CONCEPTS OF CONCURRENCY

Back in the day, Erlang's development as a language was extremely quick with frequent feedback from engineers working on telephone switches in Erlang itself. These interactions proved processes-based concurrency and asynchronous message passing to be a good way to model the problems they faced. Moreover, the telephony world already had a certain culture going towards concurrency before Erlang came to be. This was inherited from PLEX, a language created earlier at Ericsson, and AXE, a switch developed with it. Erlang followed this tendency and attempted to improve on previous tools available.

Erlang had a few requirements to satisfy before being considered good. The main ones were being able to scale up and support many thousands of users across many switches, and then to achieve high reliability—to the point of never stopping the code.

Scalability

I'll focus on the scaling first. Some properties were seen as necessary to achieve scalability. Because users would be represented as processes which only reacted upon certain events (i.e.: receiving a call, hanging up, etc.), an ideal system would support processes doing small computations, switching between them very quickly as events came through. To make it efficient, it made sense for processes to be started very quickly, to be destroyed very quickly and to be able to switch them really fast. Having them lightweight was mandatory to achieve this. It was also mandatory because you didn't want to have things like process pools (a fixed amount of processes you split the work between.) Instead, it would be much easier to design programs that could use as many processes as they need.

Another important aspect of scalability is to be able to bypass your hardware's limitations. There are two ways to do this: make the hardware better, or add more hardware. The first option is useful up to a certain point, after which it becomes extremely expensive (i.e.: buying a super computer). The second option is usually cheaper and requires you to add more computers to do the job. This is where distribution can be useful to have as a part of your language.

Anyway, to get back to small processes, because telephony applications needed a lot of reliability, it was decided that the cleanest way to do things was to forbid processes from sharing memory. Shared memory could leave things in an inconsistent state after some crashes (especially on data shared across different nodes) and had some complications. Instead, processes should communicate by sending messages where all the data is copied. This would risk being slower, but safer.

Fault-tolerance

This leads us on the second type of requirements for Erlang: reliability. The first writers of Erlang always kept in mind that failure is common. You can try to prevent bugs all you want, but most of the time some of them will still happen. In the eventuality bugs don't happen, nothing can stop hardware failures all the time. The idea is thus to find good ways to handle errors and problems rather than trying to prevent them all.

It turns out that taking the design approach of multiple processes with message passing was a good idea, because error handling could be grafted onto it relatively easily. Take lightweight processes (made for quick restarts and shutdowns) as an example. Some studies proved that the main sources of downtime in large scale software systems are intermittent or transient bugs (source). Then, there's a principle that says that errors which corrupt data should cause the faulty part of the system to die as fast as possible in order to avoid propagating errors and bad data to the rest of the system. Another concept here is that there exist many different ways for a system to terminate, two of which are clean shutdowns and crashes (terminating with an unexpected error).
Here the worst case is obviously the crash. A safe solution would be to make sure all crashes are the same as clean shutdowns: this can be done through practices such as shared-nothing and single assignment (which isolates a process' memory), avoiding locks (a lock could happen to not be unlocked during a crash, keeping other processes from accessing the data or leaving data in an inconsistent state) and other stuff I won't cover more, but were all part of Erlang's design. Your ideal solution in Erlang is thus to kill processes as fast as possible to avoid data corruption and transient bugs. Lightweight processes are a key element in this. Further error handling mechanisms are also part of the language to allow processes to monitor other processes (which are described in the Errors and Processes chapter), in order to know when processes die and to decide what to do about it.
Supposing restarting processes real fast is enough to deal with crashes, the next problem you get is hardware failures. How do you make sure your program keeps running when someone kicks the computer it's running on?Although a fancy defense mechanism comprising laser detection and strategically placed cacti could do the job for a while, it would not last forever. The hint is simply to have your program running on more than one computer at once, something that was needed for scaling anyway. This is another advantage of independent processes with no communication channel outside message passing. You can have them working the same way whether they're local or on a different computer, making fault tolerance through distribution nearly transparent to the programmer.

Being distributed has direct consequences on how processes can communicate with each other. One of the biggest hurdles of distribution is that you can't assume that because a node (a remote computer) was there when you made a function call, it will still be there for the whole transmission of

the call or that it will even execute it correctly. Someone tripping over a cable or unplugging the machine would leave your application hanging. Or maybe it would make it crash. Who knows?

Well it turns out the choice of asynchronous message passing was a good design pick there too. Under the processes-with-asynchronous-messages model, messages are sent from one process to a second one and stored in a *mailbox* inside the receiving process until they are taken out to be read. It's important to mention that messages are sent without even checking if the receiving process exists or not because it would not be useful to do so. As implied in the previous paragraph, it's impossible to know if a process will crash between the time a message is sent and received. And if it's received, it's impossible to know if it will be acted upon or again if the receiving process will die before that. Asynchronous messages allow safe remote function calls because there is no assumption about what will happen; the programmer is the one to know. If you need to have a confirmation of delivery, you have to send a second message as a reply to the original process. This message will have the same safe semantics, and so will any program or library you build on this principle.

Implementation

Alright, so it was decided that lightweight processes with asynchronous message passing were the approach to take for Erlang. How to make this work? Well, first of all, the operating system can't be trusted to handle the processes. Operating systems have many different ways to handle processes, and their performance varies a lot. Most if not all of them are too slow or too heavy for what is needed by standard Erlang applications. By doing this in the VM, the Erlang implementers keep control of optimization and reliability. Nowadays, Erlang's processes take about 300 words of memory each and can be created in a matter of microseconds—not something doable on major operating systems these days.

To handle all these potential processes your programs could create, the VM starts one thread per core which acts as a *scheduler*. Each of these schedulers has a *run queue*, or a list of Erlang processes on which to spend a slice of time. When one of the schedulers has too many tasks in its run queue, some are migrated to another one. This is to say each Erlang VM takes care of doing all the load-balancing and the programmer doesn't need to worry about it. There are some other optimizations that are done, such as limiting the rate at which messages can be sent on overloaded processes in order to regulate and distribute the load.

All the hard stuff is in there, managed for you. That is what makes it easy to go parallel with Erlang. Going parallel means your program should go twice as fast if you add a second core, four times faster if there are 4 more and so on, right? It depends. Such a phenomenon is named *linear scaling* in relation to speed gain vs. the number of cores or processors (see the graph below somewhere).

Source : http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency