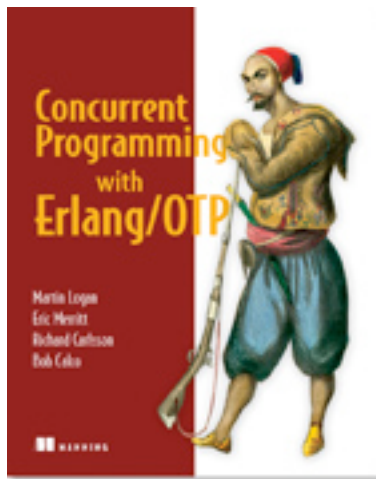


Green Paper From



Concurrent Programming with Erlang/OTP

EARLY ACCESS EDITION

Martin Logan, Eric Merritt, Richard Carlsson, and
Robert Calco

MEAP Release: August 2008

Softbound print: May 2009 (est.) | 500 pages

ISBN: 1933988789

This green paper is taken from the forthcoming book Concurrent Programming with Erlang/OTP from Manning Publications

Erlang is a programming language for which processes are the fundamental construct of the language. What is a process? When you run a word processor and an internet browser on your desktop you are using processes. Each one of those programs runs in it's own process. If your word processor crashes your internet browser typically stays running as if nothing happened. The reason for this is processes. Processes are a bit like people in that they don't share. I don't mean that people are not generous, what I mean is that if you eat food, I don't get full, and furthermore, if you eat bad food, I don't get sick. You have your own brain and internals that keep you thinking and living independently of what I do. This is how processes behave; they are separate from one another and are guaranteed not to have an effect on one another through their own internal state changes. Erlang runs entirely on processes.

Processes run independently of one another. This means that you can code simultaneous activities simultaneously. Because processes are so cheap in Erlang we can start to look at systems in a different manner. Each independent activity in an Erlang program can be coded that way. No unintuitive event loops here, no thread pooling, none of that. If your program needs 100k processes running at once to accomplish a job that is not a problem. As you will see further on, this really and fundamentally alters the way we look at systems in a way that is, as I hope to show you, much more intuitive.

Due to the fact that processes can't directly corrupt one another due to errors in their own internal state it is possible to make significant strides in fault through the use of processes as encapsulations of state. No matter how badly coded one process is it cannot corrupt the state of the rest of your system through its faults. Basically at a micro level you can have the same behavior out of your code as you see between the web browser and the word processor on your desktop.

Since processes share nothing they must communicate through copying. If one process wants to communicate with another it sends a message, that message is a copy of data that the sender has locally. These semantics make

For Source Code, Free Chapters, the Author Forum and more information about this title go to

<http://www.manning.com/logan>

distribution a natural part of Erlang. You can't share data over a wire you can only copy it. Erlang processes copy anyway which means that network programming is the same as coding on one machine. Erlang programmers look at the network as just a collection of resources, we don't much care about whether process X is running on a different machine than process Y because we are going to communicate between them in the exact same way no matter where they are located. As you can no doubt imagine this has tremendous positive impact on the ability of Erlang/OTP system to scale. Again though, more on that later!

Lastly Erlang is a functional programming language. It could have been a procedural language as well. The characteristics described above can be achieved without functional programming however functional programming and characteristics of it like referential transparency, the use of higher order functions, and it's general lack of mutable structures really lends itself nicely to the fundamental features of Erlang. Functional programming is the vehicle by which the above characteristics were codified and are expressed elegantly and succinctly in code. The power of what is above without the clarity of functional code would yield a very complex and much less enjoyable language to code in than Erlang is today.

Fundamental Features of Erlang/OTP

Processes and concurrency

Erlang is inherently concurrent. Concurrency allows us to create cleaner architectures and fully utilize the modern multi core hardware we see today. Processes are at the heart of that concurrency. A process executes code at the same time, concurrent to, other processes also executing code.

When you build an Erlang program you say to yourself, "what activities here are truly concurrent; can happen independently of one another?" Once you answer that question you go and build a system where every single one of the activities you identified becomes its own process. In stark contrast to most other languages concurrency is very cheap in Erlang, very cheap. Spawning a process boils down to a few assembly instructions and less than a millisecond. This takes some getting used to in the beginning because it is such a foreign concept, once you do get used to it however, the magic happens. Picture a complex operation that has six concurrent parts, all modeled as processes. The operation starts, processes are spawned, data is manipulated, an answer is arrived at, and at that very moment all processes involved simply die magically cascading into oblivion taking with them state, database handles, sockets, and all other sorts of stuff you don't want to handle manually.

