Elixir and the Internet of Things
Handling a Stampede

CC by https://www.flickr.com/people/t3rmin4t0r/

Doug Rohrer

Me

- Professional software developer for over 20 years
- Written in many languages on many operating systems
- Often called upon to understand and mitigate performance issues on projects
- Had never touched Elixir or Erlang (or much FP) before this project

The Problem

- 10s to 100s of thousands of connected devices
- Sometimes all connecting in a short period of time
- Need to process many messages in a short period of time (round-trip time is critical, as users are waiting for doors to unlock)
- Beyond functionally routing messages, the number one requirement for this application was to scale to very large (200k) numbers of simultaneous connections

Our Process

- Built an performance-testing tool (in Go)
- Built a potential solution (Ruby, Elixir)
- Test
- Profiled applications to find and fix bottlenecks, and retest
- Abandon the solution when there were no clear bottlenecks to fix but performance was inadequate

We Tried Ruby/EventMachine

- Pros
  - Existing solution
- Cons
  - Single-threaded event loop (callback hell)
  - High CPU overhead for AMQP
  - Unbalanced I/O - Would do all of "I" before any "O"

We Tried Ruby/Celluloid.io

- Pros
  - Agents are easier to understand than lots of callbacks and next_tick calls
- Cons
  - Uses Fibers, which are native threads on JRuby
  - Uses LOTS of memory (40GB for ~40k connections)

Why Try Erlang?

- Erlang/OTP was designed to handle this kind of problem
- {ok, concurrency} = erlang:use()
  - Concurrency is built into the runtime
  - Actor model is easy to implement
  - Known for handling the load levels we needed
- Clustering is (almost) code free
  - We need to load balance across machines
  - Erlang allows us to do this with minimal additional coding

Why Elixir and not Erlang?

- The client is predominantly a Ruby shop, and Elixir was a more comfortable choice for them as it has a more familiar syntax than Erlang
- Macros reduce "template code" bloat & potential for errors
- Can still use all Erlang libraries
- Compiles to Erlang bytecode and runs on the Erlang VM
- For all practical purposes, Elixir was >= Erlang for us

