Extracted from:

Programming Erlang, Second Edition

Software for a Concurrent World

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Erlang

Software for a Concurrent World

Second Edition

Joe Armstrong

Edited by Susannah Davidson Pfalzer

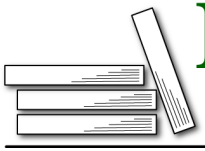
Programming Erlang, Second Edition

Software for a Concurrent World

Joe Armstrong

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Susannah Davidson Pfalzer (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-53-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—August 2013

CHAPTER 1

Introducing Concurrency

Let's forget about computers for a moment; I'm going to look out of my window and tell you what I see.

I see a woman taking a dog for a walk. I see a car trying to find a parking space. I see a plane flying overhead and a boat sailing by. All these things happen *in parallel*. In this book, we will learn how to describe parallel activities as sets of communicating parallel processes. We will learn how to write *concurrent programs*.

In everyday language, words like *concurrent*, *simultaneous*, and *parallel* mean almost the same thing. But in programming languages, we need to be more precise. In particular, we need to distinguish between concurrent and parallel programs.

If we have only a single-core computer, then we can never run a parallel program on it. This is because we have one CPU, and it can do only one thing at a time. We can, however, run concurrent programs on a single-core computer. The computer time-shares between the different tasks, maintaining the illusion that the different tasks run in parallel.

In the following sections, we'll start with some simple concurrency modeling, move on to see the benefits of solving problems using concurrency, and finally look at some precise definitions that highlight the differences between concurrency and parallelism.

1.1 Modeling Concurrency

We'll start with a simple example and build a concurrent model of an everyday scene. Imagine I see four people out for a walk. There are two dogs and a large number of rabbits. The people are talking to each other, and the dogs want to chase the rabbits.

To simulate this in Erlang, we'd make four modules called person, dog, rabbit, and world. The code for person would be in a file called person.erl and might look something like this:

```
-module(person).
-export([init/1]).

init(Name) -> ...
```

The first line, -module(person)., says that this file contains code for the module called person. This should be the same as the filename (excluding the .erl filename extension). The module name *must* start with a small letter. Technically, the module name is an *atom*; we'll talk more about atoms in Section 3.5, *Atoms*, on page ?.

Following the module declaration is an *export declaration*. The export declarations tells which functions in the module can be called from *outside* the module. They are like *public* declarations in many programming languages. Functions that are not in an export declaration are private and cannot be called from outside the module.

The syntax -export([init/1]). means the function init with one argument (that's what /1 means; it does not mean divide by one) can be called from outside the module. If we want to export several functions, we'd use this syntax:

```
-export([FuncName1/N1, FuncName2/N2, .....]).
```

The square brackets [...] mean "list of," so this declaration means we want to export a list of functions from the module.

We'd write similar code for dog and rabbit.

Starting the Simulation

To start the program, we'll call world:start(). This is defined in a module called world, which begins like this:

```
-module(world).
-export([start/0]).

start() ->
    Joe      = spawn(person, init, ["Joe"]),
    Susannah = spawn(person, init, ["Susannah"]),
    Dave     = spawn(person, init, ["Dave"]),
    Andy     = spawn(person, init, ["Andy"]),
    Rover    = spawn(dog,    init, ["Rover"]),
    ...
    Rabbit1  = spawn(rabbit, init, ["Flopsy"]),
   ...
```

spawn is an Erlang primitive that creates a concurrent process and returns a process identifier. spawn is called like this:

```
spawn(ModName, FuncName, [Arg1, Arg2, ..., ArgN])
```

When spawn is evaluated, the Erlang runtime system creates a new process (not an operating system process but a lightweight process that is managed by the Erlang system). Once the process has been created, it starts evaluating the code specified by the arguments. ModName is the name of the module that has the code we want to execute. FuncName is the name of the function in the module, and [Arg1, Arg2, ...] is a list containing the arguments to the function that we want to evaluate. Thus, the following call means start a process that evaluates the function person:init("Joe"):

```
spawn(person, init, ["Joe"])
```

The return value of spawn is a *process identifier* (PID) that can be used to interact with the newly created process.

Analogy with Objects

Modules in Erlang are like classes in an object-oriented programming language (OOPL), and processes are like objects (or class instances) in an OOPL.

In Erlang, spawn creates a new process by running a function defined in a module. In Java, new creates a new object by running a method defined in a class.

In an OOPL we can have one class but several thousand class instances. Similarly, in Erlang we can have one module but thousands or even millions of processes that execute the code in the module. All the Erlang processes execute concurrently and independently and, if we had a million-core computer, might even run in parallel.

Sending Messages

Once our simulation has been started, we'll want to send messages between the different processes in the program. In Erlang, processes share no memory and can interact only with each other by sending messages. This is exactly how objects in the real world behave.

Suppose Joe wants to say something to Susannah. In the program we'd write a line of code like this:

```
Susannah ! {self(), "Hope the dogs don't chase the rabbits"}
```

The syntax Pid ! Msg means send the message Msg to the process Pid. The self() argument in the curly brackets identifies the process sending the message (in this case Joe).

Receiving Messages

For Susannah's process to receive the message from Joe, we'd write this:

```
receive
   {From, Message} ->
     ...
end
```

When Susannah's process receives a message, the variable `From` will be bound to `Joe` so that Susannah knows who the message came from, and the variable `Message` will contain the message.

We could imagine extending our model by having the dogs send "woof woof rabbits" messages to each other and the rabbits sending "panic go and hide" messages to each other.