In this section we are going to take a look at the characteristics of processes. We will demonstrate how to start them, how lightweight and fast they are, and how to communicate between them. This will set us up to be able to talk about what you can do with them and how they enable tremendous fault tolerance and scalability other fundamental features of Erlang.

Process Creation

Erlang is easily capable of spawning hundreds of thousands of processes on a single runtime system running atop commodity hardware. Each one of these processes is separate from all the others on the system, it shares no memory with any others, and in no way can it corrupt or be corrupted by another process dying. **Processes share nothing and copy everything.** That is the central concept with regard to processes. That is what all the fuss is about. If you share nothing and copy everything then concurrency can happen without the need for locks (and deadlocks). Once complex systems become, well, less complex. Spawning a process in Erlang is extremely simple and light weight. Erlang processes are not OS threads, they are much lighter than that and take on the order of microseconds to spawn. The syntax is quite simple as well as is illustrated by this example. We are going to spawn a process that will run the function `io:format("erlang is great")`.

```
spawn(io, format, ["erlang is great"])
```

Ok, that's it. You could safely run the code above 100000 times on your laptop and have 100000 calls to the *format* function within the *io* module run with no trouble. The code in the following listing does just that. I encourage you to type the code into your favorite editor and give it a try. In the next chapter we will show you how to start an Erlang shell and run programs.

Code listing 1.1

```
%% Don't be afraid to spawn off different functions
spawn_lots(0) ->
    ok;
spawn_lots(Number) ->
    spawn(io, format, ["erlang is great"]),
    spawn_lots(Number - 1).
```

They would not all be concurrent in this example however because the `io:format` function does not recurse. `io:format` simply prints some text to the screen and then finishes up, at which point the process finishes up. Processes are simple they are just function, if you spawn off a function that recurses infinitely then your process will live infinitely. Conversely if your process runs a function like `io:format` which runs and dies quickly then your process will be short lived as well.

Process communication

Processes need to do more than spawn and run however, they need to communicate. Erlang/OTP makes this communication quite simple. Remember the previous code example where we used a little syntax I referred to as the “bang” or the !. That is how a message is sent in its most primitive form. OTP takes that to another level and we will certainly be diving into all that is good with OTP messaging later on. For now, let's marvel at the simplicity of communicating between concurrent processes without a single synchronization concern in the world illustrated in code listing 1.2.

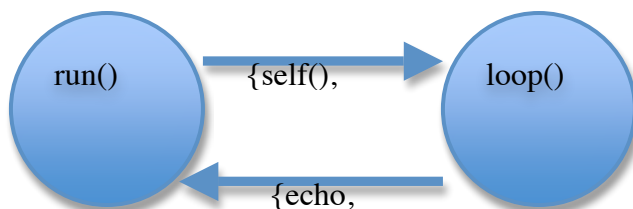
Code listing 1.2

```
run() ->
    Pid = spawn(fun() -> loop() end),
    Pid ! {self(), Msg},
    receive
        {echo, Msg} -> ok
    end.

loop() ->
    receive
        {From, Msg} -> From ! {echo, Msg}
    end,
    loop().
```

There we have it, process communication. Every call to spawn yields a process identifier that uniquely identifies the child process. This process identifier can then be used to send messages, or as we say, bang messages out to the child. Each process has associated with it a “process mailbox”. This mailbox is a list of messages that is attached to each process and which receives messages asynchronously with regard to what the process is doing. The process is free to search and retrieve messages from this mailbox at its convenience with the receive clause which is featured in our example.

In the example above a child process is spawned, the child process loops (recurses) forever each time around waiting for a message of the form {From, Msg}. When it receives such a message it sends a message back to the sender, From, of the form {echo, Msg}. Notice the use of the self() function within the run() function. The self() function is used, as you may have guessed, to get the process identifier, or PID, of the current process. This identifier is bound by the “loop” process as From and allows it to reply to its peer with its message.



For Source Code, Free Chapters, the Author Forum and more information about this title go to <http://www.manning.com/logan/>

In this section you learned that processes are independent of one another. They cannot corrupt one another because no data is shared between them. This is one of the pillars of another of Erlang's fundamental features; fault tolerance.

