

Erlang AND OTP IN ACTION

Martin Logan
Eric Merritt
Richard Carlsson

FOREWORD BY ULF WIGER





Erlang and OTP in Action

by Martin Logan
Eric Merritt
Richard Carlsson

Chapter 1

Copyright 2011 Manning Publications

brief contents

PART 1 GETTING PAST PURE ERLANG: THE OTP BASICS1

- 1 ■ The Erlang/OTP platform 3
- 2 ■ Erlang language essentials 22
- 3 ■ Writing a TCP-based RPC service 94
- 4 ■ OTP applications and supervision 119
- 5 ■ Using the main graphical introspection tools 132

PART 2 BUILDING A PRODUCTION SYSTEM147

- 6 ■ Implementing a caching system 149
- 7 ■ Logging and event handling
the Erlang/OTP way 170
- 8 ■ Introducing distributed Erlang/OTP 190
- 9 ■ Adding distribution to the cache with Mnesia 213
- 10 ■ Packaging, services, and deployment 242

PART 3 INTEGRATING AND REFINING.....259

- 11 ■ Adding an HTTP interface to the cache 261
- 12 ■ Integrating with foreign code using
ports and NIFs 291
- 13 ■ Communication between Erlang and Java
via Jinterface 332
- 14 ■ Optimization and performance 357
- 15 ■ Installing Erlang 379
- 16 ■ Lists and referential transparency 381

The Erlang/OTP platform



This chapter covers

- Understanding concurrency and Erlang's process model
- Erlang's support for fault tolerance and distribution
- Important properties of the Erlang runtime system
- What functional programming means, and how it applies to Erlang

If you're reading this book, you probably know already that Erlang is a programming language—and as such it's pretty interesting in itself—but as the title of the book indicates, our focus is on the practical use of Erlang for creating real, live systems. And for that, we also need the OTP framework. This is always included in any Erlang distribution and is such an integral part of Erlang these days that it's hard to say where the line is drawn between OTP and the plain Erlang standard libraries; hence, we often say “Erlang/OTP” to refer to either or both. Despite this close relationship, not many Erlang programmers have a clear idea of what OTP can provide

What does OTP stand for?

OTP was originally an acronym for Open Telecom Platform, a bit of a branding attempt from the time before Erlang went open source. But few people care about that now; these days, it's just OTP. Nothing in either Erlang or OTP is specific to telecom applications: a more fitting name might have been Concurrent Systems Platform.

or how to start using it, even if it has always been just a few keystrokes away. This book is here to help.

The Erlang programming language is already fairly well known for making it easy to write highly parallel, distributed, and fault-tolerant systems, and we give a comprehensive overview of the language in chapter 2 before we jump into the workings of the OTP framework. But why should you learn to use OTP, when you could happily hack away, rolling your own solutions as you go? These are some of the main advantages of OTP:

- *Productivity*—Using OTP makes it possible to produce production-quality systems in a very short time.
- *Stability*—Code written on top of OTP can focus on the logic and avoid error-prone reimplementations of the typical things that every real-world system needs: process management, servers, state machines, and so on.
- *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 is rock solid and has been thoroughly battle tested.

Despite these advantages, it's 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'd like to change this. This is, to our knowledge, the first book focused on learning to use OTP, and we want to show you that it can be a much easier experience than you may think. We're sure you won't regret it.

At the end of this book, you'll have a thorough knowledge of the concepts, libraries, and programming patterns that make up the OTP framework. You'll understand how individual programs and whole Erlang-based systems can be structured using OTP components and principles in order to be fault tolerant, distributable, concurrent, efficient, and easy to control and monitor. You'll probably also have picked up a number of details about the Erlang language, its runtime system, and some of the libraries and tools around it that you weren't already aware of.

In this chapter, we discuss the core concepts and features of the Erlang/OTP platform that everything else in OTP builds on:

- Concurrent programming
- Fault tolerance
- Distributed programming
- The Erlang virtual machine and runtime system
- Erlang's core functional language

The point is to get you acquainted with the thinking behind all the concrete stuff we dive into from chapters 2 and 3 onward, rather than starting by handing you a bunch of facts up front. Erlang is different, and many of the things you'll see in this book will take some time to get accustomed to. With this chapter, we hope to give you an idea of why things work the way they do, before we get into technical details.

1.1 Concurrent programming with processes

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 you to create fault-tolerant architectures and fully utilize the multicore hardware that is available today. But before we go any further, we should explain more exactly what we mean by the terms *concurrency* and *process*.