Some Erlang/Elixir Basics

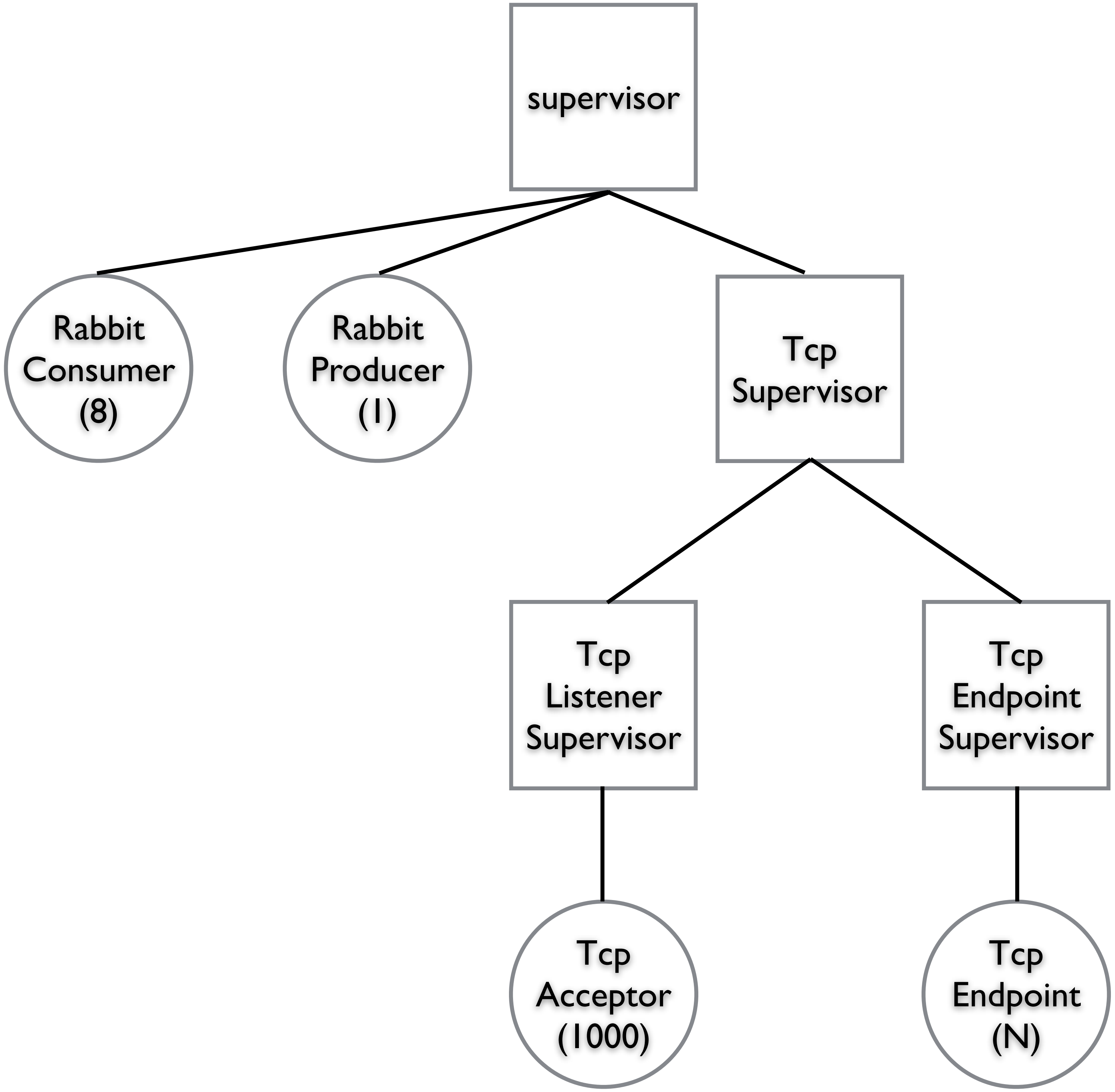
Processes

- Are not OS processes - much more lightweight (100000 on a single Erlang instance is not uncommon, millions per instance are doable)
- Can maintain state using recursive calls
- Communicate with other processes via messages and mailboxes
- Process messages in FIFO order

