Troubleshooting a Large Erlang System

Mats Cronqvist
Ericsson Hungary
H-1300 Bp.3. P.O.Box 107
HUNGARY
+3614377169
mats.cronqvist@ericsson.com

ABSTRACT

In this paper, we discuss some experiences from a large, industrial software project using a functional programming language. In particular, we will focus on programming errors.

The software studied is the AXD 301 (a multi-service switch from Ericsson AB [1]) control system. It is implemented in a functional language Erlang [2]. We will discuss how this affects programmer productivity.

There are now well over 1,000 AXD 301's deployed. Even though a properly handled AXD 301 is quite reliable, there exists a great deal of knowledge about problems that do occur in production code. We will analyze what kinds of programming errors cause these problems, and suggest some methods for preventing and, when that fails, finding the errors. We will also describe some tools that has been specifically developed to aid in debugging.

One (perceived) problem with using a interpreted, functional language is execution speed. In practice, we have found that the overhead of running in an emulator is not dramatic, and that it is often more than compensated for by the advantages. The expressiveness of the language and the absence of low-level bugs means that programmers have more time to spend on tuning the code. And since the emulator has good support for tracing, one can perform very advanced profiling, thus making the code intrinsically more effective. We will discuss a profiling tool developed for that purpose.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: *Debugging aids, Diagnostics, Distributed debugging, Error handling and recovery, Tracing*

General Terms

Performance, Design, Reliability.

Keywords

Erlang

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Erlang'04, September 22, 2004, Snowbird, Utah, USA.

Copyright 2004 ACM 1-58113-918-7/04/0009…$5.00.

1. INTRODUCTION

The AXD 301 Multi-Service Switch from Ericsson is probably one of the largest industrial software projects using a functional programming language. The AXD 301 presents a fairly complicated execution environment. It has a number of subracks (typically 1-4), each with a number of Central Processors (CP's, typically 2-4), and a number of Device Processors (DPs, typically ~10). The DPs handles the physical interfaces (Ethernet, ATM, SONET etc), and the CP's run the traffic control, configuration and administration software. The CP's are paired, with active and a standby roles, applications on the active CP can run with hot (minimal disruption of service) or warm (application restart) standby. The requirements on availability means that application unavailability should be less than 5 minutes per year.

The CP's software is written primarily in Erlang (~2.1 million lines of code), with additional third party software (mostly routing protocols). About 300 people have contributed code, almost all of which were complete Erlang beginners when they wrote their first production code.

Erlang is a functional programming language developed by Ericsson (hence open sourced [3]). It is a single-assignment, dynamically typed language. The source code is byte-compiled and executed in a garbage-collecting emulator. It has extensive support for distribution and concurrency. The execution environment means that certain common types of errors (e.g. pointer errors) are non-existing, and that certain problems, such as distribution, often becomes trivial.

The AXD 301 is considered very reliable. There are of course many contributing factors to this. The project was well run, with a pragmatic view on e.g. internal documentation. Testing was highly efficient, partly by using an automated test environment also implemented in Erlang [4]. It was staffed with programmers that may have been unfamiliar with the Erlang language, but generally had a lot of experience in the problem domain. However, from talking to the coders it seems clear that the Erlang language and runtime environment was a key factor. Some observations that are often mentioned includes;

- It's easy to learn. There is lots of code in the system that's written by people with less than a months experience of the language.
- The applications are split into a large number of processes. This means that a software error will often affect only a single process (it can affect other processes e.g. by sending signals or modifying database tables). This can of course be quite serious, e.g. by leading to a restart, but in practice the system will often keep working, although perhaps with reduced functionality.
- The emulator takes care of most of the drudgery in things such as memory management, concurrency and distribution.

Not having to spend weeks chasing elusive memory leaks is good for both productivity and morale.

