Chapter 7

Concurrency

We understand concurrency.

A deep understanding of concurrency is hardwired into our brains. We react to stimulation extremely quickly, using a part of the brain called the *amygdala*. Without this reaction, we would die. Conscious thought is just too slow; by the time the thought "hit the brakes" has formed itself, we have already done it.

While driving on a major road, we mentally track the positions of dozens, or perhaps hundreds, of cars. This is done without conscious thought. If we couldn't do this, we would probably be dead.

The world is parallel.

If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

This is why we should program in a concurrent programming language.

And yet most often we program real-world applications in sequential programming languages. This is unnecessarily difficult.

Use a language that was designed for writing concurrent applications, and concurrent development becomes a lot easier.

Erlang programs model how we think and interact.

We don't have shared memory. I have my memory. You have yours. We have two brains, one each. They are not joined together. To change your memory, I send you a message: I talk, or I wave my arms.

You listen, you see, and your memory changes; however, without asking you a question or observing your response, I do not know that you have received my messages.

This is how it is with Erlang processes. Erlang processes have no shared memory. Each process has its own memory. To change the memory of some other process, you must send it a message and hope that it receives and understands the message.

To confirm that another process has received your message and changed its memory, you must ask it (by sending it a message). This is exactly how we interact.

Sue: *Hi Bill, my telephone number is 45 67 89 12.*

Sue: *Did you hear me?*

Bill: *Sure, your number is 45 67 89 12.*

These interaction patterns are well-known to us. From birth onward we learn to interact with the world by observing it and by sending it messages and observing the responses.

People function as independent entities who communicate by sending messages.

That's how Erlang processes work, and that's how we work, so it's very easy to understand an Erlang program.

An Erlang program is made up of dozens, thousands, or even hundreds of thousands of small processes. All these processes operate independently. They communicate with each other by sending messages. Each process has a private memory. They behave like a huge room of people all chattering away to each other.

This makes Erlang program inherently easy to manage and scale. Suppose we have ten people (processes), and they have too much work to do. What can we do? Get more people. How can we manage these groups of people? It's easy—just shout instructions at them (broadcasting).

Erlang processes don't share memory, so there is no need to lock the memory while it is being used. Where there are locks, there are keys that can get lost. What happens when you lose your keys? You panic and don't know what to do. That's what happens in software systems when you lose your keys and your locks go wrong.

Distributed software systems with locks and keys always go wrong.

Erlang has no locks and no keys.

If somebody dies, other people will notice.

If I'm in a room and suddenly keel over and die, somebody will probably notice (well, at least I hope so). Erlang processes are just like people—they can on occasions die. Unlike people, when they die, they shout out in their last breath exactly what they have died from.

Imagine a room full of people. Suddenly one person keels over and dies. Just as they die, they say "I'm dying of a heart attack" or "I'm dying of an exploded gastric wobbledgog." That's what Erlang processes do. One process might die saying "I'm dying because I was asked to divide by zero." Another might say, "I'm dying because I was asked what the last element in an empty list was."

Now in our room full of people, we might imagine there are specially assigned people whose job it is to clear away the bodies. Let's imagine two people, Jane and John. If Jane dies, then John will fix any problems associated with Jane's death. If John dies, then Jane will fix the problems. Jane and John are linked together with an invisible agreement that says that if one of them dies, the other will fix up any problems caused by the death.

That's how error detection in Erlang works. Processes can be linked together. If one of the processes dies, the other process gets an error message saying why the first process dies.

That's basically it.

That's how Erlang programs work.

Here's what we've learned so far:

- Erlang programs are made of lots of processes. These processes can send messages to each other.
- These messages may or may not be received and understood. If you want to know whether a message was received and understood, you must send the process a message and wait for a reply.

- Pairs of processes can be linked together. If one processes in a linked pair dies, the other process in the pair will be sent a message containing the reason why the first process died.

This simple model of programming is part of a model I call *concurrency-oriented programming*.

In the next chapter, we'll start writing concurrent programs. We need to learn three new primitives: **spawn**, **send** (using the ! operator), and **receive**. Then we can write some simple concurrent programs.

When processes die, some other process notices if they are linked together. This is the subject of Chapter 9, *Errors in Concurrent Programs*, on page 159.

As you read the next two chapters, think of people in a room. The people are the processes. The people in the room have individual private memories; this is the state of a process. To change your memory, I talk to you, and you listen. This is sending and receiving messages. We have children; this is spawn. We die; this is a process exit.