Concurrency Oriented Programming in Erlang

Joe Armstrong
Distributed Systems Laboratory
Swedish Institute of Computer Science
joe@sics.se

17 February 2003

1 Introduction

The starting point for this paper is the observation that the real world, the world in which we live and breath and are born in and die is *concurrent*. Paradoxically, the programming languages which we use to write programs which interact with the real world are predominately sequential.

Using essentially sequential programming languages to write concurrent programs is difficult and leads to the notion that concurrent programming is difficult; indeed writing concurrent programs in a languages like Java or C++ which were designed for sequential programming is very difficult. This is not due to the nature of concurrency itself, but rather to the way in which concurrency is implemented in these languages.

There is a generation of languages, which includes Erlang[9] and Oz[6], in which writing concurrent programs is the natural mode of expression. We argue that modeling a real world application around the natural concurrency patterns that are observed in the application leads to code which is clear and easy to maintain and this has significant advantages for early adopters of the technology. Indeed several of the products developed by early adopters of Erlang have become market leaders in their commercial niches, some examples of these are given later in the paper.

This paper has a short overview of concurrent programming followed by a brief tutorial introduction to Erlang. Finally we describe a number of commercial applications that were written in Erlang.

2 Does your language have processes?

The notion of a process is well-known to programmers and computer scientists, unfortunately very few languages or operating systems implement processes in a way that makes then useful to the programmer.

Processes, for example, can be created in Java or C++, but the mechanisms for creating processes are just API's which thinly disguised the process architecture that exists in the underlying operating system.

In many languages the concurrency model of the language is the same as the concurrency model of the underlying operating system. If, for example, the underlying operating system uses a particular scheduling or time slicing policy, then this property will be inherited by the language. This also means that, for example, a concurrent Java program running on one operating system might have completely different semantics to a concurrent Java program running on a different operating system.

This dependence upon the underlying semantics of the operating system is particularly unfortunate since most widely used operating system do not support lightweight processes, nor do they allow very large numbers of processes to operate concurrently.

We argue that concurrency should be a property of the programing language and not a property of the operating system. We think that the concurrent behavior of a program should be the same on all operating systems, as regards all issues of synchronization, scheduling order, etc. We believe that the only difference in behaviour in moving a concurrent program from one machine to another is that the program will run faster on a faster machine etc; otherwise there should be no differences in behaviour which depend upon the operating system.

Operating systems which only allow a limited number of processes result in weird programing practices and in the belief that concurrent programming is “difficult”.

To illustrate this point we will use an analogy.

Suppose you could only create and use 3256 objects in Java - and that if you created more objects the system would slow down, grind to a halt and crash. Suppose you were told that using more than 1000 objects was very dangerous and that if you used more than 1000 objects your program would probably be very inefficient. Suppose that, in order to have 2000 objects it would be better to have 500 objects and use some form of 4-to-1 object merging scheme; suppose that if you wanted to use 2000 objects and have the program run efficiently you would have to have to re-write the garbage collector and implement your own object management system.

If Java had the above properties then you might come to the conclusion that object oriented programming was difficult and a bad idea. This would have nothing to do with the properties of objects *per se* and everything to do with the terrible way in which they were implemented.

To illustrate our contention that concurrency is poorly implement in most systems we have performed a number of measurements which show how long it takes to create a new process,[1] and how long it takes to send a message between a pair of processes. These measurements are performed for a number of different programming languages and the results are shown in the next two sections.

Process creation times

Figure 1 shows the time needed to create a process as a function of the total number of processes in the system. We observe that the time taken to create an Erlang process is a constant 1μs. up to 2,500 processes; thereafter it increases to about 3μs. for up to 30,000 processes. The performance of Java and C# is shown at the top of the figure. For a small number of processes it takes about 300μs. to create a process. Creating more than two thousand processes is impossible.