- The language mechanism of “selective receive” means that state machines doesn't have to handle every message in every state. This can greatly simplify many problems.
- The dynamic code replacement feature means that the debug cycle can be very short. The time between observing an error in the test lab and verifying the solution, which includes finding the problem, fixing it, recompiling, loading the new code and rerun the test case, is regularly as short as one hour.
- The emulator support for tracing and debugging is extremely powerful. One can inspect message passing between Erlang processes, the state of the processes, the workings of the emulator scheduler and garbage collector, etc. This means that it is possible to localize and optimize runtime bottlenecks in a very efficient fashion. In this paper, we will discuss tools developed for the AXD 301 project to make use of these, sometimes obscure, features easier.

We will also discuss some of the more common serious problems. These include;

- Memory “leaks”. There emulator has no real memory leaks (or rather, none has ever been observed in the AXD 301), but is possible to make the emulator allocate too much memory.
- Faulty loops (or rather recursion).
- Deadlocks.
- Race conditions.

Some tips and tricks to avoid and, failing that, finding these kinds of problems will be presented.

2. ERRORS AND ERROR DETECTION

2.1 Development Cycles

In the AXD 301 project the testing and verification is normally divided in three phases; block test, function test and system test. This is where coding errors should be found. Unfortunately there will also be errors that are not discovered until the product has been deployed.

2.1.1 Block Testing

The AXD 301 software is divided in so called blocks. Each block is supposed to be more or less functionally independent of all others, and to be small enough that a few people (preferably one person) can maintain it. When some new functionality is implemented, the developer(s) test the functionality in a simulated environment, either a block test environment (an Erlang node where they load their own code and stub code for all other blocks), or a simulated AXD 301 (4 Erlang nodes, simulating CP's, running more or less the complete CP software, with stubs to the hardware). Since the Erlang emulator that runs on the designer's workstations is identical to the one used in the AXD 301 it is relatively simple to create such simulated environments.

In block testing most bugs causing run-time errors are detected and corrected. Common such errors include; calls to non-existing functions, misnamed variables/constants and malformed pattern matches.

In addition to verifying the basic functionality of the block, this phase of testing thus includes finding bugs that, in many cases, would be found by the compiler in a statically typed language. The line between block testing and design is very thin. This is generally considered a good thing by the designers, since it is rewarding to watch your code do something (even if it is crashing). Since fixing a simple bug, recompile, load it into the emulator and re-running the test case can be often be accomplished in a minute or two, this way of working is quite viable. At the end of block test the code is considered ready for integration.

2.1.2 Function and System Testing

In function testing, all software and hardware is integrated into a fully functional AXD 301. Function testing mostly verifies the promised functionality, whereas system testing involves stress testing, long-term stability and performance issues.

At this point corrections to the code is documented in so called Trouble Reports (TRs). By investigating the TRs one can get a pretty good picture of what kinds of errors are typically found. Some 150 TR's were investigated for this paper. Almost all coding errors belonged to one of these groups;

- API mismatches
  Calling the wrong function/using the wrong arguments
- Race conditions
  Two parallel processes trying to do something incompatible, e.g. one deleting and the other reading a table object
- Wrong context (process/CP)
  Executing in the wrong context, e.g. trying to read from a table that only exists on another CP
- Typos

Some of these are related to the properties of the Erlang language. Some of the API mismatches could have been caught by a linker. But not the ones that call legal functions, just not the right one. Race conditions are more of an issue because the heavy use of concurrency in Erlang. Some typos would have been caught by a type checker.

However, it is interesting that features often claimed to be dangerous, such as dynamic typing, dynamic code loading and pattern matching, is not in practice much of a problem. Most of the errors were not coding errors, but simply a working implementation of the wrong thing.

2.1.3 Deployed Systems

Occasionally there are problems that doesn't surface until the system is deployed. It has proved difficult to get a quantitative breakdown of the error types, but interviews with third-line support staff indicates that the most common causes are

- handling errors (e.g. applying the wrong software upgrades)
- hardware problems (e.g. hard disk failures)
- sourced C code

Crashes in Erlang code is almost always associated with one of the above.

To the best of our knowledge, the Erlang emulator itself has never spontaneously crashed on a live site, and only once has an error been traced back to a problem with the emulator itself.

