Reliability with Erlang

Steve Vinoski • *Verivue*

In the last issue, I discussed the concurrency features of Erlang, a programming language created at Ericsson more than 20 years ago for implementing telecommunications systems with stringent reliability, distribution, and uptime requirements. I showed two different Erlang implementations of a function for raising a number to a power — one recursive and one based on potentially thousands of Erlang processes, which are akin to user-space threads. Users of popular languages such as Java or C++, in which threads are on the heavy side because they're tied to the underlying operating system kernel, are typically surprised by Erlang's capacity to rapidly spawn so many lightweight processes.

As we'll see in this column, Erlang's concurrency primitives provide more than just a fast way to create threads. They also enable parts of an application to monitor other parts — even if they're running on separate hosts across the network — and restart those other parts should they fail. Erlang's libraries and frameworks take advantage of these capabilities to let developers build systems with extreme availability and reliability.

Hang On

Before moving on to explore more of Erlang, let's reexamine one of the examples I used last time. Kenny MacDermid of KMD Consulting emailed me to point out that making my recursive pow/2 function tail recursive would improve its performance. A tail-recursive function simply calls itself at the very end, or tail, of its function body, thus allowing the interpreter or virtual machine underneath to optimize away stack growth and other overhead encountered with normal function calls. MacDermid's version of pow/2 uses an accumulator parameter to avoid tying the multiplication to the recursive call, as my original example did:

```
-module(pow).
-export([pow/2]).

pow(N, M) -> pow(N, M, 1).

pow(_, 0, Acc) -> Acc;
pow(N, M, Acc) -> pow(N, M-1, Acc*N).
```

Here, pow/2 (the first version of pow shown, with two arguments) just calls the second form of pow/3 and passes 1 for the initial value of accumulator Acc. The pow/3 function accumulates the answer in Acc, continually multiplying the accumulator by *N* to obtain the next accumulator value to pass to the next recursion. Finally, the exponent argument *M* reaches zero, matching the first form of pow/3, which simply returns the accumulator's value as the final answer.

This might seem an inconsequential difference, but it's definitely not. I previously found that calculating $50^{11,500}$ or greater using recursion was slower than my multiprocess pow/2 function. The multiprocess version uses message passing among exponent+1 Erlang processes rather than recursion to perform the same calculation. MacDermid's version, however, outperforms the multiprocess version up to about $50^{54,000}$, above which the multiprocess approach still runs faster — at least on my MacBook Pro.

His version obviously changes my detailed results, and I thank him for reminding me of the performance benefits of correctly using tail recursion. Fortunately, though, these results don't change the overall idea behind my original examples: process creation and interprocess communication in Erlang are far faster and cheaper than the multithreading approaches used with popular imperative languages such as Java and C++. As I mentioned earlier, the fact that Erlang processes are so inexpensive to create and destroy is critical to its support of highly reliable, long-running systems.

Reliability

Erlang was born of the need to create highly reliable telecommunications systems, for which the maximum allowable downtime — often mandated by law — typically amounts to just a few minutes per year, including the time required for upgrades. Obviously, the typical approach to fixing bugs or adding enhancements by taking the system offline, reinstalling it, and restarting it with a new version of software simply doesn't cut it in such environments. Unplanned outages due to faults and crashes can use all allowable downtime in one shot, potentially even resulting in governmental fines due to customers losing their service.

Given that middleware and enterprise-integration deployments often serve critical data for enterprises or help provide the back end for their public Web presence, reliability and availability for these systems can be quite important as well, though not to the same degree as in telecommunications environments. For example, very large enterprises often have multiple data centers around the globe to help maximize reliability and uptime. Maximum annual downtime for enterprise services is typically measured in hours or even days rather than mere minutes or seconds, but no one who owns or maintains such systems would complain if their reliability and availability could approach that of critical phone systems, as long as the costs for achieving it were reasonable.

Erlang has several important qualities that help address reliability concerns:

- *Fast, inexpensive process creation and tear-down.* As my September/October column showed, Erlang can create and destroy thousands of processes in the blink of an eye. When they're this cheap, launching many short-lived processes that carry out relatively small tasks and then go away is easy. Their brief existence means that such processes don't build up error-prone state or needlessly tie up more and more system resources over time. Should any processes crash or die, we can easily and quickly replace them with new ones.
- *Linkage between processes.* When an Erlang process spawns another process, it can ensure that links are established to immediately notify the spawning process when action is needed if the spawned process exits unexpectedly.
- *Transparent distribution.* Because of problems such as latency and disruption due to unexpected network partitioning, distribution can never be fully transparent, but Erlang's distribution capabilities get pretty close. Primarily, this is because the language included them in the design from the start, rather than as a bolt-on afterthought. Erlang's process-spawning capability makes creating processes on other hosts across the network just as easy as creating them locally.
- *Live upgrade capabilities.* Erlang supports mechanisms that let system operators and maintenance personnel load modules into running systems to replace modules that are currently in use.

Layered on top of the Erlang language is a framework called the Open Telecom Platform (OTP), which uses these features to help enable reliable systems. Despite the word "telecom" in its name, OTP is a general-purpose framework that's useful for applications in a variety of domains.