1.1.1 Understanding concurrency

Is *concurrent* just another word for *in parallel*? Almost but not exactly, at least when we're talking about computers and programming.

One popular semiformal 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 extra arms and eyes, you could sort both in parallel. Nothing requires you to do them in a certain order; hence, they're 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.

This may sound strange: shouldn't we say that tasks are concurrent only if they're happening at the same time? Well, the point with that definition is that they *could* happen at the same time, and we're free to schedule them at our convenience. Tasks that *need* to be done simultaneously aren't separate tasks at all, whereas some tasks are separate but nonconcurrent and must be done in order, such as breaking the egg before making the omelet. The rest are concurrent.

One of the nice things that Erlang does for you is help with the physical execution of tasks. As illustrated in figure 1.1, if extra CPUs (or cores or hyperthreads) are available, Erlang uses them to run more of your concurrent tasks in parallel. If not, Erlang

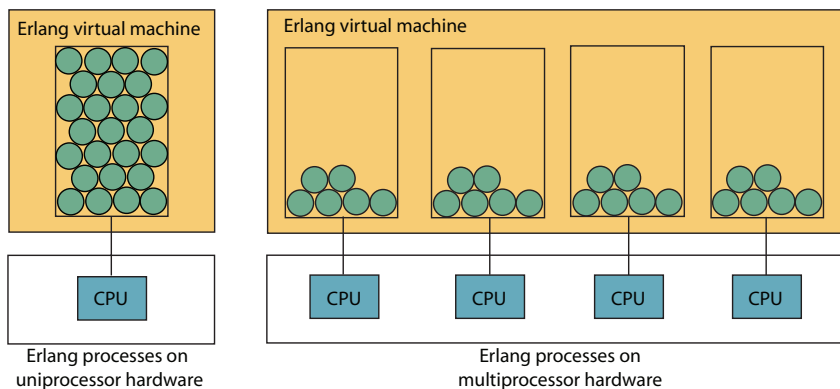


Figure 1.1 Erlang processes running on uniprocessor and on multiprocessor hardware, respectively. The runtime system automatically distributes the workload over the available CPU resources.

uses what CPU power there is to do them all a bit at a time. You won't 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.

But what if your tasks aren't concurrent, and your program must first do X, then Y, and finally Z? That is where you need to start thinking about the real dependencies in the problem you're out to solve. Perhaps X and Y can be done in any order as long as they're 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 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. But that should usually be your second concern. The most important effect of separating parts of the program that don't need to be together is that doing so 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. But first, we need a more concrete representation of the idea of having separate tasks.

1.1.2 Erlang's process model

In Erlang, the unit of concurrency is the *process*. A process represents an ongoing activity; it's an agent that is running a piece of program code, concurrent to other processes running their own code, at their own pace. Processes are a bit like people: individuals who don't share anything between them. Not that people aren't generous, but if *you* eat food, nobody else gets full; and more important, if you eat *bad* food, only you get sick from it. You have your own brain and internals that keep you thinking and living independently from everyone else. This is how processes behave;

they're separate from one another and are guaranteed not to disturb one another through their own internal state changes.

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 little to do with any other; but if the server accepts only one at a time and doesn't start handling the next request until the previous is finished, there will quickly be thousands of requests on queue if the web site is popular.

If the server instead can begin handling requests as soon as they arrive, each in a separate process, there will be no queue, and most requests will take about the same time from start to finish. The state encapsulated by each process is then 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 about the request and recycling the memory. If a bug causes one request to crash, only that process dies, while all the others keep working happily.

Because processes can't directly change each other's internal state, it's possible to make significant advances in fault tolerance. No matter how bad the code is that a process is running, it can't 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, for example, the web browser and the word processor on your computer desktop. This turns out to be very powerful, as you'll see later in this chapter when we talk about process supervision.

Because 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 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's 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. Although it may sound strange at first, this means network programming is no different from coding on a single machine!

This *transparent distribution* allows Erlang programmers to look at the network as a collection of resources—we don't much care about whether process X is running on a different machine than process Y, because the method of communication is exactly the same no matter where they're located. In the next section, we provide an overview of methods of process communication used by various programming languages and systems, to give you an understanding of the trade-offs involved.

1.1.3 Four process communication paradigms

The central problem in all concurrent systems, which all implementers have to solve, is sharing information. If you separate a problem into different tasks, how should those tasks communicate with one another? It may seem like a simple question, but some of the brightest minds out there have wrestled with it, and many approaches have been tried over the years, some of which have appeared as programming language features and some as separate libraries.

We briefly discuss four approaches to process communication that have gained mindshare over the last few years. We won't spend too much time on any single one, but this will give you an overview of the approaches current-day languages and systems are taking and highlight the differences between those and Erlang. These four are shared memory with locking, software transactional memory, futures, and message passing. We start with the oldest but still the most popular method.

SHARED MEMORY WITH LOCKS

Shared memory could reasonably be called the GOTO of our time: it's the current mainstream technique for process communication; it has been so for a long, long time; and just like programming with GOTO, there are numerous ways to shoot yourself in the foot. This has imbued generations of engineers with a deep fear of concurrency (and those who don't fear it haven't tried it yet). Still, we must admit that like GOTO, there is a low-level niche for shared memory where it probably can't be replaced.

In this paradigm, one or more regular memory cells can be read or written to by two or more processes in parallel. To make it possible for a process to perform an *atomic* sequence of operations on those cells, so that no other process is able to access any of the cells before all the operations have completed, there must be a way for the process to block all others from accessing the cells until it has finished. This is done with a *lock*: a construct that makes it possible to restrict access to a single process at a time.

Implementing locks requires support from the memory system, typically hardware support in the form of special instructions. The use of locks requires complete cooperation between processes: all must make sure to ask for the lock before accessing a shared memory region, and they must return the lock when they're done so that someone else gets a chance to use it. The slightest failure can cause havoc; so, generally, higher-level constructs such as semaphores, monitors, and mutexes, are built on these basic locks and are provided as operating system calls or programming language

constructs to make it easier to guarantee that locks are properly requested and returned. Although this avoids the worst problems, locks still have a number of drawbacks. To mention only a few:

- Locks require overhead even when the chances of collisions are low.
- They're points of contention in the memory system.
- They may be left in a locked state by failed processes.
- It's extraordinarily hard to debug problems with locks.

Furthermore, locking may work well for synchronizing two or three processes, but as the number grows, the situation quickly becomes unmanageable. A real possibility exists (in many cases, more of a certainty) of ending up with a complex deadlock that couldn't be foreseen by even the most experienced developer.

We think this form of synchronization is best left to low-level programming, such as in the operating system kernel. But it can be found in most current popular programming and scripting languages. Its ubiquitousness is likely due to the fact that it's fairly easy to implement and doesn't interfere with the programming model these languages are based on. Unfortunately, its widespread use has hurt our ability to think about concurrent issues and make use of concurrency on a large scale even though multiprocessor systems have been widely available for several years.

SOFTWARE TRANSACTIONAL MEMORY (STM)

The first nontraditional method we are going to look at is software transactional memory (STM). This mechanism can currently be found in the GHC implementation of the Haskell programming language, as well as in the JVM-based language Clojure. STM treats memory more like a traditional database, using *transactions* to decide what gets written and when. Typically, the implementation tries to avoid using locks by working in an optimistic way: a sequence of read and write accesses are treated as a single operation, and if two processes try to access the shared region at the same time, each in its own transaction, only one of them succeeds. The other processes are told that they failed and should try again after checking what the new contents are. It's a straightforward model and doesn't require anyone to wait for someone else to release a lock.

The main drawback is that you have to retry failed transactions (and they could, of course, fail repeatedly). There is also some significant overhead involved with the transaction system itself, as well as a need for additional memory to store the data you're trying to write until it's decided which process will succeed. Ideally, there should be hardware support for transactional memory just as there typically is support for virtual memory.

The STM approach seems more manageable to programmers than the use of locks, and it may be a good way to take advantage of concurrency as long as transactions don't have to be restarted too often due to contention. We still consider this approach to be at its core a variant of shared memory with locks, and one that may be more help on an operating system level than on an application programming level; but it's currently a lively research topic, and things may turn out differently.

FUTURES, PROMISES, AND SIMILAR

Another more modern approach is the use of so-called *futures* or *promises*. This is a concept with several variants; it can be found in languages like E and MultiLisp and as a library in Java, and it's similar to I-vars and M-vars in Id and Glasgow Haskell, concurrent logic variables in Concurrent Prolog, and dataflow variables in Oz.

The basic idea is that a future is a result of a computation that has been outsourced to some other process, possibly on another CPU or a completely different computer. A future can be passed around like any other object, but if someone wants to read the value and it isn't ready yet, they have to wait for it to be done. Although this is conceptually simple and makes it easy to pass around data in concurrent systems, it also makes the program brittle in case of failure of the remote process or the network in between: the code that tries to access the value of the promise may have no idea what to do if the value is still missing and the connection is dead.