3. TROUBLESHOOTING TOOLS

3.1 OTP Tools

The Open Telecom Platform (OTP) is a set of support libraries delivered with the Erlang system. It some contains well

documented troubleshooting tools [3]. In order to provide some context we'll provide a short overview here.

3.1.1 The trace BIF

The Erlang trace BIF (Built-In Function) is a way to tell the emulator to generate a trace message every time a specific event occurs. Events include calling a particular function, performing a garbage collect, creating/destroying/scheduling a process, etc. The trace messages can be processes directly by Erlang code, or sent to fail or a network socket. This mechanism is quite simple in principle, but allows one to construct very advanced debugging and profiling tools.

3.1.2 The dbg Application

`dbg` was introduced (in this form) in OTP R7. It is an layer on top of the `trace` BIF that allows you to order debug printouts on the fly. The granularity is at the Erlang function level, i.e. the information shown is the pid, the module, the function and the arguments. Key features are;

- No need for debug compiled code
- Powerful selection expressions for debug printouts
- Option to have caller displayed
- Option to have process-info displayed
- Option to have functions return value displayed

In order to reduce the amount of printouts there are three levels of filtering; pid, {Module, Function, Arity}, argument values.

The overhead is low, if it is used correctly. If the traced function (i.e. the combination of pid, {Module, Function, Arity} we specified) is not called, the cost is 0%. If the traced function is called but the argument values does not pass our filter, the cost depends on how often we call the function, but is typically a few percent. If a printout is generated, the cost is the same as the previous case plus the time it takes to generate the printout. Generating the printout can of course be a quite significant cost.

In order to start the tracing we need to specify three things. Which process(es) to trace, which function(s) to watch and under which conditions to print a message.

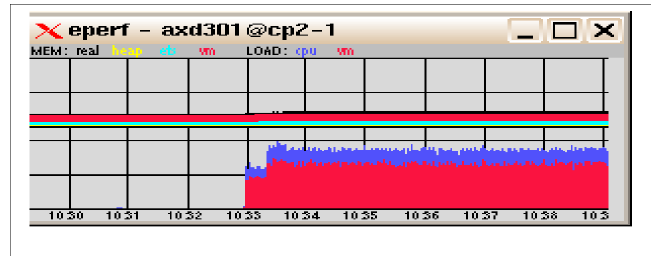
3.2 AXD 301 Tools

In this section we will describe some debugging tools developed in-house (but freely available [5]). Partly because the tools might be interesting to other people wishing to debug similar systems (the tools are freely available). But perhaps more interestingly, since they have been specifically developed to solve common problems (and been found to work), they embody a great deal of trouble-shooting knowledge. They are designed to impact the AXD 301 as little as possible. Hence they don't run on the CP's, but on an attached workstation. They are thus only useful in the test lab.

3.2.1 The eperf Tool

Often the first step in trouble-shooting is to realize that you have a problem. This might sound like a non-problem, but quite often the indication of a serious problem is that the system seems a bit sluggish. Monitoring resource utilization is vitally important to get early trouble detection.

Figure 1. Screen shot of the eperf monitoring tool.

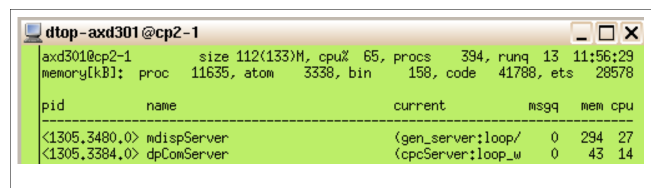


`eperf` displays the CPU time (bottom part of the display) and RAM utilization (top part) in real-time. The display scrolls to the left, and the time stamp is written at the bottom. This is of course a quite unoriginal idea, similar tools have been around for as long as the technology to display them. This particular implementation has some Erlang specific twists though. It distinguishes between the CPU time utilization of the Erlang node and of the rest of the system. It also divides the memory utilization into 4 categories, Erlang process heaps, data tables (ets), total internal use and size of the UNIX process. In Figure 1 above, it's immediately obvious (or rather, would be if it was printed in color) that at 10:33 something happened that caused the system to use about 90% of the CPU time, of which the Erlang node (i.e. the beam process) itself uses about 60%. However, it is operating in more or less constant memory space. This amount of information is often enough to get the trouble shooting off in the right direction. Even more importantly, it immediately informs the tester that there is a problem to begin with.

