Consistency Trade-offs for SDN Controllers
Colin Dixon, IBM
February 5, 2014
The promises of SDN

• Separation of control plane from data plane
• Logical centralization of control plane
• Common abstractions for network devices
Why control can’t actually be centralized

• In practice, we have a distributed cluster of controllers, rather than just one so that
  • we can tolerate faults
  • we can scale out our performance

• and maybe so that if the network partitions there are controllers on both sides
Distributed Systems Primer
Terminology

• Traditionally people talk about replicated state machines—I’m not going to

• People also talk about clients and servers—instead, I’m going to talk about clients and controllers to keep us thinking SDN

• When I say client, think switch or REST client
Things are easy when there’s one controller

- If two clients interact with a controller, it’s the same controller
- They can’t get inconsistent results
How do we get N controllers to act as one?

- Consistent if clients talk to the same controller

Clients (Switches, REST, etc.)

Replicas (Controller Instances)
How do we get N controllers to act as one?

- Problems if clients talk to different controllers

Clients (Switches, REST, etc.)

Replicas (Controller Instances)
How do we get N controllers to act as one?

- Fixed if each client talks to a majority of controllers
How do we get N controllers to act as one?

- Fixed if each client talks to a majority of controllers

Client can resolve that blue is right based on logical timestamps
How do we get N controllers to act as one?

• Or if each controller confers with a majority
Does it need to be a majority?

• Property we used was that two majorities must overlap on at least one controller

• Generalization: sloppy quorum
  • Read from R controllers
  • Write to W controllers
  • Ensure $R + W > N$, e.g., any read set overlaps with any write set at at least one controller
  • Can help scale-out and availability of reads or writes by sacrificing the other
Where this goes wrong and the CAP theorem
Network partitions make everything complex

• Before the partition, everything goes normally
Network partitions make everything complex

• With a partition, clients can read old state
Network partitions make everything complex

- With a partition, clients can read old state
CAP Theorem

• Eric Brewer conjectured that you could not be consistent, available and partition tolerant
  • Conjecture in 2000, proof in 2002
  • A system can’t be consistent, available, and partition tolerant

• There are details, but the general idea is that you have to choose 2 of the 3
  • CP: favor consistency
  • AP: favor availability
  • CA: hope partitions don’t happen
Network partitions make everything complex

On this side, writes break consistency and reads can be stale

On this side things are still happy
Network partitions make everything complex

If all clients are on this side, we’re happy

majority

minority
How modern systems deal with the CAP theorem
Favor availability (AP)

• When there’s a partition, just read from/write to as many controllers as you can and hope

• You lose consistency after one such write
  • Getting it back is hard

• In practice, these systems don’t try to get strong consistency in the first place
  • Hard to program against or reason about
  • Effectively abandons the idea of “logically centralized”

• Examples: Cassandra, MongoDB, Dynamo
Favor consistency (CP)

• When there’s a partition, the minority side(s) stops being able to write (and get stale reads)

• Thus, the minority side(s) are not available
  • This may be fine as long as the clients you care about are on the majority side
  • Great if you can make the majority side the side with the Internet :-)  

• Examples: Chubby, ZooKeeper, some SQL DBs, MegaStore, Spanner
What about CA?

• This is assuming partitions will not happen
  • Generally, people think this is a bad assumption
  • Good design + some expense can make partitions rare though:

• In general, systems that assume partitions won’t occur can’t promise C or A during a partition
  • Typically better off with a CP design that happens to be available because there are no partitions
Other fanciness

• The CAP theorem is weaker than many assert
  • Assumes you want linearizability for consistency; loosely: “all reads get the latest write”
  • Implies you must make the trade-off once

• Some modern systems dance around this
  • Detect partitions and only sacrifice C when they happen; recovering it on healing is tricky
  • Provide less than linearizability, e.g., “all your reads reflect your previous writes”
What about SDN?
Some thoughts on SDN and consistency

"we’ve found the hard problems to be with distributed state which are app specific. Not sure how the controller can help much."
Should we pick AP, CP, or something else?

• We need switches to work even if they’re partitioned off ➔ we need availability
  • So, we should pick AP over CP

• AP generally means eventual consistency
  • This means the controller and apps are not logically centralized from the programmer’s view
  • Hard for programmers to work with

• Ugh… is this what Martin was talking about?
SDN controller examples

• Floodlight
  • Cassandra (AP) deals with eventual consistency
  • Discussion w/Floodlight devs on mailing list:

• ONOS
  • ZooKeeper (CP) for mapping switches to controller instances
  • Instances own the state for their switches
  • See September 30, 2013 Tech Work Stream Call
What can we do?