[1]To be strict, we should say a lightweight process in Erlang or a thread in Java or C#

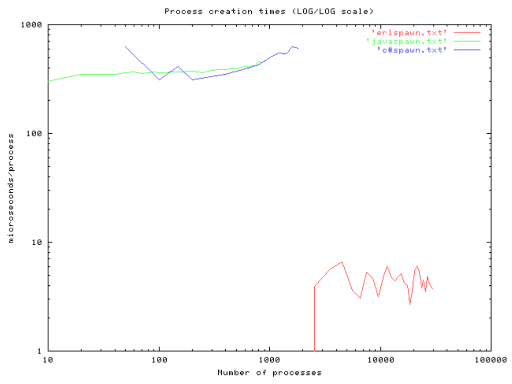


Figure 1: Process creation time

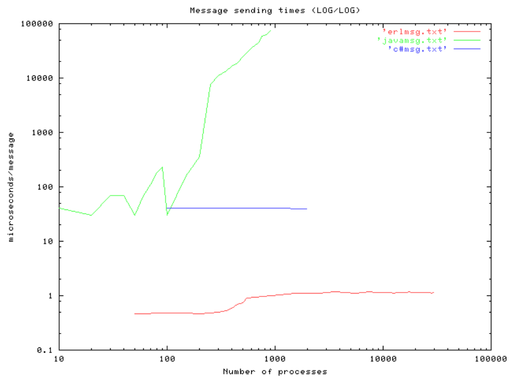


Figure 2: Process creation time

Message passing times

Figure 2 shows the time to send a simple message[2] between two concurrent processes running on the same machine as a function of the total number of processes. We see that for up to 30,000 processes the time to send a message between two Erlang processes is about 0.8μs. For C# it takes about 50μs. per message, up to the maximum number of processes (which was about 1800 processes). Java was even worse, for up to 100 process it took about 50μs. per message thereafter it increased rapidly to 10ms per message when there were about 1000 Java processes.

From the previous two sections we observe that process creation and message passing times in Erlang are one to two orders of magnitude faster than the equiva-

[2]in Java and C# this was the time to transfer private data between two threads using a synchronized method call.

lent operation on threads in Java or C#.

Processes in Java or C# are pretty much like the objects in the flawed object system which we described earlier - you can't have many of them, and if you have more than a few hundred processes the system will start mis-behaving and to do any real work with them you have to write your own scheduler and start combining multiple threads of control into single processes. All of this gives concurrent programming a bad name - and would probably make any sane person think that concurrent programming was a difficult and should be avoided whenever possible.

The opposite is true.

3 Concurrency Oriented Languages

An object oriented language is a language with good support for *objects.* A concurrency oriented language has good support for *concurrency.*

For a language to qualify as a "concurrent Oriented Language" the following criteria should apply:

- We should be able to create large numbers of processes.
- Processes creation and destruction should be an efficient operation.
- Message passing between process should be inexpensive.
- Processes should share no data and operate as if they ran on physically separated processors.

In Erlang processes are *light weight.* This means that very little computational effort is required to create or destroy a processes. Light-weigh processes in Erlang are one to two order of magnitude lighter than operating system threads.[3]

Not only are Erlang processes light-weight , but also we can create many hundreds of thousands of such processes without noticeably degrading the performance of the system (unless of course they are all doing something at the same time).

Erlang processes have *share nothing semantics* - sharing no data leads to highly efficient code. Traditionally concurrent programs used shared data, protected by semaphores of mutexs, that reason being (supposedly) to improve efficiency.

[3]Erlang programmers think of OS threads as terribly heavy weight objects.

Precisely this sharing of data leads to a number of problems which ultimately leads to performance degradation.

Firstly, and most obviously, the program cannot easily be divided to run on multiple processors should the need arise. If processes share data they cannot in any simple manner be changed to run on physically separated computers. Secondly, access of data through critical regions leads to over-synchronization of processes and in programs being more sequential than they need to be.

In Erlang, processes share no data and the only way in which they can exchange data is by explicit message passing. Erlang message never contain pointers to data and since there is no concept of shared data, each process must work on a copy of the data that it needs. All synchronization is performed by exchanging messages. The main reason for disallowing pointers in messages and for requiring processes to work with a copy of the data was to simplify programming fault-tolerant systems. Systems with distributed data containing "dangling" pointers are very difficult to program in the presence of hardware failures - we took the easy way out, by disallowing all such data structures.

