHEN HANDLING CONFIGURATION

Configuration is a side effect, meaning it creates functions that are not referentially transparent. What we pass into ensure contact is not the only thing that effects what it returns. Good functional programmers don't burry side effects deeply in their code. They keep them at the top and easily visible. In this case ensure_contact/0 is called directly from the start/2 API function, and even that makes me just a little nervous…

> Following this practice will make your code much more manageable and refactorable. The more referentially transparent functions you have the easier it is to reason about your code and refactor it. If function that is not referentially transparent is buried down deep in code it means that all the functions that rely on that function are also not referentially transparent, and that is bad.

We pull 2 different bits of config, the nodes to contact, and how long to wait for a fully connected network to be created. ensure_contact/2 takes the list of node names we get back from config and tries to use net_adm:ping/1 to contact them. Once this function gets a successful ping it returns with ok, but not before it quickly pings the other contact nodes listed just to help keep things in sync. If the function never receives a pong from net_adm:ping/1 for any of the contact nodes it returns an error. Notice the use of the pattern match ok = with the call to ensure_contact/2 to create a succeed or die contract. If we don't get an ok back our application throw a badmatch exception and will consequently never start. Next the function wait_for_nodes/2 is called en order to efficiently allow us to wait for a fully connected network to be created. This happens asynchronously and so we want a reasonable time for it to happen. The wait_for_nodes/2 function does exactly this. The next bit of code in listing 9.7 is the start/2 function modified to call ensure_contact/0.

Listing 9.7 Ensuring Contact with Remote Nodes

```
start(_StartType, _StartArgs) ->
    ensure_contact(),       #1
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.
```

Code annotation #1 shows where the new line was added. Now we ensure that we have a fully connected network of nodes before we start our application supervision tree and proceed with the business of being a simple cache. At this point we have a Mnesia enabled cache that starts up and gets itself into an Erlang cloud of nodes. Our next step then is to install resource discovery so that we can single out the other simple_cache instances from any other service types that may exist in our cloud. Once we have that accomplished we will modify our Mnesia system to connect and replicate with the other nodes in the cloud. First things first though, let's get down to using our resource discovery system.

9.3.3 – Integrate resource discovery to find other cache instances

In this section we will get the resource discovery system we built in the last chapter integrated into our simple_cache application. Resource discovery does not really fit into our simple cache per say. It is more generic than that, it could really be used by any application that, like the simple cache, needs to find other service instances within an Erlang cloud of resources. Given this property of resource discovery it makes sense to package it up in its very own application and include it along side the simple cache in our directory structure. I am not going to go deep into the details of creating an application because we have done that already with simple_cache itself. Suffice it to say that we create an app behaviour module, a supervisor behaviour module which starts the resource discovery process we write in chapter 8, and finally we create an interface module named resource_discovery for consumers of our application to interact with. Once all of that is created we should have two applications as follows in the next listing:

```
|-lib
     |- simple_cache
          |- src
          |- ebin
          |- ...
     |-  resource_discovery
          |- src
          |- ebin
          |- ...
```

The ebin directory of both resource discovery and simple cache of course contain .app files. Before we can continue coding away we need to specify that simple_cache depends on resource_discovery. The code listing below illustrates exactly how to do that.

Listing 9.8 – Modified .app File

```
{application, simple_cache,
  [{description, "A simple simple caching system"},
   {vsn, "0.3.0"},
   {modules, [simple_cache,
              sc_app,
              sc_sup,
              sc_element_sup,
              sc_store,
              sc_element,
              sc_event,
              sc_event_logger,
              sc_event_guard]},
   {registered,[sc_sup]},
   {applications, [kernel, stdlib, resource_discovery]},      #1
   {mod, {sc_app,[]}},
   {start_phases, []}]}.
```

Notice at annotation #1 that we have added resource discovery as a dependency.

With the boilerplate out of the way how do we go about getting a hold of the other cache instances in the cloud of nodes. Not a problem, after going through chapter 8 and building our resource discovery system this answer should come right to mind. We will simply add an "I have" or more exactly we add a local resource where the resource_type() is 'simple_cache' and the resource_instance() is node(), our local node. We do that simply by calling:

```
resource_discovery:add_local_resource(simple_cache, node())
```

The next step is to tell the system about our "I want" or target resource type which is simple_cache as well. We do this by adding the line:

```
resource_discovery:add_target_resource_type(simple_cache)
```

The third and final step to making us simple_cache aware is to add in the call to trade resources with the rest of the cloud and then to wait a reasonable time for those resources to be shared. Remember our resource discovery system is very asynchronous. I leave it to the reader to improve on this timeout model and make the resource discovery system more synchronous if need be.

```
resource_discovery:trade_resources(),
timer:sleep(?WAIT_FOR_RESOURCES),
```

This code, predictably I am sure, goes into our sc_app file right after the call to join the cloud of nodes. This is listed in listing 9.9:

listing 9.9 – sc_app With Resource Discovery

```
start(_StartType, _StartArgs) ->
    ensure_contact(),
    sc_resource_discovery:add_local_resource(simple_cache, node()),
    sc_resource_discovery:add_target_resource_type(simple_cache),
    sc_resource_discovery:trade_resources(),
    timer:sleep(?WAIT_FOR_RESOURCES),
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.
```

How simple was that! We now know about all the other simple cache instances in our network and they know about us. Time to replicate.

9.3.4 – Bring the Mnesia Tables into Dynamic Replication

Ok now to complete the magic. I just want to take a moment to revel in what we are about to do here, think about it. With almost no configuration, just some bootstrap to get us into a cloud of nodes, we are automatically discovering all simple_cache instances on the network and then replicating data across them providing us with a high degree of fault tolerance and nicely dynamic system. Getting the nodes into replication is not too touch, it just requires a bit of knowledge about the way Mnesia and it's schemas in the distributed context. The code to accomplish our dynamic syncing of caches is going to be put into sc_store:init/0 which will now rely on the fact that we have a fully connected cloud of nodes and that we have a fully populated resource discovery cache. The code that I am adding to sc_store is not entirely trivial so I will break the code up in to a few parts and discuss each separately.

```
init() ->
    {ok, CacheNodes} = sc_resource_discovery:fetch_resources(simple_cache),
    dynamic_db_init(lists:delete(node(), CacheNodes)).
```

Init now relies on resource discovery operations having been completed before it can fun. The first thing it does is pull all the CacheNodes that resource discovery knows about. Once it has them it calls the function we will explain next, dynamic_db_init/1. This function will either initialize the database in standalone mode, if no other simple_cache instances exist in the cloud, or it will sync with the already existing simple cache nodes if any are found. Notice that the local node is deleted from the list of CacheNodes, this is because resource discovery finds resources on the local node in the same way it finds them on remote ones. We of course don't need to replicate with the local node so it is stripped here. The code for dynamic_db_init/1 follows in listing 9.10

Listing 9.10 – dynamic_db_init/1

```
dynamic_db_init([]) ->   #1
    delete_schema(),
    mnesia:create_schema([node()]),
    mnesia:create_table(key_to_pid, [{index, [pid]},
                                     {attributes,
                                      record_info(fields, key_to_pid)}]);
dynamic_db_init(CacheNodes) -> #2
    delete_schema(),
    add_extra_nodes(CacheNodes).

%% deletes a local schema.
delete_schema() ->
    mnesia:stop(),
    mnesia:delete_schema([node()]),
    mnesia:start().
```

If no CacheNodes are found (annotation #1) we enter into the first clause of the dynamic_db_init/1 function. The first step therein is to delete any preexisting schemas. This incarnation of the cache takes a very cavalear attitude towards data in that it does not seek to use previously used schemas or data. We are storing all of our cached data in as ram_copies so no data is on disc at all – this is a cache afterall. Once the schema is deleted we create a fresh local schema and add our table to it. At this pint we have a working simple_cache that is ready to replicate its data to any other simple_cache instances that join the cloud.

If we had found other simple cache instances then we would drop into the second clause (annotation #2) of dynamic_db_init/1. It is in this clause that we bring data over from the other nodes in the cloud. Similar to the stand alone clause we first delete any old local schema. Once that is done we call the add_extra_nodes/1 function to do just that. This function is listed in listing 9.11.

Listing 9.11 - add_extra_nodes/1

```
add_extra_nodes([Node|T]) ->
   case mnesia:change_config(extra_db_nodes, [Node]) of    #1
         {ok, [Node]} ->
      Res = mnesia:add_table_copy(schema, node(), ram_copies),  #2
            error_logger:info_msg("remote_init schema type ~p~n", [Res]),

      Res1 = mnesia:add_table_copy(key_to_pid, node(), ram_copies), #3
            error_logger:info_msg("remote_init add_table copy = ~p~n",
                                  [Res1]),

      Tables = mnesia:system_info(tables), #4
            mnesia:wait_for_tables(Tables, ?WAIT_FOR_TABLES);
        _ ->
            add_extra_nodes(T)
    end.
```

This function does quite a bit, but is pretty straightforward in the end. First we run mnesia:change_config (annotation #1) and tell it to add an extra db node. We only need to do this to one of the remote simple cache instances. Mnesia works in much the same way as Erlang nodes themselves. When we bring over the remote schema from one of the other connected nodes we will be informed of the others, and the others about us. If that operation returns successfully we first pull over the remote schema (annotation #2) adding a copy for the local node. Second we do the same thing for the key_to_pid (annotation #3) table that stores our cache data. Finally we call Mnesia:system_info/1 to get a list of all the tables we have added and use the Mnesia:wait_for_tables function to await a full sync of the data in the newly added table (annotation #4). That's all folks, there we have it. A dynamically replicating cache. Go ahead and give it a whirl on the shell. Start up two nodes, get them talking, insert on one and query on the other.

We have now built a system that stores keys and PIDs into replicated storage. This replicated storage provides a key to any cache on the system. This allows any cache to look up, and subsequently communicate with the process, via its pid, that holds the value associated with the given key.

9.3 – Summary

We have done more than a little in this chapter. We have learned quite a bit about the powerful distributed soft real time data store called Mnesia. We have altered our cache to use that system for storage while minimally impacting the rest of the code base because of our smart hiding of the db implementation in the sc_store module. We have written some simple code to make our cache cloud aware. Most impressively we have incorporated the resource discovery system we built in chapter 8 and used to ultimately get Mnesia to come into dynamic replication with the other database nodes that are found in the cloud. The Erlware guys are going to be really thrilled because they can now do session storage and not have to worry about providing a bad or choppy user experience.

In the next chapter we will close the loop. This code will be taken and brought into a real release, ready to be pushed out into production. After the next chapter you will be on your way, really on your way, to becoming an Erlang professional.

10

Packaging, Services & Deployment

You have learned how to handle logging and configuration, how to distribute your apps, even how and when to use Mnesia. That's no small accomplishment. Now our task is to take that code and wrap it in a way that Erlang and OTP expects. This will make it packaged and ready to use. We introduced you to some of these ways and tools back in chapter 3 when we were talking about Applications. OTP Applications (as opposed to the more conventional general sense the word Applications is used in), provide small focused units of functionality. Usually, you must tie multiple applications together to get a useful system. The implementers of OTP have given us a way to tie many Applications together into a higher level package called a Release. Remember that Applications are these living things that have a lifecycle all their own. With that in mind, Releases also provide a means of managing the lifecycle of the Applications that are included in them. Let's do a quick refresher on Applications before we get into Releases. Applications form the building blocks for Releases and Releases won't make a lot of sense with if you don't have a thorough understanding of Applications.

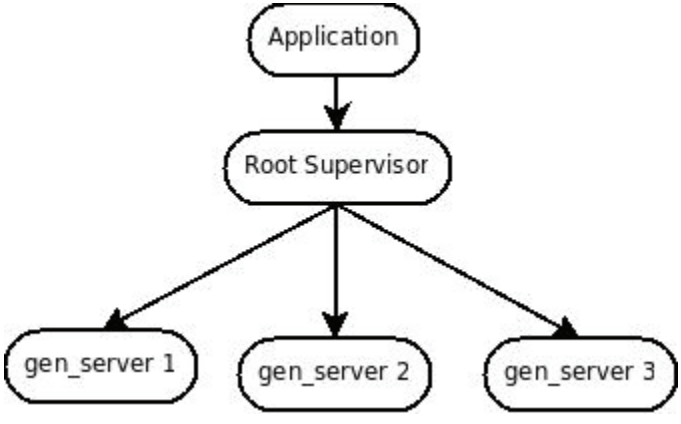
10.1 - Application Refresher

We have complained on more than one occasion about how the OTP implementers went about naming parts of their system. We complain a lot about how OTP 'Applications' are called 'Applications'. That is just huge name collision with all the rest of the world. However there is a certain logic to the name 'Applications'. OTP applications are usually living things, things that run, things with a discrete lifecycle and behavior. They are much more like the applications that the rest of the world knows than the libraries most people are used to when they think of the libraries in most languages. Think about this, when you start up an OTP Application you are starting a set of long lived processes. These long lived processes are

going to run and do interesting work for, potentially the entire lifetime of the system. So the name is somewhat justified, though very overloaded.

Applications generally have a specific hierarchy of processes, as we have talked about in the past. This hierarchy is illustrated in figure 10.1. We have the application behavior that owns the application lifecycle. It manages the root supervisor, which manages any and all long lived processes (including other supervisors) in the application.

Figure 10.1 – Application process structure



Applications, as a packaging concept, are very useful in providing consistency, a single way to package chunks of behaviour. They bring to the table a number of things, like a well defined directory structure, a well defined entry point, in the application behaviour module, canonical supervision patterns and entry points, etc. This consistency is what makes it possible to bring applications together in a formulaic way to create releases which are of course the focus of this chapter.

10.1.2 - The Simple Cache OTP Application

We have two OTP Applications in our project at this point. The resource discovery application and our simple_cache. From a directory layout perspective they both look pretty similar, actually every OTP Application looks pretty similar from a directory structure perspective and if it is a living application from a high level process perspective as illustrated in figure 10.1. Let's look at the layout for the simple cache.

```
simple_cache
  |
  |- ebin
  |- include
  |- priv
  |- src
```

The structure is recognizable; it is the same structure we have seen with every application we have created since chapter 4 when we introduced them.

Let's take a moment to reiterate the purpose of the various directories.

Table 10.1 Application Directories and Their Purpose

Directory	Purpose
ebin	The *ebin* directory is where our compiled code and *.app file ends up.
include	The *include* directory is where any header files should go.
priv	The *priv* directory is where we stick anything else (like native objects) that we care about.
src	the *src* directory is where our source code goes.

If you would like a little refresher go back and take a look at chapter four. It should get you back up to speed pretty quickly.

Remember that many applications are these living breathing versions of libraries. OTP needs a certain amount of information to know when and how these Applications are to be started what there dependencies are, what version they are, etc... This information is contained in the *.app file. This file describes our application so that OTP knows what to do with it.

Code Listing 10.1 - Application Metadata

```
{application, simple_cache,
  [{description, "A simple simple caching system"},
   {vsn, "0.1.0"},
   {modules, [simple_cache,
              sc_app,
              sc_sup,
              sc_store,
              sc_element]},
   {registered,[sc_sup]},
   {applications, [kernel, stdlib, resource_discovery]},
   {mod, {sc_app,[]}},
   {start_phases, []}]}
```

All of this information is important for the startup and proper functioning of the App. However, two pieces of information are especially important in the context of Releases. That information is the 'vsn' and the 'mod' tuples. The 'vsn' details the version of this application. You will see specifically why it's useful later. The mod entry indicates to the application infrastructure which module implements the application behaviour interface and is therefore the starting point for the application. Figure 10.4 is the application behaviour implementation for the simple_cache that the mod entry above in code listing 10.3 refers to.

Code Listing 10.2 - Application Behaviour

```
-module(sc_app).

-behaviour(application).

%% Application callbacks
-export([start/2, stop/1]).

start(_StartType, _StartArgs) ->
    sc_store:init(),
    case sc_sup:start_link() of
        {ok, Pid} ->
            {ok, Pid};
        Error ->
            Error
    end.

stop(_State) ->
    ok.
```

The start function starts the application and the stop function stops the application. Notice that in the start function for this module we do something just a bit more then start the root supervisor. We call this sc_store:init() routine. We call this out to illustrate the fact that the application behaviour implementation is the entry point, the startup point for your living breathing application. Whatever startup/shutdown functionality you have should end up here. We wouldn't suggest you add the code itself here. Put that in other, well named modules, but you should call that initialization and shutdown code here if you have it. Remember that this starting point for applications is just a behaviour implementation. That means that it conforms to a behaviour interface and that it has an associated behaviour container which contains all the functionality generic to application handling. When the start function completes and returns the pid of the top level supervisor it is returning that pid to the application container. This container is responsible for more than just managing the startup of that supervisor. It also handles things managing the loading of the very compiled object code for the application, it can also stop it and even unload it from the VM. The application container performs these functions for not just a single application but for many.

In that way it is like gen_event, one container to many behaviour implementations. The following code snippet from the shell shows the application container process and how to find it on a running ERTS VM. There is only one per VM.

```
1> regs().

** Registered procs on node nonode@nohost **
Name                  Pid          Initial Call                      Reds
Msgs
application_controlle <0.5.0>      erlang:apply/2                   12303
0
<snip>
```

The process is registered as application_controller.

Our supervisor should is pretty familiar to you by now. Here is part of it listed in code listing 10.5.

Code Listing 10.3 - simple_cache supervisor

```
RestartStrategy = simple_one_for_one,
MaxRestarts = 0,
MaxSecondsBetweenRestarts = 1,

SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},

Restart = temporary,
Shutdown = brutal_kill,
Type = worker,

    AChild = {sc_element, {sc_element, start_link, []},
           Restart, Shutdown, Type, [sc_element]},

    Event = {sc_event, {sc_event, start_link, []},
           Restart, Shutdown, Type, [sc_event]},

    Guard = {sc_event, {sc_event, start_link, [sc_event_logger]},
          Restart, Shutdown, Type, [sc_event]},

    {ok, {SupFlags, [AChild, Event, Guard]}}.
```

This is the supervisor initialization function. Notice that we have have three long lived processes that are started and owned by the Supervisor. These gen_server processes form the living heart of our application.

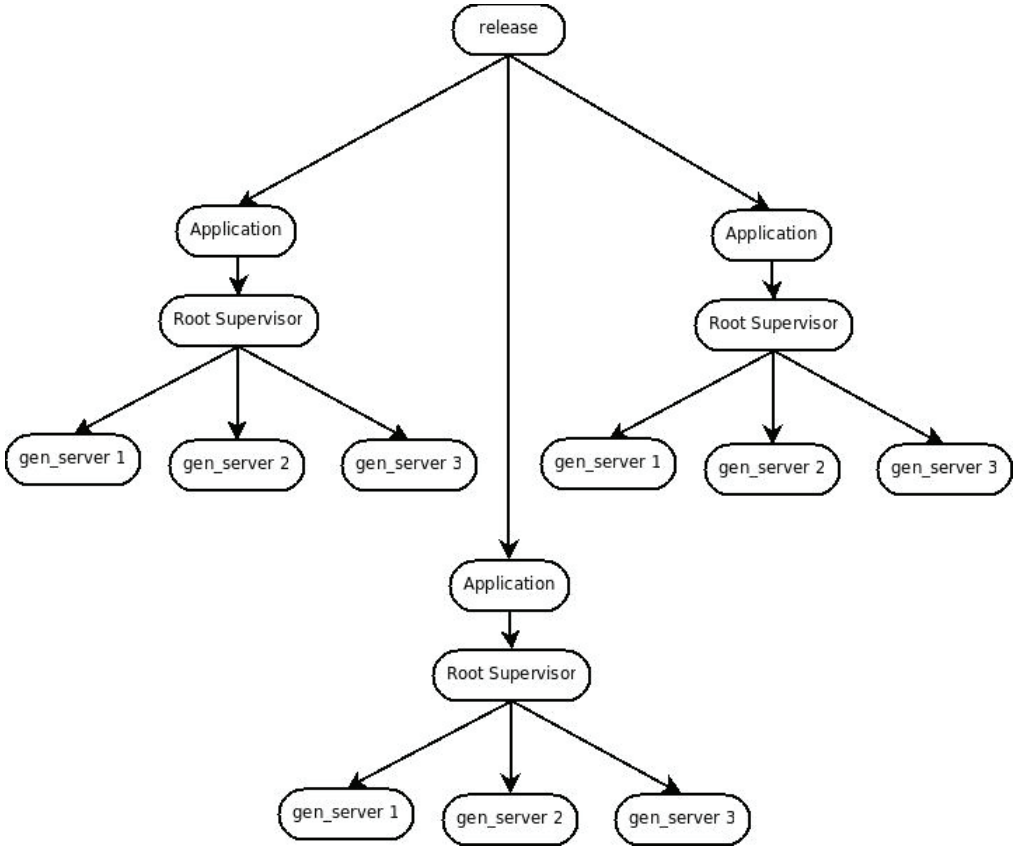
We have highlighted the biggest areas of Applications that are important for releases. We have also gone a little ways to explain why they are important. Let's start talking about how

you can tie these living chunks of functionality together in Releases which are what form running Erlang services.

10.2 – Releases Defined

Fundamentally packaging things in Erlang/OTP is a hierarchy. At the lowest level we have modules. Modules are packaged into Applications at the next level. Finally at the highest level we have Releases. Release, that's another rather overloaded term. In Erlang/OTP the term 'Release' has some pretty specific connotations. The Release is a set of Erlang/OTP Applications bundled together with metadata on how to start and manage the set of applications. This bundling is illustrated in figure 10.2

figure 10.2 – release strucutre



The applications that form a release are started together on a single running ERTS VM. Each VM can only start a single release, this makes the release a service definition. A running ERTS instance is defined by the release it was started with.

Release = Service

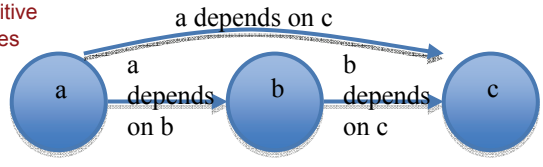
The Applications included with a release are generally the Applications required to provide the functionality required by a given single ERTS instance and all of their dependencies. For example, two of the applications that we absolutely need in our release are 'simple_cache' and 'resource_discovery'. Both of these applications depend on other applications, which depend on other applications etc; all of these need to be included in the release. In effect all of the direct and transitive application dependencies must be included in our release. Again, in the preceding figure 10.3 you can see a high level representation of how releases manage

applications. The figure illustrative the fact that a single release encompasses the lifecycle of all applications (and their dependencies) included in the release.

Dependencies

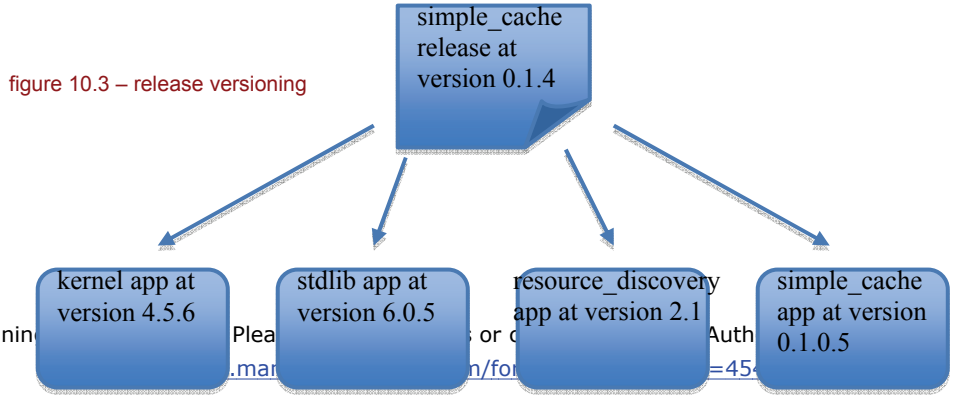
We will be talking about dependencies quite a bit in this chapter. Let's go ahead and get the terminology right now. Direct dependencies of our app, like kernel, stdlib, etc are just called 'Dependencies', while dependencies of dependencies are called 'transitive dependencies'. The distinction is important. Let's say we have application A that depends on application B. Application B depends on application C. Because application B depends on application C, application A also depends on application C. The fact that application A depends on application C is what's really called a transitive dependency. Figure 10.3 illustrates this relationship.

10.3 – transitive dependencies



Not only are releases specifying all the applications and dependencies that are required on a given ERTS instance but also which specific versions of the applications are required to run on a single VM. So simple_cache release 0.1.4 may require the application simple_cache 0.1.0.5 and resource_discovery 2.1 and kernel version 4.5.6 specifically. Versioning is a very important capability that releases bring to the table. So when we talk about release version 0.1.4 we are really talking about something like what is pictured in figure 10.3.

figure 10.3 – release versioning



At this point here is what we know about Releases. They run one per VM, they are versioned, and they aggregate a number of versioned applications along with meta data on how to start the applications. They are essentially service definitions. In the overall eco system of Erlang/OTP releases are very important and surprisingly misunderstood. To ensure that readers of this book do not misunderstand how to create releases we are now going to go through the process of creating a release for our simple cache.

10.3 – Creating a Release

We have two applications that we have developed, simple_cache and resource_discovery, and a number of other application dependencies that need to come together to form the simple_cache service. In other words for us to start a VM and have it function as a simple_cache instance we need to start that VM with, for example, the following applications:

kernel 2.13.2
stdlib 1.16.2
sasl 2.1.5.3
mnesia 4.4.10
resource_discovery 0.1.0
simple_cache 0.1.0

We also need to specify the version of the VM that we want to have these applications run on. We need to ensure that the VM we include our applications on is the VM they were compiled for. This is where the *.rel file (pronounced "dot rel file") comes into play.

10.3.1 – The Release Info File

Remember that there exists a *.app file for each Application that is created, its function is to describe many of the fundamental aspects of a given application. The *.rel file plays the same role for a Release. Listing 10.4 contains the simple_cache.rel file we are going to use as the basis for creating a Release.

Code listing 10.4 – simple_cache.rel

```
{release,{"simple_cache”,"0.1.0.4”},
```

```
{erts,"5.7.2”},
 [{kernel,"2.13.2”},
  {stdlib,"1.16.2”},
  {sasl,“2.1.5.3”},
  {mnesia,“4.4.10”},
  {resource_discovery,"0.1.0”},
  {simple_cache,"0.1.0”}]}.
```

The release info file contains an Erlang tuple. That tuple is terminated by a period just as in app files. The first element is the atom release which is followed by another tuple containing the stringified name of the release and the version of the release; in our case {"simple_cache","0.1.0.4"}. As for the version string you can put anything you like here and for all versions in this file. It is advisable that you try and stick with conventional version strings. That way they are more meaningful to your users and parseable by tools.

The next element is the ERTS version designation. This is a tuple containing the atom ‘erts’ and the version of the ERTS system required. ‘ERTS’ stands for the Erlang Runtime System. It’s the virtual machine that Erlang runs on top of. The version is the version of that virtual machine to make use of. Make note that it’s not the version of Erlang/OTP. For example, the current release of Erlang/OTP is R13B01. However, the version of ERTS is 5.7.2. You can see which ERTS version you are using if you just start up a shell.

```
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [smp:2:2] [rq:2] [async-
threads:0]    [kernel-poll:false]

   Eshell V5.7.2  (abort with ^G)
   1>
```

That first element after the “R13B01” is the ERTS version, in this example its 5.7.3.

The last element of our release info file is the list of included applications and their versions. You can see that we have simple_cache there. We also have kernel and stdlib. Why is that? That’s because this must be a complete list of all the applications your system requires. This includes not just the direct dependencies on the applications you have written like simple_cache and resource_discovery but also the transient dependencies.

The release info file is not executable. It is fundamentally just a specification and furthermore the Erlang runtime system can’t directly read it and even if it could it does not contain enough information to start a running system correctly. The release file does not contain paths to any of the artifacts it specifies. It does not point to an ERTS executable and it does not provide any information about where any of the compiled applications listed in the application dependencies list reside on the file system. This all needs to be pulled together before a real Erlang/OTP service i.e. a Release can be started. Pulling this information together along with other supporting artifacts like configuration to create a complete Release specification that we will allow us to start a working simple_cache the right way is what we will tackle in the next section.

10.3.2 – Creating the Release Specification

The *.rel file is where it all starts but it is not the end of creating a running Erlang/OTP service. This section will cover moving from the release info file to a running Erlang service. The first step toward creating a Release is to create two more files. These files represent a more complete specification of a Release. They are the *.script and *.boot files. The *.script file contains a full specification of what will be included in apps including paths to applications, what modules will be loaded and other necessary information. The *.boot file is the binary representation of the *.script file that will actually be read in by ERTS to bootstrap the Release. To create these files the first thing that needs to be done is an ERTS VM needs to be started that contains the pathing information to all the applications specified in the *.rel file. We will start a shell then and use the –pa commandline flag to add code paths for all the applications that belong to our intended release that are not already on the default path for erl (i.e. all of the applications that come with Erlang/OTP out of the box). Below is the command line we use to start the shell with the paths to the simple_cache and resource_discovery application ebin directories.

```
 Macintosh:~ martinjlogan$ erl -pa ./simple_cache/ebin -pa
./resource_discovery/ebin
```

This starts up our Erlang shell with the additional code paths we added via the –pa option to erl. The next step is to generate the actual boot and script files with the help of the systools module.

```
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0]
[kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> systools:make_script("simple_cache", [local]).
ok
```

Running the above results in the generation of two files in the current directory, simple_cache.script and simple_cache.boot. The 'local' option passed to the make_scropt/2 function stipulates that absolute paths to all applications be supplied into the script and boot files. Meaning that if you try to boot a release with the simple_cache.boot file it will require that all application code reside in exactly the places they did when the boot file was created. That is not very portable or ideal for production use. The local option is good for local testing of the sort we are doing now. The less fragile manner is to create the script and boot files without the local option. Doing so will make it such that the script and boot files expect all application code to reside in the same directory called lib. This means that our simple_cache and resource discovery apps would need to sit along side kernel and stdlib in a directory

calles lib somewhere on the fielsystem. In this case the variable $ROOT is used instead of a hardcoded path to the lib directory. This offers a less fragile and more flexible mechanism for starting a Release. For our purposes right now the local option is convenient because our apps are being developed and we have not staged them anywhere special.

We are now almost ready to start the simple_cache release. The other element we need to worry about for this particular release is configuration. You may recall that in chapter 9 we used configuration but relied on the default values we placed in our code. At this point we will actually create a configuration file to be used within our release. The vanilla name for a release configuration file is sys.config. The name can be anything so long as it ends with the extension "config". The config file like the .app file and the .rel files contains nothing more than an Erlang term terminiated with a period. Listing 10.5 contains the contents of the sys.config file for simple_cache.

Code listing 10.5 – sys.config

```
[
 %% write log files to sasl_dir
 [
 {sasl,
  [
   {sasl_error_logger, {file, "/tmp/simple_cache.sasl_log"}}
  ]},


{simple_cache,
  [
   %% Contact nodes for use in joining a cloud
  {contact_nodes, ['contact1@erlware.org', 'contact2@erlware.org']}      #1
  ]}
].
```

The format for the configuration is an Erlang a list. The list contains tuples, these tuples contain atoms indicating the name of the application and then a list of key value pairs that pertain to the application. So for simple cache above the list of key value pairs contains a single pair which is the contact nodes list pair. This is highlighted at code annotation #1.

At this point we are ready to start our release because we have all the required parts. When we start our Release we are going to specify two things. Where the simple_cache.boot file resides and where the sys.config file resides as well. In our case we have these files residing in the same directory. The following command line starts up our simple cache release.

```
erl -boot ./simple_cache -config ./sys
```

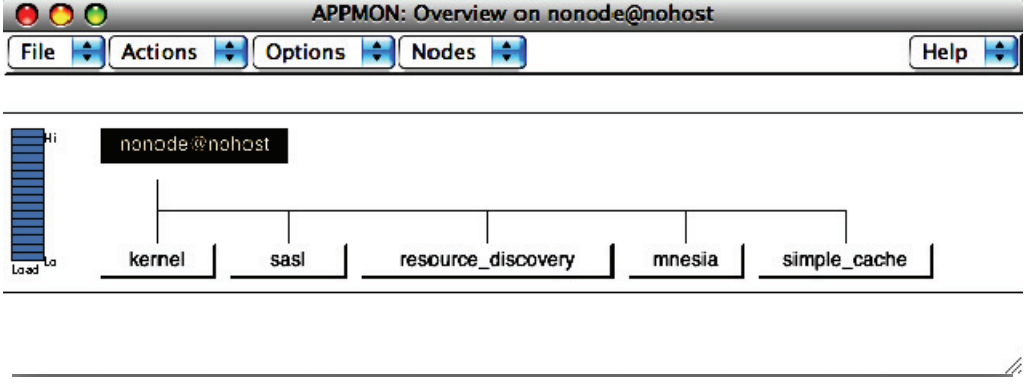
Running that commend should have produced a shell and if you implemented the logging code from chapter 7 a floury or logging output. This means the release has been successfully started. When you start a release in production you will not want the shell to start, you will want the release to start running in background. This can be accomplished by adding the –detached flag onto the command line as such:

```
erl -boot ./simple_cache -config ./sys -detached
```

For now though a shell session running in the foreground is exactly what we want so that we can quickly prove to ourselves that what we expect to be running in our Release actually is. For this we will use appmon. From the shell of the running Release execute the following:

```
1> appmon:start().
{ok,<0.72.0>}
```

This will result in the startup of the appmon utility. You should very shortly see a graphical application come to the forefront that looks just like the following screenshot.



As you can see all of the application dependencies listed in our simple_cache.rel file are up and running except stdlib which is a library app. So there we have it, a running Erlang service. Bravo! In the next section we will take what we have learned here and further its usefulness by learning howt to package up a Release for easier installation and deployment.

10.3.3 – Release Packaging

What more can we do with releases? Well, we can package them up for easy installation, distribution, and deployment. OTP provides some helpful functionality for packaging releases, but not all the functionality required for a production grade package. This is a hole in the OTP functionality and one where the community at large has had to pick up where OTP left off. For this reason we will show you a more complete process that has gained some traction in the community but is not an official OTP standard. We will start off the process of creating a Release package with the OTP provided functionality. We are going to do that by running another tool in the systools module. The make_tar function. As you may guess make_tar produces a tarball representing a component of our overall Release package. Lets dig in and run that now.

```
>> erl -pa ./simple_cache/ebin -pa ./resource_discovery/ebin
Erlang R1301 (ERTS-5.7.2) [source] [64-bit] [smp:2:2] [rq:2] [async-
threads:0] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> systools:make_tar("simple_cache").
ok
2>
```

If you list what's in the releases directory now you will see a new file called 'simple_cache.tar.gz'. That's the tarball that contains a portion of our new Release.