3.2.2 The dtop tool

Just as the `eperf` tool described above is modeled on the Solaris `perfmeter`, the `dtop` tool is basically a copy of the old Unix utility `top`. Reason being of course that `top` is extremely well thought out in terms of interface and functionality. For those unfamiliar with `top`, it is a tool that samples system resource utilization, breaks it down on a per process basis, and displays it in a text terminal. it's typically set to repeat this every 5 seconds or so, thus always displaying an up-to-date snapshot of what the system is up to.

Figure 2. Screen shot of the dtop tool



`dtop` does the same thing for an Erlang node. Typical usage is to start it up once the eperf tool has informed one that there is something wrong in the system. If the CPU usage is too high, one wants to find out which Erlang process is the culprit (no so easy given that there's typically 600-800 processes running). If the problem is memory usage, one wants to find out what kind of memory is being used (tables, process heaps, mailboxes etc.), and if it is a process heap, which process it is. An unpleasant situation is when the system is supposed to do something, but does nothing. If this is the result of a deadlock, quite often there is a message sitting in the in-queue of some process. All of these situations is amenable to analysis with the *dtop* tool.

3.2.3 *The pan tool*

pan was developed to measure what resources (CPU time and memory) a set of actions (a test case) consumes. It is designed to work in the AXD 301 environment, and should be run from an Erlang node (the host) that is external to the AXD 301 CP's. When pan is started, it will load itself on the target CP's, and start a tracing process. The target process will generate a data file containing information about the operating system and the Erlang emulator, as well as about the Erlang processes and the Erlang tables. When the target process(es) are terminated the data files will be moved to the host. It can also be run on-line, piping data from the target to the host through a socket.

The `pan:perf` function provide basic information about Erlang processes, most importantly the number of microseconds used by each type of process. we see the Erlang processes sorted after how much CPU time in microseconds they consumed (`cpu`). the `pid` is replaced by a count if there were more than one process with that `id`. the `id` is the registered name or, lacking that, the initial call. some initial calls are recognized and mangled to be more useful. `gc` is the number of garbage collects on the process, and `in` is the number of times the process was scheduled to run.

This tool demonstrates some lessons learnt while optimizing the AXD 301 code. It has been demonstrated many times that attempts to optimize with respect to CPU time consumption without continuous feedback will likely lead to minimal improvements and unreadable code. The `dtop` tool immediately shows which type of processes has the most potential for improvement.

Secondly, if the system architecture is such that there can be a large number of short-lived worker processes (like that of the AXD 301), the cumulative resource consumption of these can be the dominating factor. Therefore it is important to group all such processes and sum their CPU time usage. For example, in Figure 3 there are 38 processes grouped under the "{sysTimer,do_" tag. Even though each of them is quite cheap their total CPU time is significant. An implication of this is that one doesn't use ids (process identifiers) to identify the processes. That is to be recommended in any case, since pids are not very user friendly.

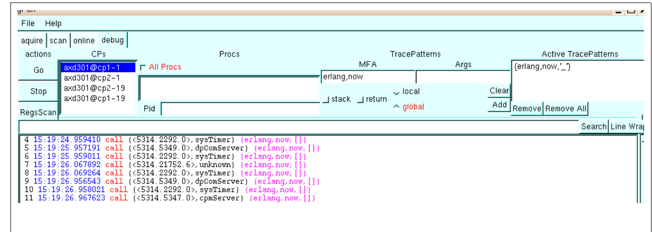
Figure 3. The pan:perf tool

```
pid             id                gc     in        cpu
=======================================================
<122.5440.0>    dpComServer       73    849     548900
<122.5574.0>    plcMemory          0    170     192779
<122.5435.0>    cpmServer          7     58     119229
<122.5569.0>    sbm                3     69      68211
38              {sysTimer,do_      0     75      13055
5               {inet_tcp_dis     10     39      11257
2               {pthTcpNetHan      8     36       8757
<122.6819.0>    pthTcpCh2         16     28       8296
<122.6229.0>    {jive_broker,      5     15       7948
<122.6818.0>    pthTcpOm1         16     27       7494
<122.6778.0>    pthOm              6     31       7431
2               {sysTimer,do_      4     16       4083
<122.6781.0>    pthOmTimer         1     22       2784
<122.17.0>      net_kernel         0      7       1697
<122.10.0>      rex                1      6       1037
```