MESSAGE PASSING

As we said in section 1.1.2, Erlang processes communicate by message passing. This means the receiving process effectively gets a separate copy of the data, and nothing it does to that copy is observable by the sender. The only way to communicate information back to the sender is to send another message in the reverse direction. One of the most important consequences is that communication works the same whether the sender and receiver are on the same computer or separated by a network.

Message passing in general comes in two flavors: *synchronous* and *asynchronous*. In the synchronous form, the sender can't do anything else until the message has arrived at the receiving end; in the asynchronous form, senders can proceed immediately after posting the message. (In the real world, synchronous communication between separate machines is only possible if the receiver sends an acknowledgment back to the sender, telling it that it's OK to continue, but this detail can be kept hidden from the programmer.)

In Erlang, the message passing primitives are asynchronous, because it's easy to implement the synchronous form when necessary by making the receiver always send an explicit reply that the sender can wait for. Often, though, the sender doesn't need to know that the message arrived—that knowledge is overrated, because nothing tells you what the receiver did next: it may have died just afterward. This asynchronous “send-and-pray” method of communication also means the sender doesn't need to be suspended while the message is being delivered (in particular if the message is sent over a slow communications link).

Of course, you don't get this level of separation between sender and receiver for free. Copying data can be expensive for large structures and can cause higher memory usage if the sender also needs to keep their copy of the data. In practice, this means you must be aware of and manage the size and complexity of messages you're sending. But in normal, idiomatic Erlang programs, the majority of messages are small, and the overhead of copying is usually negligible.

We hope this discussion has been of use to your understanding of Erlang's place in the concurrent programming landscape of today. Message passing may not be the sexiest of these techniques, but the track record of Erlang shows that from a systems engineering perspective, it seems to be the most practical and flexible.

1.1.4 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?" After you sketch out an answer to 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 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's such a foreign concept! But when you do get used to it, magic begins to happen. Picture a complex operation that has several concurrent parts, all modeled as separate processes. The operation starts, processes are spawned, data is manipulated, and a result is produced, and at that moment the processes involved disappear magically into oblivion, taking with them their internal state, their database handles, their sockets, and any other stuff that needs to be cleaned up that you don't want to have to deal with manually.

In the rest of this section, we take a brief look at how easy it is to create processes, how lightweight they are, and how simple it is to communicate between them.

CREATING A PROCESS: SPAWNING

Erlang processes are *not* operating system threads. They're much more lightweight, implemented by the Erlang runtime 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 runtime 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 a 32-bit machine can never have more than a few thousand simultaneous threads), and it still crashes if it uses more stack space than expected. Erlang processes, on the other hand, start with only a couple of hundred bytes of stack space each, and they grow or shrink automatically as required.

Erlang's syntax for creating processes is straightforward, as illustrated by the following example. Let's spawn a process whose job is to execute the function call `io:format("erlang!")` and then finish:

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

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

In chapter 2, we give an overview of the Erlang language and its syntax, but right now we hope you'll be able to get the gist of our examples without further explanation. One of the strengths of Erlang is that it's generally easy to understand the code even if you've never seen the language before. Let's see if you agree.


HOW PROCESSES TALK

Processes need to do more than spawn and run—they need to exchange information. Erlang makes this communication simple. The basic operator for sending a message is `!`, pronounced “bang,” and it's used in the form “Destination `!` Message”. This is message passing at its most primitive, like mailing a postcard. The OTP framework takes process communication to another level, and we dive into that in chapter 3; for now, let's marvel at the simplicity of communicating between two independent and concurrent processes, as illustrated in the following listing.

Listing 1.1 Process communication in Erlang

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

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



1 From contains sender ID

Take a minute or two and look at this code. You can probably understand it without any previous knowledge of Erlang. Points worth noting are a variant of the `spawn` function that gets a single reference to “the function named `ping` that takes zero arguments”; and the function `self()`, which produces the identifier of the current process, which is passed to the new process so that it knows where to reply **1**.

That's Erlang's process communication in a nutshell. Every call to `spawn` yields a fresh process identifier that uniquely identifies the new child process. This identifier can then be used to send messages to the child. Each process has a mailbox where incoming messages are stored as they arrive, even if the receiving process is currently busy, and the messages are kept there until the process decides to check the mailbox. It can then search and retrieve messages from the mailbox at its convenience using a `receive` expression, as in the example (which grabs the first available message).