I want to make it clear that Erlang and OTP aren't magical — they won't automatically make your software extremely reliable. Creating reliable systems with Erlang/OTP still requires knowledge, experience, solid code, thorough testing, and general attention to detail. Nevertheless, because the language was designed with reliability as a foremost concern, the combination of Erlang and OTP definitely has advantages over other common languages when it comes to reliable systems.

Supervisor Trees

Last time, we saw how easy it is to create a new process in Erlang: you call `spawn/1`, passing in the function that the new process should run. Calling `spawn/1` makes the new process independent of the process that creates it. If the new process dies unexpectedly, for example, the creating process receives no notification of the event. However, Erlang also provides the `spawn_link/1` function, which "links" the new process to the creating process such that, should the new process die unexpectedly, the creating process also dies.

Based on that description, `spawn_link/1`'s behavior doesn't seem all that useful. When combined with Erlang's *exit-signal trapping* feature, however, the result is a powerful monitoring and restart mechanism. The following line of code enables exit signal trapping:

```
process_flag(trap_exit, true).
```

In Erlang terms, a process that enables exit-signal trapping becomes a system process. Enabling exit-signal trapping before invoking `spawn_link/1` means that, rather than crashing if a new process crashes, the system process receives a message describing the defunct process's exit status. Depending on the reason for the process's death, the system process might take different actions, such as logging an error message, sending an event message to another process, or starting a new process to replace the one that died.

By using process-linking capability, developers can design applications so that *supervisor* processes monitor all the important processes — those that do the actual work. OTP directly supports the creation of *supervisor trees* — hierarchies of supervisor and worker processes — via its supervisor module, and OTP's support for reliable applications depends heavily on these trees.

When supervisor trees are combined with Erlang's ability to easily spawn processes on other nodes within a distributed system, the result is a substantial yet straightforward foundation for long-running, reliable, fault-tolerant applications. The distribution transparency of Erlang's process-spawning capabilities means supervisors receive exit signals from worker processes even if the workers run across the network on different hosts. By spreading processes across nodes, your application keeps running even if some of the machines crash or shut down. And with appropriate network redundancy in place between the nodes, your application can continue even if the network breaks.

Over the years, I've helped design and build several distributed fail-over and redundancy frameworks in C++ and Java, and I know from hard-won experience that getting it right can be extremely difficult and time-consuming. Much of the difficulty stems from the fact that in such frameworks, the dependencies between the foundation code and the application-specific code are such that problems in either area can bring the whole system down. In other words, even if you write a highly robust framework, it can still be brought down by one rogue piece of application code. Conversely, writing robust applications is nearly impossible if the underlying framework isn't fully robust and reliable. Either way, this phenomenon is due to the fact that the boundaries between the application and the framework in languages like Java and C++ essentially disappear at runtime, and one bad pointer, missed exception, or deadlock or livelock problem can hang or crash the whole application. Erlang/OTP, on the other hand, does an excellent job of separating the framework from the application. Together with supervisor trees, this separation greatly reduces the complexity inherent in reliable distributed applications.

Live Upgrades

When a system's availability is so critical that taking it offline for fixes and upgrades simply isn't viable, there aren't many alternatives. Ultimately, such systems require the ability to perform live upgrades, in which the software is modified as the system keeps running.

Note how redundancy can help here: if the system is composed of multiple replicas, then you first shift any users or connections off each replica and onto one of the others. You then take the newly unloaded replica offline, upgrade it, and bring it back online. This straightforward approach is not unlike what happens during automatic fail-over, but it has some things to watch out for. For example, replicas typically talk to each other continually to ensure consistency. When you bring up a newly upgraded replica, you have to ensure that it can still correctly exchange messages with the replicas that have yet to be upgraded. This requires that you either avoid changing your message structures as part of your upgrades or, more likely, that you version your messages properly. You also have to ensure that taking replicas down doesn't increase the load on the remaining replicas so much that they falter or fail.

OTP includes significant support for in-service software upgrades. It provides for loading revised modules as well as adding and deleting modules at runtime. It also provides versioning support for applications to help ensure that versions V and V+1 (or versions V and V-1, depending on your perspective) can seamlessly and correctly interact. Unfortunately, these features' richness prevents me from fully describing them in this column space, but you can refer to the OTP design principles documentation for more information (www.erlang.org/doc/design_principles/part_frame.html).

As I hinted last time, the more I learn about Erlang, the more I wish I'd found out about it a decade ago. I've spent much of my career devising ways to support the development and execution of concurrent, distributed, fault-tolerant middleware and applications; having Erlang in my toolkit 10 to 15 years ago could have saved me significant development time and prevented numerous headaches. Many frameworks claim to take all the worry out of such applications and let the developer focus only on the business logic, but Erlang and its OTP framework comes far closer to that ideal than any other framework I've seen. It handles the difficult parts of concurrency, distribution, and reliability with relative ease.

If you develop enterprise-integration or middleware applications that require high reliability, I'll offer the same advice I gave last time: go get yourself a copy of Joe Armstrong's book, *Programming Erlang.*[1] This book is very readable and is suitable for both beginners and experts alike. It will open your eyes to a better way of building reliable software.

References

1. J. Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.

Steve Vinoski is a member of the technical staff at Verivue. He is a senior member of the IEEE and a member of the ACM. Contact him at vinoski@ieee.org.