The `pan` tool also contains a profiler similar to the `fprof` tool that comes with OTP, and a debugger similar to the OTP tool `dbg`. The reason for not using the standard tools is that `pan` predates them, and that they are less suited to the AXD 301 environment. `pan` also contains a scanner that filters and displays trace messages, primarily useful when investigating massage sending.

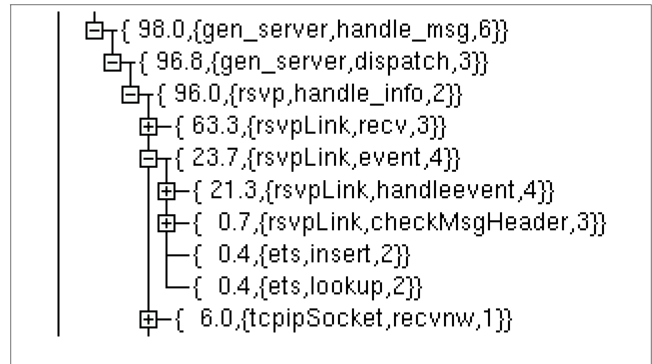
It has proven to be quite difficult to get acceptance of the pretty complicated command line interface of `pan`.

Figure 4. Screen shot of the gPan gui.



As an experiment we developed a GUI to the tool, called `gPan`. It's quite promising, and very helpful especially when running the function call tracer (the tool will keep track of the break points in a clickable list). The profiler (Figure 5) is also a lot more accessible when the data is presented in a clickable tree. The development has been somewhat hampered by the lack of a really good GUI library in OTP. gPan uses the `erlgtk` [6] interface to GTK [7], but it is quite old and no longer maintained.

Figure 5. Screen shot of the gPan profiler.



4. TROUBLESHOOTING

4.1 Run-time errors

Run-time errors are typically very easy to find. The emulator will print an informative error message, like this;

```
** exited: {undef,[{ets,insert,[1]},
                   {example,undef,0},
                   ...]} **
```

informing us that the function `undef` in the module `example` called the function `ets:insert/1`. Unfortunately, that function couldn't be found by the emulator (i.e. the function was undefined, hence `undef`). This is a powerful mechanism, and quite often enough to find the error.

However, note that the backtrace is derived from the stack, and hence doesn't display the actual call sequence, but only the functions that will be returned to. The backtrace above was produce by running this code;

```
undef() -> foo()+1.
foo() -> bla().
bla() -> ets:insert(1).
```

Note that the functions `foo/0` and `bla/0` does not appear in the backtrace. Since there is no need for the program to return to those functions after the call to `ets:insert/1` no stack frames are produced for them. `undef/1` wouldn't appear either if it wasn't for the presence of the "+" operator. Since the program needs the result of `foo/0` to evaluate `undef/0` a stack frame is created.

Run-time errors of this kind are possible because of the dynamic loading used by the Erlang emulator. Since there is no link phase, the error can't be discovered until the function is called. OTP supplies a tool, `xref`, that finds all unresolved calls within a given a set of modules. The AXD 301 project runs a custom version of this (`axdref`) for each release. The tool typically finds a number of unresolved calls. Interestingly enough, running the tool on certain older releases (predating the tool), will uncover many such unresolved calls, even though the code has been used for years by customers. This is of course because testing has found all the unresolved calls in the parts of the code that is actually used.