Let's take a look at what's inside the tarball. It should be pretty informative about what Erlang/OTP cares about.

```
>> mkdir tmp
>> mv simple_cache.tar.gz tmp
>> cd tmp
>> tar -xzf simple_cache.tar.gz
>> ls
lib releases
>> ls lib
kernel-2.13.2 mnesia-4.4.10 resource_discovery-0.1.0 sasl-2.1.5.3
simple_cache-0.1.0  stdlib-1.16.2
>> ls releases
0.1.0.4  simple_cache.rel
>> ls releases/0.1.0.4
start.boot
```

The tarball has two directories 'lib' and 'releases'. The lib directory contains all of the applications required by our release. This is one of the main reasons using the make_tar function is useful; it aggregates all the required applications needed for a release in a single place. In this case, that's kernel-2.13.2, simple_cache-0.1.0, stdlib-1.16.2, etc.

WHY OTP APPLICATION DIRECTORIES CONTAIN VERSIONS

There is a reason our app names now contain the version numbers. Erlang supports hot code reloading, to do that it needs the ability to keep multiple versions of the same codebase around. The convention of naming OTP App directories in the form <appname>-<version> allows it to do this.

So the 'lib' directory contains the apps. The releases directory contains, guess what? Release information! In this case, it contains our *.rel file and a new directory. That new directory is the version of our Release. It keeps multiple versions of a Release around for the same reason it keeps multiple versions of the App around. In that version directory is our boot file. Its renamed it to start.boot, but that's ok. It's still our boot file. Notice that the simple_cache.tar.gz file does not have any place for us to put config nor does it contain an executable for us to start up the release once it is installed. These are aspects that are not handled canonically by Erlang /OTP. Here is where we start to diverge in creating a truly functional release. We don't need to diverge much, but we do need to diverge.

The first thing to do is renaming the directories underneath the releases directory. Right now underneath releases we have the simple_cache.rel file and a directory that is the same name as the version of our release, 0.1.0.4. We will alter this by changing the name of the directory 0.1.0.4 to simple_cache-0.1.0.4, or more generically <releasename>-<releasevsn>. This is so that if the Release is to be installed along side other Releases of different names they can be differentiated. The next move to make is to place the simple_cache.rel file into the newly renamed directory. Now we manually place our sys.config file in this same directory so that it can sit along side the *.boot file.

This is almost the end of the Release creation process, but not quite the end. We need a place for executable startup files (if they exist). We suggest creating a bin directory right off the root directory of the untarred release. Within this file will go an executable script, named in this case "simple_cache", containing minimally code similar to what we used to start our release directly from the command-line in section 10.3.2. The following snippet provides an example executable script for Unix like systems.

```
#!/bin/sh

erl -boot ../releases/simple_cache-0.1.0.4/simple_cache \
```

```
-config ../releases/simple_cache-0.1.0.4/sys \
-detached
```

At the end of the day what we have just presented provides for a very useable release package. The other funny thing is that this standard is that it is exactly what the Erlang/OTP install uses. However, which it is not fully supported by the OTP tools and requires this little bit of manual work. In general, that's not a big. Its what we use in all of our projects and it works rather well. Try it, run a directory listing on the Erlang/OTP root directory. You will find a bin directory that contains erl and erlc and you will also find a common releases directory that contains all the Erlang/OTP releases you have ever installed.

```
Macintosh:erlang martinjlogan$ ls
Install     erts-5.7.3     misc           usr     bin     erts-5.6.3
     erts-5.7       erts-5.7.2     lib            releases
```

You can see the bin directory which contains erl and erlc among others. You can also see the releases directory which contains unique directories for each release. Each of these unique directories contain *.boot *.script and *.rel files all side by side. The one thing we have not yet discussed are the directories named erts-*. These directories are ERTS itself. If you really want a self contained release you can package erts along with it. This means that you will be able to distribute your release to other machines of the same OS and architecture and run them without ever having to have installed Erlang from source. We will not get too far into that in this chapter but for the more intrepid reader it is something to play with.

The last step before we move on to installing our Release is to tar it up with Erlang's tar utility as shown in the following snippet.

```
>> cd simple_cache-0.1.0.4
>> erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0]
[kernel-poll:false]
Eshell V5.7.2  (abort with ^G)
1> erl_tar:create("simple_cache-0.1.0.4.tar.gz",
                  ["lib", "releases", "bin"],
                  [compressed]).
```

With that command run we have a tarred and compressed package file named simple_cache-0.1.0.4.tar.gz. With a fully functional Release package created. Now we can move on to installing it so that it can be used.

10.3.4 - Installing a Release

OTP, out of the box, provides some functionality for installing Releases. Unfortunatly, it is not that useful. It's the standard though and, for that reason ,it's good to know. So we point you towards it. In practice it is not a very effective, nor is it a common way of installing releases. At the very least it can not cope with the changes that we have made in section 10.3.3. The function to look at if you are curious is release_handler:unpack_release/2.

So how do we make this release usable? Because the directory layout of our release structure mimics that of what gets installed by default with Erlang/OTP all that needs to be done is to extract the tar file over the actual Erlang/OTP install. It will fit perfectly. The code below shows you how to do this. (we are assuming the default Erlang/OTP install directory on Unix you should change this to match your local install.)

```
>> cd /usr/local/lib/erlang
>> erl
Erlang R13B01 (erts-5.7.2) [source] [smp:2:2] [rq:2] [async-threads:0]
[kernel-poll:false]
Eshell V5.7.2  (abort with ^G)
1> erl_tar:extract("simple_cache-0.1.0.4.tar.gz", [compressed]).
```

If you run a directory listing of erlang/bin now you will find the simple_cache executable file which of course can be run to fire up our Release. If you start up appmon again you will see all the expected Applications up and running, waiting on requests!

The process we have described here is a manual one in many aspects. There are easier ways to achieve this using some of the automated tools that exist out in the community. The authors of this book are responsible for two of them. A build system called Sinan and a package management system named Faxien which are available at Erlware.org. These are by no means standard and are certainly not the only tools available to manage these processes. At the risk of plugging our own tools we think it worthwhile to mention this because using automated tools, whether they be Faxien and Sinan or others reduces the errors associated with manual processes – not to mention it is just easier!

We have shown you the basics of packaging as OTP provides it. We have also shown you a powerful method for extending those methods to yield a powerful method for packaging and deploying your production Erlang/OTP services. You might think then that are job here is done, we understand modules, we understand applications, and we understand releases. What else is there? Well those are all artifacts but we have not really said anything about how to go about structuring your development in the context of these structures. Where do you store config files when you write code? Where do the .rel files go? How is the release started up and where do those executables go during development? There is still a lot to go into and we will do that in the next chapter!

10.4 – Conclusion

You now know how to create and manage releases. You know how to package them up and deploy them to other systems. This should give you enough information to use the existing tools built around this process or start building your own. This is a bit of a milestone, for the longest time Apps and Releases have been deep magic that only the Erlang Illuminati was aware of and used. Now you know it to and you can make good use of it in your Erlang Applications.

11

Non-native Erlang Distribution with TCP and REST

In the previous chapters we have implemented quite a nice cache for ourselves. The Erlware guys plan to take this and really enhance it over the coming months. One of the things they recognized about the simple cache is that it would be useful to more than just Erlware. The only problem with this is that currently the cache only has an Erlang interface. That means that it can only be used in conjunction with applications written in Erlang/OTP. Today, production environments contain services written in a variety of different languages. So, before open sourcing the simple cache, it needs to be rendered useful to systems implemented in languages other than Erlang. The most natural way forward would be to implement an interface, or interfaces, over common protocols. In this chapter we are going to pick two, the first will be a very simple text based protocol over TCP and the second will be a RESTful interface that implements the same functionality. There is a slight twist in store for you with regards to how we will go about implementing the RESTful interface. We are going to implement our own very scaled down webserver. This will allow us to demonstrate quite a few useful things, from new OTP behaviours to proper network coding in Erlang. In the subsequent chapter we will demonstrate other methods by which Erlang and other systems/languages can be made to communicate. These methods include Drivers and the Java interface. However, before we go any further we want to make the outline of this chapter very clear.

1. Implement text based communications interface
    a. Implement TCP server application
    b. Implement text based protocol over TCP
2. Restful interface design

3. Create RESTful interface application
    a. gen_fsm basics
    b. Create webserver with gen_fsm behaviour

The next section kicks off the development of our text based communication interface

11.1 – Implement the text based communication Interface

Text is everywhere, it is simple to use and simple to implement, which is why we are using it as a starting point for creating our communications interface into our cache. This protocol will be very straightforward and as simple as we can make it. This will be text over TCP just in a way very similar to what we have done in the past. We will make heavy use of Erlang's concurrency to make sure everything scales well. Basically, we are going to try and make this server nice and robust.

I want to take a quick detour before I explain the high level design or the TCP server we will build here. This detour is in the form of a little refresher on the semantics of accepting a TCP connection, using Erlang syntax as a guide.

TCP Socket Connection Refresher

The first step in creating the socket is to establish a listener. This is done buy calling the gen_tcp:listen function, as you can see below.

```
{ok, ListenSocket} = gen_tcp:listen(Port, [{active, true}])
```

We tell the system what port we wish to accept connections on and then we tell the system that we want the connection to be active. What this means is that we will receive messages that will contain any data that comes in off of the connection subsequent to making this socket. We can then process these messages in the normal way. The opposite of this, achieved with {active, false} known as passive mode, would require messages to be explicitly read off of the socket in the application code with a call to gen_server:recv.

FLOW CONTROL

It is worth noting that while active mode is cleaner and has a much more Erlang/OTP feel to it, it does not provide any flow control. This means a client could send data faster than the receiving end can read it off the message queue overflowing the queue and using up all available memory. This is because, in active mode, Erlang egerly reads data from the socket as quickly as it can, formulates that data as messages and sends to the designated receiver. In the vast majority of cases this is not the behaviour you want in your application. Sockets in passive mode behave differently. In passive mode you must read

the data off the socket and do something with it. So you have direct control over what data enters the system and at what rate it enters the system.

In either case, once the listen socket is established it can wait for any number of clients to actually bind to the socket it is listening on. This is done with a call to accept, a call which blocks indefinitely until a connection is established. Once established the call to accept returns with a handle to the bound socket.

```
{ok, Socket} = gen_tcp:accept(ListenSocket)
```

With this understanding of sockets we can now get back to talking about the pattern for designing a clean efficient OTP based TCP server.

IMPLEMENTING THE TCP SERVER

One effective pattern for implementing a concurrent server is to create a gen_server under a simple one of one supervisor. This gen_server sits on the socket listening for new connections. When a new connection comes in the gen_server kicks off another process to listen for incoming connections. The original gen_server then continues to handle communication with the recently accepted socket. This allows for efficient and quick accepting of connections with little or no downtime between the accept and processing of communications. This is very different from the way sockets are handled in other languages, but this is the most correct way to do it in Erlang. Figure 11.1 below illustrates this.

Figure 11.1 Simple One for One Supervisor Structure

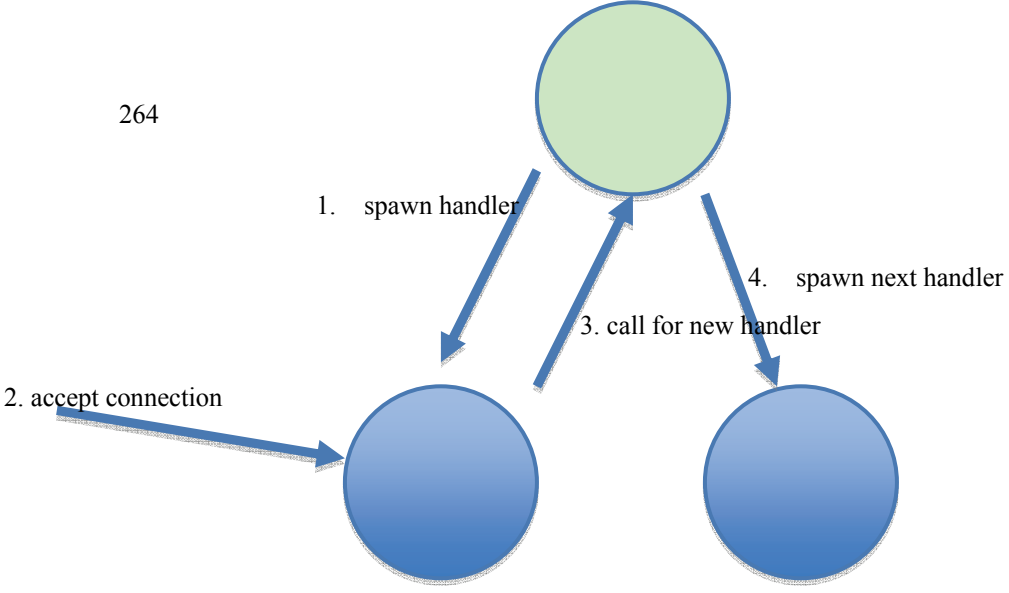


figure 11.1 – TCP Server Design

With the overall design understood we can now delve into the details of making this work. The first step along this path is creating an OTP application and the associated metadata.

11.1.2 – Creating the tcp_interface Application

Since the Text/TCP protocol is but one entry point to our cache, we will implement this communication functionality as a separate OTP application. If we wanted to add Text/UDP or HTTP/TCP later on we could follow the same pattern. A given release could include one, many, or none of these external communication applications depending on what was required, each wrapped in their own OTP Application. The point is that these OTP applications are just front ends for the cache and the cache OTP Application doesn't need to be aware of them at all. We could add any number of these external apps without any problem and without any of them interfering with each other. In any case, a new application needs to be created that we will call tcp_interface. The first step in creating the new application is the addition of all the required behaviour container applications.

Like every OTP application this one will need a .app file, an application behaviour container implementation, and a top level supervisor module, in this case it will be a simple one for one supervisor. I am not going to get too far into the details of these OTP application artifacts, these are things you are well versed in doing at this point. However, when we are done creating all of this infrastructure we should have the following file structure in place along with the simple_cache and resource_discovery applications.

```
tcp_interface
    |-- ebin
    |   `-- tcp_interface.app
```

```
    `-- src
        |-- ti_app.erl
        `-- ti_sup.erl
```

The tcp_interface application is also going to need a generic server to act as the connection handler process, we will call this one ti_server. With the ti_server added the file system tree should look as follows.

```
tcp_interface
    |-- ebin
    |   `-- tcp_interface.app
    `-- src
        |-- ti_app.erl
        |-- ti_server.erl
        `-- ti_sup.erl
```

Now we have the structure of our OTP Application, so it's time to talk a little bit about how we flesh this structure out. In the next section we will cover all of the details that make this design simple, elegant, and functional. There are a few little tricks and subtleties to getting it all put together. Upon finishing the next section you should understand all of them.

11.1.3 – Fleshing out the TCP Server

Fleshing out the server is going to require us to take some of the code we have already created in previous chapters and adding it to the TCP application in the correct places. The natural starting point is the ti_app module. There are two additions we must make to this module. The first is to add the code to create a listen socket. A listen socket must be created by a process that is not going to die throughout the life cycle of our TCP server. In the advent of the process dying we lose the ability to accept new connections. Since, in the vast majority of cases, we don't add code to the supervisor, we will add it into the ti_app module. The listen socket that we created can then be passed down into the supervisor on its creation for use with each of the children that it will spawn. The second change that is to be made in the ti_app module is that once the supervisor is done with it's initialization we must ask it to spawn the first of the connection handler processes. All of this code is present in the start/2 function, which is listed below in listing 11.1.

Code Listing 11.1 – Adding the Listen Socket and Connection Handler Startup Code

```
start(_StartType, _StartArgs) ->
    Port =
      case application:get_env(tcp_interface, port) of
            {ok, Port_} -> Port_;
            undefined   -> ?DEFAULT_PORT
        end,

    {ok, LSock} = gen_tcp:listen(Port, [{active, true}]),

    case ti_sup:start_link(LSock) of
        {ok, Pid} ->
```

```
        ti_sup:start_child(),
              {ok, Pid};
          Error ->
              Error
    end.
```

Initially we pull the value of Port, the port to listen for connections on, from configuration and via the use of a usually application configuration approach. If its not specified we allow for a default value, which is defined in the header section of the file. Second, the listen sock, LSock is created by a call to gen_tcp:listen/2, the semantics of which we discussed earlier in this chapter when we reviewed connection semantics. Last, you can see that we have added a call to ti_sup:start_child/0 and that we have added it just following successful return of the supervisor call to startup. This kicks off the initial connection handler gen_server process. This last addition to the code leads us naturally to our next set of alterations, which will be in the ti_sup module. In this module we will need to add an interface function, and export it. This interface function will be used to create new ti_server connection handlers. We will also have to modify the current start_link/0 function to take the listen socket as an argument. This will allow it to carry the socket in it's state and supply it to each of the children that it starts. Finally we will need to add code to the init/1 function so that it knows how to start the ti_server processes. The code for doing this is shown in listing 11.2. Note that I am not including any of the API documentation, but it is indeed present in the original code!

Code Listing 11.2 – Fleshing Out the Supervisor

```
-module(ti_sup).

-behaviour(supervisor).

%% API
-export([start_link/1, start_child/0]).

%% Supervisor callbacks
-export([init/1]).

-define(SERVER, ?MODULE).


 start_link(LSock) ->    #1
    supervisor:start_link({local, ?SERVER}, ?MODULE, [LSock]).

 start_child() ->    #2
    supervisor:start_child(?SERVER, []).


 init([LSock]) ->    #3
    RestartStrategy = simple_one_for_one,
    MaxRestarts = 0,
    MaxSecondsBetweenRestarts = 1,

    SupFlags = {RestartStrategy, MaxRestarts, MaxSecondsBetweenRestarts},
```

```
    Restart = temporary,
    Shutdown = brutal_kill,
    Type = worker,

AChild = {ti_server, {ti_server, start_link, [LSock]},   #4
     Restart, Shutdown, Type, [ti_server]},

    {ok, {SupFlags, [AChild]}}.
```

Starting at code annotation #1 where start_link/0 has been made, start_link/1 takes the listen socket that was passed down from the application behaviour module. The next line down you can also see that LSock is passed through the supervisor:start_link/3 call such that the LSock variable will be handed off to init/1. Code annotation #2 is where we have added the start_child function. This function prompts the simple_one_for_one type supervisor to kick off a single child of the type that was specified in it's init function. At code annotation #3 we see that init/1 has been modified to take the LSock which it passes at code annotation #4 into the child spec for the child it is configured to start. These simple modifications give us a nice OTP compliant factory for creating ti_server processes to handle incoming connections and service them – which of course leads us to the next and final module that we need to modify in order to have a working TCP server framework in place; ti_server.

The ti_server process is where we actually accept and bind connections to a socket such that we can receive TCP data coming off the wire. In this process the key is to take in the listen socket, place it in the gen_server state and then return immediately so that the supervisor can move on and spawn other children if need be. The trick is that the instant after returning control to the supervisor, i.e returning from the init/1 function which causes the start_link function to return {ok, Pid}, we jump immediately into blocking and waiting on accept/1 for a new incoming TCP connection. This is done through setting a timeout with a value 0 when returning from init/1. Since in our case the gen server mailbox is certainly empty this will cause us to bounce from init/1 to handle_info/3 without pause where we can accept any incoming socket connections. I think it is time to demonstrate the code. If you need to, come back and read this section again after the explanation of the changes made within the gen_server ti_server. The code for ti_server is shown in listing 11.3.

Code Listing 11.3 – ti_server Internals

```
-module(ti_server).

-behaviour(gen_server).

-export([
      start_link/1
     ]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
      terminate/2, code_change/3]).
```

```
-define(SERVER, ?MODULE).

-record(state, {lsock}).


 start_link(LSock) ->  #1
    gen_server:start_link(?MODULE, [LSock], []).

init([LSock]) ->
   {ok, #state{lsock = LSock}, 0}.  #2

handle_call(Msg, _From, State) ->
    {reply, {ok, Msg}, State}.

handle_cast(stop, State) ->
    {stop, normal, State}.

 handle_info({tcp, Socket, RawData}, State) ->
     gen_tcp:send(Socket, RawData),
     {noreply, State#state{request_count = RequestCount + 1}};
 handle_info(timeout, #state{lsock = LSock} = State) ->  #3
    {ok, _Sock} = gen_tcp:accept(LSock),
    ti_sup:start_child(),
    {noreply, State}.
```

Marked with code annotations #1, #2 and #3 are the fundamental steps needed to accept a socket connection in the concurrent TCP server context. Starting with annotation #1 where the listen socket is passed through from the supervisor state and then in the next line down through the gen_server:start_link/3 function to arrive finally at init/1. Annotation #2 quickly puts the listen socket handle into the state and then returns with third tuple element the timeout of 0. Notice that we don't do anything in init, we simply return immediately. This causes the start_link/1 function at annotation #1 to unblock immediately which allows the calling supervisor to continue on with its business of starting other children. The timeout 0 prompts the execution of the gen_server code shown in code listing 11.2 resulting in an immediate drop into the code starting at code annotation #3. Here the listen socket is extracted from the state and the server immediately blocks waiting on an incoming TCP connection via the call to gen_tcp:accept/1. Once this call returns we call the supervisor with ti_sup:start_child, prompting it to start another ti_server connection handler process to field the next incoming connection. This could possibly already have happened and is now queued up waiting for accept. At this point, we have an accepted connection and we are ready to receive data off out of the process mailbox. This is due to the fact that our connection was set to active when the call to listen was made. The format for incoming connections is the same as it was in chapter 3 and can be seen at code annotation #4 as "{tcp, Socket, RawData}".

This framework version of our server simply echos back whatever was sent to it. This is not going to suffice for providing an interface to our simple_cache. To do that we need to build out the protocol handling aspects of our system.

11.1.4 – Implementing the Text Communication Protocol

What we set out to do in this chapter is enable non-erlang VM's to interact with the simple_cache. Thus far we have a framework in place for accepting incoming TCP connections in an efficient manner, now we need to layer on top of it a text communication mechanism. The protocol will be very simple, and is expressed below in generic terms:

```
Function|Args
where
       Function = insert | lookup | delete
       Args = [Arg]
        Arg = term()
```

and each message has a result associated with it which is

```
Result = term()
```

I have purposely written that specification in sort of an Erlang format instead of BNF or other language specific format. We are taking this route because this is a syntaxt that you will probably be familiar with at this point. You will see similar syntax all over the place and you need to be able to read it.

Lets get into a couple of literal examples of using this protocol. Hopefully, this will help round out your comprehension of the protocol and of the specification syntax above.

```
insert|[key, {value1, value2}]
delete|[key]
```

The first line is an insertion operation. The function "insert" is specified followed by a pipe "|" to separate it from it's arguments. The arguments follow, in an Erlang list, the first element of which is an atom, 'key' and second of which is a tuple containing two atoms, 'value1' and 'value2'. The second example is of a delete operation being performed on a single argument, 'key'. This protocol is straightforward and allows external programs an easy way to get terms into and out of the cache. Now that we have a protocol defined we can dig into the parsing and processing of messages using this protocol.

11.1.5 – Text interface implementation

Implementing our interface will happen entirely in the ti_server module. We have set our TCP socket to active mode. This means that all of those messaegs will come in on our gen_server container implementation's handle_info function and so it is where we will do the heavy lifting of implementing this functionality. Our protocol is a simple request response type protocol. A client implementing our protocol properly will not send more than one request without getting a response back, every request must get a response, before another

request is sent, this effectively controls the flow, or rate of data, that is sent into each connection handler to a likely manageable level. Below in code listing 11.4 the handle info function as modified to handle our protocol is shown.

Code Listing 11.4 – Protocol Implementation In ti_server.

```
handle_info({tcp, Socket, RawData}, State) ->
    try
        Result =
      case string:tokens(Data, "|\r\n") of                    #1
                [Function, RawArgList] ->
                    {ok, Toks, _Line} =
            erl_scan:string(RawArgString ++ ". ", 1),         #2
          {ok, Args} = erl_parse:parse_term(Toks),            #3
                    apply(simple_cache, list_to_atom(Function), Args);
                _Error ->
                    {error, bad_request}
        end,
     gen_tcp:send(Socket,
            lists:flatten(io_lib:fwrite("~p~n", [Result])))          #4
    catch
        _C:E ->
            gen_tcp:send(Socket,
                         lists:flatten(io_lib:fwrite("~p~n", [E])))
    end,

    {noreply, State};
```

In practice this function could be broken down a bit into a couple of sub functions, but it is easier to show and step through in this context if we keep the code local to a single function. Starting with code annotation #1; this is where the incoming string, something like "insert|[key, value] gets parsed. String:tokens splits the string on the pipe and returns a list substrings that where delimited by that pipe. If the tokens function returns anything but a list with two elements it is a clear indication that a bad request was sent in. In the case of a bad request an error term is stored in the variable Result. Code annotation #2 shows where we scan the Argument substring we split off of the main string. The erl_scan:string/1 function here takes a string and breaks it down into Erlang tokens. Tokens are representations of Erlang syntax elements such as integers, floats and punctuation, basically representations of all the syntax that make up Erlang terms. Notice the addition of a ". ", that is period followed by a space. This is because the next function at code annotation #3 erl_parse:parse_term/1 expects the tokens it receives to be terminated in this manner. If the tokens are not terminated as such, the call to parse_term will fail. Parse term is where the tokens that represent a term are parsed and actually converted into a term. According to the protocol definition we created earlier, arguments are always passed in the form of an Erlang list. This makes it easy for us to use the built in apply function for dynamically applying function calls. After the function is applied, the result is captured and returned to the client, this happens at code annotation #4.

The technique at use here is not the most intuitive so it bears a little examination. The line below accomplishes the reverse of what was done with erl_scan:string/2 and erl_parse:parse_term/1.

```
lists:flatten(io_lib:fwrite("~p~n", [E]))
```

The line, shown again above, renders an Erlang term into a string so that we can pass it back over the socket. The first thing that is done is the call to io_lib:fwrite which works much like io:format in that it takes a prompt, in this case the ~p formatter, to pretty print whatever arguments are supplied to it in the second argument which is always a list of terms. The issue is though, that io_lib:fwrite, returns a deeply nested list, which isn't really that readable when printed out to the console. So we use the function lists:flatten/1 to flatten the nested list into a flat list. In this case, the resulting list is just a string that we can print out to the console.

That is it. We are now free to move on to the next section of this chapter, which will get us started on what will finally turn out to be a RESTful HTTP interface that is capable of carrying any type of payload to and from our cache. As we mentioned at the outset, we are going to be doing this restful interface a bit differently then others do. Not only are we going to implement the the interface itself. We are also going to implement the HTTP server that will serve that restful interface. You may consider this an extreame case of 'not invented here syndrome'. However, there is a method to our madness. Our goal in implementing the HTTP server is two fold. First to give you a complete, robust and well documented example of a TCP server. Second we want to introduce you to a behaviour that you have yet to see in this book. That behaviour is the gen_fsm behaviour. Our HTTP server will be built around this new behaviour. That said, let's dive into our server.

11.2 The restful_interface Application

Now we will undertake to create our RESTful interface application. This is quite exciting because we are going to implement a scaled down web server as part of this interface. At the end of it we will have an efficient implementation of an HTTP server and a good RESTful API. This is going to allow us to release the simple cache beyond the confines of the Erlware project.

This new application starts out with the standard application code that you learned about in chapter 4. When we have finished creating that we should have the following tree, illustrated in code listing 11.5, in your overall project structure.

Code Listing 11.5 – Restful Interface Directory Structure

```
|-- restful_interface
|   |-- ebin
```

```
|   |   `-- restful_interface.app
|   `-- src
|       |-- ri_app.erl
|       `-- ri_sup.erl
|-- resource_discovery
|   |-- ebin
|   |   `-- resource_discovery.app
|   `-- src
|       |-- rd_app.erl
|       |-- rd_server.erl
|       |-- rd_sup.erl
|       `-- resource_discovery.erl
|-- simple_cache
|   |-- ebin
|   |   `-- simple_cache.app
|   `-- src
|       |-- sc_app.erl
|       |-- sc_element.erl
|       |-- sc_element_sup.erl
|       |-- sc_event.erl
|       |-- sc_event_guard.erl
|       |-- sc_event_logger.erl
|       |-- sc_store.erl
|       |-- sc_sup.erl
|       `-- simple_cache.erl
`-- tcp_interface
    |-- ebin
    |   `-- tcp_interface.app
    `-- src
        |-- ti_app.erl
        |-- ti_server.erl
        `-- ti_sup.erl
```

Now that we have our directory structure layed out and our framework in place, we can begin our implementation. We will be building the scaled down web server with the help of the gen_fsm behaviour. This behaviour is the last of the major behaviours that we cover in this book. It is extremely useful and we think you will find it to be one of the most elegant implementations of finite state machine behaviour anywhere. This is thanks to Erlang's ability to pattern match along with some good thinking and design by the OTP team. Before we dive into implementing the web server, we explain gen_fsm in a simpler setting. Hopefully this will make it easier to understand what we are doing with the web server. Once we finish talking about FSMs we can then talk a little bit about HTTP and then get right into our implementation armed with the knowledge of these technologies.

11.2.1 – The gen_fsm Behaviour

As with all behaviours the gen_fsm is based around a set of interface functions and callbacks. Table 11.4 lists these interface functions and callbacks.

Table 11.4 The gen_fsm entry point

gen_fsm module	Callback module
gen_fsm:start_link	Module:init/1
gen_fsm:send_event	Module:StateName/2
gen_fsm:send_all_state_event	Module:handle_event/3
gen_fsm:sync_send_event	Module:StateName/3
gen_fsm:sync_send_all_state_event	Module:handle_sync_event/4
N/A	Module:handle_info/3
N/A	Module:terminate/3
N/A	Module:code_change/4

Table 11.4 represents an interface function supplied by the gen_fsm module itself and the callback that is invoked to service a call made to each of the interface functions. handle_info, terminate and code_change function the same way they do in all other behaviours and have no associated callback function.

gen_fsm is a generic way to implement specific finite state machines. To illustrate what a state machine is we want to take something that is dead simple and explain it in the context of states. To that we will use the concept of a simple two way radio. This two way radio allows one person in a pair listen to audio and the other to capture it as he speaks. Figure 11.3 illustrates the state and transitions possible for this system.

Figure 11.3

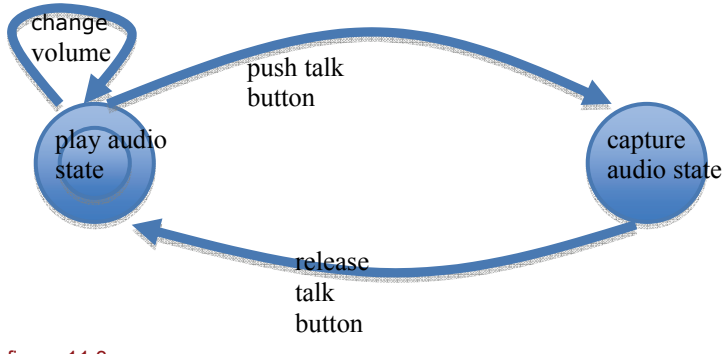
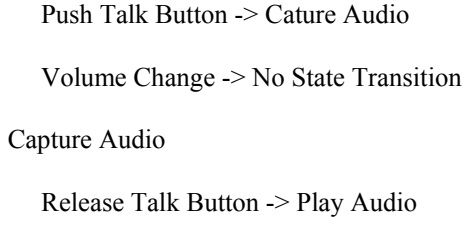


figure 11.3

TWO WAY RADIO STATE MACHINE

A state machine is defined as a set of states where actions take place, a set of events, and a set of transitions that occur as a result of events occuring while in particular states. In this case the states are the "play audio state" and the "capture audio state". There are three events, the "push talk button" event, the "release talk button" event, and the "volume change" event. When the push talk button event occurs from within the play audio state a transition to the capture audio state occurs. In this state, the action of capturing audio and transferring it to the other radio takes place. When in the capture audio state releasing the talk button causes a state transition to the play audio state. In this state, the action of playing audio captured from the other radio occurs. A "volume change" event within the play audio state causes the action within the listen state of a change up or down in the volume of the audio. A volume change action in the capture audio state is not defined and has no effect on this radio. Figure 11.4, illustrates the states and their transitions in a more consumable way.

Figure 11.4 Play Audio



Now that we understand the states and their transitions, lets take a look at how they would be implemented as a gen_fsm. We can see this in Code listing 11.6.

Code Listing 11.6 – 2 Way Radio gen fsm

```
-module(radio).
-behaviour(gen_fsm).

 -export([start_link/0]).        #1
-export([push_button/0, release_button/0, change_volume/1]).
-export([init/1, play_audio/2, capture_audio/2]).

-define(SERVER, ?MODULE).
-define(DEFAULT_VOLUME, 5).

-record(state, {volume = ?DEFAULT_VOLUME}).

start_link() ->
    gen_fsm:start_link({local, ?SERVER}, ?MODULE, [], []).

 push_button() ->        #2
```

```
    gen_fsm:send_event(?SERVER, push_button).

release_button() ->
    gen_fsm:send_event(?SERVER, release_button).

change_volume(Volume) when Volume >= -10, Volume =< 10 ->
    gen_fsm:send_event(?SERVER, {change_volume, Volume}).


init([]) ->
     spawn_player(?DEFAULT_VOLUME),
   {ok, play_audio, #state{}}.      #3

play_audio(push_button, State) ->
     stop_player(),
     spawn_capture_agent(),
   {next_state, capture_audio, State};      #5
play_audio({change_volume, Volume}, State) ->
    player_change_volume(Volume),
    {next_state, play_audio, State#state{volume = Volume}}.

capture_audio(release_button, State) ->
    stop_capture_agent(),
    spawn_player(State#state.volume),
    {next_state, play_audio, State}.
```

Taking a look at code annotation #1 we see three –export declarations. The first is for the function to start the server. The second are the functions that implement the events that this fsm can receive. The third function is the set of states that comprise our radio fsm. Skipping down to code annotation #2 we can see where our event wrapper functions start, with push_button/0 and immediately followed by release_button/0 and change_volume/1. These functions use gen_fsm:send_event to send events to the fsm we have registered in the start_link/0 function as ?SERVER. At code annotation #3 we have the last line of our init function being {ok, play_audio, #state{}}. That last line is important, but it can be a bit confusing. The second tuple indicates the initial state of our finite state machine. The third element of that tuple contains the internal state of our server, basically where we are going to stick our variables. This is exactly the same as you have seen so many times before in gen_servers. Try not to let the two different kinds of state confuse you.

