HA - A Development Experience Report

Jeff Epstein
Parallel Scientific Labs
jeff.epstein@parsci.com

Today’s talk

1. What is HA?
2. How did we do it?
3. Haskell
4. Cloud Haskell
5. Distributed consensus
6. Debugging

1. What is HA?

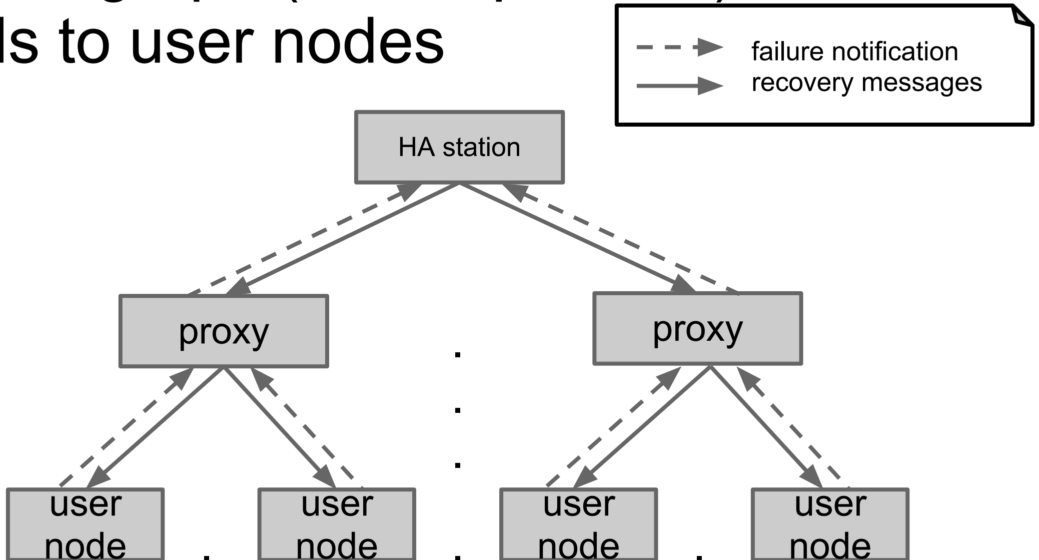
HA (High Availability) middleware manages networked clusters.

- Architected by Mathieu Boespflug and Peter Braam
- Clusters of up to tens of thousands of nodes.
- Centrally controls services running on nodes.
- Create consensus in the cluster about the location of active services.
- Handle events due to failures.
- Recovery time should be ~5 seconds.
- Cluster state must be replicated, for durability.
- Existing systems (Zookeeper, Corosync) don't scale.