A similar type of error is also illustrated by the code above, where we add 1 to the result of `foo/0`. The return value of `foo/0` is dynamically typed (like everything else in Erlang) and since it's also strongly typed an exception will be thrown if `foo/0` does not return a number;

```
** exited: {badarith,[{example,arith,0},
                        ...]} **
```

On a theoretical level, this behavior is of course pretty unsettling. It means that the program can only be trusted if all possible return values of all functions are covered in testing. On the other hand, in practice this has never caused any major problems. We very rarely see this kind of type-errors, and if we do it is more often than not caused by another bug somewhere else in the system.

The combination of strong typing and informative error logging means that even very rare errors are identified and corrected. Even if an error only occurs once in, say, 1,000 hours of testing it is almost guaranteed to be reported by the testers, and with enough information to enable the designer to find the problem.

4.2 Deadlocks

A deadlock results when two (or more) processes wait for each other. This problem is more relevant in Erlang than in most other languages because of the way it encourages massive concurrency and its usage of asynchronous message passing. A typical example is two processes that each wants to send a message and receive an acknowledgment. If the code is such that it requires the acknowledgment before proceeding, and the processes happen to send their messages simultaneously, a deadlock will occur. Even if the receives are protected with timeouts, sometimes it is not acceptable to wait for the timeout to happen. An example of such a situation is system startup, where the entire system might stall due to a deadlock.

Finding a deadlock can be quite tricky. In some situations some of the processes involved will have messages stuck in their message queues. They can then be found with e.g. the `dtop` tool described above.

If one is faced with a deadlock that times out, one can identify a the deadlock perpetrator by tracing on all processes in order to see the first thing that happens after the deadlock is resolved.

Another possible measure is to inspect the call stack of each process (by getting a backtrace with the system call Erlang`:process_info/2)`.

4.3 Runaway processes

A runaway process is a one that consumes resources (such as memory or CPU time), without doing any useful work. Typically this is the result of a non-terminating loop. There are several ways in which a runaway process can break the system; by consuming all, or most of, the CPU cycles, by using up enough memory to force swapping of the emulator, or by exceeding some system limit (e.g. by endlessly creating tables or processes).

5. POTENTIAL FOR IMPROVEMENT

The AXD 301 project did, in our opinion, by and large make the right decisions about development practices. Of course, some minor things might have been better done differently.

- Use more processes
  Generally speaking, it is more common that designers want to use too few rather than too many processes.
- Catch considered harmful
  Catching and continuing from unexpected return values or exceptions mostly prevents errors from being discovered, as opposed to happening.
- Type checking API
  Type checking the functions defined in the internal API speeds up debugging

6. CONCLUSIONS

Use of the functional programming language Erlang in a large telecom project has proved to be successful.

After the initial code-debug cycle very few coding errors, as opposed to logical or algorithmic errors, remain. This provides evidence to the claim that using a very high level language like Erlang free up the programmer to work mainly on the problem at hand, as opposed to, say, debugging core dumps or chasing memory leaks.

The Erlang environment provides mechanisms for debugging and optimizing code that allows the final product to be both stable and quite fast.

In closing, we'd like to quote Eric Raymond, whose sentiments about Python pretty much sums up our experiences with using Erlang in the AXD 301;
“[Accepting] the debugging overhead of buffer overruns, pointer-aliasing problems, **malloc/free** memory leaks and all the other associated ills is just crazy on today's machines. Far better to trade a few cycles and a few kilobytes of memory for the overhead of a scripting language's memory manager and economize on far more valuable human time.“[7]

7. REFERENCES

1. *Ericsson*. www.ericsson.com
2. Armstrong, J., Virding, R., Wikstrom, C., and Williams, M. *Concurrent Programming in ERLANG. Prentice*-Hall Europe, Hemel Hempstead, U.K., 1996.
3. *Open Source Erlan*g. www.erlang.org
4. *OTP Test Server*. www.erlang.org/project/test_server
5. *jungerl*. jungerl.sourceforge.net
6. Tony Rogvall et al. *Erlang + GTK*. erlgtk.sourceforge.net
7. *The GIMP Toolkit*. www.gtk.org
8. Eric S. Raymond, *Why Python?* www.linuxjournal.com/article.php?sid=3882