As we just mentioned the 'play_audio' state is the initial state of the fsn. Jumping down into the middle of the function that implements that state at code annotation #5 we can see that the function has two clauses. At code annotation #5 we see the implementation of the transition to the capture_audio state upon receipt of the push_button event. The second clause here is for the volume change event and the transition there is back to the play_audio state as indicated by our state diagram in figure 11.4.

States and their transitions is what the finite state machine is all about. Take a minute and review the code we have just gone over. Did you notice how each state is represented by a function and each function can have multiple transitions represented by function

clauses? If you get that then we can move onto implementing our HTTP Server. Before we get into the implementation of the HTTP server though we need to understand a bit about the HTTP protocol itself.

11.2.2 – A Quick and Dirty Introduction to HTTP

This book however is not about HTTP it is about Erlang, so we are not going to spend a ton of time on HTTP, nor are we going to implement a fully fledged complete webserver. We are going to learn what we need to know about the protocol and then implement just enough of it to create a very functional RESTful interface for our cache. We are going to use a couple of unix utilities to explore the HTTP protocol in a hands on way. The first is the nc utility, which is a utility for inspecting protocols over TCP. This utility allows us to easily bind a listening socket and inspect whatever is sent over to it. The second is the curl utility, which is a command line http client. If we combine the two we can easily see what an HTTP request looks like. We will start with GET. The first snippit is getting the socket setup with nc and instruct it to listen on port 1156:

```
Macintosh:~ martinjlogan$ nc -l 1156
```

Now that we are listening we can send over an HTTP GET request with curl.

```
Macintosh:~ martinjlogan$ curl http://localhost:1156/foo
```

Next come the results

```
Macintosh-2:~ martinjlogan$ nc -l 1156
 GET /foo HTTP/1.1          #1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
```

What we see, with the help of nc, is the http packet that curl sent over. Lets break it down. The initial request line is the first line and is annotated as #1. This line tells us the request type, GET, it provides path or "request uri" information /foo and finally the version of HTTP it is using which is 1.1. Following the initial request line are the headers. Headers are bits of information passed along over HTTP that instruct servers to do various different things or provide additional information about the request. The header names are separated from their values with the : character. Following the header will always be a blank line and then the message body if there is one. In this case there is no message body. Basically the structure of HTTP messages are as such:

1. Initial line
2. Header lines (0 up to n)
3. Blank line

4. Message body

Our restful interface will also use PUT and DELETE so let us take a quick look at packets for each of those methods.

To demonstrate PUT we will put a small file named put.txt with the word "Erlang" inside it as its only contents. Starting with the curl command. The –T option instructs curl to PUT the file following the option.

```
Macintosh:~ martinjlogan$ curl -T put.txt http://localhost:1156/foo
```

and now for the output of nc.

```
Macintosh:~ martinjlogan$ nc -l 1156
PUT /foo HTTP/1.1
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
Content-Length: 7
Expect: 100-continue

Erlang
```

Ok, very similar to the GET request we just looked at, but with a few differences. The initial request line contains the method PUT instead of GET like the last time. At the end of the packet we can see the newline followed by an actual message body. This contains the expected the contents of our file; the word "Erlang". Notice the header highlighted above though Expect: 100-continue. If you watched carefully as you send the PUT request over to nc you will have seen that the contents of the file were not sent right away. This is because the Expect header was sent with the request. The expect header was implemented in part to make the internet more efficient. When it is included in a request the server is expected to send back a 100 Continue packet instructing the client to continue on and send the message body. If the server is not prepared to handle the body it can close the connection thus saving the client from sending, potentially, a large amount of data over the wire to a server that was not even prepared to do anything with it. When we implement our web server you will see where we handle this case.

This brings us to the last of the methods that we will use in our RESTful interface, DELETE. As before we will use the combination of nc and curl to examine a packet example.

```
Macintosh:~ martinjlogan$ curl -X DELETE http://localhost:1156/foo
```

and the results

```
Macintosh:~ martinjlogan$ nc -l 1156
DELETE /foo HTTP/1.1
```

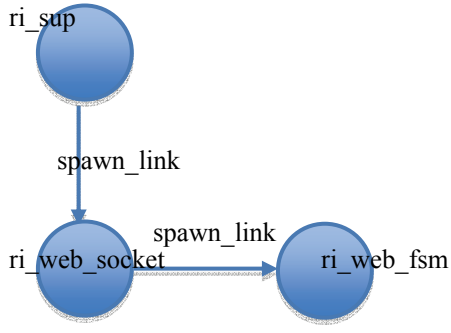
```
User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3
OpenSSL/0.9.7l zlib/1.2.3
Host: localhost:1156
Accept: */*
```

The DELETE request is very similar to that of GET with the only major difference being that in the initial request line the method is shown as DELETE instead of GET. The semantics are quite different. With our GET request we asked for the resource referenced by /foo and now we are asking to remove it. This examination of packets puts us in a good place. We are now ready to move forward and actually implement our scaled down web server.

11.2.3 – Building the HTTP Server

Each request to the web server will be handled by a pair of processes. One will handle the TCP socket and the other, the fsm, will handle parsing the HTTP protocol. When each HTTP connection is made a simple one for one supervisor will spawn a gen_server called ri_web_socket. That process will spawn and link with an FSM called ri_web_fsm. If one goes down so does the other and the connection is automatically cleaned up for us. Figure 11.4 shows the process structure.

Figure 11.5



The listen socket is created in the ri_app module and passed through the supervisor structure into each of the spawned ri_web_socket servers so that is can accept incoming connections on it. The design and interaction of the ri_sup and the ri_web_socket server in terms of how connections are accepted is exactly the same is it was for ti_sup and ti_server earlier on in this chapter. The only major difference here is the way that ti_server reads data off of the socket. Instead of using {active, true} option we are using the active once pattern. This is a very nice pattern because it offers the flow control you get with direct

reads and {active, false} but also the continuity that you get with using active true. The following code segment in listing 11.7 contains all our handle info clauses and demonstrates the use of active once.

Code listing 11.7 – Active Once Processing

```
handle_info({tcp, Socket, Packet}, #state{fsm_pid = FSMPid} = State) ->      #2
    ri_web_fsm:packet(FSMPid, Packet),
    inet:setopts(Socket, [{active,once}]),
    {noreply, State};
handle_info({tcp_closed, _Socket}, State) ->
    error_logger:info_msg("socket closed~n"),
    {stop, normal, State};
handle_info(timeout, #state{lsock = LSock} = State) ->
    error_logger:info_msg("waiting to accept an incoming connection~n"),
    {ok, Socket} = gen_tcp:accept(LSock),
    error_logger:info_msg("connection received on socket ~p~n", [Socket]),
    ri_sup:start_child(),
   inet:setopts(Socket,[{active,once}]),       #1
    {noreply, State#state{socket = Socket}}.
```

Code annotation #1 is the section of the code in which this is first activated. The listen socket is initially set up as active false in ri_app as follows.

```
{ok, LSock} = gen_tcp:listen(Port, [binary, {active, false}, {packet, raw},
{reuseaddr, true}]),
```

Then here in the timeout clause of handle_info it is accepted and the inet:setopts/2 function is used to modify the properties of the connected socket to make it active once. This means that a single message will be delivered into the process mailbox and then the socket will revert back to active false. When that single message is delivered at code annotation #2 you can see that we send it over to our ri_web_fsm with the packet/2 function and then immediately set our socket to active once again. This process repeats until we are finished receiving data. This pattern is a useful one anytime you want to be able to control flow with regard to your socket connections.

In the tradition of the first fsm example we will go through another diagram illustrating the transitions and actions that can take place in this state machine. Figure 11.5 below illustrates this.

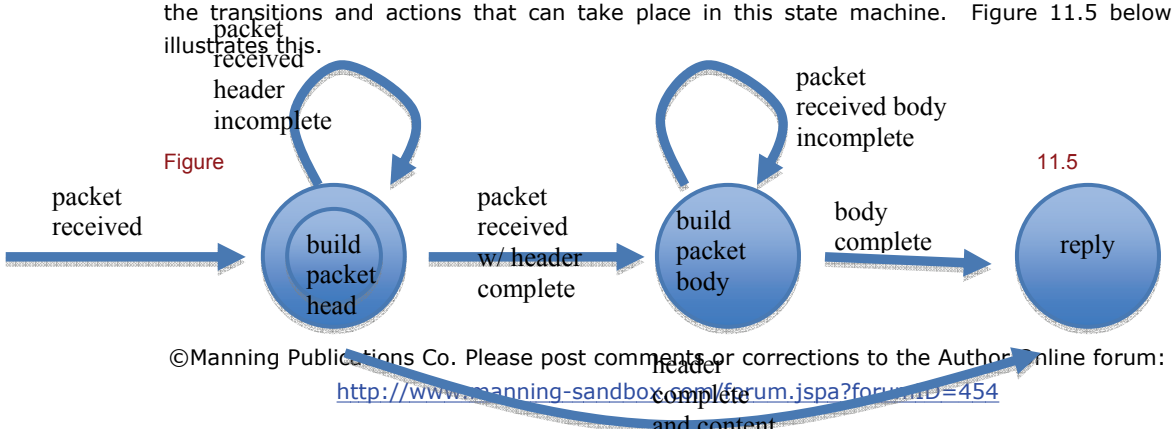


Figure 11.5

When you look at this the state machine is not too complex. There are only 3 states and the main events are receiving a connection, receiving a header packet, and receiving a body packet. This web server is not a full fledged web server so it only implements get, put, and delete minimally – just enough to support our RESTful API. Remember that our purpose here is to get you familiar with TCP programming in Erlang and the gen_fsm behaviour.

Code listing 11.8 is an excerpt of pi_web_fsm code. This is the code that handles our web serving needs. The pi comes from the fact that this web server functionality will be added onto our simple cache within an application called protocol buffers interface from which we took the prefix P.I.

Code Listing 11.8 – pi_web_fsm

```
-module(pi_web_fsm).

-behaviour(gen_fsm).

-export([start_link/1, packet/2]).

-export([
         init/1,
         handle_event/3,
         handle_sync_event/4,
         handle_info/3,
         terminate/3,
         code_change/4
        ]).

 -export([       #1
         build_message_head/2,
         build_message_body/2,
         reply/2
        ]).
-record(state, {socket_manager, head = [<<>>], body = <<>>,
content_length}).
```

```
-define(SERVER, ?MODULE).

start_link(SocketManager) ->
    gen_fsm:start_link(?MODULE, [SocketManager], []).

packet(FSMPid, Packet) ->
    gen_fsm:send_event(FSMPid, {packet, Packet}).

init([SocketManager]) ->
    error_logger:info_msg("pi_web_fsm:init/1~n"),
    {ok, build_message_head, #state{socket_manager = SocketManager}}.

build_message_head({packet, Packet}, #state{socket_manager = SM, head = Head} = State) ->  #2
    NewHead = decode_initial_request_line_and_header(Head, Packet),
    case NewHead of
        [http_eoh|Headers] ->
            CL = list_to_integer(
                  header_value_search('Content-Length', Headers, "0")),
            NewState = State#state{head = lists:reverse(Headers),
                                   content_length = CL},
      send_continue(SM, Headers),       #3
            case CL of
                0 ->
                    {next_state, reply, NewState, 0};
                CL ->
                    {next_state, build_message_body, NewState}
            end;
        _ ->
            NewState = State#state{head = NewHead},
            {next_state, build_message_head, NewState}
    end.

build_message_body({packet, Packet}, State) ->      #4
    #state{body           = Body,
           content_length = ContentLength} = State,
    NewBody  = list_to_binary([Body, Packet]),
    NewState = State#state{body = NewBody},

    case ContentLength - byte_size(NewBody) of
        0 ->
            {next_state, reply, NewState, 0};
        ContentLeftOver when ContentLeftOver > 0 ->
            {next_state, build_message_body, NewState}
    end.

reply(timeout, State) ->
   #state{socket_manager = SocketManager,
           head          = [InitialRequestLine|_],
           body          = Body} = State,
    Reply = handle_message(InitialRequestLine, Body),
    pi_web_socket:send(SocketManager, Reply),
    {stop, normal, State}.


terminate(normal, _StateName, _State) ->
```

```
    ok;
terminate(_Reason, _StateName, #state{socket_manager = SocketManager}) ->
    Err = create_http_message(500, "text/html", "500 Internal Server
Error"),
    pi_web_socket:send(SocketManager, Err),
    ok.
```

In the header portion of the code you can see how we have broken up the different types of exports. The API first containing only two functions, start_link/1 and packet/2. The start link function needs little explanation at this point. It receives SocketManager which is the process id of a process that manages the actual TCP socket setup for a given http connection. We will go into that just a bit later. The second function packet/2 is called when generating a packet event i.e. sends a packet of http data over to this FSM. Skipping down to the third export attribute defined in code annotation #1 we get to the exported state functions. For each state we wish to define as part of our FSM we export a function corresponding to the state name and the exports are placed here in the third export attribute. The first export representing the first state is build_message_head which can be seen at code annotation #2. This state as you can see from the state diagram in figure 11.5 receives packets until such time as the header is completely received and put together. Then, depending on the content length received as part of the header, transitions either to reply, or to the build_message_body state. The content length represents the number of bytes contained in the body of the http request. The first thing we do in the build_message_head state is call decode_initial_request_line_and_header/2, an internal function, passing it Head and Packet. Head is a state variable. It is a list of accumulated constructed from the data packets received. Packet is the new packet that has just arrived with a packet event. The decode_initial_request_line_and_header/2 function listed below in code listing 11.9 decodes the new packet and joins it with the accumulator Head.

Code Listing 11.9 – pi_web_fsm Decode Header Internal Function

```
decode_initial_request_line_and_header([Unparsed], Packet) ->
    decode_initial_request_line(list_to_binary([Unparsed, Packet]));
decode_initial_request_line_and_header([Unparsed|T], Packet) ->
    decode_header([list_to_binary([Unparsed, Packet])|T]).

decode_initial_request_line(Packet) ->
    case erlang:decode_packet(http, Packet, []) of
        {more, _} ->
            [Packet];
        {ok, IRLine, Rest} ->
            decode_header([Rest, IRLine]);
        Error ->
            throw({bad_initial_request_line, Error})
    end.

decode_header([Unparsed|Parsed] = Headers) ->
    case erlang:decode_packet(httph, Unparsed, []) of
        {ok, http_eoh, <<>>} ->
```

```
            [http_eoh|Parsed];
        {more, _} ->
            Headers;
        {ok, MoreParsed, Rest} ->
            decode_header([Rest, MoreParsed|Parsed]);
        {error, Reason} ->
            throw({bad_header, Reason})
    end.
```

To understand this function we need to talk again about http packets. For this we look to the example PUT packet we examined earlier, more specifically we look just at the initial part of the packet, not the body. Remember from earlier that HTTP messages have the following structure:

1. Initial line
2. Header lines (0 up to n)
3. Blank line
4. Message body

Going step by step through what happens with this function; the first packet is received by the build_message_head function, we have no idea what is contained in the packet, it is sent to the . It is sent to the decode_initial_request_line_and_header with the default Head which is [<<>>] which drops us into the first clause of the function wherein we parse out the initial line. The data from the new packet is decoded by the very convenient erlang:decode_packet/2 function and we have three scenarios,

1. Complete initial line which is appended to the list and the leftover which may be only <<>> but in either case is placed as the first thing in the list. So we have [UnparsedHeader, ParsedInitialLine] and we use that data to call into decode_header and continue on parsing the UnparsedHeader data in similar fashion.
2. (Very rare circumstances) We still don't have a complete initial line so we add what we do have to the empty binary in the Head accumulator. [UnfinishedInitialLine]
3. Something went wrong and an exception is generated and our process dies.

When all is done we return to the build_message_head function and check to see if we have completed the headers section of our http packet (which includes the "initial line" as well in this implementation). If we have not we wait for the next packet and use the decode_initial_request_line_and_header/2 function again. If we have completed the headers section, and the erlang:decode_packet/3 function will tell us when we have, we check our content length and either go to the reply state or over to the build_message_body state. There is one thing we do before we go to the next state and that is check to see if we need to send the 100-continue packet and send it if it is required. This is done in the function

below which is called from the build_message_head state function at code annotation #3. The code is below:

```
send_continue(SocketManager, Headers) ->
    case lists:keymember("100-continue", 5, Headers) of
        true  -> ri_web_socket:send(SocketManager,
                                    create_http_message(100));
        false -> ok
    end.
```

Using the key_member/3 function from lists we check to see if the "100-continue" header has been sent. If it has we create the 100 continue message with our create_http_message/1 internal function and use the ri_web_socket:send/2 function to send it back over to the client. The 100 continue packet will prompt the client to send the body of the HTTP message. If there is no 100 continue header present then this function is a no-op.

Assuming we have content, i.e. our content-length header value is greater than 0 we move on to the build_message_body state. This state is rather simple, collect packets and append them to the Body accumulator in our fsm state until the number of bytes collected matches the content length. The function is shown below.

```
build_message_body({packet, Packet}, State) ->
    #state{body           = Body,
           content_length = ContentLength} = State,
    NewBody  = list_to_binary([Body, Packet]),
    NewState = State#state{body = NewBody},

    case ContentLength - byte_size(NewBody) of
        0 ->
            {next_state, reply, NewState, 0};
        ContentLeftOver when ContentLeftOver > 0 ->
            {next_state, build_message_body, NewState}
    end.
```

Notice that we don't check for the condition where in the number of bytes collected exceeds the content length. This is because if it does then we will get a case clause exception and this process will die. When the process dies we use the catch all terminate behaviour function to simple send a 404 to the client as pictured below. This is the Erlang way, skip the error checking and let process death with a little assistance from OTP behaviours handle the rest.

```
terminate(normal, _StateName, _State) ->
    ok;
terminate(_Reason, _StateName, #state{socket_manager = SocketManager}) ->
    Err = create_http_message(500, "text/html", "500 Internal Server
Error"),
    ri_web_socket:send(SocketManager, Err),
    ok.
```

Once the body is collected, or in the case that the content was 0 and there was no body, we jump to the reply state. The way that the reply state is reached is a little different than the two states we have just covered. We get there by generating a timeout. This is done the same way you have seen it done on a few occasions with gen_servers by setting a timeout value of 0 in the return from a behaviour function as demonstrated below.

```
{next_state, reply, NewState, 0};
```

This causes the fsm to jump immediately into the reply state with the event 'timeout'. The reply state is below:

```
reply(timeout, State) ->
    #state{socket_manager = SocketManager,
           head           = [InitialRequestLine|_],
           body           = Body} = State,
   Reply = handle_message(InitialRequestLine, Body),      #1
    ri_web_socket:send(SocketManager, Reply),
    {stop, normal, State}.
```

The reply state is where we execute the action dictated by the HTTP message and then reply to the client.

Code annotation #1 is where the insert, fetch, or delete is performed on the cache depending on whether we got a GET, PUT, or DELETE sent over and then on the next line we use ri_web_socket:send/2 to send across the reply we generated as a result of performing the operation at code annotation #1.

We want to quickly present an alternate approach to using the reply state. This approach is actually preferred slightly but because we wanted to demonstrate the use of timeout 0 we added in the reply state. It also makes for a slightly less trivial FSM. The other approach that can be used is to utilize the terminate function to do what reply currently does. Instead of using timeout 0 and directing to a new state we could have returned {stop, normal, State} which would drop us into the terminate function which would tie everything together; perform the simple_cache operation and send a reply just before exiting the FSM. When playing with the code implement both and see which feels more elegant to you.

Now that we have implemented our HTTP server it's time to implement the rest service that our HTTP service will front.

11.2.4 Getting RESTful

Our REST based protocol is very simple, as REST based protocol should be. We support three actions in our protocol, GET, PUT and DELETE. Each of these entry points in our system have a specific format associated with them.

The get entry point is, perhaps, the simplest of the three. This is used for retrieving a a value from the cache. It expects the request to come in as an GET request with the following format.

```
GET /key
```

If we receive this request then we return the value in the cache associated with that key.

The delete entry point follows closely the format of the get request, with the exception that it expectes it to come in as an HTTP DELETE request. We expect a url of the following form.

DELETE /key

If we receive this request we are expected to delete the value specified by key from the cache.

Finally we have the most complex of the entry points, the put entry point. We expect this request to come in as an HTTP PUT request. The main difference between this entry point and the get/put entry point is that the message has a body. That body will contain the value we are expected to insert into the cache. We expect a url of the following form.

PUT /key

If we receive this request we are expected to insert a new value into the simple_cache. This value will be identified by the 'key' in the url with a value specified by the body of the message.

Implementing Our Protocol in the HTTP Server

We implement our protocol in the handle_message function of our HTTP server. The code is listed in code listing 11.10.

Code listing 11.10 – handle_message/2

```
handle_message({http_request, 'GET', {abs_path, [$/|Key]}, _}, _Body) ->
    case simple_cache:lookup(Key) of
        {ok, Value}        -> create_http_message(200, "text/html", Value);
        {error, not_found} -> create_http_message(404)
    end;
handle_message({http_request,'PUT',{abs_path, [$/|Key]},_}, Body) ->
    simple_cache:insert(Key, Body),
    create_http_message(200);
handle_message({http_request, 'DELETE', {abs_path, [$/|Key]}, _}, _Body) ->
    simple_cache:delete(Key),
    create_http_message(200).
```

This function pattern matches on the initial line of the HTTP message in order to figure out what operation was performed and to capture the request URI information. It then performs the appropriate simple_cache function and creates a response message.

We have now implemented all of the protocols we mentioned in the first part of this chapter. Our text interface does not give us much freedom with regard to what we can store in the cache. It is pretty limited, which is fine for storing certain types of data. It has the benefit that it is pretty easy to implement and use. Our second protocol, the REST based protocol, puts the full power of HTTP at our disposal for transporting data into and out of our cache.

11.3 – Summary

The simple cache is now capable of talking with external clients through two different protocols, our custom text based protocol and the more structured RESTful interface. Along the way we learned a very important technique/pattern for creating a concurrent TCP server, enough HTTP to be dangerous, and we rounded out our coverage of behaviors with the gen_fsm. This work has made it possible for the simple cache to be used in production environments outside of an Erlang shop which is certainly useful in today's "right tool for the right job" multi-language and multi-platform production environments. In the next chapter, we will cover how to get Erlang to interact directly with other languages like, C, Java to give you the skills for even tighter multi-language communication.

12

Drivers and Multi-Language Interfaces

At this point we have a really nice implementation of the cache. It already does a number if interesting things; things like provide a persistent store, distributes the cached objects, handles session objects, etc. Over all the Erlware guys are very happy. However, there is one small problem. Several of the objects that are cached are getting quite large. That's making those objects a bit slow to distribute and they are taking up a lot of memory. After a short analysis it looks like these large chunks of data are pretty compressible. So Erlware guys put out a requirement that the cache support some type of compression. That's generally not a problem, however, there aren't really any good compression libraries in Erlang. There are a several very good compression libraries out there of course, but they are written in C though. it seems prudent to take one of those libraries and wrap it for use in Erlang rather than try to implement a compression scheme from scratch in Erlang. This approach also gives us the opportunity to introduce you to Erlang's specialized foreign function interface.

I said Erlang's 'specialized' foreign function interface for a reason. At the highest level Erlang is no different from any other language; it allows you to make use of libraries and code written in other languages. However, as you dig deeper it becomes apparent that Erlang is actually very different. Most other languages have a foreign function interface to C. Where the languages provides some way to compile or links in C code that the language may, then call into. Erlang does it differently; it extends the message passing paradigm to interacting with foreign code as well. All of Erlang's foreign interface approaches rely on passing messages back and forth between the foreign code and the Erlang system. The method of

transport and the 'closeness' between Erlang and the foreign code are really the only things that differentiate a C interface from one in Java or Python or Ruby.

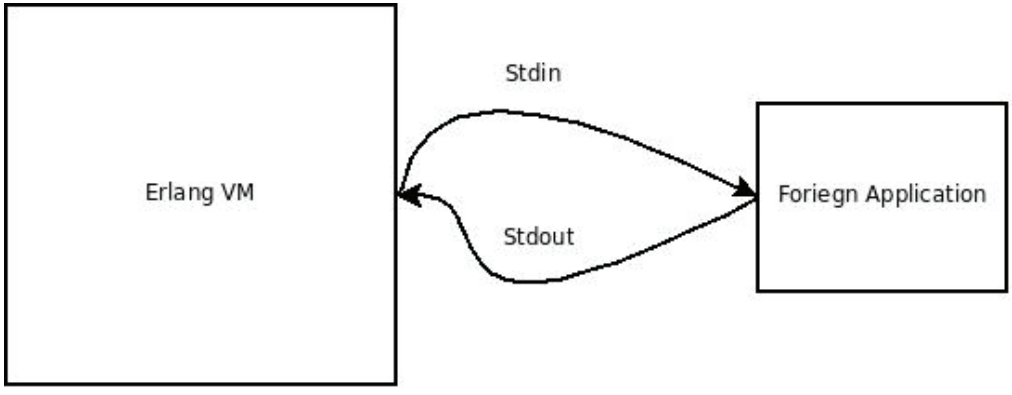
There are two types of foreign function interfaces in Erlang. These are the Port Driver and the Linked in Driver. The Port Driver and the Linked in Driver have similar high level semantics but very different low level semantics. The Port Driver method uses stdin and stdout to pass data to and from a running application via a "pipe". Think of this pipe as exactly that, a conduit for passing information back and forth. Using this method of foreign language interface is very much language neutral. For example, Erlang can actually interface to any language you want, from shell scripts to assembly to scripting languages. If it can run and read from stdin and write to stdout Erlang can interface with it via Port Drivers. The other way, the Linked in Driver is much less flexible and safe for that matter. That is because it is exactly what its name implies, A Linked in Driver, a shared object that is compiled to native code that Erlang loads into its own address space. This has the benefit of being very fast, but a Linked in Driver has the potential to segfault the entire Erlang VM. This isn't something that can happen when using a Port Driver by virtue of the fact that the Erlang VM and what it is communicating with via a pipe are different OS processes.

In this chapter we are going to talk about both of these approaches. We will talk about how to use them, more importantly we will talk about when to use them. Both of these approaches have their own benefits and drawbacks. Understanding those benefits and drawbacks of each individual approach is key to choosing the right one to use in the context of your system. All this information will be used in the end to wrap a compression library for integration with the Simple Cache.

12.1 Port Drivers

Port Drivers are the most common way to communicate with foreign code in Erlang. Port Drivers are actually rather simple. A Port Driver is just a program that can read from standard in and write to standard out. To use this program Erlang will start it up as a managed process and communicate with it over these channels. If the program being communicated with crashes then Erlang can just restart it and continue on its way (assuming of course, that you designed the program with this in mind). The program lives in its own process space and the only interaction it has with Erlang is over these standard io channels. That means that nothing the program does really has the ability to impact the running Erlang system. Considering the unsafe nature of most foreign code that's a very big benefit. You can see in Figure 12.1 how communication with the remote code progresses.

Figure 12.1 Communication Between Erlang and Remote Code



Of course, there is no such thing as a free lunch. To get this level of safety, you pay a price in speed. All data that moves between the two processes must be marshaled and unmarshalled. Not only that, but you need to define the marshalling and unmarshalling semantics. Basically you need to define a protocol through which the Erlang and the running foreign code will talk to each other. You should be rather familiar with this however due to all our hard work in chapter 11. For C, Erlang provides some libraries that you can use. For other languages, like python, you need to define your own. Depending on the application and your needs this can be very simple or very complex.

12.1.1 Creating The Port Driver

By way of example, we are going to create a very simple Port Driver that interfaces Python code to Erlang. Its whole purpose in life will be to accept strings from the Erlang side and echo those strings back to Erlang. The protocol we will use to communicate with the application will also be very simple. It is just a new line delimited string. Of course, a real application will use a more complex protocol. Don't worry; we will get into more complex real world examples later. These are just little steps to get us up to speed.

Every Port driver consists of, at the very least, two parts. A process on the Erlang side and a remote OS process that is spawned via a port. Lets dig into the Erlang side here first and then we will examine the OS process side.

THE ERLANG SIDE OF A PORT DRIVER

The Erlang process side of the equation for us is of course, yup you guessed it, a gen server. In our example, we are not going to show you all the gen server implementation code. You have seen examples of that already in many previous chapters. In explaining our implementation let's start with our API to the world. We expose just one function and that is 'send_msg'.

```
send_msg(Msg) ->
    gen_server:cast(?SERVER, {send_msg, Msg}).
```

As we do in almost all gen_server modules we simply wrap the actual gen_server call in an API function. The whole point of send_msg/1 is to simply take a string, wrap it up, and send it to the gen_server which will transmit the message over our port connection. Before we show you the code to send the message however lets deal with creating the actual port which starts the remote Python application. This happens in init/1 after we start our gen server via the start link API function. Init/1 calls the internal function create_port/0 which is shown in the next snippet.

```
create_port() ->
    PrivDir = code:priv_dir(port_example_1),
    open_port({spawn, filename:join([PrivDir, "test.py"])},
           [stream, {line, 1024}, exit_status]).
```

Unless you have a good reason not to you should drop your foreign application in the priv dir of the Erlang/OTP application that will be fronting it. That's what we have done here. We can always get the full path just by calling code:priv_dir with the name of the application. That's what we do here. To spawn the port we call the open_port/2 function from the Erlang module. We pass it a tuple with the spawn atom and the fully qualified path to the application we want to start.

The second argument to open_port is a list of options used to send details to the port subsystem on how to treat the data going back and forth between Erlang and the foreign application. In this case we specify a few options; we want it to be a stream based protocol. That means that the port subsystem is not going to try to introspect the data at all. The second option is the 'line' tuple. This tells the port subsystem that we want a newline based protocol and the maximum length of a line is 1024 characters. Finally we tell Erlang that we

want to get the exit_status as well as the data. We are going to do something interesting with this.

The next snippet contains the code that we use to send a message through our port into our remote application. You can tell from the message tag that messages received here come in from the send_msg API function.

```
handle_cast({send_msg, Msg}, State = #state{port=Port}) ->
    erlang:port_command(Port, Msg ++ "\n"),
    {noreply, State}.
```

We expect a message to always be in the {send_msg, Msg} format. When we get the message we append a newline (just to be sure) and call erlang:port_command. This is the Erlang function that actually sends the command over the pipe into our foreign application. It just takes the Port and the data to send. Since our protocol is so simple, we can get away with just sending the raw string. The next snippet contains the code for receiving the data returned from the Python code.

```
handle_info({Port, {data, {_, Data}}}, State = #state{port=Port}) ->
    io:format("~p~n", [Data]),
    {noreply, State};
```

The port subsystem just sends us raw data containing the input from the program. It doesn't know that we are using a gen_server and so cannot use the more normal gen_server functions and so all of our information comes over as out of band into the handle_info function. The data comes into handle_info as a tuple within a tuple. The outermost tuple contains the Port itself and the data tuple. Notice that we use the Port in our state to match the Port in our message tuple. That allows us to trivially guard against getting data from the wrong Port. Granted there is little chance of that happening here but it's still a good practice to get into and is a light example of programming by contract. The contract being that for this function to be executed the sending port must be the same one that is stored in state. The second part of the tuple contains the data associated with our system. The actual format of the data tuple will change with the type of streaming the port driver is requested to do. In our case, we are using a simple stream based protocol where newlines are import so we get a tuple of the form {data, {eol, Data}} where the Data is newline delimited. We could also get a tuple in the form {data, {noeol, Data}}, which just means that the data coming back wasn't newline delimited. In our case, we don't actually care. In other cases, it may be important.

Because we requested 'exit_status' from the port subsystem, another message we can get is an 'exit_status' message when the foreign application exits. The code for handling this is in listing 12.1.

12.1 Handling The Exit Status

```
handle_info({Port, {exit_status, Status}}, #state{port=Port})
  when Status == 1 ->
    io:format("BAD SHUTDOWN:~p~n", [Status]),
    {noreply, #state{port=create_port()}};
handle_info({Port, {exit_status, Status}}, #state{port=Port})
  when Status == 0 ->
    io:format("GOOD SHUTDOWN:~p~n", [Status]),
    {noreply, #state{port=create_port()}}.
```

You can use this exit status to track and even to manage the foreign application. We manage it by restarting the foreign application when it shuts down. Another potentially more elegant way of handling the foreign application is to make this gen_server a true proxy for the outer application. We would hook this application into a supervisor and make it a transient child. If the foreigh application exits with a status of 0 our server shuts down with the reason normal causing the supervisor to let it die without a restart. If the application shuts down with any other status our server dies abnormally causing the supervisor to restart it and thus causing the port to be respawned in init. For illustrative purposes our example serves us well enough but in production you may want to think about cleaving closer to the "OTP way". With our Erlang server built lets now look at the Python side of this example.

THE PYTHON SIDE OF THE PORT DRIVER

The other half of our Port Driver is the application in the foreign language. In this example it's a simple python module that just reads a line off stdin, introspects it to see if it is a command, then echos it back to stdout.

12.2 priv/test.py Python Port Driver Application

```
#!/usr/bin/env python

import sys

def main():
    """ Lets just echo everything we get on stdin to stdout,
    that should be enough to give us a good example """

    while True:
        line = sys.stdin.readline().strip()

    if line == "stop-good":          #1
            return 0
    elif line == "stop-bad":         #2
            return 1
```

```
        sys.stdout.write(line)        #3
            sys.stdout.write("\n")
            sys.stdout.flush()

if __name__ == "__main__":
    sys.exit(main())
```

Notice at code annotations 1 and 2 that if send this code the string 'stop-good' or 'stop-bad' we return a specific exit code. That's mostly so we can play around with restarts. Otherwise this code echos back whatever string is sent to it starting at code annotation number 3. In code listing 12.3 we take this code and run it in the shell to see what happens.

12.3 Running The Port Driver Example

```
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [smp:2:2] [rq:2] [async-
threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
1> application:start(port_example_1).
ok
2> pe1_py_drv:send_msg("hello").
ok
3> "hello"
```

As you can see we start up our port driver in the usual way, via the OTP Application start call. We then send it the message "hello". It happily responds back with a nice "hello" echo. It will do this for any string you want to send it. How about we play with shutdown and restart semantics for the foreign application. Remember that we set it up so that if it gets a 'stop-good' message the application exits with an error code of 0. If it gets a 'stop-bad' message it exits with an error code of 1. We can test this by just sending it those messages using the API.

```
4> pe1_py_drv:send_msg("stop-good").
ok
5> GOOD SHUTDOWN:0
6> pe1_py_drv:send_msg("stop-bad").
ok
7> BAD SHUTDOWN:1
```

Go ahead and send them both like have done here. You may notice that even though the application exits in both cases we don't have any problem sending another message. That's because we capture the exit and restart the application every time it crashes. The exit of the foreign application has no effect on the running of the Erlang node. That's the thing to take away from this section. Port Drivers are generally safe and easy to manage. For that reason if you can use them, you should use them.