PROCESS TERMINATION

When a process is done with its work, it disappears. Its working memory, mailbox, and other resources are recycled. If the purpose of the process is to produce data for another process, it must send that data explicitly as a message before it terminates.

Crashes (exceptions) can make a process terminate unexpectedly and prematurely, and if this happens, other processes can be informed of the crash. We've previously talked about how processes are independent and the fact that a crash in

one can't corrupt another, because they don't share internal state. This is one of the pillars of another of Erlang's main features: fault tolerance, which we cover in 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 aren't 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 don't 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 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 receive this signal. By default, this causes the receiver to exit as well and propagate the signal on to any other processes it's linked to, and so on, until all the processes that are linked directly or indirectly to each other have exited (see figure 1.2). This cascading behaviour allows you to have a group of processes behave as a single application with respect to termination, so that you never need to worry about finding and killing off any leftover processes before you can restart that entire subsystem from scratch.

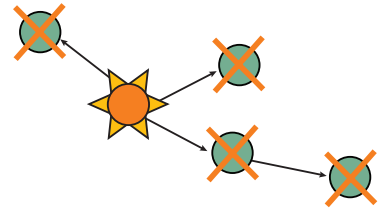


Figure 1.2 An exit signal triggered by a crashing process is propagated to all its linked processes, generally making those terminate as well so that the whole group is cleaned up.

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.

Let it crash

Rather than thrashing around desperately to save a situation that you probably won't be able to fix, the Erlang philosophy is "let it crash"—you drop everything cleanly and start over, logging precisely where things went pear-shaped and how. This can take some getting used to, but it's a powerful recipe for fault tolerance and for creating systems that are possible to debug despite their complexity.

1.2.2 Supervision and 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`, you can make a process *trap* any incoming exit signal rather than obey it. In this case, when the signal is received, it's dropped in the process's 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 typically runs code that is different from that run by ordinary *worker processes*, which don't usually trap signals. Because a system process acts as a bulwark that prevents exit signals from propagating further, it insulates the processes it's linked to from each other and can also be entrusted with reporting failures and even restarting the failed subsystems, as illustrated in figure 1.3. We call such processes *supervisors*.

The point of letting an entire subsystem terminate and be restarted is that it brings you 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 is that it isn't granular enough. Ideally, you'd like to be able to reboot only 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, you'd still be left to implement supervisors from scratch, which would require careful thought, lots of experience, and a long time shaking out

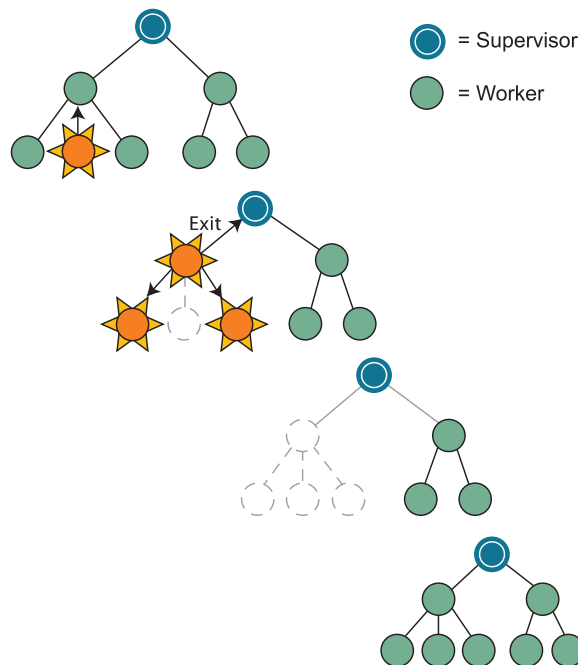


Figure 1.3
Supervisor, workers, and signals: the crash in one of the worker processes is propagated to the other linked processes until the signal reaches the supervisor, which restarts the group. The other group of processes under the same supervisor isn't affected.

the bugs and corner cases. Fortunately, the OTP framework provides just about everything you need: both a methodology for structuring 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 shouldn't be structured as a single-level hierarchy of supervisors and workers. In any complex system, you'll want a *supervision tree* with multiple layers that allows subsystems to be restarted at different levels in order to cope with unexpected problems of varying kinds.

1.2.3 Layering processes for fault tolerance

Layering brings related subsystems together under a common supervisor. More important, it defines different levels of working base states that you can revert to. In figure 1.4, you see 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.