Constructing systems, as opposed to individual programs, using the philosophy of share-nothing processes has several significant advantages:

- The system is easily distributable - to turn a non-distributed program into a distributed program can often be achieve by merely allocating the different parallel processes to different machines.
- The system is easily made fault tolerant - this can be achieved by arrangements of processes into "workers" and "observers." The worker processes perform computations, and the observer processes observe the workers and perform error recovery if anything goes wrong in a worker process. The worker and observer processes can run on the same machine (for local error recovery) or on physically separated machines (for building fault-tolerant systems).
- The system is easily scalable - this can be achieved by adding more processors and moving processes between processors.

3.1 A Web Server in Erlang

A web server is a typical application which benefits from concurrency. YAWS[4] is an web server written entirely in Erlang. A typical web server must manage large numbers of concurrent sessions - the so called "Keep Alive" persistent sessions of HTTP/1.1 benefit from the ability to handle large numbers of concurrent sessions. In an experiment involving a cluster of 16 machines we measured the performance of YAWS and Apache[1] under conditions of overload. We simulated a denial of service attack by loading the server with a large number of slow parallel sessions and then measured the throughput of the server for genuine traffic as we increased the number of attacking client processes.

Figure 3 shows the throughput of Apache and YAWS in KBytes/second as a function of the number of parallel sessions (overload) that the servers are subject to.

For low load Apache and Yaws perform equally well and the throughput is about 800 KBytes/sec.

As we increase the load up to up to 80,000 disturbing processes the throughput of the Erlang server remains essential unchanged. The Apache web server degrades slightly, but then crashes when it has to maintain more than about 4,000 parallel sessions. Recall also, as we showed earlier that both java and C# could not handle more than about 2,000 concurrent processes. What we are observing here is a limitation in the underlying operating system. The only way to avoid this is to re-implement scheduling and processes management in the run-time system for the language involved.

Erlang happily manages 85,000 processes with little observable degradation in performance - as far as the host operating system is concerned Erlang has only used one or two processes.

The difference in performance between the Erlang web server and the Apache system is dramatic. Under conditions of low load - Apache and Yaws have similar performance - under high load during a simulated denial of service attack Yaws outclasses Apache. Indeed, it proved impossible to break the Erlang server using 16 attacking machines - though it was easy to stop the Apache server.

This example, nicely demonstrates systems written in sequential languages (like C) on conventional operating systems perform badly in the presence of massive concurrency - whereas systems written in Erlang performs well.

[4]Yet Another Web Server[13]

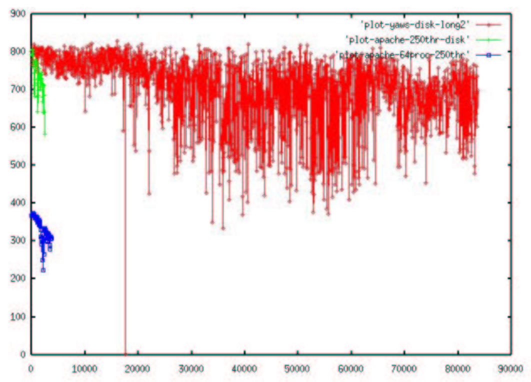


Figure 3: YAWS - an Erlang web server

3.2 Why is COP Nice?

We believe that concurrency oriented programming is desirable for a number of reasons:

- The world is parallel.
- The world is distributed.
- To program a real-world application we *observe* the concurrency patterns = no guesswork (only observation, and getting the granularity right).
- Our brains intuitively understand parallelism (think driving a car).
- Our programs are *automatically* scalable, have *automatic fault tolerance* (if the program works at all on a uni-processor it will work in a distributed network)
- Make more powerful by adding more processors.

The points need little explanation, the world is parallel and distributed - trying to model this behaviour in a sequential language is very difficult.

In teaching Erlang for programming real-world applications we emphasize the importance of observation. We try to observing the concurrent patterns in the application and the message passing channels and we map these in a 1:1 manner onto a set of Erlang processes and messages. This method takes the guesswork out of program design and replaces it by observation. Programs designed and written in this manner have a clear and logical relation to the problem that they are designed to solve - students are often surprised at how easy this is and soon learn that the observed concurrency structure of the problem drives the solution in a natural way.