We have done a very quick run through of port drivers. At this point you should understand how to create them and that they are a light and easy way to manage and communicate with foreign language applications running on the same host as an Erlang VM. Next we are going to get into Linked in Drivers. Linked in Drivers have the same basic purpose as Port Drivers. The allow you to communicate with a system not written in Erlang. That is where the similarity stops. These two paradigms have significantly different semantics.

12.2 Linked In Drivers

At the highest levels Linked in Drivers work exactly the same as Port Drivers. That is you have foreign code communicates with the VM via messages in a protocol that you define. The mechanism by which those messages travel is very different. Everything else, from the protocol definition through the actions taken can be exactly the same. That being the case though, there is one very big difference. That difference is that the Linked in Drivers live in same process space as the Erlang VM. The sole purpose of this is performance. Keep this purpose in mind because there are a lot of very negative ramifications to the Linked in Driver living in the same process space as the VM. The fundamental negative drawback to living in the same process space is that if the Linked in Driver crashes then it crashes the entire VM. This is a huge glaring issue that has the potential to destroy the fault tolerance of the system. This is exacerbated by the fact that Linked in drivers are written in C. C is a very low level language that leaves a lot of the error checking and resource management to the developer. Unfortunately, developers aren't actually very good at handling this error checking and resource management. In general, this makes code written in C and other low level languages much more error prone than similar code in a language like Erlang. Since, in this case, C code is being linked directly into the Erlang VM any one of these errors carries with it the very real potential to bring down the entire VM where all of your nice fault tolerant Erlang code lives. Supervisors will not save you here!

Linked In Drivers in Languages Other Then C

The idea that Linked in Drivers can only be written in C isn't exactly true.. You could probably write linked in drivers in other languages that are compatible with the C API and have the ability to build to a *.so file. In all likelihood you would have the same problem that you have in C. That is the ability to easily crash the Erlang VM.

Keep in mind that when you use a Linked in Driver **you are very specifically trading safety for speed**. This trade off should be made only when you are sure that you actually need the additional speed for your system. In general, when working with linked in drivers be aware of the tradeoff you are making and approach with caution.

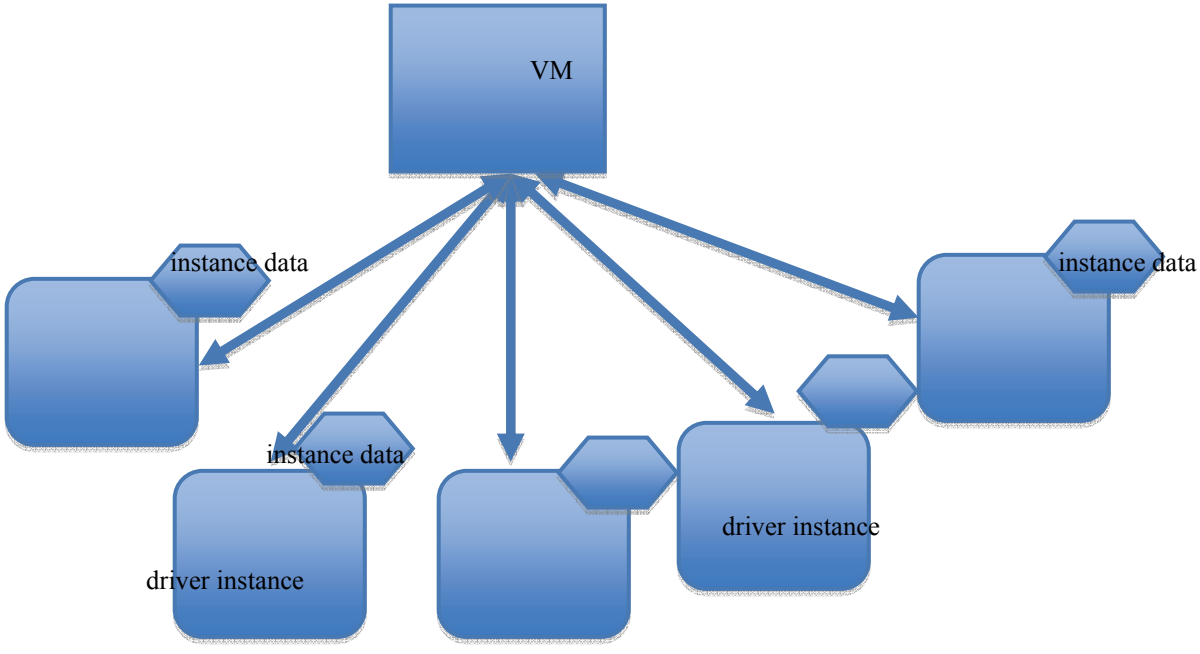
Now that we have all of the 'here be dragons' signs out on the map we can get into discussing some of the details of Linked in Drivers and get into the process of actually creating a linked in driver.

12.2.1 Understanding Linked In Drivers

As we have said Linked in Drivers share a lot of the same fundamentals as Port Drivers but the mechanism of those behaviors are different. In the case of Linked in Drivers the data that will be processed comes in on one of several callback functions that you define. These callback functions parse a request and then return a result. All of this occurs using a protocol that you, the developer, define. As with Port Drivers, the same Linked in Driver can be loaded into the system and run as a unique instance of that driver any number of times, the difference with linked in drivers is that all of these calls happen in the same OS process. This impacts how you handle the state of in your Driver. You must design your driver with the idea that it will be reentrant. This means that the same code can be called i.e. reentered by multiple threads and all invocations will produce the same result as if they had been executed alone.

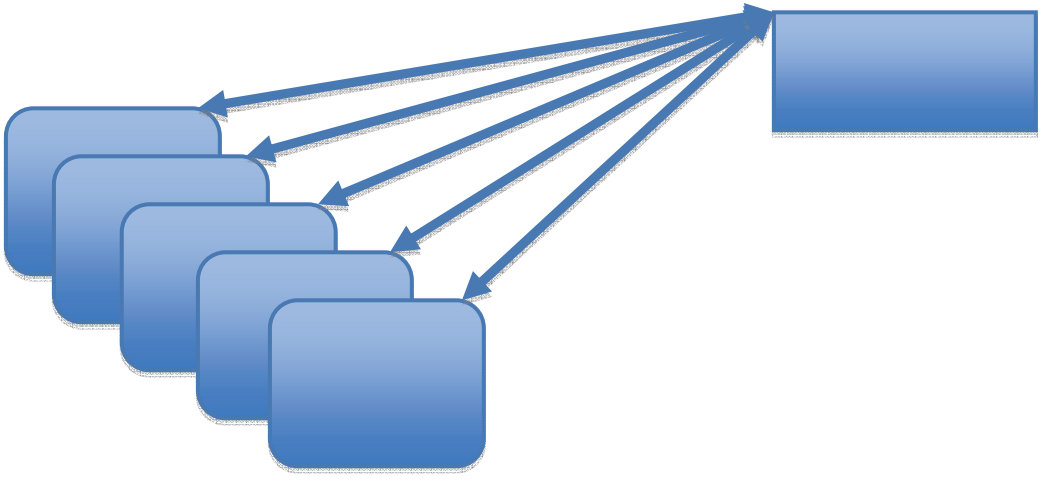
There are two types of long lived data that can be defined. The first type is of data are global variables. These are just global variables that you define in C, the same global variables create in any other context. The other type of long lived data is driver instance data. This is data that is specific to an instance of the driver. You do this via the state struct that you return from the start lifecycle function of your Driver. In the case of global variables the same data is shared across all instances of the driver. In the case of driver instance data each instance of a driver gets its own, instance specific, data. We can see a representation of this in figure 12.1.

Figure 12.1



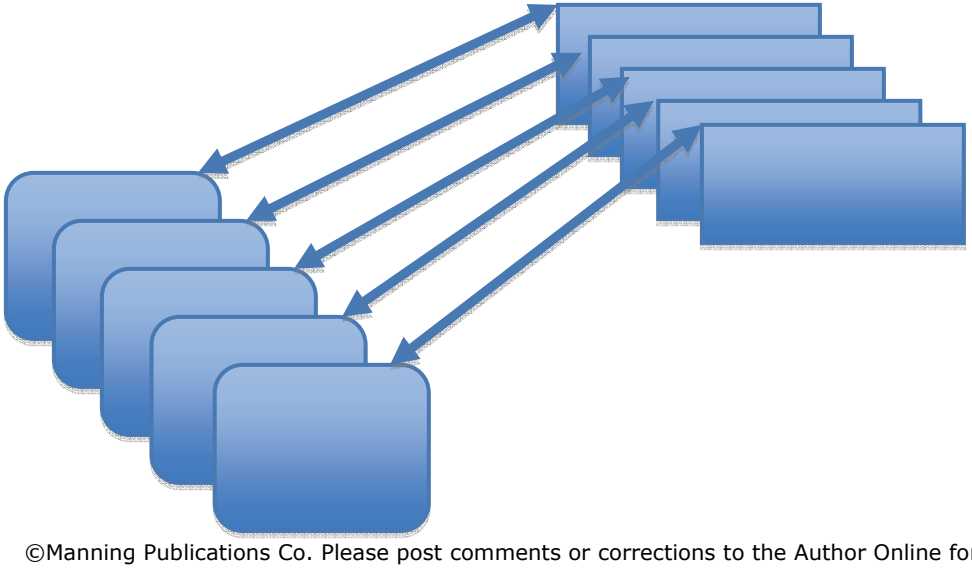
Lets say we have a very simple driver that just increments a counter every time we request it to. If we don't think about the issue of reentrance, we might just use a global variable called 'counter' to hold our count. Every time we get a request we increment that counter. We create this little driver and release it to our users. A user decides that he needs five of these counters so he loads up five instances of the driver. This is completely acceptable and supported in Erlang. Unfortunately, since we didn't think about this case when we designed the driver the instances stomps all over each other by each incrementing the same counter thus producing odd results for the user. We can see this instance of shared state in figure 12.2

Figure 12.2 – 5 instances sharing one bit of state



This isn't the behavior we want in our driver at all. Fortunately, there is a way around this. We have the ability to create a per-instance state in which we can store the instance specific state of our system. If we put our counter variable there we should get the behavior we are after. Instead of our system looking like that in figure 12.2 it will look more like figure 12.3.

Figure 12.3 – Separate state for each instance



As you can see in this figure, each instance has its own instance specific state. Nine times out of ten this is the behavior that you want and expect out of a linked in driver. As you get more advanced you may want to do other things. For now though, you can safely put all of your state in the state struct that you will initialize in your start function and return to the VM, it will then be passed into each of your callback functions.

Enough of abstractions! Lets get dive into some actual code. This example will have exactly the same function as our last example; echoing anything you send to it back to the sender. It should make some of these things we have been talking about much more concrete in your mind.

12.2.2 Building a Linked In Driver

We are going to get into the concrete details of building this Linked in Driver. We will start by creating a c file called basic_driver.c. For this simple example it is going to contain all of the code in our system. As with any C file we are going to start out by including the header files that we will need.

```
#include <erl_driver.h>
#include <ei.h>
```

In this example we are only including two header files, erl_driver.h and ei. You need the erl_driver.h header in every Linked in Driver. The ei.h header is included here because it contains the code that we are going to use for our protocol. Depending on the mechanism of your protocol you may need this one, one of the others, or none at all. We are exploring two protocols in this chapter. A simple but limited protocol defined in ei.h for this example and a more flexible protocol later on in the chapter.

Now we come to defining the state that will hold our driver instance data, as we touched on in the last section. The concrete form of that state is a struct where we store all of the variables that our driver needs from one function call to the next.

```
typedef struct _basic_drv_t {
  ErlDrvPort port;
} basic_drv_t;
```

What we are storing here is the data that represents our driver to the Erlang VM. If we had a need to store more instance data we would also include them in this struct.

The Erlang Virtual Machine communicates with the driver via a set of callback functions. Some of the functions that we need to implement are lifecycle functions, like start and stop. Others are called when data is available. Most drivers will implement the lifecycle functions, but they will only implement the communication functions that they need or want to use. In any case, these functions let the VM communicate in away that is somewhat similar to the way it communicates with any other process. There are 16 different possible callback functions that linked in drivers may implement. Fortunately, you don't have to actually implement all of them. You just need to implement the lifecycle functions and the communication functions that you are actually going to use. We are using three functions; two lifecycle functions and one communication function.

Before we get on to implementing these functions though, we need to describe them to the C compiler by declaring their prototypes. We don't actually need to put these in a header file since nothing is actually consuming a header file here.

```
static ErlDrvData start(ErlDrvPort port, char* cmd);
static void stop(ErlDrvData handle);
static void process(ErlDrvData handle, ErlIOVec *ev);
```

There are two things we would like you to notice here. The first is that we tend to name our functions after the name of the callbacks themselves. The second is that, for everything but the start function, the first argument is an ErlDrvData. That ErlDrvData represents the state that contains our driver instance data that we described above. Its passed to every callback function except start (that function creates it) so you will always have access to it.

Now that we have the callback functions that we intend to implement described to the C compiler via their prototypes we need to tell the Erlang VM about them. The VM has to know what to call for the various events that occur. We do that by defining a large ErlDrvEntry that contains pointers to the functions that we just described or NULL for those functions that we do not wish to implement. Let's look at the struct of callback function pointers that Erlang uses to find the entry points to your application.

12.4 basic_driver.c Linked In Driver Description

```
static ErlDrvEntry basic_driver_entry = {
    NULL,                              /* init */
    start,              /* startup */ #1
    stop,               /* shutdown */ #2
    NULL,                              /* output */
    NULL,                              /* ready_input */
    NULL,                              /* ready_output */
    "basic_drv",                       /* the name of the driver */
    NULL,                              /* finish */
```

```
    NULL,                                /* handle */
    NULL,                                /* control */
    NULL,                                /* timeout */
   process,              /* process */ #3
    NULL,                                /* ready_async */
    NULL,                                /* flush */
    NULL,                                /* call */
    NULL,                                /* event */
    ERL_DRV_EXTENDED_MARKER,             /* ERL_DRV_EXTENDED_MARKER */
    ERL_DRV_EXTENDED_MAJOR_VERSION,      /* ERL_DRV_EXTENDED_MAJOR_VERSION */
    ERL_DRV_EXTENDED_MAJOR_VERSION,      /* ERL_DRV_EXTENDED_MINOR_VERSION */
    ERL_DRV_FLAG_USE_PORT_LOCKING        /* ERL_DRV_FLAGs */
};
```

This is the lynch pin, the key to your driver. It describes to the Erlang VM what callbacks your Linked in Driver supports. Each position contains a pointer to a callback function that you may implement. Even though, we are only using three, start (annotation #1), stop (annotation #2) and process (annotation #3); we are going to go ahead and describe the rest so that you may use them in the future. As you can see there are a bunch of possible options, though you will never use them all.

Linked In Driver Interface Functions

Entry	Description
init	called at system start up for statically linked drivers, and after loading for dynamically loaded drivers.
startup	called when open_port/2 is invoked. return value -1 means failure.
shutdown	called when port is closed, and when the emulator is halted
output	called when we have output from erlang to the port
ready_input	called when we have input from one of the driver's handles
ready_output	called when output is possible to one of the driver's handles
driver name	name supplied as command in open_port XXX
finish	called before unloading the driver - DYNAMIC DRIVERS ONLY
handle	Reserved -- Used by emulator internally

control	"ioctl" for drivers - invoked by port_control/3)
timeout	Handling of timeout in driver
process	called when we have output from erlang to the port
ready_async	This function is called after an asynchronous call has completed.
flush	called when the port is about to be closed, and there is data in the driver queue that needs to be flushed before 'stop' can be called.
call	Works mostly like 'control', a synchronous call into the driver.
event	Called when an event selected by driver_event() has occurred

We will not be going to go into much detail on most of these. If you wish further knowledge the Erlang Documentation has it in abundance. For the most part, the short descriptions provided are plenty to get a reasonable understanding of what's going on.

Now that we understand how Erlang knows what functions to call, let's start getting into the functions that actually implement the core of our simple echo driver. The first function we are going to look at is the start function, which is listed in our next snippet. As we have discussed this function actually starts up our example and allocates the state that we will be using in the rest of this driver.

```
static ErlDrvData start(ErlDrvPort port, char* cmd) {
  basic_drv_t* retval = (basic_drv_t*) driver_alloc(sizeof(basic_drv_t)); #1
  retval->port = port;
  return (ErlDrvData) retval;
}
```

We allocate our state using driver_alloc in annotation #1. We assign the port to our state and then return that state back to the control of the VM in annotation #2. In this example that's all we actually need to do. You should, however, take note of the use of driver_alloc. The Erlang VM uses a garbage collector, as you know. The VM must have a good indication of what memory it is managing and what memory you are managing. By allocating with the driver_alloc function we basically tell the VM that it is managing the data we just allocated. In this case the VM is actually going to do that by calling our stop function.

```
static void stop(ErlDrvData handle) {
  basic_drv_t* driver_data = (basic_drv_t*) handle; #1
```

```
    driver_free(driver_data); #2
}
```

This is another lifecycle function. Here we just take the state that we created in the start function and free it (annotation #1). Once again, notice the driver_free (annotation #2). This matches the driver_alloc of the start function. You should use them in pairs, don't free something that was allocated with driver_alloc and vice versa.

In more complex Linked in Drivers it is very likely that you would be starting threads, initializing resources, etc in the start function and doing the opposite in the stop function. In our example, we don't need to actually do more then create and destroy our state.

12.6 The Process Function

```
static void process(ErlDrvData handle, ErlIOVec *ev) {
    basic_drv_t* driver_data = (basic_drv_t*) handle; #1
    ErlDrvBinary* data = ev->binv[1];
    ErlDrvTermData spec[] = {ERL_DRV_ATOM, driver_mk_atom("ok"),
                             ERL_DRV_BINARY, (ErlDrvTermData) data,
                             data->orig_size, 0,
                             ERL_DRV_TUPLE, 2}; #2
    driver_output_term(driver_data->port, spec,
                       sizeof(spec) / sizeof(spec[0])); #3
}
```

This is the one and only communication function that we are using in this example. Two things are passed into us. First is the ErlDrvData object that represents our reentrant state. The second is an ErlIOVec that contains the raw data that contains our protocol. The action that we want perform (echoing) happens here. To make that happen we are taking a term that the Erlang VM has sent us and are turning it around and sending it (in some nice wrapping) back to the caller. All we really need to do is put our reply in a format that VM can actually understand.

The very first thing we are going to do is cast our state to something we can actually use. The Erlang VM gives it to us as an ErlDrvData object, but its really just the state that we created in our start function. That being the case we can just cast it to the appropriate type (annotation #1).

All of our data comes in as an ErlIOVec object. We are not going to be trying to decode that here. There are plenty of examples of how to do that in the next section. We are, though, going to spend a few minutes looking at how we write data back to Erlang.

```
ErlDrvTermData spec[] = {ERL_DRV_ATOM, driver_mk_atom("ok"),
```

```
                        ERL_DRV_BINARY, (ErlDrvTermData) data,
                        data->orig_size, 0,
                        ERL_DRV_TUPLE, 2};

driver_output_term(driver_data->port, spec,
                   sizeof(spec) / sizeof(spec[0]));
```

Our protocol here is just going to be the Erlang binary term format. Erlang provides several libraries to marshall and unmarshall this protocol. We are using one of the simplest here. Using this library, all we need to do is describe the outgoing term to the Erlang VM (annotation #2). It does all the heavy lifting of encoding it correctly and seeing it back to the caller. We just need to create the description. We do this via a simple array of ErlDrvTermData objects. You may think of this array as little more than a stream of tags and data. In the listing above we see the tag ERL_DRV_ATOM followed by the actual atom itself, then ERL_DRV_BINARY and a description of the binary itself, followed by the size of the binary. Following this is our tuple definition. It's specified by the ERL_DRV_TUPLE tag followed by the size of the tuple. As you may have guessed the preceding two values are the contents of the tuple. This is really just a custom protocol for our system, we are using the library provided by Erlang to do the encoding and decoding.

Finally all of this is passed to driver_output_term with the appropriate parameters (annotation #3). This converts the spec to a binary form and sends that back to the caller.

12.2.2 The Erlang Side

On the Erlang side we need to load the driver into the VM and make it available for to this system. Doing this is almost exactly the same as what we did for the port driver and is shown in the next snippet.

```
load_driver() ->
  SearchDir = code:priv_dir(port_example_2), #1
  case erl_ddll:load(SearchDir, "basic_drv") of #2
    ok ->
    {ok, open_port({spawn, 'basic_drv'}, [binary])}; #3
    Error ->
      Error
  end.
```

As you can see we simply find the driver on the file system (annotation #1) and then load the driver via the erl_ddll call (annotation #2). Once its loaded (annotation #3) we can start calling it which we do in code listing 12.8.

12.8 Example Port Usage

```
test() ->
  {ok, P} = load_driver(), #1
  port_command(P, [<<"a">>, <<"b">>, <<"c">>]), #2
  receive
    Data ->
    io:format("Data: ~p~n", [Data]) #3
  after 100 ->
      io:format("Received nothing!~n")
  end,
  port_close(P). #4
```

In our driver the communication protocol is simply the Erlang binary format. That makes it very easy for us to write out messages. In this code we load the driver via the function we just described (annotation #1) and call port_command (annotation #2). port_command equates to the process function we registered as an output callback. Data sent here will end up as a parameter to the process function we describe previously. We then receive output from the port (annotation #3) and close the port (annotation #4). Remember that this is just a simple example and not really ment to be a full fleged Linked in Driver. So we have taken some shortcuts here, like not using gen_server, that we would normally not do. We did this in the interests of a short, understandable example. That being said, lets get into the details of our compression driver, a Linked in Driver done right.

12.2 Adding Compression to the Simple Cache

We have spent most of this chapter learning about about Erlang's two types of FFI, Port Drivers and Linked in Drivers. We know the differences between the two and when and how you should use them. Now we are going to take that knowledge and apply it to our cache. As we said in the introduction to this chapter, our central goal is to provide compression to our cache. We are going to do that using the open source lzjb library. This library will provide compression to the cache using the lzjb compression algorithm. This driver is going to be about bridging the lzjb compression library from C to Erlang. We are not going to spend really any time on the library itself. We are going to delve deeply into the infrastructure and method of communication.

There are quite a lot of options when it comes to using the Linked in Driver infrastructure. There are a number of different possible entry points and even several different methods of interacting with the VM within those entry points. We are going to concentrate on a single method of interaction that has worked well for us in the past and should work well for you.

This doesn't mean that it's the only approach to the problem. We think this will give you a good approach to use in developing your drivers and should give you a reasonable foundation in understanding some of the other approaches. We encourage you to explore this by reading the documentation included with Erlang's distribution.

12.3.1 The Protocol

If you remember from previous sections, communication between an Erlang VM and a Driver, whether it be a Linked in Driver or a Port Driver, is very similar to message passing. With that in mind we are going to start by specifying the protocol that we will use to communicate with our driver. In this case, our Driver is going to be very simple. Its whole purpose in life is to take a chunk of data, compress it and return that chunk of data. An additional requirement is that the data be uncompressible at some arbitrary point in the future. That requirement, by itself, gives us most of the information that we need to define the protocol. In the inteerests of simplicity, and because it lets us introduce you to some interesting things we are going to define the protocol in Erlang Terms. These Terms will be converted into a binary and sent to the Driver. The Driver will then decode those Terms do the action requested and return the result to the VM in Erlang Terms as well.

There are two sides to our system, compression and decompression. Lets address both of these at the protocol level individually. Before we do that though we should setup some of the basics. Our only method of communication is through the messages we are sending. So we need to be able to indicate to the driver the type of operation we require. We also need to be able to give each message a payload that will either be compressed or uncompressed respectively.

THE ENVELOPE

We are going to define an envelope that will contain our messages. The envelope will include the action we are requesting and the Term to be processed. We will define this as a tuple of three elements. The first being an atom describing the action to be performed, the second being the PID of the process doing the requesting and the third is the payload of data to be processed. Since we have two actions we will use two atoms to describe that purpose, 'compress' and 'decompress'. Lets take a look at a quick example. For compression we would send the following.

```
{compress, Pid, <opaque data be processed>}
```

For decompression we would send the following.

```
{decompress,  Pid, <opaque data to be processed>}
```

We are keeping the pid because passing that information along with the message is the easest way of associating the data with the caller. This is an asynchronous and reintrant library, by passing the Pid we make it easy to get the data back to the caller without a lot of complexity on the part of the gen_server that will handle this. The reasons for this will becoe more clear as we get into the Erlang side of this driver.

The 'opaque data to be processed' is in a specific format as well, but that format is defined by the action to be performed and it changes between compression and decompression. Lets take a look at those specific examples.

There is also a return envelope that is slightly simpler. This return envelope is always returned from a call. It consists of two elements. The first element is the Pid of the process that made the call followed by the data returned from the action. It looks as follows.

```
{Pid, <opaque result of action>}
```

AS with the outgoing message, the incoming result varys by the action called. We will get into that action next.

COMPRESSION

Our compression protocol is the simplest of the two. All it really is a tuple of two elements, the first is a integer that represents the size of the data to be compressed (this makes it easier to do tho memory allocation on the c side) and the second element is the binary that will be compressed. The following example illustrates this process.

```
{compress, Pid, {14, <<"Erlang Rocks!">>}}
```

As you can see we have a the envelope that we described previously, with the compression data making up the third element. In this case, we have we have a tuple with the first element being the integer 13 (the size of the following data), followed by the binary `<<"Erlang Rocks!">>`.

What gets returned to the caller when this call is made is slightly more complex. It is wrapped in the outgoing envelope that we defined. Inside that envelope it is a tuple of four elements. The first element is the atom 'compressed, it is followed by the uncompressed length of the data, followed by the length of the length of the data after compression. An example of this format follows.

```
{Pid, {compressed, 14, 180, Binary}}
```

In this particular example, compression is probably not the best choice, but it is useful for illustration. As you can see, we have the atom 'compressed' followed by the size of the uncompressed binary (14), followed by the size of the compressed binary (180), followed by the compressed data itself as a binary. It just so happens that the payload of the return message is exactly the same thing that will be sent to decompress. That should make things slightly easier on our cache interface.

DECOMPRESSION

Decompression is quite similar to compression over all. The main difference is that the message going into the system is slightly more complex and the message coming out of the system is slightly simpler. Lets decompress the example that we used in the compression section. As we mentioned before the result returned from compression is exactly the message payload that is sent to decompression. That means you have already seen it, but lets go over it in more detail.

The decompression message is wrapped in the same envelope that the compression message is wrapped in but, the details differ. The decompression message is an element of four tuples. The first element in the tuple is the atom 'compressed' (indicating that this is compressed data). The second element is the size of the uncompressed data (having this makes it easier on the C side). The third element is the size of the compressed data (also makes it easier on the c side). The final element is the binary to be decompressed. Lets look at an example.

```
{decompress, Pid, {compressed, 14, 180, Binary}}
```

Does this look familiar? It should, we just saw something very similar a few paragraphs ago. We can see the envelope with atom 'decompress' as the initial element, the Pid of the caller as the second element and the payload we just described.

The resulting message for a decompress call is much simper. It is simply the return envelope that contains the payload for decompression. That payload consists a two element tuple. This tuple contains the atom 'uncompressed' and the uncompressed binary.

```
{Pid, {decompressed, Binary}}
```

So know we know what our protocol looks like. Lets dig into how we will actually consume this protocol in our Driver.

12.3.2 The C Side

As you would expect the C side represents the main worker of our driver. It handles the actual compression and decompression duties as well as all of the marshalling and unmarsahlling.

SETTING UP THE DRIVER

Just like in our earlier example there is a bit of work that needs to be done to set up our linked in driver. Amazingly enough that work is very similar to what we have already seen. In fact, it will almost always be very similar from driver to driver. This is a case of once you have seen one you have seen them all. Though, that's the case lets go over it again anyway.

To start with we need to define the state of our driver. We do this with a simple struct.

```
struct port_state {
    ErlDrvPort port;
};
```

Once again, all we are including here is the port itself. Our compression library doesn't actually need to store any additional state. This may not be the case for you and you should make use of this state if you need to.

As is our custom in Linked in Drivers we declare the prototypes for the functions that appear in the driver at the top of our file. In this case there are three functions that do the work. The first two provided lifecycle functionality for the driver (startup and shutdown respectively). The third is the function that actually does the work of the driver.

```
static ErlDrvData
drv_start(ErlDrvPort port, char *buff __attribute__((unused)));

static void
drv_stop(ErlDrvData handle);

static void
drv_outputv(ErlDrvData handle, ErlIOVec *ev);
```

We will go into more detail on these functions in later sections. Lets move on to the next bit of interesting Driver stuff. That is the struct that defines the callbacks in this Driver to the Erlang system.

12.9 The Driver Description

```
ErlDrvEntry driver_entry = {
```

```
    NULL,           /* F_PTR init, N/A */
    drv_start,      /* L_PTR start, called when port is opened */
    drv_stop,       /* F_PTR stop, called when port is closed */
    NULL,           /* F_PTR output, called when erlang has sent */
    NULL,           /* F_PTR ready_input, called when input descriptor ready
*/
    NULL,           /* F_PTR ready_output, called when output descriptor
ready */
    "lzjb_compress", /* char *driver_name, the argument to open_port */
    NULL,           /* F_PTR finish, called when unloaded */
    NULL,           /* F_PTR handle */
    NULL,           /* F_PTR control, port_command callback */
    NULL,           /* F_PTR timeout, reserved */
    drv_outputv,    /* F_PTR outputv, reserved */
    NULL,           /* ready_async */
    NULL,           /* flush */
    NULL,           /* call */
    NULL,           /* event */
    ERL_DRV_EXTENDED_MARKER,        /* extended marker */ /* enable the
rest */
    ERL_DRV_EXTENDED_MAJOR_VERSION, /* major version */
    ERL_DRV_EXTENDED_MINOR_VERSION, /* minor version */
    ERL_DRV_FLAG_USE_PORT_LOCKING,  /* driver_flags */    /* better
locking? */
    NULL,           /* handle2 */                         /* set by the VM
*/
    NULL            /* process_exit */
};
```

This is almost the same struct that we declared in our earlier example. There are two main differences. The first is that we changed a couple of names to be a little more clear and consistent. The second is that we are going to use the outputv callback instead of the process callback. This allows us to access the data we need to parse directly and in an asynchrounous fasion. Other than that things are pretty similar.

The last thing we do in setting up our driver is just declare the entry point to Erlang and return this struct to the VM.

```
DRIVER_INIT(lzjb_compress)
{
    return &driver_entry;
}
```

As you can see this is exactly the same as the function we used in our last example. Of course, the name that we pass into the DRIVER_INIT macro changes but that's it. You should remember that this name must match the name specified in the name section of the description struct.

We that we have done all of the setup work that we need to do to complete the driver. Now its time to move into actual code that will be called. Lets get started with the lifecycle code that will start and stop the Driver.

STARTING AND STOPPING THE DRIVER

Lets get started with the drv_start function. In many Drivers this function would do things like start native threads, initialize libraries, etc. In this case of this Driver though, we don't actually need to do that. All we need to do is allocate our state and return it to the Erlang VM. Lets take a quick look at drv_start.

12.10 The drv_start function

```
static ErlDrvData
drv_start(ErlDrvPort port, char *buff __attribute__((unused)))
{
    struct port_state *drv = NULL;

    drv = (struct port_state *) driver_alloc(sizeof(struct port_state)); #1

    if (drv == NULL) {
      return ERL_DRV_ERROR_GENERAL;
    }
    drv->port = port;
    return (ErlDrvData)drv;
}
```

As you can see, in this function we are simply allocating the state of the function, nothing more. The important thing to take note of here is that we are allocating the state with the driver_alloc function (annotation #1). This allows the Erlang VM to manage that memory for us. Aside from that, we are just returning our allocated state function, populated with the port that is passed into the system.

The drv_stop function is even simpler. We have no real shutdown semantics in our Driver. Other drivers would use this function to kill threads, deinitialize libraries, etc. As we have no real shutdown semantics in our Driver, all we need to do is deallocate the state that that we allocated in the drv_start function.

```
static void
drv_stop(ErlDrvData handle)
{
    driver_free((struct port_state *)handle);
}
```

As before the main thing to be aware of here is that we deallocate with driver_free. Anytime you allocate something with driver_alloc you must deallocate it with driver_free.

No that we know how to start up and shutdown our system its time to move into more interesting things. In this case, more interesting things are comprised of a single function that handles the parsing and unparsing the protocol we described in the earlier area.

THE MEAT OF THE C DRIVER

At last we get to what we have been waiting for. The more interesting bit of the driver, that is the code that does the data marshalling and unmarshalling in the system. Lets kick off our function by declaring the things that we are going to need.

12.11 The drv_outputv Function

```
static void
drv_outputv(ErlDrvData handle, ErlIOVec *ev)
{
     struct port_state* d = (struct port_state *) handle;
     ErlDrvBinary *buff_bin = ev->binv[0];
     char *buff = buff_bin->orig_bytes;
     char atom[MAXATOMLEN];

     void * data = NULL;
     void * udata = NULL;
     ei_x_buff x_buff; #1
     ei_x_new_with_version(&x_buff); #2

     // The decoding index
     int index = 0; #3
```

There are a few things to note here. First and foremost is the us of SysIOVec. This allows us to get at the data directly and make use of the convenient ei_decode functions. I should point out here that there are other ways to do this but I find this the most convenient. The data that we are interested in is passed in as the second char array in SysIOVec. So that's the one we are going to use in our buffer.

The next piece to notice is the declaration of ei_x_buff (annotation #1). This is related to encoding our data for transmition back to the Erlang side. Erlang provides us with a nice body set of functions to encode erlang terms. Those functions are encapsulated in a library called libei. There are two versions of the function. The first are simply the ei_* functions. They take an index and a buffer and write into that buffer. The second set of functions are the ei_x_* functions. The only difference is that the ei_x_* functions take an ei_x_buff instead of a char * and an index. These ei_x_* functions grow the buffer encapsulated by the

ei_x_buff as needed so that you (the developer) don't have to worry about it. Unless I am in a place where I need to have explicit and tight control of the memory I am using I always use the ei_x_* version of the functions.

We instantiate ei_x_buff with the function ei_x_new_with_version (annotation #2). This allocates the required memory and rights a bit of header information that Erlang needs into the buffer. Finally, we declare our index (annotation #3) into the incoming data. The decode functions we call will keep this updated and allow the parsing of terms as they occure.

The next thing we are going to do is parse our outer tuple. Remember the envelope we talked about in the protocol section? That's what we are going to parse first.

