Erlang

Rebecca Bailey and Ian Ryan

Due: May-14-2008

1 Erlang: Dead and Loving It

Like the lowly mammal before it, Erlang is a programming language who's time has finally come. Created over twenty years ago in response to needs in telecommunications, the world passed it by, a funny concurrent functional language - a dead end evolution with a small target audience. Computing was obsessed with faster processors and object-oriented programming. Erlang persisted, hiding away in telecommunications software, until the day it would be rediscovered, by a new generation of programmers searching for a language uniquely suited to the demands of modern programming.

Throughout this programming language class, we've seen many examples of languages are that well suited for one task, but perform very poorly at another. Programmers need to learn multiple languages, so that they have a good tool at hand suited for any task. Every programmer learns an object-oriented language in his repertoire, Java or Ruby or any of a dozen others in their first years of college. Everyone knows an imperative language like C. Most have worked in a declarative language, such as LISP, or Prolog. Each of these programming styles has uses for which they are best suited. They each are great to have in your toolbox for real world applications.

Erlang's concurrency-oriented programming is a different paradigm than any other, and will have much use in the near future. The changing parameters of the computing world today, specifically the increasing need for web server extensibility, has created an environment that the unique traits of Erlang can really shine.

2 History of Erlang, Part 2

The 1980's were a heady time in computing. The PC was gaining traction in households, businesses were able to afford desktop machines for the first time, and somewhere Al Gore was creating the internet. Hardware costs were dropping, and programming costs were rising. The Swedish telecommunications company Ericsson needed a new programming language, to increase the productivity of their programmers. After several years of testing diverse languages the Ericsson Computer Science Lab determined that the target language must be a high level symbolic language, and have powerful concurrency primitives, as well as robust error recovery features. The only problem was that no current language met these requirements.

Over the course of the next several years, Ericsson developed Erlang. It is a functional language, with powerful pattern-matching syntax. This makes it incredibly easy to program. As a declarative language, the focus is on defining the problem to be solved, not the nuts and bolts of how to solve it. This lets programmers rapidly prototype and develop solutions. The size of code for an application written in Erlang is much smaller than the size of an equivalent application written in an imperative language like C.

Erlang was released as open source in 1998. Since then it has started to gain steam as a more common programming language. Users such as Amazon, Nortel and T-Mobile rely on Erlang to power their systems. Applications using Erlang are as varied as email systems, electronic payment systems and software testing tools.

Simply calling it Erlang is a bit of a misnomer. Properly most applications are written using the OTP libraries, which are part of the open source release. Also featured in the open source release is the run time environment, along with many useful tools for programming concurrently.

3 The Erlang Movie

Erlang has several features which make it uniquely capable as a language for writing massively scalable applications, both from a sheer programming standpoint and a durability/maintainability standpoint.

Creating concurrent processes in most languages is a difficult task; in Erlang, it is trivial. Figure 1 shows an example of spawning a new process. As can been seen, the creation of a new process uses the *spawn* primitive.

```
start_server() ->
        register(srvr, spawn(eggs, server, [ [] ] )),
        server_node() ! initiate.
```

Figure 1: Starting a new process

The concurrency model that Erlang employs is a great example of forward thinking. Other concurrent languages developed during the same time period as Erlang used shared state to communicate between processes. Erlang's creators chose to go with asynchronous message passing instead, using the actor method. Under the actor method, a process can do several things on receiving a message. It can spawn a finite number of new actors, creating new processes; it can send new messages to a finite number of other actors; it can make decisions in its own process; or it can determine how to handle the next message. In figure 2, the process passes the *initiate* message to the *server_node* process.

There are many benefits of using a message passing system instead of shared state. One such benefit is that Erlang does not need variable locks. Languages with shared state need to lock variables to ensure concrruent processes do not have side effects or draw from corrupted data. Erlang on the other hand needs no such thing. Since Erlang does not support mutable data structures, correct programming techniques are enforced, and so processes are side-effect free. This makes Erlang processes very easy to parallelize.

While the inability to change variables might seem completely inconvient, the benefits outweight the cost. Processes cannot modify data, but can only consume it. As an attempt to offset the inconvenience of this feature, the creators of Erlang included easy ways to create a copy of a variable and make required modifications while you copy the variable.