Let's assume that the processes in group A work together to produce a stream of data that group B consumes. Group B isn't required for group A to function. To make things concrete, let's say group A is processing and encoding multimedia data, and group B presents it. Let's also suppose that a small percentage of the data entering group A is corrupt in some way that wasn't 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 trying to untangle the mess; and because processes are isolated, none of the other processes are affected by the bad input. The supervisor, detecting that a process has died, restores the base state 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 what's going on and doesn't care. As long as group A pushes enough good data to group B for the latter to display something of acceptable quality to the user, you have a successful system.

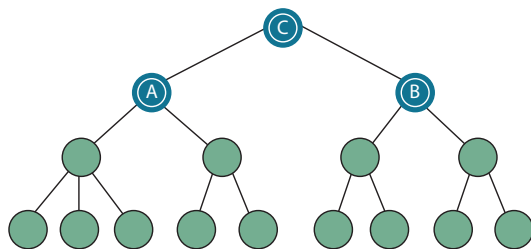


Figure 1.4

A layered system of supervisors and workers. If for some reason supervisor A dies or gives up, any still-living processes under it are killed and supervisor C is informed, so the whole left-side process tree can be restarted. Supervisor B isn't affected unless C decides to shut everything down.

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

But you're forced to share some things as long as you're running on a single machine: the available memory, the disk drive, the network connection, even the processor and all related circuitry, and, perhaps most significant, 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 you from inevitable downtime. This brings us to our next topic, which is distribution—the feature of Erlang that allows you to achieve the highest levels of fault tolerance and also make your solutions scale.

1.3 *Distributed Erlang*

Erlang programs can be distributed 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. Assuming that you manage to get the locking right, this is nice and efficient, but only until you want to move one of the threads to a separate machine. Perhaps you want to make use of more computing power or memory, or prevent both threads from dying 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 different communication mechanism necessary 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, aren't much affected by this kind of problem. As we explained in section 1.1.2, 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 rarely in Erlang. If it works on your netbook today, it could be running on a cluster tomorrow.

The fact that it's 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?"

A real-life example

At one employer, we had 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 common goal. Integration testing this cluster of 15 different applications running on 15 separate virtual machines, although doable, wouldn't have been the most convenient undertaking. Without changing a line of code, we were able to invoke all the applications on a single Erlang instance 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.

The concept demonstrated in this example 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 don't 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.

Now that you know a bit about what Erlang can do for you, we next talk about the engine at the heart of it all, to give you a better idea of what is going on under the hood when your Erlang program is running.

1.4 The Erlang runtime system and virtual machine

So what makes all of the above tick? The core of the standard Erlang implementation is something called the Erlang Run-Time System application (ERTS): this is a big chunk of code written in the C programming language, and it's responsible for all the low-level stuff in Erlang. It lets you talk to the file system and the console, it handles memory, and it implements Erlang processes. It controls how these processes are distributed over the existing CPU resources to make good use of your computer hardware, but at the same time makes it possible to run Erlang processes concurrently even if you only have a single CPU with a single core. ERTS also handles message-passing between processes and allows processes on two different machines, each in its own ERTS instance, to talk to each other as if they were on the same machine. Everything in Erlang that needs low-level support is handled by ERTS, and Erlang runs on any operating system that ERTS can be ported to.

One particularly important part of ERTS is the Erlang virtual machine emulator: this is the part that executes Erlang programs after they have been compiled to byte code. This virtual machine is known as Bogdan's Erlang Abstract Machine (BEAM) and is very efficient: even though it's also possible to compile Erlang programs to native machine code, it isn't usually necessary, because the BEAM emulator is fast enough. Note that there is no clear-cut line between the virtual machine and ERTS as a whole; often, people (including us) talk about the *Erlang VM* when they mean the emulator and runtime system as a whole.

There are many interesting features of the runtime system that you won't know about unless you dig through the documentation or spend a lot of time on the Erlang mailing list. These are at the core of what enables Erlang to handle such a large number of processes at the same time and are part of what makes Erlang unique. The basic philosophy of the Erlang language combined with the pragmatic approach the implementers have taken have given us an extraordinarily efficient, production-oriented, stable system. In this section, we cover three important aspects of the runtime system that contribute to Erlang's power and effectiveness:

- *The scheduler*—Handles the running of Erlang's processes, allowing all the ready-to-run processes to share the available CPU resources, and waking up sleeping processes when they get a new message or a timeout happens
- *The I/O model*—Prevents the entire system from stopping just because a single process wants to talk to some external device, making the system run smoothly
- *The garbage collector*—Keeps recycling memory that is no longer used

We start with the scheduler.

