Fuse

Let it crash & handle with grace
10 June 2014

Jesper Louis Andersen

The problem

Large complex Erlang systems cannot avoid:

- Many applications.
- Lots of cascading subsystems (MySQL, Caches, foreign APIs, …)

Errors in part of the system affects our system. Fingers gets pointed at **us**

Fuse

Fuse

Implement the pattern known as the **circuit breaker**:

- Analogy: a fusebox.
- System works nominally: The circuit breaker is closed.
- Errors in cascading subsystem: The fuse is melted a little bit.
- Too many errors: The fuse is blown and the circuit breaker is opened.
- After a cooldown period: The fuse cools down and heals.

Work mainly by me (Jesper) with a lot of helpful inputs from Thomas Arts.

```
http://github.com/jlouis/fuse
```

(Thanks to Erlang Solutions and Issuu for letting me use a bit of their time on this)

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

Community work

- Proper documentation of code via edoc
- Tutorial
- Uses Travis-CI
- Semantic versioning with proper git tags

Tries hard to be easy to use as a project

Furtermore:

- Implement the circuit breaker in Erlang-style with Erlang idioms

Configuration

```
MaxI = 10,
MaxT = 60,
Name = database_fuse,
Strategy = {standard, MaxI, MaxT},
Refresh = {reset, 60000},
Opts = {Strategy, Refresh},
fuse:install(Name, Opts).
```

Typical usage:

```
Context = sync, % Can also pick async_dirty here
case fuse:ask(database_fuse, Context) of
    ok -> …;
    blown -> …
end,
```

And when a timeout occurs:

```
case emysql:execute(Stmt) of
    {error, connection_lock_timeout} ->
        ok = fuse:melt(database_fuse),
        …
    …
end,
```

Performance characteristics:

Machine: Q4 2013 Macbook Pro, OSX 10.9.2, 2 Ghz Intel Core i7 (Haswell)

- 2.1 million `fuse:ask/2` calls per second.
- Linear scaling on 8 cores.
- Typically: 3-4μs per call
- My Lenovo W530 Ivy Bridge-based Core i7 running Linux 3.14.4 delivers roughly the same performance

Simple stress test, not very scientific. But gives away the general ballpark figure for the implementation. This is an upper bound in practice! Real systems will be much lower.

Somewhat disappointing at 3000-4000 nanos per lookup.

- 2 mutex locks, uncontended, and a hash, 2 DRAM accesses: 25 + 25 + 100 + 100 = 250 nanos.
- Expectation off by an order of magnitude!

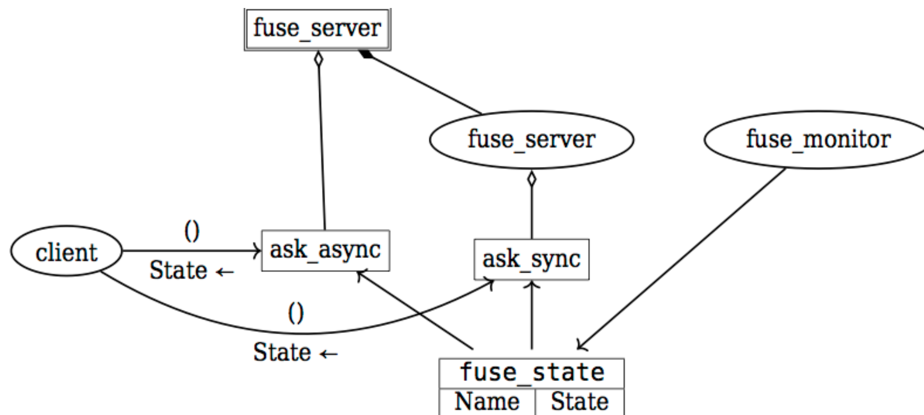
Monitoring / Eventing

Fuse contains a monitor and gen_event framework.