12.12 Parsing the Envelope

```
int arity;
if (ei_decode_tuple_header(buff, &index, &arity)) {
        encode_error(&x_buff, "tuple_expected", "tuple expected");
        goto cleanup_out;
}

if (arity != 3) {
        encode_error(&x_buff, "invalid_tuple",
                     "expected tuple of arity 3");
        goto cleanup_out;
}
```

Here we parse our first bit of the incoming code. All of our parsing is done with the ei_decode_* functions. For the most part they are all very similar. The each parse a specific Erlang Term and return that as a parameter. They return zero on success and one on failure. Tuples and lists are slightly different from other ei_decode_* functions though, in that you parse a 'header' for the tuple or list, then parse the values for that list as they occur. In this case we parse a tuple getting the arity into our arity variable.

You may notice that if we are unsuccessful we write some error information into our buffer (in the encode_error function) and then jump to a label that cleans up after us. This is a convention that I generally follow. It makes for a lot of explicit code but it works well.

Once we get the arity we check it against the arity we expect. If its off we write out a different error message with the useful information.

Next we decode the atom that describes the action requested.

```
if (ei_decode_atom(buff, &index, atom)) {
        encode_error(&x_buff, "invalid_tuple",
                     "Expected an atom as the first "
                     "element of he tuple");

        goto cleanup_out;
}
```

As you can see, this decoding follows the same pattern that we used in our tuple decoding. That is we wrap the function in an if statement, write error information into the buffer and then jump to the cleanup area when we are done. As in other cases it follows the same pattern in decoding as the other decode functions. In the case of decoding an atom we past a char * that the ei_decode_atom function will fill. It doesn't give us any indications about the size of the atom required because atoms have a fixed size limit. In this case we declared an atom of the appropriate size back at the start of the function. Here it is again though

```
char atom[MAXATOMLEN];
```

As long as we have an char * of at least MAXATOMLEN size we will be ok.

Once we have the atom that indicates what action we are going to take we need to check to see if it matches one that we know how to do. If you remember from our protocol we expect 'compress' or 'decompress' as the action indicators. So we compare this incoming atom and use it to assign to our 'act' enum. This will allow us to dispatch on this action at well. Of course, if the atom isn't 'compress' or 'decompress' we return an error.

12.13 Figuring Out What Our Action Is

```
enum action act;
if (strcmp("compress", atom) == 0)
        act = COMPRESS;
else if (strcmp("decompress", atom) == 0)
        act = DECOMPRESS;
else {
        encode_error(&x_buff, "invalid_action",
                     "Expected an action of compress or decompress");

        goto cleanup_out;
}
```

According to our protocol the last element in our envelope is the PID of the sender. We decode this just like every thing else. With the ei_decode_* functions for PID.

```
erlang_pid pid;
if (ei_decode_pid(buff, &index, &pid)) {
        encode_error(&x_buff, "expected_pid",
```

```
                        "Expected pid as the second element"
                        " of the tuple.");
                goto cleanup_out;
        }
```

We don't actually care about this PID except as a function that we will pass back to the caller. So we will just let it remain opaque to us.

Finally we get into the part of the protocol that represets the interesting part of the system. This is the data that will be compressed or decompressed depending on the action requested. Once again our data is encapsulated as a tuple and we must parse the tuple head to get that data.

```
        int narity;
        if (ei_decode_tuple_header(buff, &index, &narity)) {
                encode_error(&x_buff, "tuple_expected",
                        "Expected a tpule as the third "
                        "element in the series" );
                goto cleanup_out;
        }
```

We decode the new arity and check that against what we expect for the compress and decompress actions.

12.14 Checking the Arity of the Body

```
        // We expect a tuple of the form {Action, Pid, Len, Arity}
        if (COMPRESS == act && narity != 2) {
                encode_error(&x_buff, "expected_compress_tuple",
                        "Expected a tuple of arity two as the correct"
                        "indicator for compression.");

                goto cleanup_out;
        } else if (DECOMPRESS == act && narity != 4) {
                encode_error(&x_buff, "expecting_decompression_tuple",
                        "Expecting tuple of arity four as the correct"
                        "decompression tuple");
                goto cleanup_out;
        }
```

Now that we know the arity is good we can jump into decoding the action specific parts of the message. We do this by checking our 'act' enum and then decoding the relevant pieces of the incoming stream.

12.14 Parsing the Compress Data

```
        if (COMPRESS == act) {
```

```
long data_len;
if (ei_decode_long(buff, &index, &data_len)) {
        encode_error(&x_buff, "expected_long",
                     "Expected long as the first element in"
                     " the compression tuple");
        goto cleanup_out;
}

data = malloc(data_len);
if (! data) {
        // this is really bad
        encode_error(&x_buff, "unable_to_alloc_data",
                     "Unabel to alloc space for data");
        goto cleanup_out;

}

long real_data_len;
if (ei_decode_binary(buff, &index, &data, &real_data_len)) {
        encode_error(&x_buff, "expecting_fourth_element_binary",
                     "Unabel to decode binary");
        goto cleanup_out;
}

if (data_len != real_data_len) {
        encode_error(&x_buff, "length_doesnt_match",
                     "The compressable data length specified "
                     "and  what was actually provided do "
                     "not match");
        goto cleanup_out;
}
```

For the most part, this goes exactly as it has up to this point. We decode each piece of the stream as we expect it using the correct ei_decode_* function. The only real difference here from what you have already seen is the ei_decode_binary function. It takes an already allocated char * to write data into, it also updates a long with the actual size of the data. This is the main reason we pass in the size of the expected data as part of the protocol. Sizing your arrays correctly is extremely important. Finally we check the size that was expected against the size that was actually returned and create an error if they don't match.

Once all the decoding is done we get to actually compress the data. It's a bit anticlimactic really. We just allocate a structure that's big enough to hold the data and call lzjb_compress on it, passing in all of the relevant links.

```
// Just in case the data isn't terribly compressible leave
// room for markers.
long out_data_len = real_data_len + 40;
udata = malloc(out_data_len);
```

```
long comp_len = lzjb_compress(data, udata,
                              real_data_len, out_data_len);
```

Once we have the data compressed and ready, its time to return it to the caller. We do that by encoding it into the set of terms required by the protocol and closing out our function. To do this we use the ei_x_encode_* functions. They are very similar to the decode functions that we have already used, just sort of going in the opposite direction.

First we create the envelope for our return message. This is represented by the tuple_header and the pid that was passed in erlier.

```
// Regardless of what we return its wrapped in this tuple
ei_x_encode_tuple_header(&x_buff, 2);
ei_x_encode_pid(&x_buff, &pid);
```

Then we encode our data. The four pieces of data that we encode are the atom 'compressed', the length of the uncompressed data, the length of the compressed data and the data itself. We encode each part as a term in the buffer using the relevant ei_x_encode_* functions.

```
ei_x_encode_tuple_header(&x_buff, 4);
ei_x_encode_atom(&x_buff, "compressed");
ei_x_encode_long(&x_buff, htonl(real_data_len));
ei_x_encode_long(&x_buff, htonl(comp_len));
ei_x_encode_binary(&x_buff, udata, comp_len);
```

The other side of our function is decompressing. It follows a model very similar to the compression side of the function. We decode our elements of the decompression tuple using the same ei_decode_*

12.15 Parsing the Decompression Side

```
} else if (DECOMPRESS == act) {

        if (ei_decode_atom(buff, &index, atom)) {
                encode_error(&x_buff, "expecting_atom",
                             "Expecting atom 'compressed' "
                             "indicating compressed data "
                             "to uncompress.");
                goto cleanup_out;
        }

        if (strcmp(atom, "compressed") != 0) {
                encode_error(&x_buff, "expecting_atom",
```

```
                        "Expecting atom 'compressed' "
                        "indicating compressed data "
                        "to uncompress.");
        goto cleanup_out;
}

long uncompressed_size;
if (ei_decode_long(buff, &index, &uncompressed_size)) {
        encode_error(&x_buff, "expecting_uncompressed_size",
                        "Expecting a long indicating the "
                        "uncompressed size of the data to be "
                        "uncompressed.");
        goto cleanup_out;
}

uncompressed_size = ntohl(uncompressed_size);

long compressed_size;
if (ei_decode_long(buff, &index, &compressed_size)) {
        encode_error(&x_buff, "expecting_compressed_size",
                        "Expecting a long indicating the "
                        "uncompressed size of the data.");
        goto cleanup_out;
}

compressed_size = ntohl(compressed_size);

data = malloc(compressed_size);
long real_data_len;
if (ei_decode_binary(buff, &index, data, &real_data_len)) {
        encode_error(&x_buff, "expecting_binary",
                        "Unable to decode compressed binary "
                        "expected.");
        goto cleanup_out;
}

if (real_data_len != compressed_size) {
        encode_error(&x_buff, "data_len_doesnt_match",
                        "THe length of the binary to be "
                        "uncompressed and the length specified "
                        "in tuple do not match!");
        goto cleanup_out;
}
```

Once again it's a bit anticlimactic but we finally get to the point of the action here. We take our data and call the decompression function on it.

```
udata = malloc(uncompressed_size);

/* -1 if unco data != out_length */
lzjb_decompress(data, udata, uncompressed_size);
```

Once we have our data nicely decompressed we can wrap it up in the outgoing packet and return it to our user. We use the exact same ei_x_encode_* functions that we called before.

```
            // Regardless of what we return its wrapped in this tuple
            ei_x_encode_tuple_header(&x_buff, 2);
            ei_x_encode_pid(&x_buff, &pid);

            ei_x_encode_tuple_header(&x_buff, 2);
            ei_x_encode_atom(&x_buff, "decompressed");
            ei_x_encode_binary(&x_buff, udata, uncompressed_size);

    }
```

Now our data is all wrapped up and everything is ready to ship back to the user. So we call the driver function 'driver_output'. It takes the contents of our ei_x_buff and writes it back to the caller. We then go on to cleanup after ourselves by freeing the data that we allocated previously.

```
    cleanup_out:
     driver_output(d->port, x_buff.buff, x_buff.index);

        ei_x_free(&x_buff);
        if (data)
                free(data);

        if (udata)
                free(udata);
}
```

That's it! We are done with the C side. This stuff seems complex but really it isn't. Its just a matter of encoding and decoding the terms as we expect them along with some reasonable error handling.

12.3.4 The Erlang Side

We have finally gotten to the Erlang side of our linked in Driver. As you can imagine at this point we are going to be building an OTP app. We will, of course, have an application Behaviour impelementation, a root supervisor and an gen_server Behaviour container. The application behaviour implementation and the root supervisor are very, very similar to all of the implementations you have seen to this point. In the interests of space I am going to skip describing them in this chapter. I am going to jump right into the gen_server that forms the core of our compression library. This is quite similar to what you have already seen as well,

but there are a few, very distinct differences. It is these differences that we are going to dive into next.

SETTING UP THE DRIVER

As in all gen_server container implementations the init function serves as the entry point for initialization of the module. It is exactly the same in this implementation. The primary concern in this particular case is locating the actual shared object and loading that shared object. Fortunately, if we follow a few simple conventions doing these things isn't too hard at all. Lets take a look at code lising 12.16 which contains the initialization code for our gen_server implementation.

12.16 Initializing the Erlang Side of the Linked In Driver

```
init([]) ->
    ok = open(),
    {ok, #state{port = open_port({spawn, ?LIB}, [binary])}.

 open() ->
  case erl_ddll:load(code:priv_dir(lzjb), ?LIB) of #1
    ok -> ok;
    {error, already_loaded} -> ok;
    _ -> exit({error, could_not_load_driver})
  end.
```

We have our init function, it calls the open function that contains the meat of our application. The open call is the one we are interested in. I calls the function that loads the driver and handles the exit value. Notice the erl_ddll:load function call (annotation #1). This is the function call that actually loads the shared object into the vm. It takes two arguments, the path to the shared object and the name of the shared object itself. Another thing to notice in this statement is the call to code:priv_dir. We have talked about this in the past, but I am going to go over it again. Every non-erlang thing in your OTP application should go into the 'priv' directory of the OTP application. In this case, that would be our shared object. By putting it in the 'priv' directory we make it easy to find and load using functionality provided by the vm. This is the convention in Erlang and you would save quite a bit of time and effort by following it. This allows us to find the path to our shared object by simply calling 'code:priv_dir' with the name of our OTP application as its argument.

Once we load up the driver we pattern match on two clauses, 'ok' and 'already_loaded'. For our purposes, these two clauses are completely valid responses so we return ok for them. In the case, of any other response we return an error.

THE MEAT OF THE ERLANG DRIVER

The truly interesting parts of our gen_server implementation are spread across several functions. First we have our API functions, 'compress' and 'decompress'. Then the handle_call functions that actually do the processing. Finally, we have the handle_info calls that handle the response from the port and forward the data to the correct place.

Lets get started on this piece by looking out our API functions. The two functions we expose are 'compress' and 'decompress'. You can probably guess what they are the API for.

```
compress(data) ->
    gen_server:call(?SERVER, {compress, Data}).

decompress(Data = {compressed, _, _, _}) ->
    gen_server:call(?SERER, {decompress, Data}).
```

As is the case for nearly all of our gen_server APIs, all we really do is turn around and do a gen_server call with the protocol that we expect. The decompress side of our API has an explicit requirement around the shape of the tuple that is passed into it. We do validate the shape of this tuple in our API. This will give the caller explicit and immediate feedback if he passes in a tuple that doesn't conform to this shape.

The handle_call functions that process these messages are much more interesting.

```
handle_call({compress, Data}, From, State) ->
    compress_term(From, Data),
    {noreply, State};
handle_call({decompress, Data = {compressed, _, _, _}}, From, State) ->
    decompress_term(From, Data),
    {noreply, State}.
```

In both of these cases we simply call the function related to actually processing the requested action. Two things you should notice here are the fact that we pass the 'From' argument to our call and that we return noreply. The are both related to each other. Our communication with the port is asynchronous, like most other communication in Erlang. So we can't actually return a result to the caller at that moment. So we keep track of the from command and pass that to the linked in driver to be return with the result. We then pass 'noreply' back to the gen_server container to indicate that we don't have a reply at this moment.

Lets take a look at our compression and decompression functions. The take the data provided from our handle_call and handle cast functions and encode them into the protocol we described previously.

12.16 The Compression/Decompression Encoding Functions

```
compress_term(Pid, Term) ->
    call_port(encode_compress(Pid, term_to_binary(Term)).
decompress_term(Pid, Data) ->
    call_port(encode_decompress(Pid, Data)).

encode_compress(From, Data) ->
    term_to_binary({?COMPRESS, From, {size(Data), Data}}).

encode_decompress(From, Data) ->
    term_to_binary({?DECOMPRESS, From, Data}).
```

These functions do two things. They take the data provided and format them in the way the protocol requires. They then take that code and convert it to the binary the Linked in Driver expects.

Finally we have the entry point for the data sent from the Linked in Driver to our gen_server. The Linked in Driver doesn't have any understanding at all about gen_servers. So all of the messages it sends goes through the handle_info functions. Lets look at our handle info function.

```
handle_info({Port, {data, Data}}, State = #state{port=Port}) ->
    case binary_to_term(Data) of
     {From, Result = {compressed, _, _, _}} ->
         gen_server:reply(From, Result);
     {From, {decompressed, DecomData}} ->
         gen_server:reply(From, {decompressed, binary_to_term(DecomData)})
    end,
    {noreply, State}.
```

This handle_info call takes the data sent from the port, converts it from a binary to a term and the sends the result back to the process that made the originial request. You may notice that we are using the From pid that we sent to the port driver originally. When we get the data correctly converted to terms we use the gen_server:reply function to reply to the gen_server:call that originally made the request. This is a feature of the gen_server that isn't heavily used, but when you need it it is quite handy.

Now that we have both the C side of our linked in driver and the Erlang side of our Linked in Driver complete its time to look into how to integrate it with the rest of the cache.

12.3.5 Integrating With The Cache

Integrating with the existing cache is actually quite easy. We just need to modify the cache such that when a value is inserted it is compressed, if compression is turned on, and when it is retrieved it is decompressed, if it was previously compressed.

12.17 Compressing Values inserted into the Cache

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_event:replace(Key),
       sc_element:replace(Pid, compress(Value)); #1
     {error, _Reason} ->
         sc_event:create(Key, Value),
       sc_element:create(Key, compress(Value)) #2
    end.

compress(Value) ->
    case application:get_env(simple_cache, compress) of
     {ok, true} ->
         lzjb:compress(Value);
     _ ->
         Value
    end.
```

We integrate compression by changing the insert function of the simple_cache module in two ways. First we add the compress function call to any place we add a value to the cache. For both create and replace we call the compress function. Then we add the compress function. The compress function takes the value passed to it. It checks to see if compression is enabled by looking at the applications environment. If compression is enabled, then it calls the compression function in lzjb and returns the compressed value. If compression isn't enable it simple returns the value. This allows compression to be somewhat opaque to the insert function.

The ability for compression to be turned on and off makes things a bit tricky on the decompression side. At any time, we could have a mix of compressed and decompressed values in the cache. That means that the decompression function needs to be a bit more discerning of the values it tries to decompress.

12.18 Decompressing Values in the Cache

```
lookup(Key) ->
    sc_event:lookup(Key),
    case sc_store:lookup(Key) of
     {ok, Pid} ->
       decompress(sc_element:fetch(Pid)); #1
     {error, Reason} ->
         {error, Reason}
    end.

 decompress(Data = {compressed, _, _, _}) -> #2
    lzjb:decompress(Data); #3
decompress(Data) ->
    Data. #4
```

Fortunately, we know what compressed data looks like, so we can actually dispatch on that pattern. You can see that we do this in the head of the decompress function, (annotation #2) if it looks like compressed data we call lzjb:decompress on it (annotation #3), if it doesn't we pass the data back unchanged (annotation #4). To actually use the decompress function, we call it on every piece of data returned from the cache (annotation #1).

With that our long journey through the world of Drivers, both Port and Linked in, is complete. We have a fully functioning compression and decompression system integrated into the cache in a way that is very performant. We can have both uncompressed and compressed data mixed into the cache without any problem and we have a compression OTP app that we can use whenever and wherever we need it.

12.4 Summary

In this chapter we looked at the two methods of communication with foreign code that is available to erlang. We also discussed the caveats of using each of these methods. The use of the Port Driver is, by far, the preferred solution from a safety standpoint. However, in some cases speed may be of the essence. In those cases it may be necessary to use a linked in driver. In either case, you now have the tools you need to create either style of interface.

13

Communication between Erlang & Java via JInterface

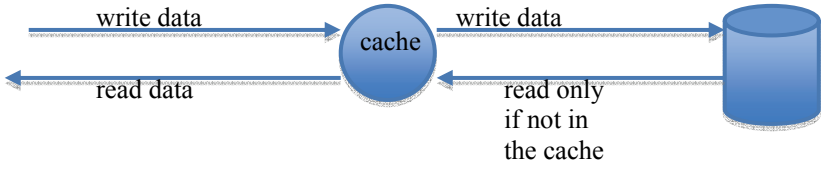
In our last chapter we talked about interfacing directly between Erlang and Foreign code via Port Drivers. That's a useful approach but not the most convenient one to take for every mode of interaction. In this chapter we are going to talk about a different manner of interaction. It revolves around a non Erlang application stack appearing as, and communicating with, an Erlang cluster. In essence, the foreign application masquerades as an Erlang node and talks with its Erlang and non Erlang peers over the Erlang distribution protocol that we talked about in Chapter 8 and Chapter 9. Fortunately, for us the folks behind the Erlang language have already done a lot of the heavy lifting for us and provided solid libraries in Java and C to make this relatively easy. It is also possible to find libraries for other languages such as Python for example.

Since having rounded out our cache in part two of this book and making it suitable for use in enterprise environments other people and organizations have picked it up and started using it for their projects. One of the groups that is using it needs to persist the data with something more long lived than in memory Mnesia. Their requirement is to pretty much persist all the data they ever insert into the cache forever! That is quite a requirement but we can handle it. To help them out we are going to add the ability to store our cached objects in an HBase cluster. HBase is the Hadoop database. It offers a very fast and reliable store for big data sets. Here is the description of Hbase from their own site:

> *"HBase is the Hadoop database. Use it when you need random, realtime read/write access to your Big Data. This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware."*

Integrating with the HBase database gives us the ability to use HBase's robust and well understood storage and distribution model to back up our distributed cache thus providing it with the ability to store loads and loads of data. The interaction with the Hbase system would work something like this. When doing a lookup in the cache the cache will check to see if the data is already present in the cache, if so, it will return it. If not it will pull that data from Hbase, cache it, and then return it. On writes the cache will write both into the cache and out to Hbase. Users of this incarnation of the cache will use it basically to speed up access to a reliable big data capable backing store like Hbase and to persist data reliably long after its cache lease time has expired. Figure 13.1 demonstrates the cache interaction.

figure 13.1 – cache access



There are a number of ways we could integrate Hbase, from implementing the HBase protocol stack directly in Erlang to calling the HBase RESTful API. The purpose of this chapter is to introduce you to the concept of foreign nodes and so we are going to glue the hbase java interface directly into Erlang via Jinterface. Jinterface is a library that allows a Java application to masquerade as an Erlang node. We will create a Java application that allows an Erlang node to put things into and retrieve things from an HBase database which is of course written in Java.

This is a book about Erlang and not about HBase and consequently we are going to give you enough information to get a working HBase node up and running with the requisite table descriptions and related requirements, however, we are not going to spend a large amount of time on HBase itself.

13.1 – Erlang/Java Integration with JInterface

Before we dive into HBase and the Erlang integration layer let's take a little time to talk about JInterface itself. For the most part JInterface is a library in Java that tries to expose Erlang concepts to the Java runtime. It does not do this in an idiomatically Java way. The library has the very distinct feel of exposing the Erlang layer with as little change as possible. For our purposes, coming from Erlang to Java, this is actually a pretty good thing. Nearly every Erlang construct has a matching construct in a Java Class. This is true for large abstractions like the Node and Mailbox to more granular abstractions like Tuples and Lists. Lets go through these abstractions and talk about how they map from Erlang to Java.

13.1.1 - The Node

The root of any Erlang system is the node. In the JInterface library the concept of the Erlang node maps directly onto the OtpNode class. The node provides the means of connecting to and interacting with other Erlang and non-Erlang nodes. Like a normal Erlang node it starts with a node name and optionally, cookies and a port. The node name and the cookies serve the exact same purpose that they server for a native Erlang node. We discussed this back in the chapter 8 and 9 on distribution, feel free to go back and re-acquaint yourself with that information. To get ourselves moving lets show a small example of how to start an OtpNode.

```
OtpNode node = new OtpNode(nodeName);
```

It really is as simple as that. OtpNode is a pretty interesting class that hides all of the underlying communication, connection handling etc from the implementer. Now we have a node, however, to actually do anything with the node we need to create a mailbox.

13.1.2 - Mailboxes

For the most part mailboxes are exactly that, mailboxes that you use to interact with other nodes in the cluster. They are equivalent to mailboxes that serve Erlang Processes and

perform the same function. However, they are not processes themselves. JInterface allows you, the Java programmer, to manage threads in any way that you would like. Therefore gives you direct access to this mailbox abstraction so that your threads can interact with nodes as they will.

We create the mailbox by asking an OtpNode object to create a mailbox for us. We have two kinds of mailboxes that can be created, one with a name and one without. If we create a mailbox with a name that name is registered in the exact same way a registered process is. That is we can send messages to the mailbox using a registered name just like any other registered process. If we create the mbox without a name then it acts like any other anonymous process and we need the PID to interact with it. Yes, JInterface mailboxes also have a PID like any other Erlang process. Let's look at how these are actually created in the next snippet.

```
OtpMbox mbox = node.createMbox("our_registered_name");
```

To create an mbox without a registered name we just do the same without actually passing in the name to our node.

```
OtpMbox mbox = node.createMbox();
```

Once we have the mailbox we can use it to send and receive messages. The two fundamental APIs we are going to use to do this are the send and receive methods of the OtpMbox class. There are several variations of these methods, but I will leave it to you to take a look at the javadocs for the variations. Sending and receiving data requires that we marshal the data to and from the native Erlang format. Fortunately, for us JInterface gives us the tools we need to be able to do that.

Mapping Erlang Data structures Onto Java

Anything we send to or receive from an Erlang node or anything masquerading as an Erlang node needs to be marshaled to and from the native Erlang types. We do this by using the type mapping classes that JInterface provides for us. These classes are a direct representation of the Erlang types in Java.

Table 13.1 Erlang Java Mappings

Erlang Type	Java Class
Atom	OtpErlangAtom

Binary	OtpErlangBinary
Floating point types	OtpErlangFloat, OtpErlangDouble
Int types	OtpErlangByte, OtpErlangChar, OtpErlangShort, OtpErlangUShort, OtpErlangInt, OtpErlangUInt, OtpErlangLong
List	OtpErlangList
Pid	OtpErlangPid
Port	OtpErlangPort
Ref	OtpErlangRef
Tuple	OtpErlangTuple
Term	OtpErlangObject

I think the best way to understand how to use these classes is to go through a set of examples. The next snippet is an Erlang term.

```
{this_is_an_atom, "This is a string", 22}
```

Lets map this onto java now. Unfortunately, the java side of things is going to be quite a bit more verbose if, for no other reason, then that we can't use a declarative structure to marshal the objects. That's ok though, it's not too bad.

```
OtpErlangAtom thisIsAnAtom = new OtpErlangAtom("this_is_an_atom");
OtpErlangString thisIsAString = new OtpErlangString("This is a string");
OtpErlangShort thisIsAShort = new OtpErlangShort(22);

OtpErlangTuple tuple = new OtpErlangTuple(OtpErlangObject[]{thisIsAnAtom,
                                                            thisIsAString,
                                                            thisIsAShort});

mbox.send(tuple);
```

As you can see it is verbose. However, the mapping between Java and Erlang is very direct as well. The main things to remember are that types map directly from Erlang to Java and that the elements of compound objects like tuples and lists must be mapped in their own right. If you keep those in mind you shouldn't have any problem with marshaling and unmarshaling objects. Let's take a quick look at moving the data back in the other direction, basically marshalling from Erlang into Java. .

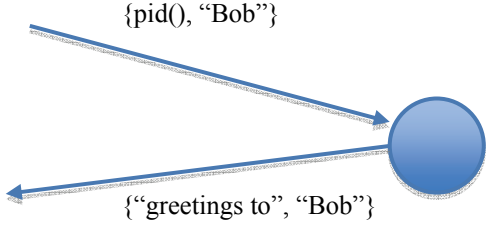
```
OtpErlangObject o = mbox.recieve();

OtpErlangTuple tuple = (OtpErlangTuple) o;
String thisIsAnAtom = ((OtpErlangAtom) o.elementAt(0)).toString();
String thisIsAString = ((OtpErlangString) o.elementAt(1)).toString();
short thisIsAShort = ((OtpErlangShort) o.elementAt(2)).shortValue();
```

As you can see the mapping rather easily goes the other direction as well. In this case, we get an object back from receive that we then proceed to decode. As you may have surmised you must know the format of the data that is coming to you. As long as you know what you are receiving its not to difficult to convert the data to native java types. You map the data onto the Erlang mapping object and then call the relevant conversion function.

Remember, that our ultimate goal here is to present an interface between our cache and HBase using Java. To be able do that we need to understand how to communicate between Erlang and Java. With that in mind, let's go through a complete example. The example code will take a pid and a name sent to it in the form of a tuple and return a greeting for that name in the form of a tuple back to the pid that was sent in. Figure 13.2 demonstrates the data flow.

figure 13.2 – example data flow



Our example code starts in the next snippet with the main method.

```
public static void main(String[] args) throws Exception {
              if (args.length < 2) {
                     usage();
                     return;
              }

              JinterfaceExample main = new JinterfaceExample(args[0],
                                                              args[1]);
              main.process();
       }
```

All we do here in the main main method is take the node name and the process name that where passed in on the command line and pass it into the JinterfaceExample

constructor, then call main.process(). The real magic of initialization happens in the constructor itself.

```
public JinterfaceExample(String nodeName, String mboxName)
                throws IOException {
        super();

        _node = new OtpNode(nodeName);
        _mbox = _node.createMbox();
        _mbox.registerName(mboxName);
}
```

As you can see the constructor takes in the name of the node and the name of the registered mbox. We use these names to create the node representation of our system which will the handle all of the incoming/outgoing connection management. We then use that node to create the mailbox. The method in code listing 13.1 actually handles the incoming messages and sends responses.

13.1 JInterface Example Process Method

```
public void process() {
   while (true) {
     try {
       OtpErlangObject o = _mbox.receive();
         if (o instanceof OtpErlangTuple) {
     OtpErlangTuple msg = (OtpErlangTuple) o;       #1
           OtpErlangPid from = (OtpErlangPid) msg.elementAt(0);
           String name = ((OtpErlangString)
                          msg.elementAt(1)).toString();

           OtpErlangString greetings =
                new OtpErlangString("Greetings to");

              OtpErlangTuple outMsg = new OtpErlangTuple(        #1
                                              New OtpErlangObject[]{
                                                  _mbox.self(),
                                                 greetings,
                                        new OtpErlangString(name)});

                                    _mbox.send(from, outMsg);        #3
                         }
                } catch (Exception e) {
                       System.out.println("" + e);
                }
        }

}
```

In this method we create an endless loop whose only purpose is to process incoming messages from the system. We get those messages in the form of a two element tuple that

contains the PID of the sender and the name as a string. Beginning at code annotation number 1 we deconstruct that tuple so that we can pull out the PID and the name. Starting at code annotation number 2 a response tuple is created. The response tuple contains the greeting as a string and the name. We then send the new message back to the originator of the triggering message at code annotation number 3. We do this using the basic constructs that we have talked about up to this point.

So far we have looked at how we can map Terms from Erlang to Java and back again. We have talked, to some extent, about the higher level abstractions like Nodes and Mailboxes. We have taken a good look at how to marshall and unmarshall information to and from Erlang terms in Java. Now lets take all that knowledge and put it together.

13.1.3 - Putting It All Together

The Jinterface library needs an EPMD mapping daemon, that handles Node discovery for Erlang, running already. EPMD basically is a name server for Erlang. It maps symbolic node names onto actual machine addresses. The issue here is that it does not start one on its own. The first Erlang node that is started on any given machine starts the EPMD daemon automatically. This means is that we need to start an Erlang node before our java node or even a separate EPMD daemon process by itself first. Simply making sure an Erlang node is started is the most common approach to this issue however.

In Chapter's 8 or 9 we talked quite a bit about distribution, so you already know how to start a shell with distributed semantics. So let's start the node with a name or sname and a cookie in the next snippet.

```
erl -sname test1 -setcookie test
Erlang R13B01 (erts-5.7.2) [source] [64-bit] [smp:2:2] [rq:2] [async-
threads:0] [hipe] [kernel-poll:false]

Eshell V5.7.2  (abort with ^G)
(test1@my.host.name)1>
```

Our goal here is to communicate between Erlang and Java. We can't very well do that without and Erlang node running. So we have an Erlang node setup and running for us now. Great! Let's get the Java node up and running so we can connect to it and have an Erlang and a Java node communicating. This after all is exactly what we need to get the Simple Cache (an Erlang node) and Hbase (Java) communicating. We need fire up java with the correct classpath. That classpath is going to vary based on where your JInterface is installed and what version of it is present on the local machine. Once you have that it merely becomes a point of calling the correct command. That command should look look much like what is in the following snippet (all on one line).

```
java -cp ./bin:/usr/local/lib/erlang/lib/jinterface-
1.5.1/priv/OtpErlang.jar org.erlware.jinterface_example.JinterfaceExample
test2 test
```

Take note of the –cp part of this command. This is where the class path that pulls in the actual jinterface code goes. The argument to this flag is the main thing that may change from machine to machine. Everything else should stay very similar. Notice, also, the 'test2', 'test', parts at the end of the command line. These are the arguments that are passed in to main which represent the node name and the mail box name. With this java process up and running we can go back to our Erlang shell and play a bit. The next snippet illustrates a message transfer to and from our java node.

```
(test1@my.host.name)1>  {test, 'test2@my.host.name'} ! {self(), "Hello"}.
{<0.39.0>,"Hello"}
(test1@my.host.name)2> receive Data -> Data end.
{<5569.1.0>,"Greetings to","Hello"}
(test1@my.host.name)3>
```

We are using Erlang's built in distribution primitives to send and receive messages. In this case we are sending a simple tuple message (the greeting we talked about earlier) to our Java node. To get the greeting back we are just yanking it out of the process mailbox associated with our shell via receive.

Our goal in this Chapter is to make an HBase based storage mechanism available to Erlang. The approach we have taken is to build his interface up in Java using the JInterface library. The simplicity of mapping between Erlang and Java is actually very helpful to us, as you have seen. Now that we understand the basics of messaging from Erlang to Java and back we can use that knowledge to implement the custom interface between our simple cache and an HBase cluster. We will use many techniques that are similar to ones we have already used in our example. We will be providing a consistent well defined interface and a stable, usable implementation.

13.2 - Building the Simple Cache to HBase Bridge

Our goal here is to build an interface directly between an HBase cluster and our simple_cache server. There is a bit of setup work that we need to do before we get started. That setup work mostly revolves around getting HBase started and running. We will give a quick intro to starting HBase and getting it up and running. As we said at the beginning of this chapter though, this isn't a book about HBase so we are not going into too many details. However, we are going to give you enough information to get an HBase node up and running and get the tables that we require for our cache going as well. Let's get started with that and then we will jump in feet first.

13.2.1 - Installing HBase

The first step is getting HBase installed on our system. HBase comes as a single all in one download that you can simply untar and run. HBase does have one requirement that is somewhat non-obvious. You must have an sshd installed on your system. If you are trying these exercises out on a Linux server you probably won't have a problem. However if you are trying these on a laptop or desktop system, like we did, you just might. You should consult the documentation for your system to figure out how to install sshd.

Once sshd is installed you we can get started installing HBase. To do that download the hbase tarball. For us this was hbase-0.20.0. Untar it as you see in the next snippet.

```
$> tar -xzf hbase-0.20.0.tar.gz
```

Substituting your version of HBase for the 0.20.0 version we use here, of course. Once that is done you can start up HBase from the location in which you untared it. That's a really nice feature that we can appreciate here. Running HBase is pretty easy as well. Just cd into the root HBase install.

```
$> ./bin/start-hbase.sh
```

This is where sshd is important. It's going to ssh back into the box it's on to gather certain bits of information that it needs to run. In so doing it's going to ask you for your password a couple of times. It's no problem at all but probably something you should be aware of. Once installed we move on to setting it up.

13.2.2 - Setting up HBase

For our needs there isn't a whole lot of setup that we need to actually do. We just need to start the Hbase shell and create a table that we can use to store the key/value information from the cache. Starting the shell is demonstrated in the next snippet.