Trying to force a naturally concurrent problem into a sequential framework is difficult and error prone.

We can speculate that are brains are especially suited for perceiving patterns of concurrency since we manage complex tasks involving hundred or thousands of parallel activities without conscious thought - if this were not possible everyday activities like driving a car would be impossible.

The final point, was discussed earlier. Scalability, and fault-tolerance are often achieved by a simply moving processes between processors - this is a consequence of the “share nothing” philosophy of program construction.

3.3 What is Erlang/OTP?

Erlang is a concurrent programming language with a functional core. By this we mean that the most important property of the language is that it is concurrent and that secondly, the sequential part of the language is a functional programming language.

The sequential sub-set of the language expresses what happens from the point it time where a process receives a message to the point in time when it emits a message. From the point of view of an external observer two systems are indistinguishable if they obey the principle of observational equivalence. From this point of view, it does not matter what family of programming language is used to perform sequential computation.

It is interesting to note that during it’s life Erlang started off with a logical core (Prolog[2]) which later evolved into a functional core. The functional core is a dynamically typed, strict, higher-order functional language and is, in it’s own right, a small and powerful programming language.

One of the surprising things about Erlang is that the functional core language is itself a useful programming language - so surprisingly even purely sequential applications written in Erlang often outperform applications in languages which were designed for purely sequential processing. Indeed, at the 2002 Erlang users conference Balagopalakrishnan and Krishnamachar[4] reported work at Lucent technologies showing that Erlang was six times faster than Perl for a number of applications.

OTP[7, 11] stands for Open Telecom Platform, OTP was developed by Ericsson Telecom AB for programming next generation switches and many Ericsson products are based on OTP. OTP includes the entire Erlang development system together with a set of libraries written in Erlang and other languages. OTP was originally designed for writing Telecoms application but has proved equally useful for a wide range of non-Telecom fault-tolerant distributed applications.

In 1998 Ericsson released Erlang and the OTP libraries as open source.

Since its release OTP has been used increasingly outside Ericsson for a wide range of commercial products, we will describe some of these later in the paper.

OTP represents the largest commercial use of a functional programming language outside academia.

4 Erlang in 11 minutes

The next few sections proved a simple introduction to Erlang through a number of examples, for more examples see[9].

4.1 Sequential Erlang in 5 examples

1 - Factorial

```
-module(math).
-export([fac/1]).

fac(N) when N > 0 ->  N * fac(N-1);
fac(0)            ->  1.

> math:fac(25).
15511210043330985984000000
```

2 - Binary Tree

```
lookup(Key, {Key, Val, _, _}) ->
  {ok, Val};
lookup(Key,{Key1,Val,S,B}) when Key<Key1->
  lookup(Key, S);
lookup(Key, {Key1,Val,S,B}) ->
  lookup(Key, B);
lookup(Key, nil) ->
  not_found.
```

3 - Append

```
append([H|T], L) -> [H|append(T, L)];
append([],    L) -> L.
```

4 - Sort

```
sort([Pivot|T]) ->
  sort([X||X <- T, X < Pivot]) ++
  [Pivot] ++
  sort([X||X <- T, X >= Pivot]);
sort([]) -> [].
```

5 - Adder

```
> Add = fun(N) -> fun(X) -> X + N end end.
#Fun
> G = Add(10).
#Fun
> G(5).
15
```

4.2 Concurrent Erlang in 2 examples

1 - Spawn

```
Pid = spawn(fun() -> loop(0) end)
```

2 - Send and receive

```
Pid ! Message,
.....

receive
  Message1 ->
    Actions1;
  Message2 ->
    Actions2;
    ...
after Time ->
  TimeOutActions
end
```

4.3 Distributed Erlang in 1 example

1 - Distribution

```
...
Pid = spawn(Fun@Node)
...
alive(Node)
...
not_alive(Node)
```

4.4 Fault tolerant Erlang in 2 examples

1 - Catch/throw

```
...
case (catch foo(A, B)) of
   {abnormal_case1, Y} ->
      ...
   {'EXIT', Opps} ->
      ...
   Val ->
      ...
end,
..
foo(A, B) ->
  ...
  throw({abnormal_case1, ...})
```

