Erlang in Production

Tips and tricks for running Erlang in production
11 June 2014

Jesper Louis Andersen

Introduction

The system is built, it is deployed and it seems to work.

You might have good extensive test suites. Stress tests seem to work as well.

The question is: What **now** ?

My experience only

At various customers, I have seen different kinds of situations over the years.

Many of these can be avoided with the right tooling or mindset, early on.

Erlang helps because it makes performance highly predictable, even in the presence of system failures, but…

Ideology

- If we have metrics, we can see what is going wrong
- If we don't have metrics, we need to trace the live production system

It is not always the case we can reproduce the error in test

Complex systems are hazardous—it **will** eventually go wrong.

Large complex systems will fail

- Systems are highly dynamic (Richard Cook)
- The operating point is constantly moving

Typical things:

- Developers deploy new code
- A user finds a new way of using your system
- Data amass over time, changing response/query times
- Hardware fails
- Your system relies on a distributed system, which then fails

Failures in complex systems (Cook)

- Complex systems are intrinsically hazardous
- Complex systems have defense in depth
- They never fail due to single-points-of-failure
- Complex systems always have flaws in them
- They always run in degraded mode
- The catastrophe is just around the corner
- There is no root-cause
- You can't do post-accident analysis in the right light
- Human interaction is both a problem and a defense
- All actions on the system is a gamble with risk factors

More about failure

Cook has even more information in "Why complex systems fail"

Regard your production system as a highly dynamic environment which can go wrong at any point in time

Approach

- Basic stuff: basho/lager, otp/alarm_handler, otp/os_mon
- Metrics: boundary/folsom (feuerlabs/exometer)
- Cascading dependencies: fuse (mmzeeman/breaky, klarna/circuit_breaker)
- Overload handling: safetyvalve (uwiger/jobs, basho/sidejob)
- Tracing: recon (otp/dbg, massemanet/eper/redbug)

Basics

Use erlang releases and configuration

- The .app file can contain default configuration
- Provide a standard `sys.config`

Do note the trick from `erl -man config`

```
sys.config:

 [{myapp,[{par1,val1},{par2,val2}]},
  "/home/user/myconfig"].
```

Target test, use this to override in production.

Would really like chubby/etcd-style configuration here!

Lager

- Use lager as your logging framework.
- Plug in handlers into lager so you can syslog, can forward log statements to other servers, etc
- Handlers are specified in the configuration. Gives your devops flexibility.

alarm_handler

The `alarm_handler` is very very useful.

- Set and clear alarms when things go awry
- Write your own alarm handler. Export alarms to Nagios or what is in your environment.
- Also write lager log statements when alarms go off

os_mon

Enable the os_mon application

- It raises/clears alarms!
- Monitors cpu usage
- mem_sup is a beast! Can be configured to report if any process uses more than 5% memory

mem_sup has caught lots of problems early on in production for me!

Metrics

Metrics

Metrics is a tool which tells you how a system is currently behaving

- We need to be able to point fingers
- Often, the error is in a cascading subsystem

Folsom

Folsom is a metric subsystem

- Doesn't persist metrics, only collects them
- Write a tool to push/pull metrics to other systems
- Different metrics, like counters, gauges, spirals, histograms

Using folsom

We focus on histograms here for latency calculations. Other types are similar in structure.

- Histograms run a replacement reservoir

```
histogram_timed(Name, Thunk) ->
    {Time, Result} = timer:tc(Thunk),
    Diff = Time div 1000,
    _ = folsom_metrics:notify(Name, Diff, histogram),
    Result.
```

This will **dynamically** create **Name**, so you don't have to add it beforehand.

You can also pre-add metric types to the system.

Obtaining folsom results

```
Stats = folsom_metrics:get_histogram_statistics(Name),
Value = folsom_metrics:get_metric_value(Name),
```

- Go for the percentiles in the histogram.
- 50th percentile (median)
- 95th percentile (slow queries)
- 99th percentile (really slow queries)

VM Metrics

The module is `folsom_vm_metrics`.

```
folsom_vm_metrics:get_statistics()
```

What to collect

EVERYTHING

- Cost is low. Knowing is better than guessing!
- API endpoints
- Database query times
- Cache hit/miss rates. Also track Type-II misses.
- Every cascading subsystem
- Internal heavyweight processing
- File/Line/Severity log lines!

Systems I run regularly collect more than 2000 different probes

Tracing

Tracing / Recon

Recon is a tool by Fred Hébert

```
http://ferd.github.io/recon/index.html
```

This should be in all your production systems

```
recon:proc_window(reductions, 5, 100).
```

- We want `reductions` from `erlang:process_info(Pid)`
- We want the top 5 Pids
- We want a window of 100ms

Can be used to find problematic processes quickly.

Online Tracing

Avoid printf-style debugging. Avoid excessive logging. Trace running systems when mistakes are present.

```
recon_trace:calls({Mod, Fun, MatchSpec}, Max, Opts)

recon_trace:calls({erlang, now, fun(_) -> return_trace() end}, 6, [{scope, global}])
```

- Traces `erlang:now()` for up to 3 calls (there are two messages per call)
- The `scope` can also be `local` useful for non-exported functions

Low impact on the system. Can be used in production. The `Max` is an overload protector.

Cascading dependencies

Scenarios

Typical scenarios:

- Nuclear power plants
- Finance
- Medical equipment

Circuit breakers

Install a fail-safe against the problem. Use this to trigger if the problem occurs. For software:

- Databases
- Systems over which we have no control
- Frequently failing systems

Characteristics

No resource buildup in the Erlang system.
- Quick response on failure, close to 4µs.
- Clients can discriminate long processing time from genuine known failure.
- Excellent place for monitoring

Fuse

Fuse implements the circuit breaker pattern

Using

```
-define(FUSE_RESET, reset_fuse).
reset_test(_Config) ->
    ct:log("Installing a fuse, then resetting it should clear out timers"),
    ok = fuse:install(?FUSE_RESET, {{standard, 2, 60}, {reset, 5000}}),
    ok = fuse:ask(?FUSE_RESET, sync),
    ok = fuse:melt(?FUSE_RESET),
    ok = fuse:melt(?FUSE_RESET),
    ok = fuse:melt(?FUSE_RESET),
    blown = fuse:ask(?FUSE_RESET, sync),
    ok = fuse:reset(?FUSE_RESET),
    ok = fuse:ask(?FUSE_RESET, sync),
    3 = proplists:get_value(one, folsom_metrics:get_metric_value('reset_fuse.melt')),
    2 = proplists:get_value(one, folsom_metrics:get_metric_value('reset_fuse.ok')),
    1 = proplists:get_value(one, folsom_metrics:get_metric_value('reset_fuse.blown')),
    ok.
```

Overload handling

Goto

goto other slide deck :)

Thank you

Jesper Louis Andersen
http://erlang-solution.com (http://erlang-solution.com)
@jlouis666 (http://twitter.com/jlouis666)