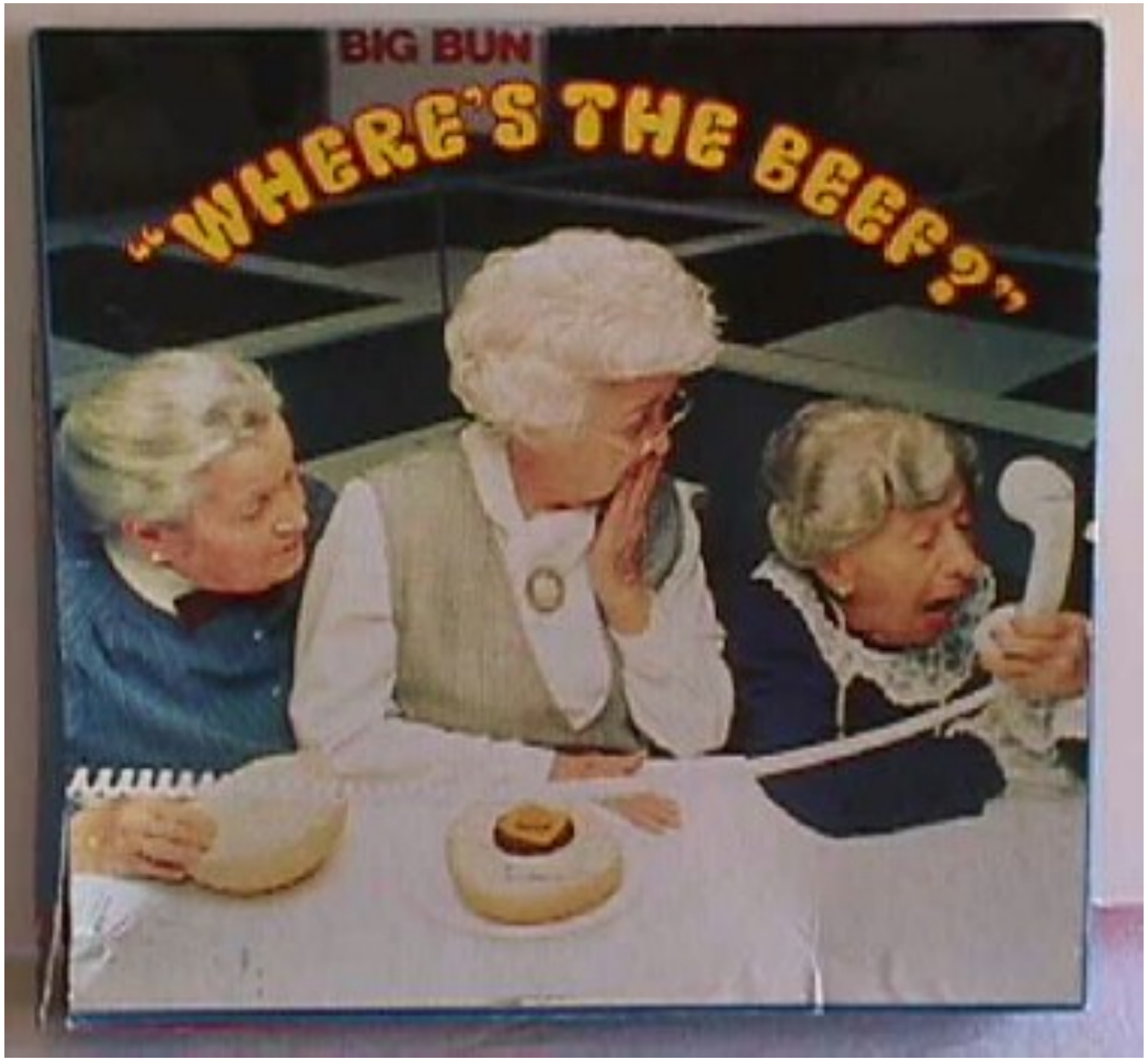
OTP - Open Telecom Platform

- Abstraction above simple processes that provides templates for actors
- Supervisors and Workers
  - If worker dies, supervisor is notified and can take action
- Defined, well understood pattern for handling state and message passing among processes