- Monitors fuses and uses the alarm_handler to raise alarms
- Monitors uses a skewed hysteresis algorithm to avoid flapping of state
- You can subscribe to events and handle cases yourself

Implementation

(Visual Erlang notation)



Implementation (2)

Architecture:

- Split into a fast and slow path
- Fast path is ask → ok
- Slow path is when a melt happens

Design:

- ETS keeps the fuse state ({read_concurrency, true})
- Melting is forwarded to the `fuse_server`
- The fuse_server maintains `[Ts¹, Ts², ...]` recent-first melts. Cuts out too old timestamps. (stolen from `supervisor`)

Low hanging fruit in the fuse_server, but I don't care yet.

Correctness

Correctness

Fuse sits on the mission-critical path. Prime candidate for extensive testing.

- Quviq QuickCheck is used to test random use case scenarios.
- Thomas Arts provided helpful input in how to structure the tests.

Built "Property based test first": The QuickCheck model is built first or at the same time as the implementation, a bit at a time.

Correctness (2)

- Both positive and negative testing on a stateful model
- Parallel testing
- Uses PULSE. This forces the system into a particular random schedule. Finds subtle race conditions

Correctness (3)

We use requirements testing:

```
…
Group install:
R03 - Installation of a fuse with invalid configuation
R04 - Installation of a fuse with valid configuration
…
Group Melt:
R11 - Melting of an installed fuse
R12 - Melting of an uninstalled fuse
…
Group ask/1:
R15 - Ask on an installed fuse
R16 - Ask on an uninstalled fuse
```

There are 16 requirements in all.

Run Example

```
2> fuse_eqc:r(seq, {5, sec}). ..........
OK, passed 1213 tests

48.20% {fuse_eqc,elapse_time,1}
12.71% {fuse_eqc,melt_installed,1}
10.03% {fuse_eqc,run,4}
9.44% {fuse_eqc,install,2}
9.10% {fuse_eqc,ask_installed,1}
3.79% {fuse_eqc,reset,1}
2.93% {fuse_eqc,fuse_reset,1}
1.94% {fuse_eqc,ask,1}
1.87% {fuse_eqc,melt,1}

12.93% "R11 Melt installed fuse "
11.18% "R15 Ask installed fuse"
8.51% "R07 Run on ok fuse"
7.71% "R10 Run uninstalled fuse "
```

- The model has good coverage

QuickCheck Implementation

- Aspiring to math, we use QuickCheck models as "lemmas" for more QuickCheck models.
- Once a "lemma" is done, we can build on top of it.

In fuse, **time** is important. Build a model for time and QuickCheck that model. Then use it in the real model.

QuickCheck implementation (2)

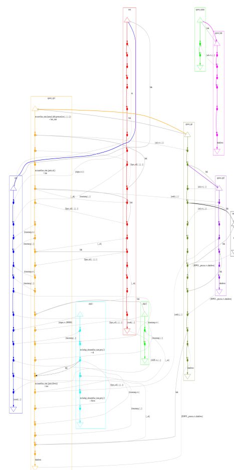
The monitor code is orthogonal to the rest of the code.

- Stack is: ETS Table → fuse_monitor → alarm_handler

Use the eqc_component QuickCheck feature to **mock** the alarm_handler and then test the fuse_monitor in isolation

The PULSE problem

- The fuse implementation allows you to pick a Context as either sync or `async_dirty`.
- If running in async dirty mode, PULSE finds a linearazibility problem.



The timing problem

QuickCheck uncovered that the **client** can't take the timestamp and then push it into the fuse_server.

- Sounds good in principle: work in the clients parallelizes and can happen concurrently
- Is bad in practice: latency skews, in **particular** under distribution
- Client takes timestamp, gets blocked. Then injects timestamp too late. Time now goes backwards.

Complicated scenario. Better to have the fuse_server take the timestamp.

Found easily by quickcheck. Suggested different implementation of the system.

Results from using Quickcheck

- Slower development speed by a factor of 3-5.
- **NO** bugs found yet while in production. No maintenance time used. Can focus on other things, much better use of our time.
- Code is very lean. QuickCheck suggests what code to weed out.
- Rather than trying to handle corner cases, make them illegal by returning `badarg`, then remove the code!

Overall, the use of QuickCheck in this project is a big win. Maintenance time is very expensive, but we avoid it entirely.

Thank you

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