Editor: Steve Vinoski • vinoski@ieee.org

Concurrency with Erlang

Steve Vinoski • Verivue

In the last issue, guest columnist Rachid Guerraoui of EPFL discussed the difficulties of developing multithreaded systems for today's (and tomorrow's) multicore CPUs, and he presented the notion of "free objects," an extension of object-oriented (OO) programming, as a possible solution. Virtually everyone who has written a multithreaded library or application has had to deal with the mysterious program behavior and hard-to-find bugs that come with the territory. Despite a great deal of research and practice applied over the years in the hope of making things easier for developers, writing a concurrent application remains a difficult task for all but the elite developers Guerraoui identified as "concurrency aristocrats." It's not an exaggeration to say that writing correct multithreaded programs is beyond many programmers' technical abilities.

One of the primary reasons concurrency is so hard is that popular imperative programming languages such as Java and C++ essentially require state to be shared among threads. In such languages, program pathways provide access for reading and writing thread-independent state variables. With multiple threads running on multiple CPUs, more than one pathway can be active simultaneously, and without the appropriate safeguards in place, the threads can read partial values and overwrite each other's data, resulting in bogus values and application instability.

Programmers writing multithreaded applications in languages like Java and C++ spend much of their time determining what state is shared among threads and how best to protect its integrity within the running application. Finding all the shared state isn't always easy. Assuming the programmer can even find it, he or she must then possess the skills, experience, and patience necessary to determine the best way to serialize access to it. Adding to the difficulty is the locking granularity used to protect shared state. If locking is too coarse-grained, the application tends toward single-threading, because only one thread at a time can obtain the lock that predictably surrounds a large portion of the code. Such applications make poor use of multicore CPUs and tend to be slow. If locking is too fine-grained, on the other hand, the chances for deadlock increase greatly as different threads are increasingly likely to obtain locks in different orders. Getting it right takes time: fine tuning what state needs protection, and at what granularity, sometimes requires months of development time. Idioms, patterns, and frameworks can help out partially, but they introduce restrictions and trade-offs of their own. Ultimately, all it takes is one developer overlooking one small piece of shared state to cause a large system to crash in production — usually at the worst possible time for the most important customer.

One way to avoid the problems with shared state is to simply avoid it, but that's impractical in a language like C++ or Java. Doing so requires a combination of libraries or frameworks such as those based on actor models and message passing, as Guerraoui described. It also requires significant programmer discipline because the programming languages themselves can't do anything to help developers completely circumvent shared state. Rather than using the programming language as intended, the developer is forced to write in what's at best a one-off dialect or worse, a whole new one-off language, using the special idioms and frameworks designed to help avoid sharing state among threads.

A better way to avoid shared state is to switch to a programming language specifically designed to do exactly that. Erlang is one such language.

What Is Erlang?

Erlang is getting a fair bit of attention these days because of widespread concerns around how to

 1089-7801/07/$25.00 © 2007 IEEE

best utilize multicore CPUs. Although many now hearing of it for the first time think Erlang is a new language, it's actually been around for more than two decades. Erlang started life in the telecommunications industry at Ericsson, where its developers sought to create a language for building telecommunications switches. They required their language to help them build highly concurrent, fault-tolerant, highly available, and distributed services that supported live upgrades and ran with virtually zero downtime. In 1998, after years of development and honing to meet these stringent requirements, Erlang became open source software. You can get it at www.erlang.org.

My own attraction to Erlang arose from my continual research into what it takes to develop reliable and practical distributed computing and middleware-integration systems. I first heard about it nearly two years ago, and the more I looked into it, the more I found that Erlang provided canned solutions for vexing problems I'd spent a lot of time on over the years. For example, it's difficult to use traditional languages like C, C++, or Java on top of Unix, Linux, or Windows to write distributed systems and middleware that support concurrency, replication, load balancing, and automatic failover. In fact, it's so hard that even if you've done it before, you can still very easily get it wrong. Consequently, when building a new system, most developers tend to focus first on the nonreplicated, nonreliable version, seriously considering the really hard failover and load-balancing parts only after they get that working.