2 - Monitor a process

```
...
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
receive
  {'EXIT', Pid, Why} ->
    ...
end
```

4.5 Bit syntax in 1 example

Erlang has a "bit syntax" for parsing bit aligned data fields in packet data. As an example we show how to parse the header of an IPv4 datagram:

`Dgram` is bound to the consecutive bytes of an IP datagram of IP protocol version 4. We can extract the header and the data of the datagram with the following code:

```
-define(IP_VERSION, 4).
-define(IP_MIN_HDR_LEN,5).

DgramSize = size(Dgram),
case Dgram of
 <<?IP_VERSION:4, HLen:4,
  SrvcType:8,TotLen:16,ID:16,Flgs:3,
  FragOff:13,TTL:8,Proto:8,HdrChkSum:16,
  SrcIP:32,DestIP:32,Body/binary>> when
   HLen >= 5,4*HLen =< DgramSize ->
    OptsLen = 4*(HLen-?IP_MIN_HDR_LEN),
    <<Opts:OptsLen/binary,Data/binary>>
    = Body,
  ...
end.
```

4.6 Behaviors

Many common programming patterns[5] in Erlang can be captured in the form of higher-order functions.

For example, a universal *Client - Server* with dynamic code update can be written as follows:

```
server(Fun, Data) ->
  receive
    {new_fun, Fun1} ->
      server(Fun1, Data);
    {rpc, From, ReplyAs, Q} ->
      {Reply, Data1} = Fun(Q, Data),
      From ! {ReplyAs, Reply},
      server(Fun, Data1)
  end.
```

Here, the semantics of the server is completely determined by the function `Fun`. By sending a message of the form `{new_fun, Fun'}` the semantics of the server will change without having to stop the system.

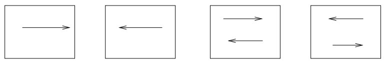
The server is accessed by calling the routine `rpc` which is as follows:

[5]Called behaviors in Erlang.

```
rpc(A, B) ->
  Tag = new_ref(),
  A ! {rpc, self(), Tag, B},
  receive
    {Tag, Val} -> Val
  end
```

4.7 Programming Simple Concurrency Patterns

How can we program common concurrency patterns? The following diagram shows the four most common concurrency patterns:



Reading from the left, these can be programmed as follows:

```
Cast
  A ! B

Event
  receive A -> A end

Call (RPC)
  A ! {self(), B},
  receive
    {A, Reply} ->
      Reply
  end

Callback
  receive
    {From, A} ->
      From ! F(A)
  end
```

These four concurrency patterns account for a large proportion of all programming.

Entire industries are built around the remote procedure call (the first of our patterns with two messages) - well known protocols like HTTP and SOAP are just disguised remote procedure calls with bizarre and difficult to parse syntaxes.

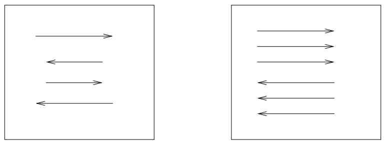
The callback model (the second pattern with two messages) leads to entire school of “callback” programming - commonly found in windowing systems.

4.8 Programming Complex Concurrency Patterns

The last section dealt with the four simplest concurrency patterns, it is tempting to ask how we should program complex concurrency patterns.

The following diagram illustrates two slightly more complex concurrency patterns.

The figure to the left shows a callback occurring *within* a remote procedure call, to the right is a *parallel* set of remote procedure calls (here we dispatch three queries and wait for the return values *which might come back in any order.*



These are simply programmed in Erlang. Callback within a PRC can be written:

```
A ! {Tag, X},  g(A, Tag).
g(A, Tag) ->
  receive
    {Tag, Val} -> Val;
      {A, X} ->
        A ! F(X),
        go(A, Tag)
  end.
```

and parallel RPC:

```
par_rpc(Ps, M) ->
  Self = self(),
  Tags = map(
    fun(I) ->
      Tag = make_ref(),
      spawn(
        fun() ->
          Val = rpc(I, M),
          Self ! {Tag, Val}
          end),
      Tag
    end, Ps),
  yield(Tags).

yield([]) ->
    [];
yield([H|T]) ->
    Val1 = receive
             {H, Val} -> Val
   end,
    [Val1|yield(T)].
```

4.9 Commercial application of Erlang

In 1998 Erlang and the OTP system was released into the public domain subject to an Open Source license. Since that time a number of different companies have adopted Erlang and used it to build commercial products.

