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

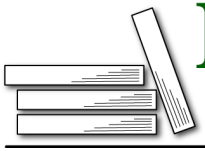
Programming Erlang, Second Edition

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 13

Errors in Concurrent Programs

Handling errors in concurrent programs involves a completely different way of thinking than handling errors in sequential programs. In this chapter, we'll build upon the principles you learned about in Chapter 6, *Error Handling in Sequential Programs*, on page ?, extending the ideas to concurrent programs.

We'll look at the underlying philosophy of error handling and at the details of how errors are propagated between processes and trapped by other processes. Finally we'll round off with some small examples that form a basis for programming fault-tolerant software.

Imagine a system with only one sequential process. If this process dies, we might be in deep trouble since no other process can help. For this reason, sequential languages have concentrated on the prevention of failure and an emphasis on *defensive programming*.

In Erlang we have a large number of processes at our disposal, so the failure of any individual process is not so important. We usually write only a small amount of defensive code and instead concentrate on writing *corrective code*. We take measures to detect the errors and then correct them after they have occurred.

13.1 Error Handling Philosophy

Error handling in concurrent Erlang programs is based on the idea of *remote detection and handling of errors*. Instead of handling an error in the process where the error occurs, we let the process die and correct the error in some other process.

When we design a fault-tolerant system, we assume that errors will occur, that processes will crash, and that machines will fail. Our job is to detect the errors after they have occurred and correct them if possible. Users of the

system should not notice any failures or suffer any loss of service while the error is being fixed.

Since we concentrate on cure rather than prevention, our systems have very little defensive code; instead, we have code to clean up the system after errors have occurred. This means we will concentrate on how to detect errors, how to identify what has gone wrong, and how to keep the system in a stable state.

Detecting errors and finding out why something failed is built into the Erlang VM at a very low level and is part of the Erlang programming language. Building groups of processes that observe each other and take corrective action when errors are detected is provided in the standard OTP libraries and is described in Section 23.5, *The Supervision Tree*, on page ?. This chapter is about the language aspects of error detection and recovery.

The Erlang philosophy for building fault-tolerant software can be summed up in two easy-to-remember phrases: “Let some other process fix the error” and “Let it crash.”

Let Some Other Process Fix the Error

Processes are arranged to monitor each other for health. If a process dies, some other process can observe this and perform corrective actions.

For one process to observe another, we must create a *link* or *monitor* between the processes. If the linked or monitored processes dies, the observing process is informed.

Observing processes work transparently across machine boundaries, so a process running on one machine can monitor the behavior of a process running on a different machine. This is the basis for programming fault-tolerant systems. We cannot make fault-tolerant systems on one machine since the entire machine might crash, so we need at least two machines. One machine performs computations, and the other machines observe the first machine and take over if the first machine crashes.

This can be thought of as an extension to handling errors in sequential code. We can, after all, catch exceptions in sequential code and try to correct the error (this was the subject of Chapter 6, *Error Handling in Sequential Programs*, on page ?), but if this fails or if the entire machine fails, we let some other process fix the error.

Let It Crash

This will sound very strange to you if you come from a language like C. In C we are taught to write *defensive code*. Programs should check their arguments

and not crash. There is a very good reason for this in C: writing multiprocess code is extremely difficult and most applications have only one process, so if this process crashes the entire application, you're in big trouble. Unfortunately, this leads to large quantities of error checking code, which is intertwined with the non-error-checking code.

In Erlang we do exactly the opposite. We build our applications in two parts: a part that solves the problem and a part that corrects errors if they have occurred.

The part that solves the problem is written with as little defensive code as possible; we assume that all arguments to functions are correct and the programs will execute without errors.

The part that corrects errors is often *generic*, so the same error-correcting code can be used for many different applications. For example, in database transactions if something goes wrong in the middle of a transaction, we simply abort the transaction and let the system restore the database to the state it was in before the error occurred. In an operating system, if a process crashes, we let the operating system close any open files or sockets and restore the system to a stable state.

This leads to a clean separation of issues. We write code that solves problems and code that fixes problems, but the two are not intertwined. This can lead to a dramatic reduction in code volume.

Why Crash?

Crashing immediately when something goes wrong is often a very good idea; in fact, it has several advantages.

- We don't have to write defensive code to guard against errors; we just crash.
- We don't have to think about what to do; we just crash, and somebody else will fix the error.
- We don't make matters *worse* by performing additional computations after we know that things have gone wrong.
- We can get very good error diagnostics if we flag the first place where an error occurs. Often continuing after an error has occurred leads to even more errors and makes debugging even more difficult.
- When writing error recovery code, we don't need to bother about why something crashed; we just need to concentrate on cleaning up afterward.