1.4.1 *The scheduler*

The process scheduler in ERTS has evolved over the years, and these days it gives you a flexibility matched by no other platform. Originally, it was there to make it possible to have lightweight Erlang processes running concurrently on a single CPU, regardless of what operating system you were using. ERTS generally runs as a single operating system process (usually found under the name `beam` or `werl` in OS process listings). Within this, the scheduler manages its own Erlang processes.

As threads became available in most operating systems, ERTS was changed to run a few things like the I/O system in a different thread from the one running Erlang processes, but there was still only one thread for the main body of work. If you wanted to use a multicore system, you had to run multiple ERTS instances on the same machine. But starting in May 2006 with release 11 of Erlang/OTP, support for symmetric multiprocessing (SMP) was added. This was a major effort, allowing the Erlang runtime system to use, not one, but multiple process schedulers internally, each using a separate operating system thread. The effect can be seen in figure 1.1.

This means there is now an $n:m$ mapping between Erlang processes and OS threads. Each scheduler handles a pool of processes. At most m Erlang processes can be running in parallel (one per scheduler thread), but the processes within each pool share their time as they did when there was only one scheduler for all processes. On top of this, processes can be moved from one pool to another to maintain an even balance of work over the available schedulers. In the latest releases of Erlang/OTP, it's even possible to tie processes to schedulers depending on the CPU topology of the machine, to make better use of the cache architecture of the hardware. That means, most of the time, you as an Erlang programmer don't have to worry about how many CPUs or cores you have available: you write your program as normal, trying to keep

your program separated into reasonably sized parallel tasks, and let the Erlang runtime system take care of spreading the workload. A single core or 128 cores—it works the same, only faster.

One caveat is that inexperienced Erlang programmers have a tendency to rely on effects of timing, which may make the program work on their laptop or workstation but break when the code moves to a server with multiple CPUs where timings may be much less deterministic. Hence, some testing is always in order. But now that even laptops often have at least two cores, this sort of thing is getting found out much earlier.

The scheduler in Erlang is also involved in another important feature of the runtime system: the I/O subsystem. This is our next topic.

1.4.2 I/O and scheduling

One of the things that many concurrent languages get wrong is that they don't think much about I/O. With few exceptions, they make the entire system or a large subset of it block while any process is doing I/O. This is annoying and unnecessary, considering that Erlang has had this problem solved for the last two decades. In the previous section, we talked about the Erlang process scheduler. Among other things, the scheduler allows the system to elegantly handle I/O. At the lowest levels of the system, Erlang does all I/O in an event-based way, which lets a program handle each chunk of data as it enters or leaves the system in a nonblocking manner. This reduces the need to set up and tear down connections, and it removes the need for OS-based locking and context switching.

This is an efficient method of handling I/O. Unfortunately, it also seems to be a lot harder for programmers to reason about and understand, which is probably why we only 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 paper “The C10K Problem” back in 2001; it's out of date in some respects now, but it's still relevant and worth reading. It should give you a good overview of the problem and the approaches available to solve it. All of these approaches are complex and painful to implement; that is why the Erlang runtime system does most of it for you. It integrates the event-based I/O system with its process scheduler. In effect, you get all the benefits with none of the hassle. This makes it much easier to build highly available systems using Erlang/OTP.

The last feature of ERTS that we want to explain is memory management. This has more to do with processes than you may think.

1.4.3 Process isolation and the garbage collector

As you probably know or assume, Erlang manages memory automatically, like Java and most other modern languages. There is no explicit de-allocation. Instead, a so-called *garbage collector* is used to regularly find and recycle unused memory. Garbage collection (GC) algorithms constitute a large and complicated field of research, and we can't go through any details here; but for those of you who know a bit and are curious, Erlang currently uses a straightforward generational copying garbage collector.

Even with this relatively simple implementation, programs in Erlang don't tend to suffer from pauses for GC, like systems implemented in other languages. This is mostly due to the isolation between processes in Erlang: each has its own areas of memory, allocated when the process is created and de-allocated again when the process dies. That may not sound important, but it is. First, it means that each process can be individually paused for garbage collection while all the others keep running. Second, the memory used by any single process is usually small, so traversing it can be done quickly. (Although some processes use large amounts of memory, those typically aren't the ones that are also expected to respond quickly.) Third, the scheduler always knows when a process was last running, so if it hasn't been doing any work since the last time it was garbage collected, it can be skipped. All this makes life simpler for the Erlang garbage collector and lets it keep pause times small. Furthermore, in some cases it's possible for a process to be spawned, do its job, and die again without triggering any garbage collection at all. In those cases, the process acted as a short-lived memory region, automatically allocated and de-allocated with no additional overhead.