Fault Tolerance

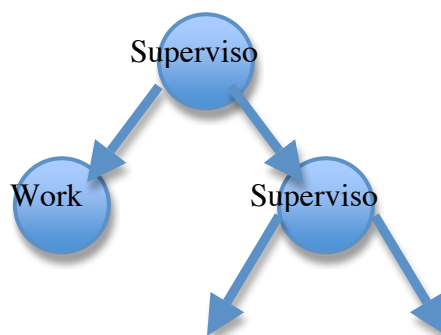
Fault tolerance is worth its weight in gold in the real world. Programmers are not perfect and neither are requirements. In order to deal with imperfections in code and data, just like dealing with a significant personal relationship that is not perfect, we need to have systems that are fault tolerant. The systems need to be able to deal with mistakes and not go to pieces each time one occurs. When a process dies an exit signal is generated. All processes that are "linked" to the terminating process will receive the exit signal. By default when this signal is received by another process, it dies as well. Previously I spoke about cleaning up complex state via processes. This is basically how it is done. Processes encapsulate all their state, sharing none of it and can therefore die safely without corrupting the rest of the system. This is as true for a group of linked processes as it is for one. One process dies, all its collaborators die, and all the complex state that was created is now snuffed out of existence cleanly and easily saving programmer time and reducing errors. Instead of failing when state gets too complex to unwind in Erlang you just remove the state cleanly without affecting the rest of your code and start over. That is a potent aspect of Erlang's overall fault tolerance.

Supervision and the trapping of exit signals

One of the major ways fault tolerance is achieved in OTP is to override this link chaining exit signal behavior. Calling the function `process_flag(trap_exit, true)` allows a process to catch an exit signal generated as a result of a linked process terminating. When the exit signal is received it is received in the processes mailbox as a standard message of the form `{EXIT, Pid, Reason}`.

Given that a process can trap an exit signal from another process but be assured that the other processes exit can in no way impact its own functioning we can implement "supervision". Supervision means that one process can detect the death of another and take intelligent action about how to deal with the failure.

Processes are started by a supervisor in some prescribed manner and order. The supervisor can be told how to restart the processes with respect to one another in the event of a failure of any single process. This restarting of processes essentially brings us back to our base state, which we know to function properly.

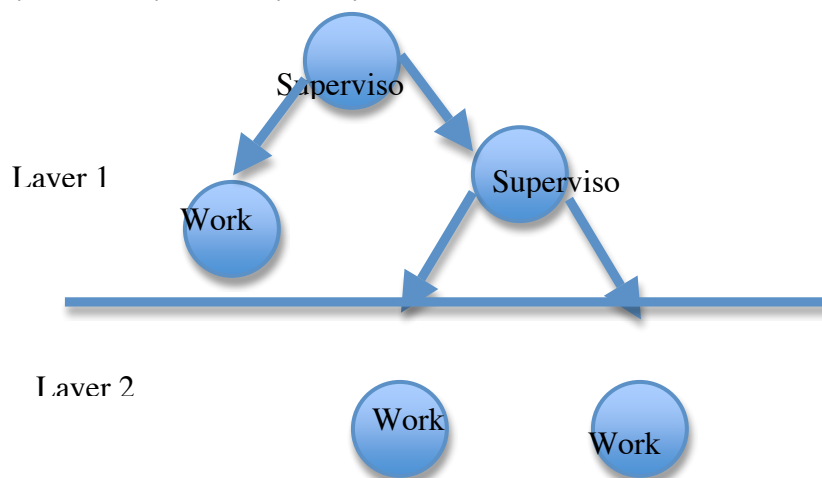


For Source Code, Free Chapters, the Author Forum and more information about this title go to <http://www.manning.com/logan/>

Think of it like rebooting your computer. Rebooting is a great way to clear up messes and restart from a point that is known to work. The problem with it is that it is not granular enough, ideally what you would like to be able to do is reboot just a part of it and the smaller the part the better. Erlang processes provide the mechanism for very granular “reboots”. Mapping out how your system is restarted is done through a powerful method known as layering.

Layering processes for fault tolerance

Layering is the process of bucketing related activities together. More importantly it is the process of defining working base states that we can restart to. In the diagram below you can see that there are two distinct groups of worker processes supervised separately from one another.