```
$>./bin/hbase shell
```

You should probably note that there is a space between 'HBase' and 'shell' instead of a dash. Shell is treated like an argument to hbase not an executable all on its own.

Now that we have a shell we need to create the tables that we are going to need as well. To be clear they aren't actually tables, not in the relational sense, however, that term works just fine for our needs. If you would like to know more we encourage you to pick up one of the books out there on HBase.

Let's create our table. They will consist of some type of unique identifier and a binary value. We will do this using the HBase create command.

```
hbase(main):006:0> create 'cache', {NAME => 'value'}
0 row(s) in 4.1560 seconds
hbase(main):007:0>
```

We are basically creating a table, or a map actually, called 'cache' with a single field called value. HBase stores everything as a binary value so we don't actually need to specify any type like you would on a normal database. Congratulations, we just brought up a single node Hbase cluster complete with table and all. Now that we have our single node HBase cluster up and running let's talk about our interface application itself.

13.2.3 - The Simple Cache to HBase Bridge

This application will give us a way to provide an method to get and put key/value pairs into our HBase cluster. In essence it will provide a bridge from our Simple Cache to an HBase cluster. The interface it supports is very similar to the one the Simple Cache itself supports. It consists of a single module with three functions get, put and delete. Put gives us the ability to store any Erlang term we would like into the cache. The format for the message is the action identifier (put in this case), followed by the PID of the sending process, followed by a unique token which identifies the transaction, followed by a key and then a value. This provides enough information to both put the value into the cache and receive confirmation as to its success. The next snippet shoes the code for put on the Erlang side.

```
put(Node, Key, Value) ->
    Ref = make_ref(),
    {hbase_server, Node} ! {put, self(), Ref, Key, term_to_binary(Value)},
    receive
     {put_result, Ref, ok} ->
         ok
    after 1000 ->
         error
    end.
```

The second part of our API is get. The format of this message is the action identifier, the sender (self) followed by the unique identifier, followed by the key. What gets returned to us a tuple that represents the result.

```
get(Node, Key) ->
    Ref = make_ref(),
    {hbase_server, Node} ! {get, self(), Ref, Key},
    receive
     {get_result, Ref, Value} ->
         {ok, Value}
    after 1000 ->
```

```
        error
    end.
```

The third part of our API is delete. This massage we send is almost exactly the same as get with the exception of the action identifier which is delete. This is because neither get nor delete need any payload of data.

```
delete(Node, Key) ->
    Ref = make_ref(),
    {hbase_server, Node} ! {delete, self(), Ref, Key},
    receive
     {delete_result, Ref, ok} ->
        ok
    after 1000 ->
        error
    end.
```

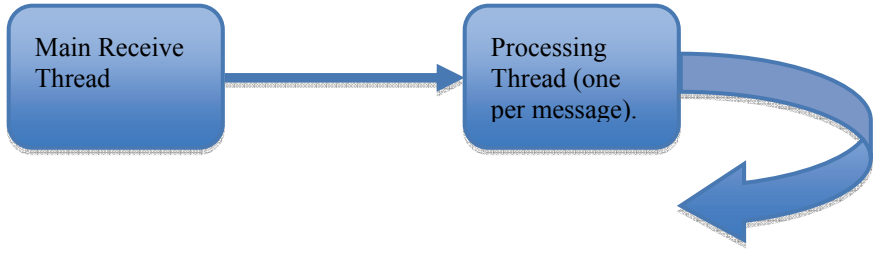
Notice that one of the arguments to all three of these functions is a node name. This allows our interface to be configuration free with regard to what node it needs to be talking to. With the Erlang side done in just those few strokes let us turn to the Java side of the Hbase bridge implementation.

13.2.4 – The Java Implementation

The Java/JInterface side of our example is just a tad more complex then the Erlang side. We have made an effort to keep it simple and understandable while still providing a reasonably robust implementation. The Java side of our application takes Erlang messages, decomposes them, processes them, and then returns the result. We have made it more interesting by taking and spinning off a Java thread to process each message. This would usually be inefficient but we make it a little bit more efficient by using a thread pool class that the Java standard library provides for us. This way we get to have a little fun and mirror, to a small extent, some of the asynchronicity of Erlang. Of course, Java isn't as geared towards an asynchronous approach as Erlang. Fortunately, most of the complexity of multithreading and asynchronicity is hidden the classes that we are using in JInterface.

The shape of our application is minimal. We have a Java class that represents the main entry point of our node. This main class sets up the Jinterface node and the receiving mailbox. It then goes into a loop pulling messages off of the mailbox as they come into the system. For each mailbox that it pulls off kicks off a thread that processes that message and does the requested action. Once the action is complete it sends a response back to the caller who originated the request. You can see a visual depiction of this setup in figure 13.3.

Figure 13.3 High Level View of the HBase Node



Java is an object oriented language and so every piece of what we just talked about is represented by a class. In this case, the main entry point for the system is in SimpleCacheHBaseMain cass, it also does the initial parsing of the message and kicks of the handler thread. The handler thread is embodied by the HBaseTask class. There are other tertiary and support classes that we will get into, but these two are the main workers in the system.

Choke Points

One thing to be aware of in the design of this system is that we are using a single OTPMbox in our implementation. This provides an implicit bottleneck to message flow. For our purposes in this application that is acceptable. However, it may not be acceptable for other types of applications. This is a common problem in Erlang applications and many of the same approaches that you would use in an Erlang application to address the same problem can be used here.

THE ENTRY POINT – SIMPLECACHEHBASEMAIN

Lets get started by looking at the main entry point to our HBase node, the SimpleCacheHBaseMain class. We will start by looking at its constructor and definition.

13.2 The SimpleCacheHBaseMain Class

```
public class SimpleCacheHBaseMain {

    private HBaseConnector _conn;
    private ExecutorService _exe;
    private OtpNode _node;
    private OtpMbox _mbox;
```

```
    public SimpleCacheHBaseMain(String nodeName,
                                String cookie,
                                String serverName)
                    throws IOException {
            super();

            _conn = new HBaseConnector();       #1
            _exe = Executors.newFixedThreadPool(10); #2
            _node = new OtpNode(nodeName, cookie); #3
            _mbox = _node.createMbox(); #4
            _mbox.registerName(serverName); #5
    }
```

The constructor does a lot of work to set up the system. Most of that work is abstracted into other classes that we use to our advantage. Some of these classes we wrote (HBaseConnector, which we will go into detail on later), others are provided by the Jinterface library (OtpNode and OtpMbox) and others are provided by the Java standard library (ExecutorService). First we create an instance of the HBaseConnector (annotation #1). It opens up a connection to the HBase instance running on localhost from which we will be pulling data. Next we create the thread pool that we are going to sue to manage all of the worker threads that will be running to process the incoming data (annotation #2). Then we are using OtpNode to create a new JInterface node that starts talking to other nodes with the help of the EPMD daemon (annotation #3). Finally we create the mbox and register it with the node (annotation #4 and annotation $5). So there is actually a lot going on here, but all of that complexity is abstracted away, making it very easy to step a new node. The code in listing 13.3 shows the main method of the SimpleCacheHBaseMain class. The method will provide the main entry point for our system.

13.3 The Main Method

```
    public static void main(String[] args) throws Exception {
            if (args.length < 3) {
                    usage();
                    return;
            }

            String nodeName = args[0];      #1
            String cookie = args[1];
            String serverName = args[2];

            SimpleCacheHBaseMain main = new SimpleCacheHBaseMain(nodeName,
cookie, serverName);
            main.process();

    }
```

The main method is the one entry point for the system. It takes the command line args and starts running the node. This main method takes the node name, cookie, and server

name off the command line starting at annotation #1. It also instantiates the SimpleCacheHBaseMain class and calls the process method. The process method itself is where the magic lies and it is illustrated in code listing 13.4.

13.4 The Process Method

```
   while (true) {
     try {
        OtpErlangObject o = _mbox.receive();
        assert (o instanceof OtpErlangTuple);        #1
        OtpErlangTuple msg = (OtpErlangTuple) o;       #2
        OtpErlangPid from = (OtpErlangPid) msg.elementAt(0);
        OtpErlangRef ref (OtpErlangRef) msg.elementAt(1);
        String action = ((OtpErlangAtom) msg.elementAt(2)).toString();
        String key = ((OtpErlangList) msg.elementAt(3)).toString();

        HBaseTask task = null;
        if (msg.arity() == 3) {       #3
           data = ((OtpErlangBinary) msg.elementAt(3)).binaryValue();

           task = new HBaseTask(_mbox, _conn, from, action, key, data);
        } else if (msg.arity() == 2) {        #4
                task = new HBaseTask(_mbox, _conn, from, action, key, null);
        } else {          #5
                OtpErlangTuple res = new OtpErlangTuple(new OtpErlangObject[] {
                new OtpErlangAtom("error"),
                new OtpErlangAtom("invalid_request") });

                _mbox.send(from, res);
                return;
        }

      _exe.submit(task);          #6

   } catch (Exception e) {
                System.out.println("" + e);
                        }
   }

}
```

We start out in the same way as we did in our example code for our greeting server by setting up an endless loop. This endless loop is just where we sit in a loop on the message receive and wait for incoming messages the same way as is done in Erlang/OTP servers. When we get a message into the mailbox we pull it off the queue and process it. First we make sure that the message we receive is a tuple with an assertion at code annotation #1. For right now, if the incoming message is not a tuple we really don't handle that other than by printing an exception to stdout. Without a properly formed tuple we don't know who the sender is and so cannot even provide a reply if we wanted to. As this code moves further forward in its readiness for production we will at least want to log out the fact that we got a

message that we didn't know how to process. For our purposes though, the purposes of a Java example, this will work just fine.

If the message actually is a tuple we start processing it. At code annotation #2 we start with decomposing the message by pulling off the message elements, casting them to the types we are actually expecting and continuing on with our processing. As you can imagine there are a bunch of things that can go wrong in this scenario. The biggest problem we might face is a message that is formatted incorrectly, either it has a wrong number of elements or it the elements are not of the type that we actually expect. In either case an exception is going to be thrown and we will cease to process the message. Basically, we discard it and continue processing other messages in the queue. Basically this falls into the same scenario as having a message that was not actually a tuple; we don't have any idea of who the sender is and so cannot return some useful reply. The only alternative we have is to log the problem and continue processing.

The incoming tuple may have two or three elements depending on the type of message it is, if it is a get or delete message and contains no data its only going to have two elements, if it's a put message it will have three. If it has three elements (annotation #3) we create an HBase task and give it a valid data value. If it has two elements we do the same thing, but pass null as the data value (annotation #4). If it has any other number of elements then there is an error and we attempt to send that error back to the caller (annotation #5).

Finally, once we have a good HBaseTask object that is read to run, we kick it off by passing it to the thread pool we created in our constructor and asking it to run the task (annotation #6).

Erlang vs. Java in Message Decomposition

I wanted to take a moment here and just point out the difference in decomposing a message in an imperative language like Java vs. a declarative language like Erlang. In Java it takes some ten or so lines of code to decompose this simple tuple. The same code in Erlang looks like this.

```
case Message of
  {Msg, From, Action, Key} ->  do something here;
  {Msg, From, Action, Key, Value} -> do something else here;
End.
```

The difference in clarity and brevity is significant. Now we are not doing this to pick on Java or really even promote Erlang. We are showing this to give you an idea of the power of declarative constructs verses imperative constructs for this type of task.

THE WORKER - HBASE

All of the interesting things that happen happen in HBaseTask. Let's dive into our HBaseTask object and take a look at what it is actually doing. As with any class that implements Threadable, the main entry point is the run method. In our case, this is where we dispatch to the appropriate handler method based on the action requested.

13.5 The Run Method

```
public void run() {
        try {
                if (_action.equals("get")) {
                        doGet();
                } else if (_action.equals("put")) {
                        doPut();
                } else if (_action.equals("delete")) {
                        doDelete();
                }
        } catch (IOException ioe) {
                OtpErlangTuple res = new OtpErlangTuple(
                                new OtpErlangObject[] {
                                new OtpErlangAtom("error"), _ref,
                                new OtpErlangList(ioe.getMessage()) });
                _mbox.send(_from, res);
        }
}
```

The run method of our HBaseTask takes the action specified in our tuple and dispatches it to the appropriate method to handle it. If you remember the tuple that the Erlang node sent us has already been parsed and provided to us in a consumable way in our main class. Basically, if the method is a 'get' message we call the doGet method of our class, if it's a 'put' we call the do put message, etc. If we get something besides that we return an error to the calling process.

Remember that we want to provide a simple way to manipulate key/value pairs in HBase from the Erlang side. To be able do that we are going to support three actions, they are 'doGet', 'doPut', and 'doDelete'. The 'doGet', 'doPut', and 'doDelete' methods actually do the work of the various actions. Let's take a look at what's going on there, starting with doGet.

```
        private void doGet() throws IOException {
                byte[] data = _conn.get(_key);

                OtpErlangTuple res = new OtpErlangTuple(new OtpErlangObject[] {
                                new OtpErlangAtom("get_result"), _ref,
                                new OtpErlangBinary(data) });

                _mbox.send(_from, res);

        }
```

doGet does exactly what it seems to do. It gets the value specified by the key and returns it to Erlang in the correct format. We use our HBaseConnection API to retrieve the value, format it and send it back to Erlang. Let's take a look at the put elements of our API.

```
        private void doPut() throws IOException {
                _conn.put(_key, _value);

                OtpErlangTuple res = new OtpErlangTuple(new OtpErlangObject[] {
                                new OtpErlangAtom("put_result"), _ref,
                                new OtpErlangAtom("ok") });

                _mbox.send(_from, res);

        }
```

Once again the doPut is a mirror image of the doGet. We use our HBaseConnection again to put the value in by the key specified. Then we send a good result back to Erlang. In either case, if an exception is thrown we send an error back to the Erlang node with details. Finally, we have the doDelete.

```
        private void doDelete() throws IOException {
                _conn.delete(_key);

                OtpErlangTuple res = new OtpErlangTuple(new OtpErlangObject[] {
                                new OtpErlangAtom("delete_result"), _ref,
                                new OtpErlangAtom("ok") });

                _mbox.send(_from, res);
        }
```

The delete function just allows for removing a key from the HBase system.

We now have a working node that will interact with HBase to get, put and delete values. However, we have hidden the core interaction with the HBase system behind the HBaseConnector class. Lets explor that class and how it interacts with HBase a little.

INTERACTING WITH HBASE – THE HBASECONNECTOR CLASS

The fundamental purpose of our system is to interact with HBase. The HBase API is a general API and can be somewhat baroque. To simplify using HBase in our system we are going to wrap the API and make it a little more useful in the context of the simple cache. This API just gives us a way of interacting with HBase to gather or store the data that we are interested in. This API is contained within a class called HBaseConnector. The constructor for that class is shown in the next snippet.

```
public class HBaseConnector {
      private HTable _table;       #1

      public HBaseConnector() throws IOException {
              super();

              _table = new HTable(new HBaseConfiguration(), "cache");
      }
```

The HBaseConnector class has a single member variable (annotation #1) which holds an HBase object. This HBase object provides us access to an HBase table. Access to HBase is provided in the class HTable. That class has quite configurable, and so takes an HBaseConfiguration object. For our purposes we don't actually need to configure the HTable, the defaults work just fine. So when we create the HTable object we pass in an empty configuration object along with the name for the table we will be accessing.

After we have initialization down we just need to create 'get' and 'put' methods that reflect the needs of the simple cache. As saw when we talked about setting up HBase our table is very simple. Consisting of two columns 'key' and 'value'. Let's get started with the 'get' method that retrieves a value out of the system.

```
      public byte[] get(String key) throws IOException {
              Result result = _table.get(new Get(key.getBytes())); #1
              NavigableMap<byte[], NavigableMap<byte[], byte[]>> res =
result.getNoVersionMap(); #2

              return res.get("value".getBytes()).get("".getBytes()); #3
      }
```

We get the value out of HBase using the native HBase API. Using the table we we instantiated in the constructor, we make a get request (annotation #1). We do that by using the Get object to designate the entry that we are interested in. Notice that we call the 'getBytes' method of the key. We do that because HBase only understands byte arrays, and makes no assumptions as to the types of data that in stores. This means that any time we interact with HBase it has to be via byte arrays as well. This is one of the reasons that we are wrapping the API. The get method of our table object returs a Result object that we can use to retrieve the actual value that we are interested. However, first we need to crate a

NavigableMap that will describe the data to HBase and allow us to access it (annotation #2). Finally we can get the result out of our navigable map. Getting the actual value of out HBase is not quite as clear as one would hope. We take the navigable map and call the get method on it with the name of our field, converting that field to bytes as we do (remember HBase only understands byte arrays). That gives us another navigable map object that contains the different values identified by their domain. When we put objects into HBase we specify an empty domain, so we want to do the same thing when we retrieve them, so we pass in an empty byte array. This will give us the bytes that make of the value identified by the key passed in.

The 'put' method is much simpler but posses basically the same functionality, just reversed.

```
public void put(String key, byte[] value) throws IOException {
            Put put = new Put(key.getBytes()); #1
            put.add("value".getBytes(), "".getBytes(), value); #2

            _table.put(put); #3
    }
```

We create a Put Object, specifying the key in the constructor (annotation #1). We then add the value, in our default empty domain (annotation #2). Notice again that we are only passing in byte arrays to HBase. Finally, we use the table object that we instantiated in the constructor of our class to actually insert the value into the system.

Finally we have the delete method. This method is also very similar to the 'put' and 'get' methods.

```
    public void delete(String key) throws IOException {
            Delete del = new Delete(key.getBytes()); #1

            _table.delete(del); #2
    }
```

First we create the HBase Delete object (annotation #1), then we pass that to the delet method of the table object (annotation #2). Once again, we only pass byte arrays to HBase.

Our original goal here was to provide an way to manipulate HBase values from Erlang. In persuit of that goal we have defined a basic API for interacting with HBase. We now continue by taking a look at the core part of our system, the SimpleCacheHBaseMain class. This class provides the core functionality for the Java node that will be providing services to our system. Code listing 13.2 shows its few member variables and a constructor.

That's our API. It provides the minimal interface, while still providing all of the functionality. The only thing left to do is embed it into the simple_cache. Let's take a look at that.

INTERACTING WITH THE SIMPLE CACHE

To integrate with our HBase application with the simple_cache we need to modify the lookup and insert functions so that they understand how to interact with our HBase system. The more complex of these two interactions is the insert function. Let's start there.

INTEGRATING HBASE INTO LOOKUP

First we must modify the simple_cache lookup API to fail over to HBase when we can't find the value locally. We do this failover in code listing 13.5.

13.5 The simple_cache Lookup

```
lookup(Key) ->
    sc_event:lookup(Key),
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_element:fetch(Pid);
     {error, not_found} ->
         case simple_cache_hbase:get(?NODE, Key) of
            {ok, Value} ->
                insert(Key, Value),
                Value;
            _ ->
                {error, not_found}
         end;
     {error, Reason} ->
         {error, Reason}
    end.
```

The change here that we should focus on is the addition of the 'error, not_found' clause of the result. Basically, if the value isn't found in the lookup we want to do two things. First and foremost we want to check to see if the value is in HBase. If it is great we pull it out and reinsert it into the cache. The cache is much quicker then HBase for our general lookups and we want to it be in the cache if at all possible. So anytime we find it in HBase but not in the cache we reinsert it into the cache and return the value.

INTEGRATING HBASE INTO INSERT

The second function that we need to look at is the insert function. All this function does is make sure that the value is inserted into HBase whenever it is inserted into the cache. This is somewhat trivial but pretty damn important.

13.6 Insert Function

```
insert(Key, Value) ->
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_event:replace(Key, Value),
         sc_element:replace(Pid, Value),
         simple_cache_hbase:put(?NODE, Key, Value);
     {error, _Reason} ->
         {ok, Pid} = sc_element:create(Value),
         sc_store:insert(Key, Pid),
         sc_event:create(Key, Value)
    end.
```

The only thing we have added here is an insert into the HBase cluster when a new value is inserted into the system.

INTEGRATING HBASE INTO DELETE

Of course, we need to keep our values clean. So when a value is deleted from the system it needs to be deleted from our HBase table as well. To do that we add a delete call to the delete interface of the simple_cache.

```
delete(Key) ->
    sc_event:delete(Key),
    case sc_store:lookup(Key) of
     {ok, Pid} ->
         sc_element:delete(Pid),
         simple_cache_hbase:delete(?NODE, Key);
     {error, _Reason} ->
         ok
    end.
```

Notice the call to the delete function of the simple cache right after the delete call to sc_element. With this we make sure that the value is removed from the cache and HBase is not reseeding the value when it is looked up.

Summary

Now you understand how to create and include a non-Erlang node into your system and interact with that. There are a huge number of Java Libraries and Projects out there, you can use the techniques you learned here can be extrapolated any of these libraries you care to use. We have actually just covered the most used approach to integrating JInterface into your cluster. We haven't covered every single feature of the JInterface library. However, this chapter should have given you the knowledge you need to continue learning about JInterace on your own. . It is a worthwhile endeavor to read the documentation around JInterface that is included in the Erlang/OTP distribution.

14

Optimization and Performance

"There is no such thing as fast, only fast enough"

I remember the first time I met Joe Armstrong, one of the creators of Erlang, back in 2002 at a functional programming conference here in the states. He was filled with tons great sayings and computer science maxims some original and some quoted many of which stuck with me. One of my favorites was "Make it work, then make it beautiful, then if you really, really have to, make it fast." but he went on to say "90% of the time if you make it beautiful it will already be fast, so really, just make it beautiful!" I relate to you this quote because I want you to go into this chapter with a real grasp that modifying your code for efficiency sake, and that alone, should be done as a last resort and only after your code has been made beautiful and has still proven not to be fast enough. This chapter is about that small percentage of cases when making it beautiful does not do the trick. Optimization should only be on your mind when you really need to get down to business and shave off those extra milliseconds and microseconds; which is what the Erlware team needs to do right now. It turns out that having a repository of Erlang software available for download is something that lots of people want. With the new improved site speed and functionality, thanks in part to the Simple Cache' more and more people are hitting the Erlware.org site. Speed is starting to be come an issue once again. Throwing hardware at the problem isn't really an option in this case, so horizontal scaling just isn't a viable approach. The Erlware team has got to optimize their code to save every clock cycle they can. Since the code is already beautiful that means that all they are left with is low level optimization.

Once you get the glaring errors out of the way, the only real way to be successful at performance improvement is to be systematic. Once expectations have been set, measurements have been taken, base lines established, and bottle necks discovered then an engineer can go in, refactor, then measure again to see whether or not the performance has

been positively effected. This is exactly the way the Erlware team will approach tuning. In this chapter we will share the tools and tricks they employ to get the job done.

We will break this chapter down into a few high level sections.

1. How to approach performance tuning.
2. How to measure along with two tools that can help you in your measurement efforts.
3. Performance caveats and improvements that can be made with judicious handling of the basic Erlang data types as well as basic erlang built-ins.
4. Improvements that can be made with Erlangs processes and functions.

14.1 – How to approach performance tuning

Performance tuning is said to be an art, and in many respects it is. Some people have great skill at divining bottlenecks and cleverly removing them. This chapter will not focus on that because such art is largely a result of the intuition that comes with years of practical experience and know how. In this chapter we will focus more on the science and look to approach performance tuning in a methodical formulaic manner.

The Erlware group approaches this problem with the following 4 steps.

1. Determine performance goals

Based on the number of actual web site hits they receive and the transaction volume that generates on different parts of the system they determine what the acceptable performance goals are currently. They also take a look at the historical rise in traffic and their current promotional efforts and take a stab at what the traffic will look like 6 months from now. Using this data they come up with goals for the performance tuning effort.

The goals that come out of this effort should have the following characteristics, known by the SMART acronym.

Specific	Peak CPU usage, throughput per unit time, etc…
Measurable	Should be possible to verify that the goals are attained
Attainable	The effort should make every effort to, and

	expect to, achieve them
Realistic	If the goal is only at attainable with tremendous exertion by the whole team then it is not realistic. The goal should be realistic with regard to current resources and motivation.
Timely	Say when you need to finish this. Don't leave the tuning effort open ended. Time boxing it focuses your thinking and allows you to brush away the unimportant.

–

2. ESTABLISH A BASELINE

Since the goals are all measurable base lines can be established. Run stress tests to find out where you stand currently with regard to the metrics you have set out by which to measure your goals. Where does your system sit with regard to CPU consumption and throughput at the onset of tuning? The broader baseline you can get the easier things will be when trying to determine what kind of impact you have made with any future refactoring.

3. PROFILE CODE AND REFACTOR BOTTLENECKS

The third step is where action is taken. In this step we profile the code base to discover bottlenecks and then refactor, ideally one at a time, the bottlenecks that are discovered.

4. MEASURE THE RESULTS

Rerun the stress tests after refactoring a bottleneck discovered during step 3. Determine if the results were positive or negative. Check the results against what you stated were your performance goals. If you have reached the goal, congratulations! If not, and likely one iteration of refactoring and measuring will not get you there, then go back to step 3 and continue profiling, refactoring, and measuring until you reach your goal or time runs out and you determine that it is time to go and get more money and more hardware.

In order to be effective we need to set SMART goals and have a solid baseline but those things are a bit beyond the scope of this book. What is right within the scope of this book is the practice and the tools used to profile Erlang code. We will dig into that topic with the next section.

14.2 – How to profile Erlang code

Profiling and measuring is the most reliable method for determining where performance bottle necks are in your system. With practice you can learn to spot the big gotchas and glaring performance pitfalls in Erlang code, you will also be able to spot many quick wins, but at the end of the day tracking down most performance bottlenecks requires profiling no matter how good you are. Erlang/OTP puts a number of profiling tools at your disposal, we will cover two major ones here; fprof and cprof. We will first dive into fprof for no other reason than we believe it to be one of the highest value of all the performance measureing tools shipped with Erlang/OTP.

14.2.1 – profiling in Erlang

There are two profiling tools avaible in Erlang, fprof and eprof. fprof is a new profiling tool distributed with Erlang, it happens to be the successor to eprof. Fprof has a number of improvments of over the venerable eprof. Fprof has much lower runtime impact than does eprof, though that is certainly not to say that fprof does not have a noticeable runtime impact. Fprof also has much more functionality then eprof. Both eprof and fprof are included in the standard Erlang/OTP distro and still documented. Both systems are built using Erlang's trace functionality. While profiling we are only going to solely at fprof. it gives you a wealth of useful information, in a digestible format, and is quite easy to use. This is not to say that reading and interpreting the results is not tricky at times; it can be. All in all fprof is a nice go to tool when performance issues first need to be diagnosed.

We are going to get started by creating code to profile. Once we run this code its going to give us a good example of the kind of output the fprof generates. Then we will go through this output line by line to get explain whats going on and how its meaningful in the context of profiling. Finally we will dig into some of the caveates and gotchas in using fprof.

RUNTIME_TOOLS APPLICATION REQUIRED

fprof depends on dbg which is part of the runtime_tools application. This must be present on your system in order to use fprof. runtime_tools comes with Erlang/OTP but if you are installing via alternate means you may need to verify you installed this application package.

CREATING THE CODE

Below in code listing 15.2 is a very simple little module that executes some ineffectual code. We create this code for the sole purpose of profiling it. This should give you a reasonable introduction to the profiling with fprof.

Code Listing 14.2 – fprof example code

```
-module(profile_ex).

%% API
-export([run/0, run_with_fprof/0]).

run_with_fprof() ->
    fprof:trace(start),     #1
      run(),
    fprof:trace(stop),    #2
    fprof:profile(),     #3
    fprof:analyse({dest, atom_to_list(?MODULE)}).    #4

run() ->
     spawn(fun() -> looper(1000) end),
     spawn(fun() -> funner(1000) end).

looper(0) ->
     ok;
looper(N) ->
     integer_to_list(N),
     looper(N - 1).

funner(N) ->
     funner(fun(N_) -> integer_to_list(N_) end, N).

funner(_Fun, 0) ->
     ok;
funner(Fun, N) ->
     Fun(N),
     funner(Fun, N - 1).
```

Executing the run_with_fprof/1 function will create a file called "profile_ex" in the working directory. Before we analyze the contents of that file we will explain quickly how it is generated. At code annotation #1 the fprof server is started and tracing commences. At annotation #2 we stop the server and clear the tracing data from the node. At annotation #3 profile/0 is called to take all the trace data accumulated in the fprof server from the function call tracing and turn it into raw profiling data. The results of this compilation step are then taken and analyzed by the fprof:analyse/1 function, at code annotation #4, and finally written out to a file, in human readable format, at the destination specified by the {dest, <filename> option. In this case we name the output file the same as the module name. There variations on how to start a trace and arrive finally at analyzed human readable output. We leave the study of those options as an exercise for the reader at a some later

date. The method for generation is not nearly as important as the ability to analyze the file which is what we will get into next.

ANALYZING FPROF OUTPUT

What follows are a few salient snippets from analyzed profile data residing in the profile_ex file. We will discuss these snippets explaining what they mean and through the discussion walk you through how to analyze performance with fprof.

```
%                                                    CNT       ACC       OWN
[{ totals,                                          4689,   78.590,   78.552}].
```

The two lines above are the first profiling lines in the fprof file and are found pretty much at the very top of the file. These numbers represent, as indicated by the heading "totals", the sums of the various things that fprof tallies. The column headed "CNT" is the total number of function calls that occurred from the start of tracing to the end. "ACC" represents the total amount of accumulated time of all function calls from start to finish. "OWN" represents an addition of the total time spent in the specific function, it does not include time spent in other functions called from within the profiled function. These times can differ significantly if there is a lot of other code running that is not under profile. Here the only thing running is our code and so these times are very much the same. Moving on to the next important section of the output file we see what follows.

```
%                                                    CNT       ACC       OWN
[{ "<0.47.0>",                                      2666,undefined,   44.376},
%%
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.1.118133018>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-1-',0}]}].
```

What we are looking at is the section heading for all the profile data for one of the two processes we spawned in the run function. Another process group heading can be found further down in the trace file. The headers continue to signify the same things here. We have a CNT of 2666 function calls. OWN adds up to 44.376 milliseconds. With a quick search we can find the heading for our second spawned process a little bit further down in the file.

```
%                                                    CNT       ACC       OWN
[{ "<0.46.0>",                                      2011,undefined,  33.924},
%%
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.0.9423607>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-0-',0}]}].
```

Actually looking at the Pids for these processes we can ascertain from the numbers, <0.46.0> vs <0.47.0> that the second process in the file was actually the first one spawned. Adding up the CNT and the OWN from this header and the header representing our second process yields just about the totals shown in the initial file level header. The small bit missing comes from the third process we find at the bottom of our trace file which is actually spawned by fprof itself. Note that in both cases ACC is undefined, this is because it is not something that makes sense to define for a process. ACC will come into play next as we get into the function call stack trace figures. So far from looking at our trace file we can see that two application processes were spawned and that <0.47.0> took up 44.376 with a count of 2666 while <0.46.0> rated 33.924 with a count of 2011. Lets dive down and find out why process <0.47.0> was longer running.

CAVEATES WHILE USING FPROF

Looking down into the meat of this trace file it is plain to see that interpreting the results can be a little bit confusing. Things will not always seem to add up, in most cases they do, but in some cases they actually will not. These tricky things include the following

- Tallying ACC times for example is a tricky operation particularly when the function exists as part of a complex mutually recursive operation.
- On some systems which don't allow for very granular CPU time measurements when the actual ERTS system is subjected to OS scheduling. When this happens it can appear that a function burns more time than it should even when doing nothing. If you see something that looks really unreasonable the best check would be to run the profile again and compare the results.

These phenomena do not, in general prove to be, too troublesome. However, reading these files is a little bit art and a little bit science. Don't get hung up on trying to track down every microsecond. Take in the whole file and you will be able to quickly zero in on where time is being spent.

An easy first place to start the investigation of profile analysis data is to take a look at suspension and the garbage collection times. Garbage collection is the time the system spends collecting all the unused memory that was allocated during runtime that is no longer needed. The garbage collector is also responsible for growing and shrinking process heaps as needed. Suspension is the time a process spends unscheduled. A process with low priority will spend more time in suspension than one with high priority as would a process that does a lot of io, or blocks to receive messages. Focusing in on garbage collection and suspension will be able to tell us if the difference in the time spent in one process vs. the other was due

to either suspension or garbage collection. Processes can be suspended due to a variety of reasons such as for a call to yield, entering a receive block, timing, waiting for io, etc… Suspend, which is shown as a function for display purposes but is not actually a call to a function, always shows an OWN time of 0 so as to avoid counting this time in the overall time spent in a process. Looking at suspend for both processes we can see that they are fairly equivalent.

for <0.47.0>

```
{[{{erlang,apply,2},                          1,   34.035,    0.000},
  {{profile_ex,funner,2},                       1,    0.037,    0.000}],
 { suspend,                                   2,   34.072,    0.000},  %
 [ ]}.
```

and for <0.46.0>

```
{[{{profile_ex,looper,1},                       1,   34.699,    0.000},
  {{erlang,apply,2},                          1,    0.188,    0.000}],
 { suspend,                                   2,   34.887,    0.000},  %
 [ ]}.
```

You may ask how we know the clauses above relate to suspend itself and not to one of the other functions listed in the stanza. The function focused on by a stanza is indicated by a % sign after that function. The above functions are information about the functions that called the function in focus and ay functions below which there are not in this case would be functions that the function in focus subsequently called. In this case you can see an empty list due to the fact that suspend called nothing else of course. Garbage collection typically happens during suspension and it is also one of those pseudo function calls. Our processes don't actually call garbage collect. Looking at the time spent garbage collecting between both processes we can see that it is roughly equivalent and not a cause for the difference in execution times.

for <0.47.0>

```
{[{{profile_ex,'-funner/1-fun-0-',1},            6,    0.214,    0.214}],
 { garbage_collect,                           6,    0.214,    0.214},  %
 [ ]}.
```

and for <0.46.0>

```
{[{{profile_ex,looper,1},                       6,    0.174,    0.174}],
 { garbage_collect,                           6,    0.174,    0.174},   %
 [ ]}.
```

Ruling those two things out we move on to look at the function calls and operations actually performed in each and where the time came from. In <0.47.0> the next line down from the process header information is the following:

```
{[{undefined,                                  1,    78.448,     0.009}],
 { {erlang,apply,2},                           1,    78.448,     0.009},   %
 [{{profile_ex,'-run/0-fun-1-',0},               1,    44.404,     0.001},
  {suspend,                                    1,    34.035,     0.000}]}.
```

The undefined at the top only means that the function that called into what was traced was called before the tracing actually started. We can ignore it. Below that we see erlang apply/2 which is again not something that we care about, moving down to the third line we see the fun that our process was spawned around. We will start walking down the stack there. The next stanza down contains the % sign after {profile_ex,'-run/0-fun-1-',0} which indicates that the stanza focuses on that function.