• We want to provide programmers with something easy (or at least easier) to use

• We want to provide network connectivity and services on both sides of a partition

• CAP says we can’t have it all, all of the time
  • We need to look at some of the fanciness
Forking timeline consistency

• When there’s a partition, the minority side forks after some sentinel timeout, e.g., 1 sec
  • Declares new group by fiat (orange)
• Creates two, internally consistent, timelines
Resolving forks

- When the partition heals, something must merge the two timelines
- Often easier than you’d expect
  - JGroups provides view changes and view merges
Exploiting locality

- Divide networking into regions
  - have one controller or a small number of controllers responsible for each region
Exploiting locality

- Divide networking into regions
  - have one controller or a small number of controllers responsible for each region
Exploiting locality

- Divide networking into regions
  - have one controller or a small number of controllers responsible for each region

```
Controllers

Network Devices

less communication east-west here

more communication north south here
```
Exploiting locality

• This is (loosely) what ONOS does
  • The MD-SAL could provide metadata to automatically distribute state per region

• Locating controllers near their devices can
  • Minimize latency/improve performance
  • Lower risk of separating devices from controllers

• Controller for each region can be a cluster also with lower risk of intra-cluster partitions
Scale-out
Using majorities prevents scale-out

• If we have N controllers and use majorities, then \((N/2)+1\) must be involved in each action.

• That means our cluster can handle less than twice the load of a single controller!

• That’s not exactly scale-out!
Scaling out while staying consistent

• If the workload is read heavy
  • Read from K controllers for a small K, e.g., 1 or 2
  • Write to (N-K)+1 controllers; guarantees overlap
  • Reads are fast and scale out, writes are slow

• If you can partition your state
  • Map each partition to a subset of the controllers
  • Use majorities of the subsets
  • Get scale-out of N/(size of subsets)
What about OpenDaylight?
What we do today

• Clustering Services
  • Based on Infinispan replicated caches
    • Which are in turn based on JGroups
    • Covered in June 24, 2013 Tech Work Stream Call
    • It seems as though it’s AP, but not clear
  • JGroups seems to have a more formal model

• MD-SAL
  • Uses ZeroMQ for replication
  • Doesn’t seem to have a formal model
What we can do going forward

• ZooKeeper
  • Strong consistency, stops working on minority side of partitions if they happen

• JGroups
  • Seems to provide flexible ways to handle partitions and healing
  • Help better expose this through Infinispan

• Akka
  • Seems to be best of breed AP with well-defined consistency both with and without partitions
Design Summit on consistency after lunch in Napa room
Backup Slides
Consistency in the controller vs. the network

• Academic work on network updates:
  • frenetic, pyretic, Mahajan & Wattenhofer’s consistent updates, etc.

• Focuses on maintaining invariants in forwarding behavior, e.g., no loops, even during updates to forwarding tables
  • Very useful, but different from this talk
What about 2PC, active-backup, etc.

• Two Phase Comment (2PC)
  • Special case of sloppy quorum with $R=1$, $W=N$

• Active-backup
  • Special case where $N=2$, $R=1$, $W=2$
  • Active replicates writes to backup so $W=2$
References and pointers if you’re interested
References

• Academic papers:
  • Pyretic [NSDI ‘13]
  • Consistent Updates [HotNets ‘13]
  • CAP for Networks [HotSDN ‘13]
  • Exploiting Locality [HotSDN ‘13]
  • Frenetic (Network Update) [SIGCOMM ‘12]
  • Logically Centralized [HotSDN ‘12]
  • Kandoo [HotSDN ‘12]
  • F1 [SIGMOD ‘12]
  • Spanner [OSDI ‘12]
  • Onix [OSDI ’10]
  • MegaStore [SIGMOD ’08]
  • Dynamo [SOSP ‘07]
References

• Open source systems:
  • CP: ZooKeeper
  • AP: Cassandra, MongoDB, Riak, Akka, Infinispan?
References

• Long blog post looking at a lot of data on whether network partitions are rare or common and if they can be prevented
  • http://aphyr.com/posts/288-the-network-is-reliable
  • Conclusion is that they can be made very rare with significant expertise and expense

• CAP Twelve Years Later
  • http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
  • Eric Brewer’s recent thoughts on the CAP theorem
References

• The road to Akka Cluster and beyond
  • http://www.slideshare.net/jboner/the-road-to-akka-cluster-and-beyond
  • Good overview of distributed systems—much better than this deck—and a bit about Akka