Group A is not required for the functioning of the processes in Group B. The processes in Group B work together to produce a stream of data that group A consumes. Let's say group B is processing and encoding multimedia data and group A presents it just to make things concrete. Let's further suppose that 15 percent of the data coming in to group B is corrupt in some way not initially predicted when coding the server. This audio data causes weird corruption inside one of the processes within group B. Because processes are isolated this corruption does not effect any other process outside the one being corrupted and because Erlang is fail fast that process dies immediately without so much as trying to untangle the strange mess that it's state has turned into. The supervisor realizing that a process has died restores the base state we prescribed for it and the system chugs on from a known point. The beauty of this is that the presentation system has no idea this is going on, and really does not care. So long as group B pushes enough good data up to group A for it to display something that is of acceptable quality to the user – we have a successful system a system that if not written with processes that fail fast and are intelligently layered would be a tangled crashing mess of exceptions.

So to take this back to the computer rebooting analogy what we have done is effectively isolate cooperation and dependence in our system and in so doing defined little kernels that can be powered down and then back up, all in fractions of a second, to keep our system chugging along even in the face of unpredicted errors. None of that would be possible if processes shared data and could corrupt one another. If that was the case we would have no

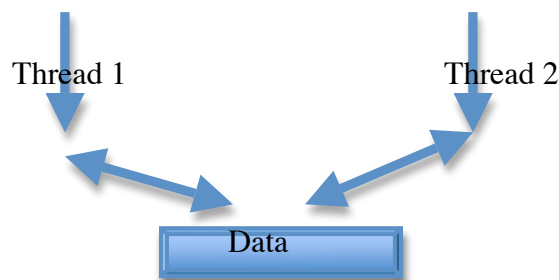
For Source Code, Free Chapters, the Author Forum and more information about this title go to <http://www.manning.com/logan/>

guarantee that group B's corruption would not adversely affect the operation of group A and layering would not make much sense.

The isolation of errors on a single machine is certainly great for improving fault tolerance. As we have seen the properties of fail fast and isolation can give rise to layering which lets a system achieve a potentially acceptable quality of service even in the face of errors from bad programming, bad data, or is most often the case, bad requirements. However, there are some things that we have to share when we are on a single machine, like the memory that machine possess in total, it's hard disk, and it's very circuitry, and perhaps most significantly a single plug into a single outlet! If one of these things goes wrong (or is unplugged) no amount of layering or processes will save us from the inevitable zero quality of service we will be delivering. This idea brings us nicely to our next topic, which is distribution. So far we have covered processes and how fault tolerance is achieved, now we are going to cover distribution a natural and fundamental feature of Erlang. Distribution is the feature that is ultimately what is going to allow us to scale as well as achieve the highest levels of fault tolerance.

Distributed Erlang

Erlang distributes very naturally due to the properties of the language. Share nothing copy everything lends itself to natural distribution. Take for example two threads running happily, and sharing memory between them as a communication method as pictured in the diagram below.



This is wonderful, and quite efficient, well until you want to move thread B onto another machine. Now the programmer is typically forced to alter his code in order to take into account the completely different communication mechanism he needs to utilize in this new context. Share memory simply does not work over the network. Erlang does not suffer from this problem. Since data is never shared but always copied via messaging processes on a single machine communicate just as well and in the same manner as processes on a remote machine.

At one employer we had quite a number of different Erlang applications running on our network. We probably had at least 15 distinct types of self-contained OTP applications that all needed to cooperate to achieve a single goal. Integration testing this cluster of 15 different applications, running on 15 different VMs would not have been the most convenient undertaking. With not a single line of code changed we were able to simply invoke all of the applications on a single VM and test them. They communicated with one another on that single VM in exactly the same manner with exactly the same syntax as when they communicated across the network. This is due to a concept known as location transparency. Location transparency means that when I send a message to a process through its process ID or other unique identifier I do not have to care where that process is located, as long as it is running the message will be delivered into its mailbox.

Think about this in terms of your current scaling problems. How much easier does scaling become if you can move processes off one machine and onto a number of others without having to materially alter your code? How much more fault tolerant does your system become when you can easily run redundant applications so that the failure of any one can be picked up by a running neighbor?

We have just seen the power of the fundamental attributes of Erlang/OTP programming.