The features of the runtime system that we have described in this section make it possible for an Erlang program to have a large number of processes running that make good use of the available CPUs, perform I/O operations, and automatically recycle memory, all while maintaining soft real-time responsiveness. Understanding these aspects of the platform is important for understanding the behaviour of your systems after they're up and running.

Finally, before this chapter is done, we'll say a few words about the functional programming aspect of Erlang. It won't be much, because we get into much more detail about the Erlang language in the next chapter.

1.5 Functional programming: Erlang's face to the world

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

To summarize what functional programming is, the main ideas are that functions are data, just like integers and strings; that algorithms are expressed in terms of function calls, not using loop constructs like `while` and `for`; and that variables and values are never updated in place (see appendix B for a discussion of referential transparency and lists). Those may sound like artificial restrictions, but they make perfectly good sense from an engineering perspective, and Erlang programs can be very natural and readable.

Erlang isn't a “pure” functional language—it relies on side effects. But it limits these to a single operation: message passing by copying. Each message is an effect on the world outside, and the world can have effects on your processes by sending

them messages. But each process is, in itself, running an almost purely functional program. This model makes programs much easier to reason about than in traditional languages like C++ and Java, while not forcing you to program using monads as in Haskell.

In the next chapter, we go through the important parts of the Erlang programming language. For many of you, the syntax will feel strange—it borrows mainly from the Prolog tradition, rather than from C. But different as it may be, it isn't complicated. Bear with it for a while, and it will become second nature. After you're familiar with it, you'll 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?

1.6 Summary

In this chapter, we've gone over the most important concepts and features of the Erlang/OTP platform that OTP is built on: concurrent programming with processes and message passing, fault tolerance through links, distributed programming, the Erlang Run-Time System and virtual machine, and the core functional language of Erlang. All these things combine to provide a solid, efficient, and flexible platform on which you can build reliable, low-latency, high-availability systems.

If you have previous experience with Erlang, much of this may not be news to you, but we hope our presentation was of interest and pointed out at least a few aspects you hadn't thought about before. Still, we haven't talked much about the OTP framework yet; we'll wait until chapter 3, but then things will move quickly, so enjoy this background reading while you can. First, chapter 2 will give you a thorough overview of the Erlang programming language.

Erlang AND OTP IN ACTION

Martin Logan • Eric Merritt • Richard Carlsson

Erlang is an adaptable and fault tolerant functional programming language originally designed for the unique demands of the telecom industry. With Erlang/OTP's interpreter, compiler, database server, and libraries, developers are finding they can satisfy tough uptime and performance requirements in all kinds of other industries.

Erlang and OTP in Action teaches you the concepts of concurrent programming and the use of Erlang's message-passing model. It walks you through progressively more interesting examples, building systems in Erlang and integrating them with C/C++, Java, and .NET applications, including SOA and web architectures. This book is written for readers new to Erlang and interested in creating practical applications.

Build apps that...

- Never deadlock on a shared resource
- Keep running, even during code upgrades
- Recover gracefully from errors
- Scale unchanged from one to many processors
- Handle many simultaneous connections, and
- Maintain fast response times

A core developer for Erlware, **Martin Logan** has worked with Erlang since 1999. **Eric Merritt** is a core developer for Erlware and the Sinan build system. An Erlang pioneer, **Richard Carlsson** is an original member of the High-Performance Erlang group.

For online access to the authors and a free ebook for owners of this book, go to manning.com/ErlangandOTPinAction



“An enormous amount of experience, combined.”
—From the Foreword by Ulf Wiger, Erlang Solutions Ltd.

“Full of practical, real-world code samples.”
—Greg Donald
Vanderbilt University

“Illuminates how to do things the Erlang way.”
—John S. Griffin
Overstock.com, Inc.

“The missing link on the Erlang learning curve.”
—Ken Pratt, Ruboss Technology Corporation

“An indispensable resource.”
—David Dossot
Programmer and Author

ISBN 13: 978-1-933988-78-8
ISBN 10: 1-933988-78-9