The more I looked into Erlang, the more I realized that, along with its Open Telecom Platform (OTP) libraries, it already included proven solutions for the hard areas of building reliable distributed concurrent software. After all, Erlang/OTP is what powers the Ericsson AXD301 ATM switch, which reportedly has an uptime of 99.9999999 percent (that's nine nines). Unfortunately, the more I learned about Erlang, the more I also had that sinking feeling about how much time I'd spent over the years trying to (re)invent the wheels that Erlang/OTP already provides.

Joe Armstrong, the language's principle inventor, just published a new book, entitled *Programming Erlang*,[1] which I believe is a game changer. It's accessible enough for the average developer, and yet it covers virtually every difficult problem you'll encounter when trying to develop distributed applications that exhibit high concurrency, fault tolerance, and reliability. It's both a beginner's guide to Erlang and an advanced bible for those interested in developing systems that can run with essentially no downtime.

Erlang 'processes,' which are essentially user-space threads rather than Unix processes or kernel threads, communicate only via message passing.

Programming in Erlang

As Armstrong explains, Erlang is a functional language that wholly embraces the "shared nothing" concept — for example, unlike in Java or C++, its "variables" can't be changed once they're bound to a value. In Erlang, the = operator supports pattern matching rather than traditional assignment. If the value on the right of the = matches the value on the left, then the statement succeeds. When an unbound variable appears on the left, Erlang sets its value to match what's on the right, thus making the assignment succeed. If the variable appears thereafter on the left of the = operator, the statement succeeds only if the right-hand side has the same value.

What's so important about single-assignment variables? Because Erlang variables are immutable, they don't need concurrency protection. The book also explains how single-assignment variables help with program correctness: if a variable has an incorrect value, there's only one place you need to look in your code to find out why.

Avoiding shared variables allows for higher degrees of program parallelization, assuming threads aren't too heavyweight. Erlang "processes," which are essentially user-space threads rather than Unix processes or kernel threads, communicate only via message passing. Because of concurrency's central role, Erlang's processes are very lightweight and cheap to create. For example, on my 2.33-GHz Intel Core 2 Duo MacBook Pro with 2 Gbytes of RAM, Erlang launches 1 million processes in 0.51 second. Contrast that with C++: on the same system, I can create only 7,044 threads in 1.3 seconds using the `pthreads` library, after which the system returns an error indicating a lack of resources. Java 5 can create more threads than C++, but it's just as slow; it takes 48.6 seconds to create and join 250,000 threads, and runs out of resources after creating 431,430 threads in 83.3 seconds.

As far as the language itself goes, Erlang is similar to other functional languages in that it encourages list processing and tail recursion. Consider, for example, a simple function that raises a number to a power (Erlang

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

pow(N, M) ->
    Pids = pow_spawn(self(),M),
    Vals = lists:map(fun(P) -> P ! N end, Pids),
    lists:foldl(
        fun(_,Total) -> receive X -> X*Total end end,
        1, Vals).

pow_spawn(Pid,0) ->
    [spawn(fun() -> receive _ -> Pid ! 1 end end)];
pow_spawn(Pid,M) ->
    [spawn(fun() -> receive X -> Pid ! X end end)
     | pow_spawn(Pid,M-1)].
```

Figure 1. Erlang concurrency. Implementing the `pow/2` *function using multiple Erlang processes and message passing eliminates the need for traditional error-prone multithreading constructs.*

already provides this, but we'll write our own just as an example):

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

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

Here, we define a module named `pow`, which contains a function named `pow/2` (Erlang function names include their *arity*, or number of arguments). First, consider the final line in the example; it defines `pow(N,M)` recursively. Its recursion is broken by the definition of `pow` just above it, where the second argument is 0. Here we see Erlang's pattern matching at work: when we pass any number for N and $M \neq 0$, we invoke the recursive function, but when M is 0, we know that raising a number to the 0th power always returns 1, which is just what the `pow(_,0)` version does. The underscore used in place of N indicates that we don't use its value in the calculation. Finally, the second line of this example simply exports the `pow/2` function for use outside the `pow` module.

Let's try this function in the interactive Erlang shell:

```
1> c(pow).
2> pow:pow(2,3).
8
3> pow:pow(4,5).
1024
```

The first line compiles the `pow` module (the `1>` out front is the shell prompt). The second and third lines invoke `pow:pow` (which has the form `module:function`) to calculate 2^3 and 4^5, respectively.

Now let's bring concurrency into the picture. Rather than calculate the result recursively, let's do it with processes, message passing, and list processing. The example in Figure 1 is admittedly contrived, but I'll use it anyway because it manages to cram some important concepts into just a few lines of code.

Here, the `pow` function operates by first spawning one process for each power to be raised, plus one. For example, if we wanted to raise 2 to the 3rd power, we'd get 4 processes. The spawning of the processes takes place in the `pow_spawn(Pid,M)` function, via a list comprehension that comprises a call to `spawn` as the head of the list and a recursive call to `pow_spawn` as the tail of the list, thus building a list of process IDs for spawned processes. To let those processes communicate with us, we give each the process ID of our initial process, represented in the code by `self()`. The processes simply wait to receive a message, and when they do, all but the last one created simply turn around and send the same messages to the process ID passed to them — our initial process. The last process, which is required in order to break the recursion in the `pow_spawn(Pid,M)` function, always sends the value 1 back to the initial process.

After spawning our processes, we use `lists:map` (provided by Erlang) to send each of them the value N, which is to be raised to the power M. The processes send us back a total of M values of N and one value equal to 1, all of which we store in a list. We then call `lists:foldl` (also provided by Erlang) to walk the list and multiply all the entries together into `Total` to get our answer.

This code shows several interesting features of Erlang:

- In just a few lines, Erlang supports the creation and coordination of multiple processes and their results.
- As with all functional languages, Erlang functions are first-class objects that can be passed as arguments, bound to variables, and returned from other functions. The `spawn` statements use anonymous functions to instruct each spawned process on what to do.
- A process can receive messages from other processes by simply calling `receive`. This function can also do pattern matching, so that multiple `receive` statements can cover all the message forms your process expects to receive. In fact, given the right arguments, `receive` can effectively parse message headers by matching expected values to what's actually received in various fields.

This form of matching can easily handle bit fields normally relegated to C++ or C.

- To send a message to another process, we just need to specify the target `process ID`, followed by `!` and the message to send; Erlang does the rest. This works regardless of whether the process is running locally or on another machine across the network. Creating such a process simply requires calling a different form of the spawn function, which takes the target node name as the first argument.
- Locks, condition variables, and other traditional multithreading constructs simply aren't necessary in any of this code.

On the same MacBook Pro described earlier, the original recursive code takes 4.45 msecs to calculate 50^{2000}, whereas this contrived approach performs the same calculation in 16.84 ms. Considering that the calculation results in a number consisting of 3,398 digits, and that the latter approach spawns 2,001 processes, sends a message to and receives a response from each, and then walks the list of returned values to calculate the result, this performance disparity is completely reasonable. In fact, it's also far smaller than most developers would expect. Even more surprising is that as the exponent value increases, the contrived multiprocess approach's performance starts to achieve parity with that of the recursive version, eventually exceeding it when the exponent gets large enough — about 50 to the 11,500th power on my MacBook Pro.

Have you ever devised what you thought was a good solution to a really difficult problem, only to find that the answer would've been almost trivial with a different tool or approach? As Guerraoui pointed out last time, many developers are comfortable with OO programming. I'd like to advise such developers not to let Erlang's functional nature scare you away, but I know that many will ignore me rather than use a system that's purpose-built for developing robust, reliable, and highly concurrent programs. Instead, they'll continue to plod along trying to solve their problems with the same low-level approaches that have already proven many times over to be highly error-prone.

This column barely scratches the surface of Erlang. What I've shown in this limited space is that Erlang concurrency is extremely straightforward and inexpensive. Next time, I'll cover more of the language, focusing on the OTP libraries and their features that enable the development of software systems with virtually zero downtime. Meanwhile, do yourself a favor and get a copy of *Programming Erlang*. Even if you never write a single line of production Erlang code, reading and understanding this excellent book and the language it describes will make you a better developer.

Reference

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

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