Safetyvalve

Verified load regulation

Jesper Louis Andersen
Erlang Solutions Ltd.
jesper.louis.andersen@erlang-solutions.com

Jun, 2013

Overview

- What is Load regulation?
- How do you use `safetyvalve (sv)` ?
- How is it tested?

What is this about

- Overload protection
- Load regulation
- Load normalization
- Certify a system to a limit

Safetyvalve

- SV is a load regulation framework, like jobs or `overload`
- Ask the framework when you may run, do not run if it says no.

Safetyvalve .2

- *concurrency* how many may run at the same time
- *queueing* when the system is overloaded, enqueue extra work
- *frequency* rate at which work is started

Safetyvalve .3

- A *Token Bucket Regulator* Adds *tokens* which are needed to dequeue
- There is a small *surplus* of tokens

Safetyvalve .4

- Queueing is *necessary* in some workloads and not relevant for others.
- Set Queue Size to 0
- TBRs allows to take short bursts

CoDel queueing

- Experimental feature!
- Some queue is good, a standing queue is bad
- Leads to bufferbloat (TCP/IP is a grave example)
- Van Jacobson, Kathleen Nichols
- Arrival is *not* Poisson (2006)!

CoDel queueing .2

- Idea: measure *sojourn* time in the queue
- If too long a sojourn, begin rejecting work
- producer *MUST* react on work rejection and lower rate
- With a TCP-like additive rate component, this is self-tuning!

CoDel queueing .3

- Stamp packet on arrival
- On Dequeue, check time
- If above a target limit for too long, begin rejecting
- Details: “Controlling Queue Delay, 2012; Jacobson, Nichols”

Example

```
{safetyvalve,
    {queues, [
       {pg_q, [{hz, 500},
               {rate, 20},
               {token_limit, 30},
               {size, 50},
               {concurrency, 32}]}]}},
```

Example .2

```
with_pg(QueryFun) ->
      {ok, C} = pg_pool:obtain_connection(),
      QueryFun(C),
      pg_pool:putback(C).
```

Example .3

```
run_query(QueryFun) ->
    case sv:run(pg_q, fun() ->
                          with_pg(QueryFun)
                      end) of
      {ok, Res} -> {ok, Res};
      {error, Reason} -> {error, Reason}
    end.
```

Internally, *ask*/*done* pairing

Manifesto

- Inspired loosely by the 'Dogme 95' manifesto
- Know to *Test* before you code
- A *Test* is a QuickCheck model
- (*Banish* unit tests from the project)
- Any feature must be *dogfooded* by a real-world user
- Never discuss indentation

QuickChecking Safetyvalve

- Philosophy: Many small, naive test cases
- Use Erlang QuickCheck as an *amplifier* to make naive tests powerful
- Prefer two tests each capturing different aspects

QuickChecking Safetyvalve

- Simplify!
- Trick 0: Do *not* care about post-conditions
- Trick 1: Degenerate model: Queue size=1, Concurrency=1, Bucket size=1. Only then add more complexity

QuickChecking Safetyvalve

- Simplify!
- Trick 0: Do *not* care about post-conditions
- Trick 1: Degenerate model: Queue size=1, Concurrency=1, Bucket size=1. Only then add more complexity
- Trick 2: Do not *track*, only *count*

QuickChecking Safetyvalve .2.5

- Trick 3: Control Internal state, hack the code so it can be queried

QuickChecking Safetyvalve .2.5

- Trick 3: Control Internal state, hack the code so it can be queried
- Trick 4: Control time, inject it! Replenish tokens from the model

QuickChecking Safetyvalve .2.5

- Trick 3: Control Internal state, hack the code so it can be queried
- Trick 4: Control time, inject it! Replenish tokens from the model
- Trick 5: Issue command, wait until no more processes does work (fixpoint)
- Fixpoint is handled by `erlang:process_info/1` by running it twice and tracking reductions/state changes

QuickChecking Safetyvalve .2.5

- Trick 3: Control Internal state, hack the code so it can be queried
- Trick 4: Control time, inject it! Replenish tokens from the model
- Trick 5: Issue command, wait until no more processes does work (fixpoint)
- Fixpoint is handled by `erlang:process_info/1` by running it twice and tracking reductions/state changes
- Trick 6: Simpler `statem`; only check the *last* thing you do.

QuickChecking Safetyvalve .2

- 3 “bits” yields 8 different states
- There are 3 commands: `replenish`, `ask` for work, mark work as `done`
- 24 possible states, many of them can be coalesced
- Reality: 10 cases to handle
- Doable in `statem` models

Example:

- Let $\{C, K, T\}$ be Concurrency, Kueue (size), and Token counts
- Queue, no tokens: $\{C, 0, 0\} \rightarrow_q \{C, 1, 0\}$
- Done, with tokens: $\{1, 1, 1\} \rightarrow_d \{1, 0, 0\}$
- Postconditions *query* to verify internal state

Generalize:

- Done, with tokens: $\{1, 1, 1\} \rightarrow_d \{1, 0, 0\}$
- Done, with tokens: $\{C, K, T\} \rightarrow_d \{C, K - 1, T - 1\}$ where $C > 0, K > 0, T > 0$
- Add time to the model
- Add arbitrary process crashes to the model (new test case!)

- Has found 4-5 bugs already in literally no lines of code
- All the bugs were of the nasty-class
- 383 SLOCs (huge win!)
- CoDel also has an EQC model (Simple at the moment)

Example, if time allows

Questions

?