- It simplifies the system architecture, so we can think about the application and error recovery as two separate problems, not as one interleaved problem.

That's enough of the philosophy. Now let's start drilling down into the details.

Getting Some Other Guy to Fix It

Letting somebody else fix an error rather than doing it yourself is a good idea and encourages specialization. If I need surgery, I go to a doctor and don't try to operate on myself.

If something trivial in my car goes wrong, the car's control computer will try to fix it. If this fails and something big goes wrong, I have to take the car to the garage, and some other guy fixes it.

If something trivial in an Erlang process goes wrong, I can try to fix it with a `catch` or `try` statement. But if this fails and something big goes wrong, I'd better just crash and let some other process fix the error.

13.2 Error Handling Semantics

In this section, you'll learn about the semantics of interprocess error handling. You'll see some new terms that you'll come across later in the chapter. The best way to understand error handing is to quickly read through the definitions and then skip to the next sections for a more intuitive understanding of the concepts involved. You can always refer to this section if you need to do so.

Processes

There are two types of processes: *normal processes* and *system processes*. `spawn` creates a normal process. A normal process can become a system process by evaluating the BIF `process_flag(trap_exit, true)`.

Links

Processes can be linked. If the two processes `A` and `B` are linked and `A` terminates for any reason, an error signal will be sent to `B` and the other way around.

Link sets

The *link set* of a process `P` is the set of processes that are linked to `P`.

Monitors

Monitors are similar to links but are one-directional. If `A` monitors `B` and if `B` terminates for any reason, a "down" message will be sent to `A` but not the other way around.

Messages and error signals

Processes collaborate by exchanging *messages* or *error signals*. Messages are sent using the `send` primitive. Error signals are sent automatically when a process crashes or when a process terminates. The error signals are sent to the link set of the process that terminated.

Receipt of an error signal

When a system process receives an error signal, the signal is converted into a message of the form `{'EXIT', Pid, Why}`. `Pid` is the identity of the process that terminated, and `Why` is the reason for termination (sometimes called the *exit reason*). If the process terminates without an error, then `Why` will be the atom `normal`; otherwise, `Why` describes the error.

When a normal process receives an error signal, it will terminate if the exit reason is not `normal`. When it terminates, it also broadcasts an exit signal to its link set.

Explicit error signals

A process that evaluates `exit(Why)` will terminate (if this code is not executing within the scope of a `catch` or `try` primitive) and broadcast an exit signal with the reason `Why` to its link set.

A process can send a "fake" error signal by evaluating `exit(Pid, Why)`. In this case, `Pid` will receive an exit signal with the reason `Why`. The process that called `exit/2` does not die (this is deliberate).

Untrappable exit signals

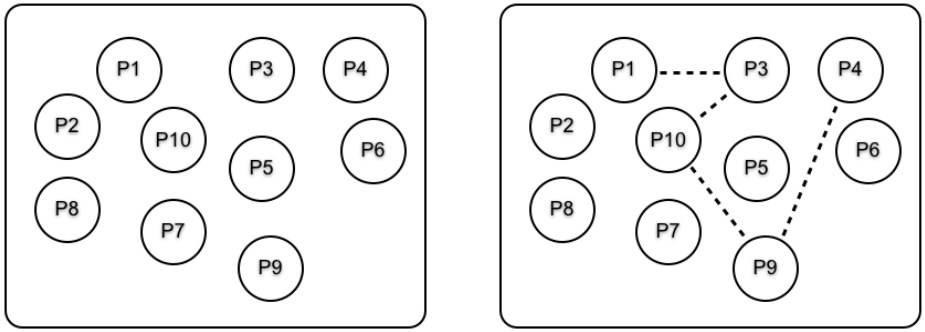
When a system process receives a *kill signal*, it terminates. Kill signals are generated by calling `exit(Pid, kill)`. This signal bypasses the normal error signal processing mechanism and is not converted into a message. The exit kill signal should be reserved for rogue processes that refuse to die using any of the other error handling mechanisms.

These definitions might look complicated, but a detailed understanding of how the mechanisms work is usually not necessary to write fault-tolerant code. The default behavior of the system tries to do "the right thing" as regard to error handling.

The next sections use a series of diagrams to illustrate how the error mechanisms work.

13.3 Creating Links

Imagine we have a set of unrelated processes; this is shown on the left side of the following figure. The links are represented by dashed lines.