The key point we should remember here is that our programming model is based on *observation* of the real world. We have three modules (`person`, `dog`, and `rabbit`) because there are three types of concurrent things in our example. The `world` module is needed for a top-level process to start everything off. We created two dog processes because there are two dogs, and we created four people processes because there were four people. The messages in the program reflect the observed messages in our example.

Rather than extending the model, we'll stop at this point, change gears, and look at some of the characteristics of concurrent programs.

1.2 Benefits of Concurrency

Concurrent programming can be used to improve performance, to create scalable and fault-tolerant systems, and to write clear and understandable programs for controlling real-world applications. The following are some of the reasons why this is true:

Performance

Imagine you have two tasks: `A`, which takes ten seconds to perform, and `B`, which takes fifteen seconds. On a single CPU doing both, `A` and `B` will take twenty-five seconds. On a computer with two CPUs that operate independently, doing `A` and `B` will take only fifteen seconds. To achieve this performance improvement, we have to write a concurrent program.

Until recently, parallel computers were rare and expensive, but today multicore computers are commonplace. A top-end processor has sixty-four cores, and we can expect the number of cores per chip to steadily increase in the foreseeable future. If you have a suitable problem and a

computer with sixty-four cores, your program might go sixty-four times faster when run on this computer, but only if you write a concurrent program.

One of the most pressing problems in the computer industry is caused by difficulties in parallelizing legacy sequential code so it can run on a multicore computer. There is no such problem in Erlang. Erlang programs written twenty years ago for a sequential machine now just run faster when we run them on modern multicores.

Scalability

Concurrent programs are made from small independent processes. Because of this, we can easily scale the system by increasing the number of processes and adding more CPUs. At runtime the Erlang virtual machine automatically distributes the execution of processes over the available CPUs.

Fault tolerance

Fault tolerance is similar to scalability. The keys to fault tolerance are independence and hardware redundancy. Erlang programs are made up of many small independent processes. Errors in one process cannot accidentally crash another process. To protect against the failure of an entire computer (or data center), we need to detect failures in remote computers. Both process independence and remote failure detection are built into the Erlang VM.

Erlang was designed for building fault-tolerant telecommunications systems, but the same technology can be applied equally well to building fault-tolerant scalable web systems or cloud services.

Clarity

In the real world things happen in parallel, but in most programming languages things happen sequentially. The mismatch between the parallelism in the real world and the sequentiality in our programming languages makes writing real-world control problems in a sequential language artificially difficult.

In Erlang we can map real-world parallelism onto Erlang concurrency in a straightforward manner. This results in clear and easily understood code.

Now that you've seen these benefits, we'll try to add some precision to the notion of concurrency and parallelism. This will give us a framework to talk about these terms in future chapters.

1.3 Concurrent Programs and Parallel Computers

I'm going to be pedantic here and try to give precise meanings to terms such as *concurrent* and *parallel*. We want to draw the distinction between a concurrent program, which is something that could potentially run faster if we had a parallel computer, and a parallel computer that really has more than one core (or CPU).

- A *concurrent program* is a program written in a concurrent programming language. We write concurrent programs for reasons of performance, scalability, or fault tolerance.
- A *concurrent programming language* is a language that has explicit language constructs for writing concurrent programs. These constructs are an integral part of the programming language and behave the same way on all operating systems.
- A *parallel computer* is a computer that has several processing units (CPUs or cores) that run at the same time.

Concurrent programs in Erlang are made from sets of communicating sequential processes. An Erlang process is a little virtual machine that can evaluate a single Erlang function; it should not be confused with an operating system process.

To write a concurrent program in Erlang, you must identify a set of processes that will solve your problem. We call this act of identifying the processes *modeling concurrency*. This is analogous to the art of identifying the objects that are needed to write an objected-oriented program.

Choosing the objects that are needed to solve a problem is recognized as being a hard problem in object-oriented design. The same is true in modeling concurrency. Choosing the correct processes can be difficult. The difference between a good and bad process model can make or break a design.

Having written a concurrent program, we can run it on a parallel computer. We can run on a multicore computer or on a set of networked computers or in the cloud.

Will our concurrent program actually run in parallel on a parallel computer? Sometimes it's hard to know. On a multicore computer, the operating system might decide to turn off a core to save energy. In a cloud, a computation might be suspended and moved to a new computer. These are things outside our control.

We've now seen the difference between a concurrent program and a parallel computer. Concurrency has to do with software structure; parallelism has to do with hardware. Next we'll look at the difference between sequential and concurrent programming languages.

1.4 Sequential vs. Concurrent Programming Languages

Programming languages fall into two categories: sequential and concurrent. Sequential languages are languages that were designed for writing sequential programs and have no linguistic constructs for describing concurrent computations. Concurrent programming languages are languages that were designed for writing concurrent programs and have special constructs for expressing concurrency in the language itself.

In Erlang, concurrency is provided by the Erlang virtual machine and not by the operating system or by any external libraries. In most sequential programming languages, concurrency is provided as an interface to the concurrency primitives of the host operating system.

The distinction between operating system– and language-based concurrency is important because if you use operating system–based concurrency, then your program will work in different ways on different operating systems. Erlang concurrency works the same way on all operating systems. To write concurrent programs in Erlang, you just have to understand Erlang; you don't have to understand the concurrency mechanisms in the operating system.

In Erlang, processes and concurrency are the tools we can use to shape and solve our problems. This allows fine-grained control of the concurrent structure of our program, something that is extremely difficult using operating system processes.

Wrapping Up

We've now covered the central themes of this book. We talked about concurrency as a means for writing performant, scalable, and fault-tolerant software, but we did not go into any details as to how this can be achieved. In the next chapter, we'll take a whirlwind tour through Erlang and write our first concurrent program.