As mentioned previously, one of the goals of the creators of Erlang was for the new language to have robust exception-handling. This is an area that it excels in. An Erlang application is constructed of many small, independently running processes. When a new process is spawned, it sends a linking message to its controlling process. By doing so, this makes it easy to create a graph of processes. This is invaluable for error recovery, as we'll see in a moment.

```
{login_attempt, user_already_logged_in} ->
        io:format("client: You are already logged in.
        Please try again later or choose a different
        User Name. ~n"), self() ! terminate;
```

Figure 2: Exiting a process on an error message

When a process encounters an error and exits, the process exits cleanly, and a message is sent to the controlling process. The controlling process then handles the error message, and the application can continue running.

By using the graph of processes, its easy for the application to trace the graph, searchin for processes which were effected by the error. Once found, these processes are easy to restart or resurrect. Other processes in the application that are independent of this error will continue running just fine.

Exception handling is one important aspect of reliability. Erlang has a second way to increase uptime for applications. Rather uniquely, Erlang allows modules to be changed while the application is running, without disturbing other unaffected portions of the system. This is very helpful for handling bug fixes, so that servers aren't down. Both the old version of a module and the new version of a module can co-exist in an application until the codeswitch atom is invoked. When codeswitch is called, the interpreter searches for a different version of the module, and replaces the current version with the new. This makes it possible to upgrade software without taking down applications or servers.

Many of the features that make Erlang good are inherited from the parent class of functional languages. Like most functional languages, Erlang does not allow standard iteration through for or $while$ loops. Instead, iteration is achieved through the use of recursion. By doing so, a programmer must break down a problem into small subproblems, simplifying complex programming goals. This is just one of the features that lets Erlang applications be smaller than applications written in other languages.

As a functional language, functions are first-class citizens in Erlang. They can be passed as arguments to other functions, returned, or even bound to variables. This makes writing a generic quicksort for undetermined variable types trivially easy.

Function names can be overloaded, if the second function has a different arity, or number of arguments, then the first function. Erlang views the functions as completely separate, with no relation. An example can be seen in figure 3.

```
-module(math).
-export([mult/1, mult/2]).

mult(X) ->
        X * X.
mult(X,Y) ->
        X * Y.
```

Figure 3: Overloading function names

Erlang is completely dynamically typed. Any value can be bound to any variable. This isn't to say that Erlang have no type system. There exist a small number of primitive types: atoms, tuples, functions and several others. Compound variables are created from the tuple type. Erlang supports limited OO programming through tuples, where the first field in a tuple is the name of the "class." Erlang does not support advanced OO features.

Similar to other functional languages, variable names in Erlang can be seen only by the local function. There are no global variables allowed in Erlang. Variable names don't need to be declared in any way before use.

```
> M = 1.
1
> N = "hello".
"hello"
> M < N.
true
```

Figure 4: Mismatched types

One interesting feature of Erlang's type system is that you can use boolean comparison between types. Figure 4 show a comparison between an integer and a string.

Many of the applications Erlang is used for have real time run-time requirements. Fast responses are required. The Erlang interpreter supports an incremental garbage collection routine, so there is never a point where garbage collection halts the response for long.

As mentioned earlier, Erlang has pattern matching syntax. Its easy to test a variable to match expected patterns, and return the result based on the matching. Erlang also allows matching to generic terms, like the * operator on a command prompt. This can be seen in figure 5.

4 Erlang Stinks

Given all the great things about Erlang, why hasn't it taken off since it was created 20 years ago? What made some of its comtemporary languages successful, while Erlang has been a backwater of development? Erlang has some serious flaws that keep it from reaching common usage.

```
fac(0) -> 1.
fac(N) -> N * fac(N-1)

fun({ClientID, _, _, _, _}) ->
```

Figure 5: Pattern Matching

One of the downfalls of Erlang is its archaic and often inconsistent syntax. Erlang is derived from LISP and Prolog, as discussed earlier, and takes the worst syntax from each of its predecessors. This leads to confusing syntax requirements, when viewed in comparison with a more modern example. For example, in Erlang an if statement has to have a true branch. If no branch evaluates to true, the process will throw an error message and exit. The inability to have if conditions without bothering with an *else* statement only means more coding for a programmer, but it is indicative of generally poorly thought out syntax.

The use of punctuation is also remarkably unfriendly, to a new Erlang programmer. The application of periods, commas, semi-colons is arcane at best. Half the statements have no terminating punctuation, further confusing the issue. Even a language as old as C has consistent statement termination.