Supervisor Tree - Owsla



Code Walkthrough - Overview



An Accidental Acceptor Pool

Public Domain by http://pdphoto.org/PictureDetail.php?mat=pdef&pg=5277

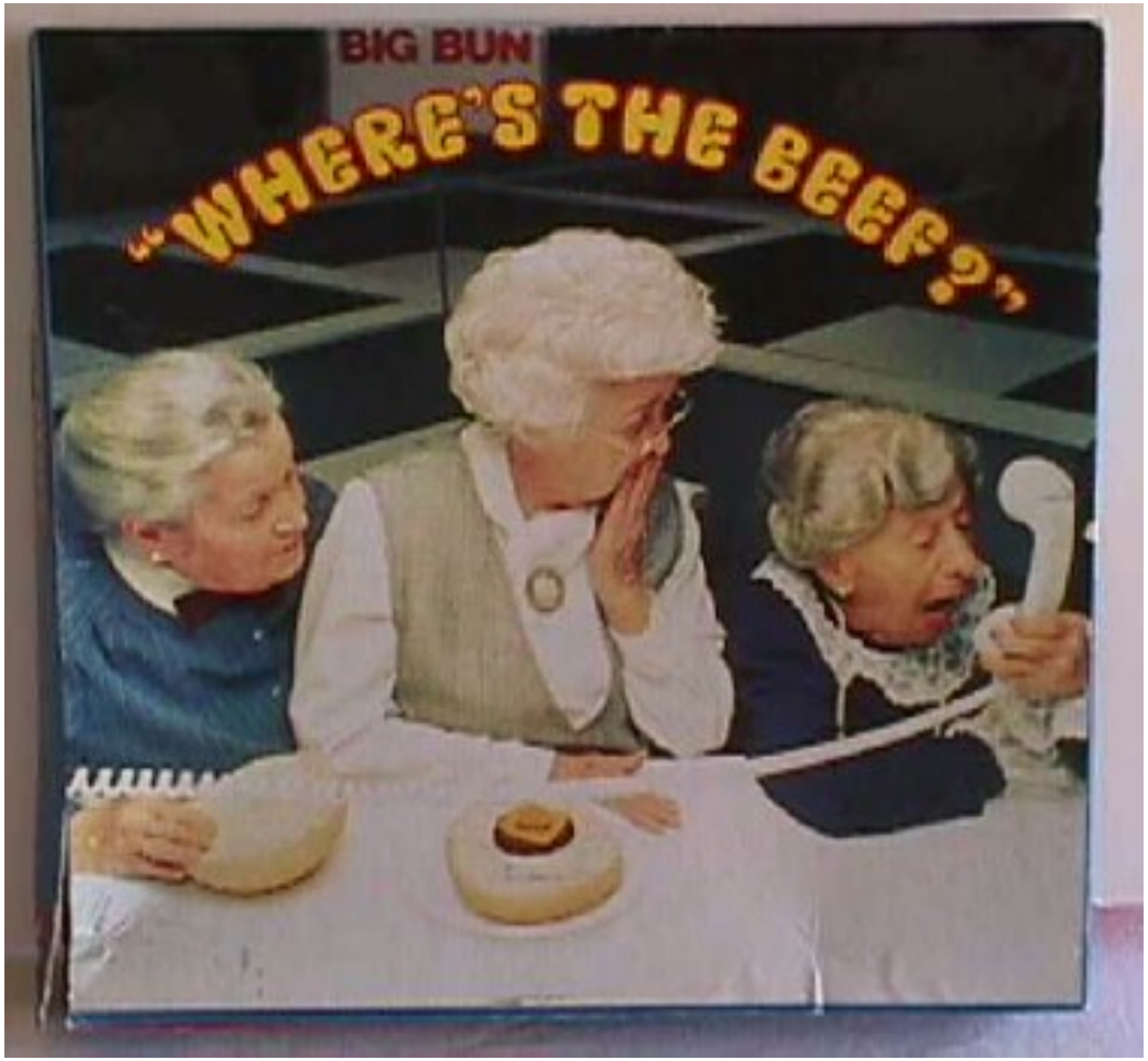
Why do I need an Acceptor Pool?

- Single-threaded listen/accept loops artificially limit connections/second
- Acceptors can all share the listen socket, with only 1 being signaled for each connection
- Allows for concurrent acceptance of a very high number of connections/second (in our case, the goal was 1000/second)

Should I build my own like you did?

- NO
  - We didn’t really know what an acceptor pool was when we started writing this code
  - Had we known, we would have found an existing library and used it instead.
- What libraries are out there?
  - Writing Erlang? Check out Ranch: https://github.com/extend/ranch
  - If you’re writing Elixir, check out reagent: https://github.com/meh/reagent

Code Walkthrough - Tcp*



Taking Control of Restarts

CC Archie McPhee : www.mcphee.com/blog/

Houston, We Have a Problem

- Load-testing tool would open 5-10k connections per instance to our server
- We'd run several instances of the load tool while testing
- Every time we stopped one of them, the entire TcpSupervisor tree would crash/restart, taking down all of the live connections

Restart Strategies

- one_for_one
  - If a child crashes, restart it
- one_for_all
  - if any child crashes, restart all of them
- rest_for_one
  - If a child crashes, it and all children started after it are restarted
- simple_one_for_one
  - like one-for-one, but designed for starting children dynamically.

Maximum Restart Frequency

- MaxT - the time window in which to measure restarts (in seconds) - default is 5
- MaxR - The maximum number of restarts that can occur within MaxT seconds before the supervisor will tear itself down - default is 5

Child Specifications: Restart

- permanent - always restart this child when it crashes (unless MaxR is exceeded)
- transient - restart if the child ‘terminates abnormally’
- temporary - never restart, even if supervisor restart strategy tells you otherwise. These, therefore, aren’t counted when looking at MaxR

Resources/Questions?

- http://elixir-lang.org
- http://erlang.org
- irc: #elixir-lang
- http://jeetkundoug.wordpress.com/2014/01/13/elixir-and-the-internet-of-things
- Me: @jeetkundoug
- My partners in crime on this project
  - @eymiha - Dave Anderson
  - @jvoegele - Jason Voegele