```
{[{{erlang,apply,2},                           1,    44.404,     0.001}],
 { {profile_ex,'-run/0-fun-1-',0},               1,    44.404,     0.001},
%
 [{{profile_ex,funner,1},                        1,    44.403,
0.002}]}.
```

Nothing too interesting there in that the function spent very little time in OWN. What the stanza does indicate is that from here the next function called was funner.

```
{[{{profile_ex,'-run/0-fun-1-',0},             1,    44.403,     0.002}],
 { {profile_ex,funner,1},                      1,    44.403,     0.002},
%
 [{{profile_ex,funner,2},                      1,    44.401,     0.013}]}.
```

Similarly here we have very little time spent in own but we see that funner/1 calls funner/2 which is where we will go next. Looking at our code we can see that funner/2 is where this process should spend most of it's time.

```
{[{{profile_ex,funner,1},                        1,    44.401,     0.013},
  {{profile_ex,funner,2},                      884,     0.000,
17.443}],
 { {profile_ex,funner,2},                      885,    44.401,    17.456},
%
 [{{profile_ex,'-funner/1-fun-0-',1},          885,    26.908,    18.194},
  {suspend,                                    1,     0.037,     0.000},
  {{profile_ex,funner,2},                      884,     0.000,
17.443}]}.
```

Now we are getting somewhere, we have found a function that takes up a considerable amount of time in OWN. Digging down just a bit from here it is easy to see upon comparison

with our other process that the addition of the fun in the funner process is where the extra time is being spent. Because integer to list occurs in both processes.

<0.47.0>

```
{[{{profile_ex,'-funner/1-fun-0-',1},          885,    8.500,    8.500}],
 { {erlang,integer_to_list,1},               885,    8.500,    8.500},     %
 [ ]}.
```

<0.46.0>

```
{[{{profile_ex,looper,1},                     1000,    9.505,    9.505}],
 { {erlang,integer_to_list,1},               1000,    9.505,    9.505},   %
 [ ]}.
```

Looking at the code it is easy to see that the funner process will be more time consuming than the looper, after all we added a fun just for the extra overhead. The important thing is here is not that you proved something obvious with fprof, but that you have gained an understanding of how to read these files and determine where time is being spent is processing. There is one other interesting thing about the traces I showed you here that you may be wondering about. That is why is the count showing us how many times we called integer_to_list/1 in process <0.47.0> 885 and the count in <0.46.0> 1000 as we would expect from looking at the code. Erlang is concurrent, it is important to keep that in mind, this is not a sequential system we are working with. When profiling, when doing anything with Erlang, remember that so much of it is concurrent. Take a look at how this example code was started:

```
run_with_fprof() ->
    fprof:trace(start),
   run(),    #1
    fprof:trace(stop),
    fprof:profile(),
    fprof:analyse({dest, atom_to_list(?MODULE)}).



run() ->
    spawn(fun() -> looper(1000) end),
    spawn(fun() -> funner(1000) end).
```

The function run/1 is called at code annotation #1 which causes looper/1 and funner/1 to be spawned off and then we immediately proceed to call fprof:trace(stop) which shuts down our profiling, even before the funner process has had a chance to finish it's 1000 iterations. At 885 iterations we stop watching and generate our output. To prove this to your self, run the same code with the addition of a sleep call as pictured in the next example.

```
run_with_fprof() ->
    fprof:trace(start),
    run(),
    timer:sleep(5000),
    fprof:trace(stop),
    fprof:profile(),
    fprof:analyse({dest, atom_to_list(?MODULE)}).
```

Now in the output you can see that 1000 calls to integer_to_list/1 were made within funner!

Fprof provides a simple programmatic interface to measuring time spent in functions that spawn multiple modules and processes. It's output can be a little bit tricky to read but at the end of the day it is providing a wealth of useful data cheaply. The next tool, cprof, does not provide the same depth of information that fprof does but it is truly simple and easy to use as well as versatile.

14.2.2 – counting with cprof

cprof is a very simple tool for counting function calls. Its main point of interest and the reason you would use it instead of fprof is that it has very minimal impact on the running system. It is the most suitable of the tools we will cover here for doing analysis in production, should that be required. Since we are already familiar with the code and the operation of the module contained within profile_ex.erl we will use profile_ex again for demonstrating the functionality of cprof. The shell session shown in code listing 15.3 demonstrates the simplest way to use cprof.

Code listing 14.3 – demonstrating cprof

```
Eshell V5.7.2  (abort with ^G)
1> l(profile_ex).
{module,profile_ex}
 2> cprof:start().   #1
5359
3> profile_ex:run().
<0.43.0>
 4> cprof:pause().   #2
5380
 5> cprof:analyse(profile_ex).   #3
{profile_ex,3007,
          [{{profile_ex,looper,1},1001},
           {{profile_ex,funner,2},1001},
           {{profile_ex,'-funner/1-fun-0-',1},1000},
           {{profile_ex,run,0},1},
           {{profile_ex,module_info,0},1},
           {{profile_ex,funner,1},1},
```

```
            {{profile_ex,'-run/0-fun-1-',0},1},
            {{profile_ex,'-run/0-fun-0-',0},1}]}
6> cprof:stop().
5380
7>
```

I really like this tool because it is so simple and easy to use. At code annotation #1 cprof is started. No special compilation, you need do nothing in the code being counted to enable it. This makes it a very useful tool. Many times you will just be interested in the counts of a specific module instead of the system as a whole. Cprof allows us to do this if we just call start with the specific name of the module to be profiled.

```
2> cprof:start(profile_ex).
10
```

This is advisable for further limiting the impact of the analysis. Either way the next step we take at code annotation #2 is to pause cprof causing it to stop counting what ever it is that we were measuring which in this case was the call to profile_ex:run/1. Code annotation #3 is the call to analysis which dumps the output to the terminal as fairly readable erlang term. We can see from our term that the results are much as expected. We see the full 1000 calls to both looper and funner. Notice though that the integer_to_list/1 calls don't show up. The reason for that is that because cprof uses breakpoints to collect counts only beam code can be evaluated. BIFs are built right into the emulator, are not beam code, and therefore can't be counted. cprof is a simple, easy way to find out what functions are being called and how many times, nothing more nothing less.

There are other related tools to look at, such as cover for code coverage and the instrument module for doing memory usage analysis but in general their usage for doing performance tuning is less common. With these two tools in your arsenal you have a really nice footing for doing some very powerful performance analysis on your systems. This will allow you to optimize the way your processes, functions, types and operators are used. This allows you to measure which performance related refactorings were positive and which were not. Knowing what works and what does not out of a list of options for implementation is the best practical tool you have for shaving off microseconds, milliseconds, and depending on the situation even minutes or hours off execution times so that you can reach your performance goals.

We now have tools in our arsenal that allow for performance profiling to be done. The other part of step three

"Profile code and refactor bottlenecks"

is the refactoring bottle necks part. The next section will start providing us the information we need to be able to critically look at Erlang code and figure out where resources were used poorly or at least could be more efficiently used.

14.3 – Performance related to primitive data types and BIFs

One place to get started in peformance tuning is Erlang's primitive data types and built in functions. These are some of the most fundamental building blocks of your code and some of the most called. It is important to know how to deal with them in an efficient manner and understand any caveats associated with each. To get our discussion kicked off lets deal with the most basic of the basic and start with the primitive types.

14.3.1 – The primitive types

The table below lists the fundamental primitive types in Erlang along with their associated sizes.

Data Type	Size in Memory
Integer	1 word
Big Number	3 words up to all the memory available
Float	4 words on 32-bit architectures 3 words on 64-bit architectures
Atom	1 word for each + the atom text once per node. Never garbage collected
Pid	1 word to represent a local process. 5 words to represent a remote process.
Fun	9 to 13 words + size of environment.

The basic types of integers, floats, pids, and funs really don't need to come with any advice about how to use them. They are what they are. It can be helpful to know the size of each of these but only in pretty unusual circumstances.

BIG NUMBERS

Erlang allows for numbers of arbitrary length and precision. When numbers get too large to hold in the space prescribed by standard C datatypes on a given system they are stored as Big Numbers. To the Erlang programmer these Big Numbers look just like integers, but in the Erlang VM they are not. Though they are quite efficient they are still significantly more expensive then numbers that can be represented by ints and floats in the VM. It is

worthwhile to know that there is some cost associated with numbers as they grow larger, however.

ATOMS

Atoms present one of the biggest hidden issues in Erlang programming, one that bites almost all novice Erlang hackers at least once. Atoms are not garbage collected, meaning that once memory is allocated for an atom that memory can never de-allocated during the life of a given ERTS system. This means that anytime an atom is create it never goes away, for the life of the system. I can't count the number of times I have seen someone arbitarryl create atoms on the fly without realizing what they are doing. An example would be a system that handles packets of information from the outside world, creating one tuple for each packet, where each packet has some string that is converted to an atom when the information is parsed. I have seen this many many times. I also personally spent many, many hours diagnosing a memory leak the first time I did this, way back when. Atoms should only be used to represent finite sets of data. Atoms themselves however are very nicely implemented and are the data structure of choice for tagging values in message locally and across the wire. So feel free to use atoms, just don't create arbitrary atoms from untrusted sources of data.

Moving on here we will remain within the realm of primitive types but we will start to talk about those types that serve to contain and aggregate the primitive types we just covered.

Erlang contains three fundamental examples of primitive types that can store or aggregate. The following table contains the size in memory that corresponds to each of those types.

Data Type	**Size in Memory**
Binary	3 to 6 words for the representation + data
Tuple	2 words for the representation + the size of each element
List	1 word per element + the size of each element

BINARY

The binary type is an aggregate type in that it stores or aggregates one or more of some more fundamental type. In this case bits and bytes. As with all aggregate types it is possible add to it, pattern match on it, and send it as an argument to a function or as a payload of a message. There are two basic types of binaries, small ones and large ones. For the most part the idea that there are two types of binaries is transparent to the Erlang developer.

However, its very useful to know the difference including when and how they are created and where. The small ones, those up to 64 bytes are called "Heap Binaries" because they are stored on the process heap, pretty much as all other types are. It is the second type, "Reference-Counted Binaries", that are different in an important way. They have an interesting property that can be exploited for efficiency purposes. I am going to say right here, very clearly, that exploiting this property for efficiency is ugly and I hope you don't ever have to do it. Reference-counted binaries are not stored on the process heap they are stored on a special global heap and are shared between processes. Hence the need to count references to them in order to allow for garbage collection. How is this useful to you? Well in general all message sends consist of copying the message from one process heap to another. In the case of large objects that copy operation can be quite expensive. You can get around this by using large binaries. Because the reside on the special global heap the binary itself is never copied, only a reference to the binary gets copied around in the process heap. This is something you should really only use as a last resort.

Binaries can also offer a large opportunity for screwing up when working with them; or depending on how you look at it a large opportunity for performance improvement with a little refactoring. One way to quickly get an idea of whether or not you are using binaries efficiently is to compile your code after setting the following environment variable:

```
export ERL_COMPILER_OPTIONS=bin_opt_info
```

The bin_opt_info flag, which can also be added as a direct flag to erlc, elicits the compiler to print out quite a bit of data about how binaries are used within the code. This can give you some really good direction when looking to optimize usage. With binaries covered we move on to the rich topic of lists.

TUPLE

Tuples are refreshingly straightforward and efficient. They don't really have any performance caveats associate with them. They are not always a suitable data structure for every task but when they are they are quite efficient. Having said that we can move on to lists.

LIST

Lists present some interesting performance issues in Erlang. Lists in Erlang are always singly linked. That means that to access any element in the list you must start at the head and traverse each element until you get to the one you are interested in. That may not sound that bad but think about the common case of building up a list. Its not uncommon to build lists by appending the new element to the end of the list. However, in a sigly linked list

this means that for every element you add to the end of the list you must traverse all elements in the list. This very quickly starts presenting performance issues. So how do we solve this problem. Instead of appending to the end of the list we just prepend to the head. That's just a single operation for every element we prepend to the front of the list. Well you might say, then your list is reversed when you are done! Well that is true, we can solve that problem by just reversing the list when we are done with it. It adds a bit of additional overhead but is still much, much more efficient then building the list by appending to the end. Take a look at code listing 14.4. In that listing we build the same list list in two very different ways. In the first example we build it in the inefficient way, by appending to the end of the list. In the second example, we build it in a much bore efficient way by prepending to the front of the list.

14.4 Inefficient and Efficient List Building

```
build_list_inefficient(0, Acc) ->
    Acc;
build_list_inefficient(Count, Acc) ->
    build_list_inefficient(Count - 1, Acc ++ [Count]).

build_list_efficient(0, Acc) ->
    lists:reverse(Acc)
build_list_efficient(Count, Acc) ->
    build_list_efficient(Count - 1, [Count | Acc]).
```

This concept that all list in Erlang are singly linked is important to keep in mind. Given this property it follows that when two lists are added together it can't be done in constant time. This is because one of the lists in the append operation must be walked from front to back and each element added one by one to the front of the other list. An example of where this knowledge would come in handy is when analyzing the following functions to determine which is best optimized.

Option number 1:

```
map(Fun, [H|T], Acc) -> map(Fun, T, Acc ++ [Fun(H)]);
map(_Fun, [], Acc)   -> Acc.
```

Or option number 2:

```
map(Fun, [H|T], Acc) -> map(Fun, T, [Fun(H)|Acc]);
map(_Fun, [], Acc)   -> [].
```

The subtle difference between number 1 and number 2 is that number 1 appends the value returned by the execution of Fun onto the end of the accumulator with ++ which is

inefficient both in terms of space and time. If the accumulator, Acc, was the list [2,3,4,5] and Fun(H) returns 1 then we have something like the following sequence of actions to join these lists.

```
[2|1]
[3|2,1]
[4|3,2,1]
[5|4,3,2,1]
```

The actual implementation would vary slightly but it would be of this general time complexity, 4 operations to join those two lists as well as the fact that a copy of each of the elements in the first list must be made to accomplish the append. map number 1 is even worse though because it does this exact same thing on each iteration. Number 2 is quite a bit more efficient. It only needs to do a simple constant time operation to append 1 element to the beginning of the accumulator each time through. An even worse way to implement number one would be to use lists:flatten. This function not only makes a copy of the first list but also of the second in order to produce an entirely new list. Avoid doing the following:

```
map(Fun, [H|T], Acc) -> map(Fun, T, lists:flatten([Acc, Fun(H)]));
map(_Fun, [], Acc)   -> Acc.
```

In the case above we would only be making the extra copy of the single element produced by Fun(H). Lists flatten in a scenario where the second list was long would have more of an impact. This means that when you want to bring lists together the ++ operator or the lists:append function should be used over the lists flatten operator which should only really be used in the case of deep lists that truly need to be flattened.

Now we have covered Erlang's primitive types. We have talked about each one in turn, including the caveates for their use and abuse. Now you should be able to make decisions obut when and how to use them in your code. We can now move on to higher level constructs knowing that we have a nice base to build on. The next piece we will layer on then will be the operators and built in functions (BIFs).

14.3.2 – Erlang's built in functions and operators

Erlang defines a number of operators and a number of built in functions (BIFs). These are generally used to manipulate primitive types and other basic functionality. Both operators and BIFs are implemented directly into the VM as C code which makes them generally quite efficient. That being the case, there are still a few performance implications and caveats to consider. In our discussion of lists and their properties we have already covered the ++ operator. We will start here then with the related operator; --

OPERATOR - -

The -- operator deletes the elements found in the right hand side list from those found in the left hand side list. The following example is pasted in from the shell.

```
1> [1,2,3] -- [1,2].
[3]
```

The issue with this function is that the performance is quite bad. Think about what you know about how lists are implemented and then reason about the performance of this operation in pretty much the worst case and you will most likely arrive at the right answer. According to the official Erlang documentation on this operator: "the '--' operator has a complexity proportional to the product of the length of its operands". If the second list is very short this function would not fair poorly basically O(n) but if both lists are long then the performance can be quite bad indeed. There are a variety of other means at your disposal to accomplish this subtraction all of which are referenced in the Erlang documentation online. From an operator to a BIF we now move to list_to_atom/1

LIST_TO_ATOM

Remember that atoms are never garbage collected so be careful when using this function. The usage of this function should not be on sets where the potential number of different lists being turned into atoms is large or infinite.

SIZE/1 VS LENGTH/1

This is a source of confusion as well as a topic we need to cover in terms of efficiency issues. Take a look at the shell session below to see how the size function is used.

```
2> size(<<"hello">>).
5
3> size({h,e,l,l,o}).
5
4> size("hello").
** exception error: bad argument
     in function  size/1
        called as size("hello")
```

The size function works nicely on tuples and binaries but throws an exception when called on lists. Length on the other hand is for use with lists and will throw an exception when used on either tuples or binaries.

```
5> length(<<"hello">>).
```

```
** exception error: bad argument
     in function  length/1
        called as length(<<"hello">>)
6> length({h,e,l,l,o}).
** exception error: bad argument
     in function  length/1
        called as length({h,e,l,l,o})
7> length("hello").
5
```

Why there is not a single function overloaded to do all of these things? We don't know but it probably has something to do with performance. Of late, we have seen a proliferation of new 'size' bifs. like tuple_size/1, byte_size/1 and bit_size/1 which allow the compiler to optimize further and offer up hints to tools like Dialyzer. One difference we are aware of is that all of the size family of functions operate on constant time, whereas the length function, efficient as it is being a built in function and therefore implemented in C right into the VM, operates in O(n) time complexity. It works by actually counting iteratively the number of elements in a list. When speed is a concern taking the length of very long lists may be a liability. That is it, as you can see operators and BIFs are relatively caveat and performance risk free, but not totally. We then close out the section on basic types, operators and built in functions and with that we are ready to take on some of the more complex entities in the Erlang language taxonomy with a move into the discussion of processes and functions each of which enclose the primitive types, operators, and BIFs that define the work given processes and functions can do.

14.4 – Processes and functions

Processes provide the fundamental execution environment of any Erlang system, even if you just write a library module with not a single spawn call, that code will be executed within a process. For that reason we will start this section with a discussion of processes.

14.4.1 - Processes

As we have learned from chapter 1 on, Processes are very cheap in Erlang. Spawning many, many concurrent execution environments is what Erlang is all about. With that said, what you choose to execute within each of those environments, and in some cases how you choose to tune that environment, makes a large impact on how efficiently you can spawn processes as well as how many you can spawn. While the actual time to spawn a process can be measured in microseconds the initialization of those functions are a different story. The start functions for a gen_server may takes quite a bit longer to finish than your average call to raw spawn due to initialization. When a gen_server or any of the other behaviours for that matter starts up the first thing it does is call the init/0 function in the callback module

specified. When the init/1 function returns it is then that the start or start_link function used to start the behaviour process returns and unblocks the caller. This is to enable deterministic startup of many processes such that when the start up function unblocks and a process id is returned that the process is ready and fully initialized. A string of dependent processes can be started up each being sure the processes it depends on are fully initialized. These properties are requisite for developing coherent and predictable concurrent systems. This start up and initialization is accomplished via a library called proc_lib. Below is an implementation that blocks the startup function until the init function, executed by the newly spawned process, has finished its execution.

Code Listing 14.4 – proc_lib example

```
-module(my_server).
-export([start_link/0]).
-export([init/1]).

start_link() ->
   proc_lib:start_link(my_server, init, [self()]).   #1

init(Parent) ->
    register(?MODULE, self()),
   proc_lib:init_ack(Parent, {ok, self()}),    #2
    loop().

loop() ->
    receive
        Msg ->
            handle_msg(Msg),
            loop()
    end.
```

At code annotation #1 proc_lib:start_link/3 is used to spawn a process. The function takes the arguments Module, Function and Arguments which indicate which function to spawn with and how to spawn that function. In this case my_server:init(self()) is called. Self is the process id of the calling process, the one that is to be linked to, the one that is doing the actual spawning. Passing this parent pid to the child process via the init function allows the code at code annotation #2 to function. After all the registration steps have been accomplished by the init/1 function the proc_lib:init_ack function is called to send a message back to the parent process indicating that the proc_lib:start_link or start function can unblock.

PROCESS LOOPS

As mentioned earlier processes are just functions that loop, which means loop recursively. This means that they better be properly tail recursive or the stack will grow to consume

all memory available and finally crash the system. Notice that the loop function in our code sample is properly tail recursive.

All behaviours have the basic functionality illustrated in code listing 15.1. This means that proc_lib is called whether you have need of an initialization step or not.

Often times in Erlang programming we see systems designed to make use of many processes coming and going rapidly. Take for example a proxy, or even closer to home a web server like the one we wrote in chapter 13. A process is spawned for every incoming http request. Under heavy load processes can be coming and going quite rapidly. When these conditions arise a trace will quickly make it apparent that a lot of time is spent in process initialization i.e. proc_lib. The lighter and more transient the processes being spawned the greater the proportion of time spent in proc_lib. Due to this when speed becomes a major concern and you are spawning many many temporary processes it can be helpful to stray away from OTP, away from proc_lib, and roll your own very light weight server consisting of a raw call to non-blocking spawn which spawns a process off directly around the process loop or even at times just around the single non looping function that needs to be executed quickly and concurrently.

With architectures where many processes are being spawned and dying quickly there is another rather tricky optimization that can be undertaken. This has to do with sizing the heap of each process to avoid any garbage collection or heap memory allocation subsequent to the process being spawned. Each process in Erlang has its own heap for storing variables and runtime data. The heap is grown as memory is required by a given process, it is also shrunk as unused memory exceeds the predefined default process heap size. The predefined process heap size is 233 words. You can see what the word size is on your system by running the following command on the shell:

```
1> erlang:system_info(wordsize).
4
```

On my system it is 4 bytes because I am on a 32-bit if I was running a 64-bit system the size would be 8 bytes. Given a 4 byte word size the default heap size on my system is 932 bytes and consequently when a process is spawned it grabs that amount of memory. Once a process on my system is spawned if it is required to store on its heap more than 932 bytes of memory the garbage collector may be required to run on the process heap to accommodate the additional storage requirements. This takes time, not a lot mind you, but it takes time and we are covering this since this chapter is about shaving off microseconds no matter how ugly the manner in which we do it. If you know about how much memory your rapidly coming and going processes need to store during their lifetime it is possible to size the default heap size using the +h option to erl itself, or by using

```
erlang:system_flag(min_heap_size, MinHeapSizeWords)
```

or even on a more granular level programmatically by passing

```
{min_heap_size, Words}
```

into the options argument list when spawning via the erlang:spawn_opt function. We have seen an example of when it is advisable to stray away from OTP and roll your own server. We have also covered a way to tune your system to take advantage of knowledge of the memory requirements of frequently spawned processes. We are now going to move from the processes to what gets executed within it; functions will be covered in the next section.

15.4.2 – Functions

Processes do all the work in Erlang systems. Erlang processes exist around functions. When the function ends the process spawned around it goes with it. Making the best and most efficient use of functions is critical to shaving milliseconds and microseconds off when that kind of time has become precious. We will start with a table showing the relative time signatures of the different ways functions can be invoked.

Function invocation type	Relative time
Local function call function()	Very fast
External function call module:function()	Almost as fast as local invocation
Fun invocation fun()	3 times more expensive than a local function
Applying an external function apply(Mod, Func, Args)	6 times more expensive than a local function

It is of course impossible to give absolute numbers because of the fact that the speed at which these different functions are invoked is entirely dependent on the hardware ERTS is running on.

When functions execute in clauses and pattern matching comes into play another technique is to place constants together so that the compiler can optimize the way it searches the clauses. Let me elaborate on that a little bit.

```
handle_message(Msg) when is_atom(Msg) -> handle_msg(atom_to_list(Msg));
handle_messages("stop")   -> do_stop();
handle_messages("go")     -> do_go();
handle_messages("report") -> do_report();
handle_messages("calc")   -> do_calculate().
```

The preceding function will be faster than the one that follows.

```
handle_messages("stop") -> do_stop();
handle_messages("go")   -> do_go();
handle_message(Msg) when is_atom(Msg) -> handle_msg(atom_to_list(Msg));
handle_messages("report") -> do_report();
handle_messages("calc")   -> do_calculate().
```

The reason the second example is slower than the first is because the first has all it's matchable patterns lined up in a row consecutively with no naked variables in the middle. The compiler can take all the known patterns and create a very efficient structure for use when pattern matching across them. The problem with the second example is it has a naked variable in the middle of the clauses. That variable can technically match any of the patterns that follow it as in: Msg = "report" and Msg = "calc". This constrains the compiler from reordering the clauses such that it can only create efficient structures to capture the matching of the "stop" and "go" patterns separate from the "report" and "calc" patterns. Other than this constraint you need not really worry about function clause matching. In the past it used to be the case that placing the least used clause last in a list of clauses was advisable, this is no longer the case because typically the compiler will reorder the clauses to be most efficient. While we are on the topic of myths lets talk about the _ pattern. As in:

```
function(WorkType, Time) -> caluculate_pay(WorkType, Time);
function(_, 0)           -> 0
```

I would advise that you never use it and instead use the _VariableName convention for unused variables as in the following:

```
function(WorkType, Time) -> caluculate_pay(WorkType, Time);
function(_WorkType, 0)   -> 0
```

You can even use just a variable, though it will generate a compiler warning. I highly advise that your code compile clean without warnings as usually those warnings are bugs that will only be discovered at run time. In any case the compiler can see when a variable is unused and it will optimize for that ensuring that no binding is done. So the very hard to read and understand _ can be replaced with the much more descriptive _VariableName with no loss of performance. With single function executions covered we can cover a bit about recursion which prompts many many single function executions in a short time all of which can add up if clauses are not designed correctly, pattern matching efficiency is not taken into account, and the recursion itself is not entirely optimal

There has been lots of talk about how body recursive functions are much slower than tail recursive ones. This is no longer true for the most part. Erlang versions long gone had this

property but the compiler and ERTS itself has evolved quite a bit since those days and now with many optimizations in place this is no longer the case. Body recursive functions are often times much more elegant and in some cases faster. Think about building up a list in order. The body recursive function will do that naturally whereas the tail recursive variant will have to reverse it at the end. The tail recursive version is also quite a bit more elegant. One thing that should be made known though is that tail recursive functions do operate in a constant memory footprint and so if memory is at a premium the tail recursive variety is generally better than the body recursive. In all cases with recursion style changes measuring is really the way to go. Performance characteristics have been shown to vary over tail vs body recursive functions on different hardware platforms. They certainly vary on other charicertistics like the number of recursions done and the structure of the clauses. And with that we end our coverage of all things, or at least many thing, performance related.

14.5 – Summary

The Erware team ran into some performance related issues with their systems due to a sharp rise in the traffic they were getting to their site. Realizing that performance tuning was required they followed a methodical 4 step process.

1. Determine performance goals
2. Establish a baseline
3. Profile code and refactor bottlenecks
4. Measure the results

In this chapter we covered the important Erlang developer related skills required to effectively execute on the methodology outlined above. We covered profiling tools so that we could profile our code and discover bottle necks and then we covered information needed to examine Erlang code with a critical eye so that bottle necks could be unblocked and performance goals met.

15 Make it Faster

"There is no such thing as fast, only fast enough"

With the last chapters focus on language to language communication we have covered near the last of what we need to cover to provide you the skills and knowledge you need to write solid production quality Erlang code. With this chapter we are going to cover the last but not least of the topics that will complete what we set out to teach you in this book; performance tuning and best practice.

I remember the first time I met Joe Armstrong, one of the creators of Erlang, back in 2002 at a functional programming conference here in the states. He was filled with tons great sayings and computer science maxims some original and some quoted many of which stuck with me. One of my favorites was "Make it work, then make it beautiful, then if you really, really have to, make it fast." but he went on to say "90% of the time if you make it beautiful it will already be fast, so really, just make it beautiful!" I relate to you this quote because I want you to go into this chapter with a real grasp that modifying your code for efficiency sake, and that alone, should be done as a last resort only after your code has been made beautiful and has still proven not to be fast enough. This chapter is about those 10% of cases when making it beautiful does not do the trick and you really need to get down to business. Some of the techniques in this chapter are going to be trivial in terms of speed improvement, ways of shaving of milliseconds, in some cases microsecond while other techniques will make a large impact in how fast your code will run. This chapter is broken up into a few high level sections.

1. Performance caveats and improvements related to primitive data types and BIFs.
2. Performance caveats and improvements related to processes and functions.
3. Profiling code.

15.1 – Performance related to primitive data types and BIFs

Starting with Erlangs' primitive types operators and built in functions makes sense in this discussion of performance. These are some of the most fundamental building blocks of your code and it is important to know how to deal with them in an efficient manner and understand any caveats associated with each. To get our discussion kicked off lets deal with the most basic of the basic and start with the primitive types.

15.1.1 – The primitive types

The table below lists the fundamental primitive types in Erlang along with their associated sizes.

Data Type	Size in Memory
Integer	1 word
Big Number	3 words up to all the memory available
Float	4 words on 32-bit architectures 3 words on 64-bit architectures
Atom	1 word for each + the atom text once per node. Never garbage collected
Pid	1 word to represent a local process. 5 words to represent a remote process.
Fun	9 to 13 words + size of environment.

The basic types of integers, floats, pids, and funs really don't need to come with any advice about how to use them. They are what they are. It can be helpful to know the size of each of these but only in pretty unusual circumstances.

BIG NUMBERS

Erlang allows for numbers of arbitrary length and precision. When numbers get too large to hold in the space prescribed by standard C datatypes on a given system they are stored as Big Numbers. To the Erlang programmer these Big Numbers look just like integers. The Erlang implementation of Big Nums is quite efficient and therefore you need not pay much attention to them. It is worthwhile to know that there is some cost associated with such numbers however.

ATOMS

Atoms present one of the biggest caveats in Erlang programming, one that bites almost all novice Erlang hackers at least once. Atoms are not garbage collected, meaning that once memory is allocated for an atom that memory can never de-allocated during the life of a given ERTS system. Couple this fact with the fact that registering a process requires an

atom name and you have a recipe for a memory leak. I can't count the number of times I have seen someone register processes that come and go as part of the life of a system. An example would be a system that handles packets of information from the outside world, one process for each packet, where each packet has a unique ID number which is used to register the process handling it. I have seen this many many times. I also personally spent many, many hours diagnosing my Erlang memory leak the first time I did this way back when. Atoms should only be used to represent finite sets of data. Atoms themselves however are very nicely implemented and are the data structure of choice for tagging values in message locally and across the wire. Moving on here we will remain within the realm of primitive types but we will start to talk about those types that serve to contain and aggregate the primitive types we just covered.

Erlang contains three fundamental examples of primitive types that can store or aggregate. The following table contains the size in memory that corresponds to each of those types.

Data Type	Size in Memory
Binary	3 to 6 words for the representation + data
Tuple	2 words for the representation + the size of each element
List	1 word per element + the size of each element

BINARY

The binary type is an aggregate type in that it stores or aggregates one or more of some more fundamental type. In this case bits and bytes. As with all binary types it is possible add to it, pattern match on it, and send it as an argument to a function or as a payload of a message. There are two basic types of binaries, small ones and large ones. The small ones, those up to 64 bytes are called "Heap Binaries" because they are stored on the process heap, pretty much as all other types are. It is the second type, "Reference-Counted Binaries", that have an interesting property that can be exploited for efficiency purposes. I am going to say right here, very clearly, that exploiting this property for efficiency is ugly and I hope you don't ever have to do it. Reference-counted binaries are not stored on the process heap and they can infact be shared between processes, hence the need to count references to them in order to allow for garbage collection. This means that if you have a large amount of data that you want to send from process to process on a particular node and that data is in the form of a binary all that needs to be sent from one process to another is send a small reference object from one process to the other. This of course does not work over the network.

Binaries can also offer a large opportunity for screwing up when working with them; or depending on how you look at it a large opportunity for performance improvement with a little refactoring. One way to quickly get an idea of whether or not you are using binaries efficiently is to compile your code after setting the following environment variable:

```
export ERL_COMPILER_OPTIONS=bin_opt_info
```

The bin_opt_info flag, which can also be added as a direct flag to erlc, elicits the compiler to print out quite a bit of data about how binaries are used within the code. This can give you some really good direction when looking to optimize usage. With binaries covered we move on to the rich topic of lists.

TUPLE

Tuples are refreshingly straightforward and efficient. They don't really have any performance caveats associate with them. They are not always a suitable data structure for every task as you know from having dealt with them rather often in this text but when they are they are quite efficient. Having said that we can move on to lists.

LIST

Lists present a rich field for performance improvements and for shooting ones self in the foot. We will cover just about all of them here so that you will be an expert after reading this section. Lists are structures that can only be accessed from their first element. There is no way to hold a reference to any element from the second to the end of the list. This means that right at the outset we have to be careful about how we build up lists. Because we have a pointer to the head of the list and can access it in constant time when we build up a list we need to add elements to the beginning of the list. Take the example below where we have a list called letters with two elements; the atoms y and x. To add to this list we must add to the front as we see below in the manner that we add z.

```
Letters = [y,x]
[z|Letters]
```

There is simply no other way to do it. Given this property it follows that when two lists are added together it can't be done in constant time. A list append operation takes O(n); in this case n is always equal to the size of one of the lists entirely. This is because one of the lists in the append operation must be walked from front to back and each element added one by one to the front of the other list. This does not mean that lists are slow or anything of the sort but it is wise to be aware of this property when absolute speed becomes a necessity. An

example of where this knowledge would come in handy is when analyzing the following functions to determine which is best optimized.

Option number 1:

```
map(Fun, [H|T], Acc) -> map(Fun, T, Acc ++ [Fun(H)]);
map(_Fun, [], Acc)   -> Acc.
```

Or option number 2:

```
map(Fun, [H|T], Acc) -> map(Fun, T, [Fun(H)|Acc]);
map(_Fun, [], Acc)   -> [].
```

The subtle difference between number 1 and number 2 is that number 1 appends the value returned by the execution of Fun onto the end of the accumulator with ++ which is inefficient both in terms of space and time. If the accumulator, Acc, was the list [2,3,4,5] and Fun(H) returns 1 then we have something like the following sequence of actions to join these lists.