Like many languages, Erlang is interpreted. Modern languages like Ruby and Python have excellent interpreters, even the old man LISP has a fairly flexible interpreter. Erlang's, on the other hand, doesn't allow composition of a function in the interpreter. A function must be written in another program, and then loaded into the interpreter. If you made a mistake, the process must be repeated.

Another problem with Erlang is that it is not actually side-effect free. There are primitives in the language that have side effects. While it is definitely possible to code side-effect free applications, the fact that Erlang is impure in this sense is troubling.

Finally, Erlang has shared state! One of the biggest draws to Erlang is its ability to easily (and without consequence) parallelize processes in applications because of the lack of shared state. But Erlang actually has a module called ets, which allow large amounts of data to be stored in the interpreter, and be accessed by multiple processes at the same time. As no data locks are provided, this can lead to data corruption with concurrent read/writes.

5 Erlang Strikes Back[1]

All the above is interesting in an educational or historical sense, but what makes it relevant to today's programming life? What brings a 20 year old language out of the books and into modern programming? There are certainly much more recent and trendy languages one could program in. As mentioned previously, Erlang is really uniquely suited to many of the new areas of application for computers in today's society.

Twenty years ago, the driving force for new computer hardware was speed. A faster clock was how manufacturers sold a new processor. There is a limit to how fast a clock can go before it starts losing integrity, and our chips appear to be nearing that limit. This has left the manufacturers in quite a bind over how to sell processors. They have adapted in the last several years, leaning on multi-core processors more and more. Performance of an Erlang application scales almost linearly with the number of processors. Combined with the rising number of processors on chips, Erlang is a clear candidate for becoming the dominant concurrent language.

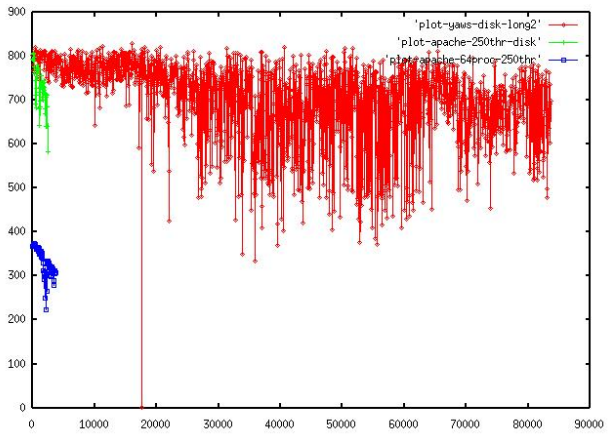

Figure 6: Apache vs YAWS

A second key factor that is leading toward Erlang ascendancy is the scalability of applications written in it. Creating new processes is simple and easy, with very low overhead. There are many examples of Erlang applications outperforming more standard language applications in high demand situations. YAWS (Yet Another Web Server), written in Erlang, outperforms other web server applications by factors of 10 on large traffic loads. The explosion of the internet in the last ten years is great news for Erlang. Figure 6 shows a YAWS web server (in red) compared to two variations of Apache, in terms of concurrent web sessions hosted.

[1]Yes, I know its not a Mel Brooks title.

6 Blazing Erlangs

In spite of its flaws, Erlang has a lot to offer as a programming language. Once you learn its quirky syntax and punctuation, the pattern matching lets you write easy-to-read and terse programs. The low overhead for spawning and maintaining new threads lets anything from a messenger service to a real-time database be programmed in Erlang.

While its impossible to predict the future accurately (flying cars are just around the corner, I swear), it seems likely that as concurrency in programming rises in importance and massive online applications continue to increase in demand, so too will Erlang rise as the dominant concurrent language.

References

Armstrong, Joe. "Apache vs. YAWS." http://www.sics.se/ joe/apachevsyaws.html

"Erlang (programming language)." Http://en.wikipedia.org/wiki/Erlang_programming_language

"Erlang is Icky." http://www.kimbly.com/blog/000057.html

"ErlyWeb: The Erlang Twist on Web Frameworks." http://erlyweb.com/

Eriksson, Klaus and Armstrong Williams. "Programming Rules and Conventions." http://www.erlang.se/doc/programming$_rules.shtml$

"FAQ About Erlang." http://www.erlang.org/faq/faq.html

Sadan, Yariv. "Erlang Does Have Shared Memory." 13 May 2008. http://yarivsblog.com/

Wegrzanowski, Tomasz. "My First Impressions of Erlang." http://t-a-w.blogspot.com/2006/09/my-first-impressions-of-erlang.html