The following major products are discussed:

- AXD301
- GPRS
- Nortel SSL accelerator
- Bluetail - *mail robustifier*

AXD301

The AXD301[5, 8] is a fault-tolerant carrier-class ATM (Asynchronous Transfer Mode) switch manufactured by Ericsson Telecom AB.

The AXD has 11% of the world market share for carrier class ATM switches making it the market leader in this market segment.

The measured reliability[10] is quoted as being 99.9999999% (9 nines) corresponding to a down time of 31 ms. year! - this makes it one of the most reliable switches ever made. Some of the techniques used to achieve this reliability are described in[12].

British Telecom use the AXD301 in their telephony and data backbone, the system is currently handling 30-40 million calls per week per node and is the world's largest telephony over ATM network.

The AXD301 has 1.7 million lines of Erlang, making it the largest functional program ever written.

Bluetail

Bluetail AB was founded in 1998 by the author and his colleagues to exploit the Erlang technology. Three months after we started our first product, the Bluetail mail robustifier[3], had been programmed and sold.

Bluetail's motto was *Bringing Reliability to the Internet* - we made a number of products designed to increase the reliability of existing Internet services. These products could be brought to market quickly (usually the entire product cycle was about three months) and were programmed predominantly in Erlang.

In 2000 Bluetail was acquired by Alteon Web Systems, and in the same year Alteon was acquired by Nortel Networks, but continued operations under its own name. Since 2000 Alteon have produced a number of dedicated servers written predominately in Erlang.

Alteon SSL Accelerator

Following the purchase of Bluetail AB by Alteon Web Systems the first product developed in Erlang was the Alteon SSL accelerator. This product rapidly became the most popular product[6] for dedicated SSL appliances 48 percent market share in the first half of 2002.

[6]according to Infonetics Research

GPRS

The Ericsson GRPS system is written mainly in Erlang - it is market leader having 45% of world market.

Minor products

In addition to the major products a number of minor commercial products are known to be under development - these range from traditional Telecoms applications to transaction based banking systems.

As of 14 February 22 Source source projects using Erlang were underway, together with a large number of individual projects.

References

[1] `http://www.apache.org/`.

[2] J. Armstrong, S. Virding, and M. Williams. Use of Prolog for Developing a New Programming Language. In C. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, 1992. Association for Logic Programming.

[3] Joe Armstrong. Increasing the realibility of email services. In *Proceedings of the 2000 ACM symposium on Applied computing 2000*, pages 627–632. ACM Press, 2000.

[4] Anand Balagopalakrishnan and Bagirath Krishnamachari. Helga - a call load generator written in erlang/otp. In *Erlang 2002 User Conference*. Ericsson, November 2002.

[5] Stafan Blau and Jan Rooth. Axd301 - a new generation atm switching system, 1998. Ericsson Review Number 1, 1998.

[6] Mozart Consortium. The Mozart Programming System, January 1999. Available at `http://www.mozart-oz.org/`.

[7] Ericsson. *Open Telecom Platform—User's Guide, Reference Manual, Installation Guide, OS Specific Parts*. Telefonaktiebolaget LM Ericsson, Stockholm, Sweden, 1996.

[8] Jan Höller. Voice and telephony networking over atm, 1998. Ericsson Review Number 1, 1998.

[9] C. Wikström J. Armstrong R. Virding and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, NJ, 1996.

[10] private communication.

[11] Seved Torstendahl. Open telecom platform, 1997. Ericsson Review Number 1, 1997.

[12] Ulf Wiger, Gösta Ask, and Kent Boortz. World-class product certification using erlang. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 24–33. ACM Press, 2002.

[13] Claes Wikström. Yaws - yet another web server. In *Erlang 2002 User Conference*. Ericsson, November 2002.