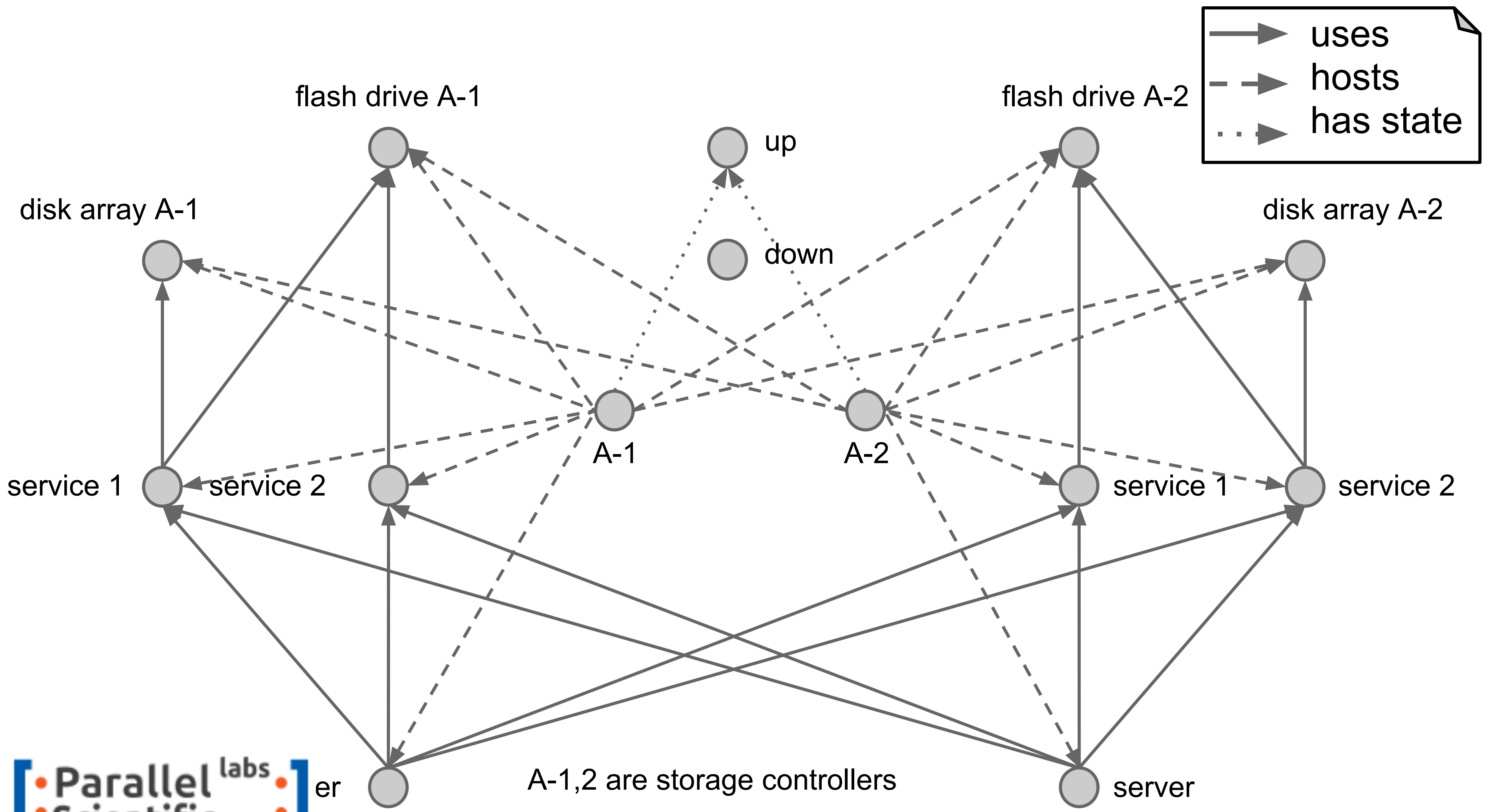
Parallel labs Scientific

1. What is HA? Cluster architecture

- Events (failure notification, etc) are sent by nodes, aggregated on a proxy node, and are enqueued at the HA station
    - The station's message queue is replicated
- Events are processed by the HA coordinator
    - Updates node state graph (also replicated)
    - Issues commands to user nodes



1. What is HA? State graph



A-1,2 are storage controllers

2. How did we do it?

- Haskell
- Cloud Haskell
- Distributed consensus (Paxos)

Parallel labs Scientific

3. How we used Haskell

- Functional data structures
  - The cluster state is a purely functional graph.
  - Replicated (impurely) between cluster managers.
  - Most of our code, however, has an imperative feel
- Easier refactoring, thanks to strong typing.
- Laziness
  - Mainly caused problems in the form of space leaks, especially in low-level networking code.
  - Moreover, laziness is problematic in a distributed setting: message passing implies (the possibility of) network communication, which requires strictness. Maintaining the locality-independent abstraction thus requires strictness in messaging

4. Cloud Haskell

- A library for distributed concurrency in Haskell. It provides an actor-style message-passing interface, similar to Erlang.
    - Processes communicate by message-passing
      no shared data
    - Communication abstraction is the same, whet
      processes are co-located or distributed.
- Originally presented in Jeff Epstein's 2011 MPhil thesis.
- Since then, completely rewritten and greatly extended, by Edsko de Vries, Duncan Coutts, and Tim Watson.

4. Cloud Haskell (cont’d)

- The CH programming model is a good fit for this project
    - Messaging abstraction helps design interacting components
    - Refactoring is easy
    - This is not a “big data” project *per se*, in that it does not itself move around lots of data; throughput is not a strength of CH
- Pluggable backends (for network transport)
    - Provided TCP transport layer
    - Additional transport layers can be implemented for different underlying networks
    - Guarantees ordering of messages, which we do not need - we wish we could turn off ordering properties
    - We implemented proprietary InfiniBand-based transport
    - Matching connection semantics of transport interface to semantics of underlying protocol is hard
    - Space leaks in particular are a problem
- Pluggable frontends (for cluster configuration)
    - Lets nodes find each other
    - Default is not sufficient for our deployment
    - Required awkward workaround for querying nodes on a remote host, based on a fixed listening port and a static file of node addresses

5. Distributed consensus (Paxos)

- Consensus: making different systems agree
  - Well-known, proven algorithm for consensus (Lamport 1989, 1998, 2001)
  - Used for replicated state (Lamport 1995)
- We implemented this as a reusable general-purpose library on top of Cloud Haskell
  - The algorithm is a good match for CH's programming model
    - client, acceptor, proposer, learner, leader as CH processes
    - Small and readable implementation (~ 1.5 kLOC)
  - Used for synchronizing distributed state graph
  - Used for synchronizing distributed message queue
- Challenges
  - Debugging Paxos is hard; untyped messages in CH make it worse
  - Getting reasonable performance is hard (MultiPaxos helps)
  - Liveness issues

6. Debugging

- The CH programming model means it’s easy to debug a “distributed” application on a single machine
- Nevertheless, debugging distributed code is hard: race conditions, missing messages
  - We would like a distributed ThreadScope
  - Instead, we mainly used logging
- We built a deterministic CH thread scheduler
  - Replaces CH message-handling primitives
  - Not a “real” scheduler (that is, does not touch GHC runtime)
- We verified core replication algorithm with Promela model, a verification modeling language

Parallel labs Scientific