```
[2|1]
[3|2,1]
[4|3,2,1]
[5|4,3,2,1]
```

The actual implementation would vary slightly but it would be of this general time complexity, 4 operations to join those two lists as well as the fact that a copy of each of the elements in the first list must be made to accomplish the append. map number 1 is even worse though because it does this exact same thing on each iteration. Number 2 is quite a bit more efficient. It only needs to do a simple constant time operation to append 1 element to the beginning of the accumulator each time through. An even worse way to implement number one would be to use lists:flatten. This function not only makes a copy of the first list but also of the second in order to produce an entirely new list. Avoid doing the following:

```
map(Fun, [H|T], Acc) -> map(Fun, T, lists:flatten([Acc, Fun(H)]));
map(_Fun, [], Acc)   -> Acc.
```

In the case above we would only be making the extra copy of the single element produced by Fun(H). Lists flatten in a scenario where the second list was long would have more of an impact. This means that when you want to bring lists together the ++ operator or the lists:append function should be used over the lists flatten operator which should only really be used in the case of deep lists that truly need to be flattened. With that we can say we are done with Erlangs' primitive types. We have covered all of the fundamental primitive types and can now move on to higher level constructs knowing that we have a nice base to

build on. The next piece we will layer on then will be the operators and built in functions (BIFs).

15.1.3 – Erlang's built in functions and operators

Erlang defines a number of operators and a number of built in functions (BIFs). These are generally used to manipulate primitive types and other basic functionality. Both operators and BIFs are implemented directly into the VM as C code which makes them generally quite efficient. Even still there are still a few performance implications and caveats to consider. In our discussion of lists and their properties we have already covered the ++ operator. We will start here then with a somewhat related operator; --

OPERATOR - -

The – operator deletes the elements found in the right hand side list from those found in the left hand side list. The following example is pasted in from the shell.

```
1> [1,2,3] -- [1,2].
[3]
```

The issue with this function is that the performance is quite bad. Think about what you know about how lists are implemented and then reason about the performance of this operation in pretty much the worst case and you will most likely arrive at the right answer. According to the official Erlang documentation on this operator: "the '--' operator has a complexity proportional to the product of the length of its operands". If the second list is very short this function would not fair poorly basically O(n) but if both lists are long then the performance can be quite bad indeed. There are a variety of other means at your disposal to accomplish this subtraction all of which are referenced in the Erlang documentation online. From an operator to a BIF we now move to list_to_atom/1

LIST_TO_ATOM

Remember that atoms are never garbage collected so be careful when using this function. The usage of this function should not be on sets where the potential number of different lists being turned into atoms is large or infinite. What if we wanted to find out how many of these non garbage collected atoms we had in a given list or tuple? We may want to use one of the functions we cover next in our discussion of the bifs size/1 and length/1.

SIZE/1 VS LENGTH/1

This is a source of confusion as well as a topic we need to cover in terms of efficiency issues. Take a look at the shell session below to see how the size function is used.

```
2> size(<<"hello">>).
5
3> size({h,e,l,l,o}).
5
4> size("hello").
** exception error: bad argument
     in function  size/1
        called as size("hello")
```

The size function as you can plainly see works nicely on tuples and binaries but throws an exception when called on lists. Length on the other hand is for use with lists and will throw an exception when used on either tuples or binaries.

```
5> length(<<"hello">>).
** exception error: bad argument
     in function  length/1
        called as length(<<"hello">>)
6> length({h,e,l,l,o}).
** exception error: bad argument
     in function  length/1
        called as length({h,e,l,l,o})
7> length("hello").
5
```

Why there is not a single function overloaded to do all of these things the world is not yet sure; in fact things have gone the other way, mostly in the name of efficiency with the introduction of the built in functions tuple_size/1, byte_size/1 and bit_size/1 which allow the compiler to optimize further and offer up hints to the Dialyzer. All of the size family of functions operate on constant time however whereas the length function, efficient as it is being a built in function and therefore implemented in C right into the VM, operates in O(n) time complexity basically having to count iteratively the number of elements in a list. When speed is a concern taking the length of very long lists may be a liability. That is it, as you can see operators and BIFs are relatively caveat and performance risk free, but not totally. We then close out the section on basic types, operators and built in functions and with that we are ready to take on some of the more complex entities in the Erlang language taxonomy with a move into the discussion of processes and functions each of which enclose the primitive types, operators, and BIFs that define the work given processes and functions can do.

15.2 – Processes and functions

Processes provide the fundamental execution environment of any Erlang system, even if you just write a library module with not a single spawn call, that code will be executed within a process. For that reason we will start this section with a discussion of processes.

15.2.1 - Processes

As we have learned from chapter 1 on, Processes are very cheap in Erlang. Spawning many, many concurrent execution environments is what Erlang is all about. With that said, what you choose to execute within each of those environments, and in some cases how you choose to tune that environment, makes a large impact on how efficient you can spawn processes as well as how many you can spawn. While the actual time to spawn a process can be measured in microseconds the start functions for a gen_server however, which will and should typically represent most of the processes in a well designed OTP system, takes quite a bit longer to unblock than your average call to raw spawn due to initialization. When a gen_server or any of the other behaviours for that matter starts up the first thing it does is call the init/0 function in the callback module specified. When the init/1 function returns it is then that the start or start_link function used to start the behaviour process returns and unblocks the caller. This is to enable deterministic startup of many processes such that when the start up function unblocks and a process id is returned that the process is ready and fully initialized. A string of dependent processes can be started up each being sure the processes it depends on are fully initialized. These properties are requisite for developing coherent and predictable concurrent systems. This start up and initialization is accomplished via a library called proc_lib. Below is an implementation that blocks the startup function until the init function, executed by the newly spawned process, has finished its execution.

Code Listing 15.1 – proc_lib example

```
-module(my_server).
-export([start_link/0]).
-export([init/1]).

start_link() ->
   proc_lib:start_link(my_server, init, [self()]).   #1

init(Parent) ->
    register(?MODULE, self()),
   proc_lib:init_ack(Parent, {ok, self()}),    #2
    loop().

loop() ->
```

```
receive
    Msg ->
        handle_msg(Msg),
        loop()
end.
```

At code annotation #1 proc_lib:start_link/3 is used to spawn a process. The function takes the arguments Module, Function and Arguments which direct it which function to spawn with. In this case my_server:init(self()) is called. Self is the process id of the calling process, the one that is to be linked to, the one that is doing the actual spawning. Passing this parent pid to the child process via the init function allows the code at code annotation #2 to function. After all the registration steps have been accomplished by the init/1 function the proc_lib:init_ack function is called to send a message back to the parent process indicating that the proc_lib:start_link or start function can unblock.

PROCESS LOOPS

As mentioned earlier processes are just functions that loop, which means loop recursively, which means they better be tail recursive or the stack will grow to consume all memory available and finally crash the system. Notice that the loop function in our code sample is definitely tail recursive.

All behaviours have the basic functionality illustrated in code listing 15.1. This means that proc_lib is called whether you have need of an initialization step or not.

Often times in Erlang programming we see systems designed to make use of many processes coming and going rapidly. Take for example a proxy, or even closer to home a web server like the one we wrote in chapter 13. A process is spawned for every incoming http request. Under heavy load processes can be coming and going quite rapidly. When these conditions arise a trace will quickly make it apparent that a lot of time is spent in process initialization i.e. proc_lib. The lighter and more transient the processes being spawned the greater the proportion of time spent in proc_lib. Due to this when speed becomes a major concern and you are spawning many many temporary processes it can be helpful to stray away from OTP, away from proc_lib, and roll your own very light weight server consisting of a raw call to non-blocking spawn which spawns a process off directly around the process loop or even at times just around the single non looping function that needs to be executed quickly and concurrently.

With architectures where many processes are being spawned and dying quickly there is another rather tricky optimization that can be undertaken. This has to do with sizing the heap of each process to avoid any garbage collection or heap memory allocation subsequent to the process being spawned. Each process in Erlang has its own heap for storing variables

and runtime data. The heap is grown as memory is required by a given process, it is also shrunk as unused memory exceeds the predefined default process heap size. The predefined process heap size is 233 words. You can see what the word size is on your system by running the following command on the shell:

```
1> erlang:system_info(wordsize).
4
```

On my system it is 4 bytes because I am on a 32-bit if I was running a 64-bit system the size would be 8 bytes. Given a 4 word size the default heap size on my system is 932 bytes and consequently when a process is spawned it grabs that amount of memory. Once a process on my system is spawned if it is required to store on its heap more than 932 bytes of memory the garbage collector will be required to augment the process heap to accommodate the additional storage requirements. Similarly if the storage requirements later shrink then the garbage collector will most likely need to shrink the heap size. This takes time, not a lot mind you, but it takes time and we are covering this since this chapter is about shaving off microseconds no matter how ugly the manner in which we do it. If you know about how much memory your rapidly coming and going processes need to store during their lifetime it is possible to size the default heap size using the +h option to erl itself, or by using

```
erlang:system_flag(min_heap_size, MinHeapSizeWords)
```

or even on a more granular level programmatically by passing

```
{min_heap_size, Words}
```

into the options argument list when spawning via the erlang:spawn_opt function. We have seen an example of when it is advisable to stray away from OTP and roll your own server. We have also covered a way to tune your system to take advantage of knowledge of the memory requirements of frequently spawned processes. We are now going to move from the processes to what gets executed within it; functions will be covered in the next section.

15.2.2 – Functions

Processes do all the work in Erlang systems. Erlang processes exist around functions. When the function ends the process spawned around it goes with it. Making the best and most efficient use of functions is critical to shaving milliseconds and microseconds off when that kind of time has become precious. We will start with a table showing the relative time signatures of the different ways functions can be invoked.

Function invocation type	Relative time
Local function call function()	Very fast
External function call module:function()	Almost as fast as local invocation
Fun invocation fun()	3 times more expensive than a local function
Applying an external function apply(Mod, Func, Args)	6 times more expensive than a local function

It is of course impossible to give absolute numbers because of the fact that the speed at which these different functions are invoked is entirely dependent on the hardware ERTS is running on.

When functions execute in clauses and pattern matching comes into play another technique is to place constants together so that the compiler can optimize the way it searches the clauses. Let me elaborate on that a little bit.

```
handle_message(Msg) when is_atom(Msg) -> handle_msg(atom_to_list(Msg));
handle_messages("stop")   -> do_stop();
handle_messages("go")     -> do_go();
handle_messages("report") -> do_report();
handle_messages("calc")   -> do_calculate().
```

The preceding function will be faster than the one that follows.

```
handle_messages("stop") -> do_stop();
handle_messages("go")   -> do_go();
handle_message(Msg) when is_atom(Msg) -> handle_msg(atom_to_list(Msg));
handle_messages("report") -> do_report();
handle_messages("calc")   -> do_calculate().
```

The reason the second example is slower than the first is because the first has all it's matchable patterns lined up in a row consecutively with no naked variables in the middle. The compiler can take all the known patterns and create a very efficient structure for use when pattern matching across them. The problem with the second example is it has a naked variable in the middle of the clauses. That variable can technically match any of the patterns that follow it as in: Msg = "report" and Msg = "calc". This constrains the compiler from reordering the clauses such that it can only create efficient structures to capture the matching of the "stop" and "go" patterns separate from the "report" and "calc" patterns. Other than this constraint you need not really worry about function clause matching. In the past it used to be the case that placing the least used clause last in a list of clauses was advisable, this is no longer the case because typically the compiler will reorder the clauses to be most efficient. While we are on the topic of myths lets talk about the _ pattern. As in:

```
function(WorkType, Time) -> caluculate_pay(WorkType, Time);
```

```
function(_, 0)            -> 0
```

I would advise that you never use it and instead use the _VariableName convention for unused variables as in the following:

```
function(WorkType, Time) -> caluculate_pay(WorkType, Time);
function(_WorkType, 0)   -> 0
```

You can even use just a variable, though it will generate a compiler warning. I highly advise that your code compile clean without warnings as usually those warnings are bugs that will only be discovered at run time. In any case the compiler can see when a variable is unused and it will optimize for that ensuring that no binding is done. So the very hard to read and understand _ can be replaced with the much more descriptive _VariableName with no loss of performance. With single function executions covered we can cover a bit about recursion which prompts many many single function executions in a short time all of which can add up if clauses are not designed correctly, pattern matching efficiency is not taken into account, and the recursion itself is not entirely optimal

There has been lots of talk about how body recursive functions are much slower than tail recursive ones. This is no longer true for the most part. Erlang versions long gone had this property but the compiler and ERTS itself has evolved quite a bit since those days and now with many optimizations in place this is no longer the case. Body recursive functions are often times much more elegant and in some cases faster. Think about building up a list in order. The body recursive function will do that naturally whereas the tail recursive variant will have to reverse it at the end. The tail recursive version is also quite a bit more elegant. One thing that should be made known though is that tail recursive functions do operate in a constant memory footprint and so if memory is at a premium the tail recursive variety is generally better than the body recursive. In all cases with recursion style changes measuring is really the way to go. Performance characteristics have been shown to vary over tail vs body recursive functions on different hardware platforms. They certainly vary on other charicertistics like the number of recursions done and the structure of the clauses. To really find out what is better measure performance with one type and measure performance with the other. With measurement you will truly be able to determine what works best when it comes to the recursive style of your functions. This is a nice lead in to the last section of this chapter which covers exactly how to measure i.e. performance profile an Erlang system.

15.3 – How to profile Erlang code

Profiling and measuring is the most reliable method for determining where performance bottle necks are in your system. With practice you can learn to spot the big gotchas and glaring performance pitfalls in Erlang code, you will also be able to spot many quick wins, but at the end of the day tracking down most performance bottlenecks requires profiling no matter how good you are. Erlang/OTP puts a number of profiling tools at your disposal, we will cover two major ones here; fprof and cprof. We will first dive into fprof for no other reason than we believe it to be one of the highest value of all the performance measureing tools shipped with Erlang/OTP.

15.3.1 – profiling with fprof

fprof is the successor to eprof. eprof had more runtime impact than does fprof, though that is certainly not to say that fprof does not have a noticeable runtime impact. eprof also had less functionality than fprof but is still included in the standard Erlang/OTP distro and still documented. Both systems are built using Erlangs trace functionality. We say that fprof is high value because it gives you a wealth of useful information, in a digestible format, and is quite easy to use. This is not to say that reading and interpreting the results is not tricky at times; it can be. In this section we will run through in detail how to interpret the results of an fprof trace. All in all fprof is a nice go to tool when performance issues first need to be diagnosed.

RUNTIME_TOOLS APPLICATION REQUIRED

fprof depends on dbg which is part of the runtime_tools application. This must be present on your system in order to use fprof. runtime_tools comes with Erlang/OTP but if you are installing via alternate means you may need to verify you installed this application package.

Below in code listing 15.2 is a very simple little module that executes some ineffectual code. The point of the execution is that we can profile it.

Code Listing 15.2 – fprof example code

```
-module(profile_ex).

%% API
-export([run/0, run_with_fprof/0]).

run_with_fprof() ->
    fprof:trace(start),     #1
      run(),
    fprof:trace(stop),   #2
    fprof:profile(),     #3
```

fprof:analyse({dest, atom_to_list(?MODULE)}). #4

```
run() ->
    spawn(fun() -> looper(1000) end),
    spawn(fun() -> funner(1000) end).

looper(0) ->
    ok;
looper(N) ->
    integer_to_list(N),
    looper(N - 1).

funner(N) ->
    funner(fun(N_) -> integer_to_list(N_) end, N).

funner(_Fun, 0) ->
    ok;
funner(Fun, N) ->
    Fun(N),
    funner(Fun, N - 1).
```

Executing the run_with_fprof/1 function will create a file called "profile_ex" in the working directory. Before we analyze the contents of that file we will explain quickly how it is generated. At code annotation #1 the fprof server is started and tracing commences. At annotation #2 we stop the server and clear the tracing data from the node. At annotation #3 profile/0 is called to take all the trace data accumulated in the fprof server from the function call tracing and turn it into raw profiling data. The results of this compilation step are then taken and analyzed by the fprof:analyse/1 function, at code annotation #4, and finally written out to a file, in human readable format, at the destination specified by the {dest, <filename> option. In this case we name the output file the same as the module name. There variations on how to start a trace and arrive finally at analyzed human readable output. We leave the study of those options as an exercise for the reader at a later date. The method for generation is not nearly as important as the ability to analyze the file which is what we will get into next.

What follows are a number salient snippets from analyzed profile data residing in the profile_ex file. We will discuss these snippets explaining what they mean and through the discussion walk you through how to analyze performance with fprof.

```
%                                                CNT       ACC       OWN
[{ totals,                                      4689,   78.590,   78.552}].
```

The two lines above are the first profiling lines in the fprof file and are found pretty much at the very top of the file. These number represent, as indicated by the heading "totals" the sums of the various things that fprof tallies. The column headed "CNT" is the total number of function calls that occurred from the start of tracing to the end. "ACC" represents the total amount of accumulated time of all function calls from start to finish. "OWN" represents

an addition of the total time spent in individual functions not including time spent in functions called subsequently from within a given function. These times can differ significantly if there is a lot of other code running that is not under profile. This is due to context switching in the VM. Here the only thing the VM is really doing is running our code and so these times are very much the same. Moving on to the next important section of the output file we see what follows.

```
%                                                    CNT       ACC       OWN
[{ "<0.47.0>",                                     2666,undefined,    44.376},
%%
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.1.118133018>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-1-',0}]}].
```

What we are looking at is the section heading for all the profile data for one of the two processes we spawned in the run function. Another process group heading can be found further down in the trace file. The headers continue to signify the same things here. We have a CNT of 2666 function calls. OWN adds up to 44.376 milliseconds. With a quick search we can find the heading for our second spawned process a little bit further down in the file.

```
%                                                    CNT       ACC       OWN
[{ "<0.46.0>",                                     2011,undefined, 33.924},
%%
 { spawned_by, "<0.38.0>"},
 { spawned_as, {erlang,apply,["#Fun<profile_ex.0.9423607>",[]]}},
 { initial_calls, [{erlang,apply,2},{profile_ex,'-run/0-fun-0-',0}]}].
```

Actually looking at the Pids for these processes we can ascertain from the numbers, <0.46.0> vs <0.47.0> that the second process in the file was actually the first one spawned. Adding up the CNT and the OWN from this header and the header representing our second process yields just about the totals shown in the initial file level header. The small bit missing comes from the third process we find at the bottom of our trace file which is actually spawned by fprof itself. Note that in both cases ACC is undefined, this is because it is not something that makes sense to define for a process. ACC will come into play next as we get into the function call stack trace figures. So far from looking at our trace file we can see that two application processes were spawned and that <0.47.0> took up 44.376 with a count of 2666 while <0.46.0> rated 33.924 with a count of 2011. Lets dive down and find out why process <0.47.0> was longer running.

Looking down into the meat of this trace file it is plain to see that interpreting the results can be a little bit confusing. Things will not always seem to add up, in most cases they do,

but in some cases they actually will not. Tallying ACC times for example is a tricky operation particularly when the function exists as part of a complex mutually recursive operation. Do not get hung up on this. Another caveat is that on some systems which don't allow for very granular CPU time measurements when the actual ERTS system is subjected to OS scheduling. When this happens it can appear that a function burns more time than it should even when doing nothing. If you see something that looks really unreasonable the best check would be to run the profile again and compare the results. This phenomenon does not in general prove to be too troublesome. In general reading these files is a little bit art and a little bit science. Don't get hung up on trying to track down every microsecond. Take in the whole file and you will be able to quickly zero in on where time is being spent.

An easy first place to start the investigation of profile analysis data is to take a look at suspension and the garbage collection times. Comparing these will be able to tell us if the difference in the time spent in one process vs. the other was due to either suspension or garbage collection. Processes can be suspended due to a variety of reasons such as for a call to yield, entering a receive block, timing, waiting for io, etc… Suspend, which is shown as a function for display purposes but is not actually a call to a function, always shows an OWN time of 0 so as to avoid counting this time in the overall time spent in a process. Looking at suspend for both processes we can see that they are fairly equivalent.

for <0.47.0>

```
{[{{erlang,apply,2},                          1,   34.035,    0.000},
  {{profile_ex,funner,2},                       1,    0.037,    0.000}],
 { suspend,                                   2,   34.072,    0.000},  %
 [ ]}.
```

and for <0.46.0>

```
{[{{profile_ex,looper,1},                       1,   34.699,    0.000},
  {{erlang,apply,2},                          1,    0.188,    0.000}],
 { suspend,                                   2,   34.887,    0.000},  %
 [ ]}.
```

You may ask how we know the clauses above relate to suspend itself and not to one of the other functions listed in the stanza. The function focused on by a stanza is indicated by a % sign after that function. The above functions are information about the functions that called the function in focus and ay functions below which there are not in this case would be functions that the function in focus subsequently called. In this case you can see an empty list due to the fact that suspend called nothing else of course. Garbage collection typically happens during suspension and it is also one of those pseudo function calls. Our processes don't actually call garbage collect. Looking at the time spent garbage collecting between

both processes we can see that it is roughly equivalent and not a cause for the difference in execution times.

for <0.47.0>

```
{[{{profile_ex,'-funner/1-fun-0-',1},              6,    0.214,    0.214}],
 { garbage_collect,                              6,    0.214,    0.214},  %
 [ ]}.
```

and for <0.46.0>

```
{[{{profile_ex,looper,1},                          6,    0.174,    0.174}],
 { garbage_collect,                              6,    0.174,    0.174},   %
 [ ]}.
```

Ruling those two things out we move on to look at the function calls and operations actually performed in each and where the time came from. In <0.47.0> the next line down from the process header information is the following:

```
{[{undefined,                                    1,   78.448,    0.009}],
 { {erlang,apply,2},                             1,   78.448,    0.009},   %
 [{{profile_ex,'-run/0-fun-1-',0},                 1,   44.404,    0.001},
  {suspend,                                      1,   34.035,    0.000}]}.
```

The undefined at the top only means that the function that called into what was traced was called before the tracing actually started. We can ignore it. Below that we see erlang apply/2 which is again not something that we care about, moving down to the third line we see the fun that our process was spawned around. We will start walking down the stack there. The next stanza down contains the % sign after {profile_ex,'-run/0-fun-1-',0} which indicates that the stanza focuses on that function.

```
{[{{erlang,apply,2},                               1,   44.404,    0.001}],
 { {profile_ex,'-run/0-fun-1-',0},                   1,   44.404,    0.001},
%
 [{{profile_ex,funner,1},                           1,   44.403,
0.002}]}.
```

Nothing too interesting there in that the function spent very little time in OWN. What the stanza does indicate is that from here the next function called was funner.

```
{[{{profile_ex,'-run/0-fun-1-',0},                 1,   44.403,    0.002}],
 { {profile_ex,funner,1},                          1,   44.403,    0.002},
%
 [{{profile_ex,funner,2},                          1,   44.401,    0.013}]}.
```

Similarly here we have very little time spent in own but we see that funner/1 calls funner/2 which is where we will go next. Looking at our code we can see that funner/2 is where this process should spend most of it's time.

```
{[{{profile_ex,funner,1},                          1,   44.401,    0.013},
  {{profile_ex,funner,2},                        884,    0.000,
17.443}],
 { {profile_ex,funner,2},                        885,   44.401,   17.456},
%
 [{{profile_ex,'-funner/1-fun-0-',1},            885,   26.908,   18.194},
  {suspend,                                      1,    0.037,    0.000},
  {{profile_ex,funner,2},                        884,    0.000,
17.443}]}.
```

Now we are getting somewhere, we have found a function that takes up a considerable amount of time in OWN. Digging down just a bit from here it is easy to see upon comparison with our other process that the addition of the fun in the funner process is where the extra time is being spent. Because integer to list occurs in both processes.

<0.47.0>

```
{[{{profile_ex,'-funner/1-fun-0-',1},          885,    8.500,    8.500}],
 { {erlang,integer_to_list,1},                885,    8.500,    8.500},     %
 [ ]}.
```

<0.46.0>

```
{[{{profile_ex,looper,1},                     1000,    9.505,    9.505}],
 { {erlang,integer_to_list,1},               1000,    9.505,    9.505},   %
 [ ]}.
```

Looking at the code it is easy to see that the funner process will be more time consuming than the looper, after all it is easy to see that we added a fun for the extra overhead. The important thing is through proving something obvious with fprof you have gained an understanding of how to read these files and determine where time is being spent is processing. There is one other interesting thing about the traces I showed you here that you may be wondering and that is why is the count showing us how many times we called integer_to_list/1 in process <0.47.0> 885 and the count in <0.46.0> 1000 as we would expect from looking at the code. Erlang is concurrent, it is important to keep that in mind, this is not a sequential system we are working with. When profiling, when doing anything with Erlang, remember it is all about concurrency. Take a look at how this example code was started:

```
run_with_fprof() ->
    fprof:trace(start),
   run(),    #1
    fprof:trace(stop),
    fprof:profile(),
    fprof:analyse({dest, atom_to_list(?MODULE)}).


run() ->
    spawn(fun() -> looper(1000) end),
    spawn(fun() -> funner(1000) end).
```

The function run/1 is called at code annotation #1 which causes looper/1 and funner/1 to be spawned off and then we immediately proceed to call fprof:trace(stop) which shuts down our profiling, even before the funner process has had a chance to finish it's 1000 iterations. At 885 iterations we stop watching and generate our output. To prove this to yourself run the same code with the addition of a sleep call as pictured in the next example.

```
run_with_fprof() ->
    fprof:trace(start),
    run(),
    timer:sleep(5000),
    fprof:trace(stop),
    fprof:profile(),
    fprof:analyse({dest, atom_to_list(?MODULE)}).
```

Now in the output you can see that 1000 calls to integer_to_list/1 were made within funner! Fprof is an invaluable tool. It provides a simple programmatic interface to measuring time spent in functions that spawn multiple modules and processes. It's output can be a little bit tricky to read but at the end of the day it is providing a wealth of useful data cheaply. The next tool, cprof, does not provide the same depth of information that fprof does but it is truly simple and easy to use as well as versatile.

15.3.2 – counting with cprof

cprof is a very simple tool for counting call counts. It has the very desirable property of very minimal runtime performance impact. It is the most suitable of the tools we will cover here for doing analysis in production should that be required. Since we are already familiar with the code and the operation of the module contained within profile_ex.erl we will use profile_ex again for demonstrating the functionality of cprof. The shell session shown in code listing 15.3 demonstrates the simplest way to use cprof.

Code listing 15.3 – demonstrating cprof

```
Eshell V5.7.2  (abort with ^G)
```

```
1> l(profile_ex).
{module,profile_ex}
 2> cprof:start().   #1
5359
3> profile_ex:run().
<0.43.0>
 4> cprof:pause().   #2
5380
 5> cprof:analyse(profile_ex).   #3
{profile_ex,3007,
          [{{profile_ex,looper,1},1001},
           {{profile_ex,funner,2},1001},
           {{profile_ex,'-funner/1-fun-0-',1},1000},
           {{profile_ex,run,0},1},
           {{profile_ex,module_info,0},1},
           {{profile_ex,funner,1},1},
           {{profile_ex,'-run/0-fun-1-',0},1},
           {{profile_ex,'-run/0-fun-0-',0},1}]}
6> cprof:stop().
5380
7>
```

I really like this tool because it is so simple and easy to use. At code annotation #1 cprof is started. The number returned represents the number of functions that received call break points. That is how cprof does it's magic. It uses very non intrusive beam breakpoints to count function call invocations. This means that no special compilation is required. Instead of just calling start here we could also have opted to have been more specific and just set break points for the profile_ex module as in

```
2> cprof:start(profile_ex).
10
```

This is advisable for further limiting the impact of the analysis. Either way the next step we take at code annotation #2 is to pause cprof causing it to stop counting what ever it is that we were measuring which in this case was the call to profile_ex:run/1. Code annotation #3 is the call to analysis which dumps the output to the terminal as fairly readable erlang term. We can see from our term that the results are much as expected. We see the full 1000 calls to both looper and funner. Notice though that the integer_to_list/1 calls don't show up. The reason for that is that because cprof uses breakpoints to collect counts only beam code can be evaluated. BIFs are built right into the emulator, are not beam code, and therefore can't be counted. cprof is a simple, easy way to find out what functions are being called and how many times, nothing more nothing less. There are other tools to look at, such as cover for code coverage and the instrument module for doing memory usage analysis but in general their usage for doing performance tuning is less common. With these two tools in your arsenal you have a really nice footing for doing some very powerful analysis on your systems which is what will allow you to optimize the way your processes, functions, types and operators are used which at the end of the day allows you to measure what really works and what does not. Knowing what works and what does not out of a list of options for

implementation is the best practical tool you have for shaving off microseconds, milliseconds, and depending on the situation even minutes or hours off execution times.

15.4 – Summary

In this chapter we have covered the properties and caveats surrounding the Erlang/OTP system from basic data types to complex modules and processes. This knowledge gives you the power to program cleanly from the get go, optimize usage for speed when you need to, and measure what you have done to prove that it works. There is more information to be read about this topic within the Erlang/OTP documentation itself. We suggest that you do take the time to cover those docs. In either case though at this point with this coverage of efficiency you can really start to call yourself a well rounded Erlang programmer with the necessary tools to build and just as importantly maintain real Erlang production systems. Well done.

Appendix A

Installing Erlang

There are several ways of installing Erlang, depending on your operating system and your personal preferences. Erlang currently runs on modern versions of Windows, on Mac OS X, Linux, and most other Unix-like operating systems, and on the VxWorks real-time operating system.

A.1 – Installing Erlang on Windows

If your operating system is Windows, just download and run the latest installer `.exe` file from `www.erlang.org/download.html`. This includes documentation, and sets up menus and icons for starting the special `werl` shell on Windows (see Section 2.1.1).

A.2 – Installing Erlang on Mac OS X, Linux, or other Unix-like systems

On Unix-like systems, you may build and install Erlang from source code, which is the best way to ensure you have the latest version, or, on some systems, you can use a package manager such as Ubuntu's `synaptic` to automatically download and install the official Erlang package for your system—but note that this might not be the latest release of Erlang.

A.2.1 – Compiling from source

Point your browser at `www.erlang.org/download.html` and get the latest source package. Once downloaded, un-`tar` the package, `cd` into the directory and run the `./configure` script. It is possible to add a `--prefix=…` flag if you would like to install to a location other than the default, which is `/usr/local/lib/erlang`. For example:

```
./configure --prefix=/home/jdoe/lib
```

Once the code is configured properly (but see below if it doesn't), just run

```
make
```

and then

```
make install
```

Note that you will probably need to do the install step with root privileges, if you are installing to the default location. On many systems these days that means running it as

```
sudo make install
```

After installing, it should be possible for you to just enter "`erl`" to start Erlang, or "`erlc`" to run the Erlang compiler.

A.2.2 – Resolving configuration problems

To compile Erlang from source, some libraries and tools must be installed already on your system. Some of the more common things that may not be installed by default are:

- A fully working GCC compiler environment
- Ncurses development libraries

Try installing any missing packages and run configure again, before you run `make`. (The exact package names may vary with your system.)

Some libraries, if missing, will only cause a warning in the configure step that some Erlang applications will not be built. If you do not need these applications, you can go ahead and run `make`, but otherwise you need to install the missing packages and reconfigure. Some typical examples are:

- OpenSSL development libraries
- ODBC development libraries
- Java

If you are running with some applications disabled and later find out that you need one of them, you can just do the ./`configure`, `make`, `make install` again after getting the missing packages.

A.4 – Installing Erlang using the Faxien package manager

Using Faxien is quite a bit faster than installing from source. Faxien is a package management system for Erlang, much like Gem for Ruby. It works by searching the online community package repositories for whatever it may be that you are asking for; in this case it is the Erlang virtual machine emulator `erl` itself.

Start by pointing your browser at the Erlware open source community website `www.erlware.org/downloads.html`, and download the Faxien universal launcher. Once you have the launcher, which is a small Python script, make it executable and run it. If you prefer not to install to the default location which is `/usr/local/erlware`, you can supply a prefix to the installer as follows:

```
faxien-launcher-universal-3.4.py /home/jdoe/
```

Doing so will install Faxien itself as well as any subsequent packages installed using it into `/home/jdoe/erlware`.

Once Faxien is installed, you can just run

```
erlware/bin/faxien install-release erl
```

When that is finished, both the `erl` virtual machine emulator and the `erlc` compiler will be in the `erlware/bin` directory for you to use.

Appendix B
Lists and referential transparency

So, what is the *point* of Erlang's list data type, you may ask (and in particular if you are used to languages like Java, Python, et cetera, where you work a lot with arrays, buffers, and whatnot. For every element you also need a pointer to the next one, using valuable memory, and you can't even add elements on the right-hand side of the list! Pah!

This is all true, but in Erlang, you are never allowed to modify the value of something if it could cause someone else to get his data changed behind his back. This is the main thought behind the fancy words *referential transparency.*

B.1 – A definition of referential transparency

Referential transparency is really a simple concept. It all boils down to this:

> If you get hold of a value (a "term"), and give it a name (let's say "X") to keep track of it for the time being, then you are *guaranteed* that X remains unchanged no matter what, even if you pass a reference to X to some other part of the program. In other words, values kept in variables (or parts of those values) are never changed behind your back.

B.2 – The advantages of referential transparency

The same reasoning is behind why strings are constant in Java: you don't want to be embarrassed by printing something rude instead of your intended "More tea, Vicar?" just

because you happened to pass a reference to that string to another, rather naughty function, before you went on and printed the string.

On a more serious note, there are several reasons why this is done for *all* data in Erlang:

- It makes programming a *lot* less error prone – a very important property when you have million-line projects with dozens or even hundreds of programmers involved.
- Things that used to work fine when you were running the code in a single process don't suddenly need rewriting if you want to divide the work over two or more processes: there can be no "covert channels" between different parts of the program that stop working when you split the code into separate processes (perhaps running on multiple machines).
- It allows the system to do creative things behind the scenes with respect to memory management and multithreading, since it knows that there will be no write accesses to already existing data structures.

Hence, referential transparency is not merely a nice property enjoyed by people of a theoretical persuasion – it has important consequences for the stability and scalability of your program, not to mention for readability, debuggability, and speed of development.

B.3 – The connection to lists

Getting back to lists, this referential transparency guarantee means that if you already have a list (maybe you got it as a an argument to function), you cannot possibly add elements to the end of that list, because if you did, then anybody else who still had a reference to it would discover that an extra last element has materialized from nowhere. That's not allowed in Erlang.

However, adding "to the left" (using list cells) is no problem, because the original list is never disturbed—we have simply created a new cell that points to the first cell of the original list, saying "I'm like that list over there, but with this new element here added to the head".

Hence, list cells are an elegant solution to the problem of growing lists dynamically in a referentially transparent system, and furthermore, the implementation is so simple that cells work very efficiently indeed: many extremely clever people have tried (for decades) to come up with a solution that would allow both "adding to the end" as well as lower memory usage, and at the same time preserving the transparency property, but those solutions have tended to be A) very complicated indeed, and B) slower.