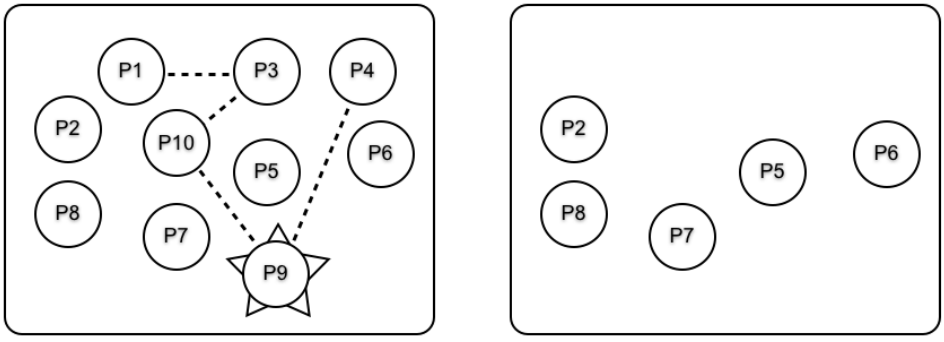
To create links, we call the primitive link(Pid), which creates a link between the calling process and Pid. So, if P1 calls link(P3), a link is created between P1 and P3.

After P1 calls link(P3), P3 calls link(P10), and so on, we arrive at the situation shown on the right side of the figure. Note that the link set of P1 has one element (P3), the link set of P3 has two elements (P1 and P10), and so on.

13.4 Groups of Processes That All Die Together

Often you'll want to create groups of processes that all die together. This is a very useful invariant for arguing about the behavior of a system. When processes collaborate to solve a problem and something goes wrong, we can sometimes recover, but if we can't recover, we just want to stop everything we were doing. This is rather like the notion of a transaction: either the processes do what they were supposed to do or they are all killed.

Assume we have some linked processes and that one of the linked processes dies. For example, see P9 in the following figure. The left side of the figure shows how the processes are linked before P9 dies. The right side shows which process are still alive after P9 has crashed and all error signals have been processed.

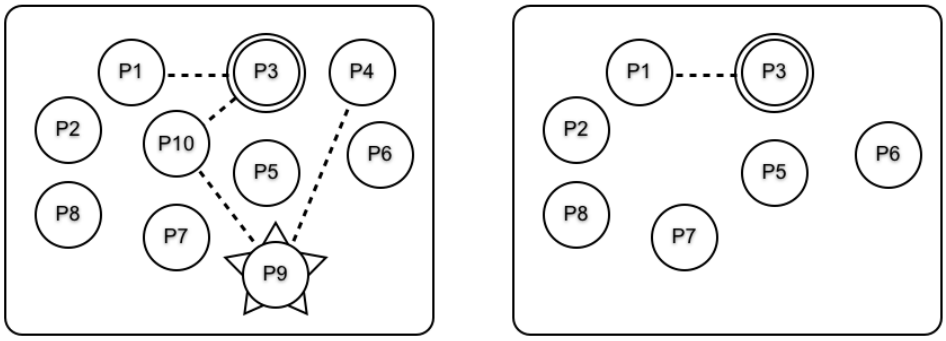


When P9 dies, an *error signal* is sent to processes P4 and P10. P4 and P10 also die because they are not system processes, and error signals are sent to any processes they are linked to. Ultimately, the error signals propagate to all the linked processes, and the entire group of linked processes dies.

Now if any of the processes P1, P3, P4, P9, or P10 die, they all die.

13.5 Setting Up a Firewall

Sometimes we don't want all our linked process to die, and we want to stop the propagation of errors through the system. The following figure illustrates this; here all linked process up to P3 die:



To achieve this, assume that P3 has evaluated process_flag(trap_exit, true) and become a system process (meaning that it can trap exit signals). This is shown with a double-circle border on the right side of the figure. After P9 crashed, the propagation of errors stopped at P3, so P1 and P3 did not die. This is shown on the right side of the figure.

P3 functions as a *firewall*, stopping errors from propagating to other processes in the system.

13.6 Monitors

Monitors are similar to links but with several significant differences.

- Monitors are unidirectional. If A monitors B and B dies, then A will be sent an exit message but not the other way around (recall that links were bidirectional, so if A and B were linked, the death of either process would result in the other process being informed).
- When a monitored process dies, a "down" message and not an exit signal is sent to the monitoring process. This means that the monitoring process does not have to become a system process in order to handle errors.

Monitors are used when you want asymmetry in error handling; links are used when you want symmetric error handling. Monitors are typically used by servers to monitor the behavior of clients.

The next section explains the semantics of the BIFs that manipulate links and monitors.