The Raft Consensus Algorithm

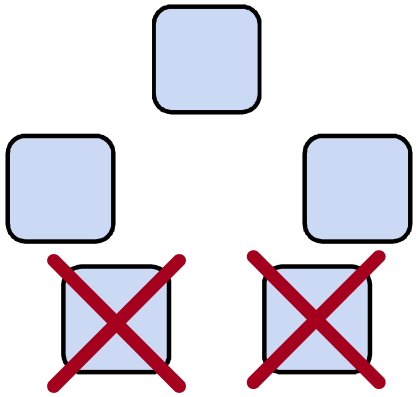
Diego Ongaro John Ousterhout

Stanford University

http://raftconsensus.github.io

What is Consensus?

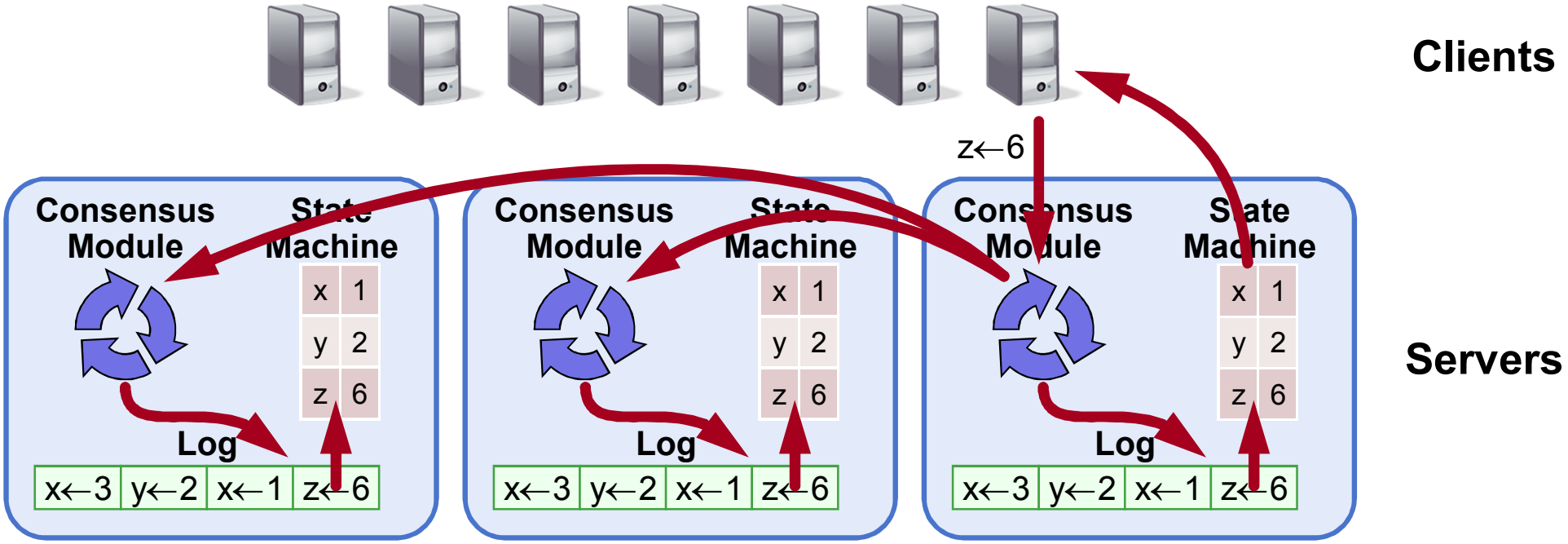
- **Agreement on shared state (single system image)**
- **Recovers from server failures autonomously**
  - Minority of servers fail: no problem
  - Majority fail: lose availability, retain consistency



Servers

- **Key to building consistent storage systems**

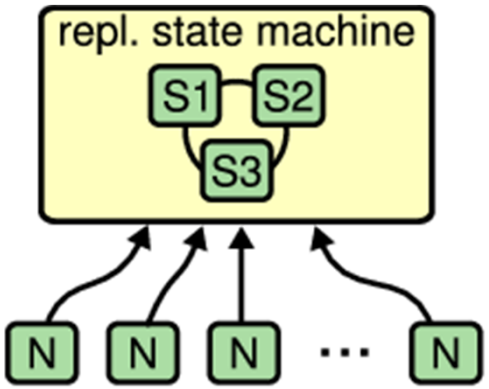
Replicated State Machines

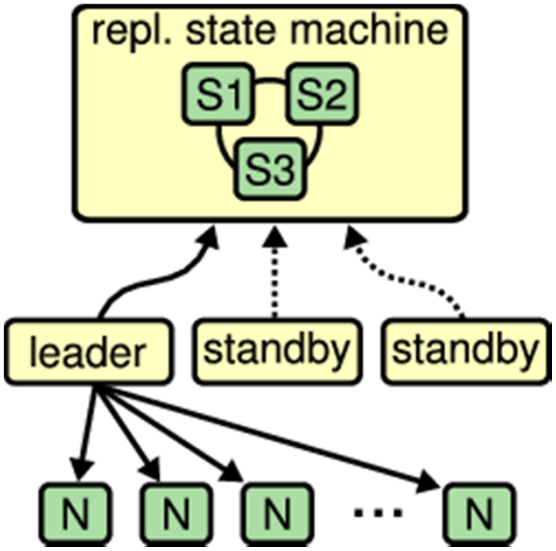


- **Replicated log ⇒ replicated state machine**
  - All servers execute same commands in same order
- **Consensus module ensures proper log replication**
- **System makes progress as long as any majority of servers are up**
- **Failure model: fail-stop (not Byzantine), delayed/lost messages**

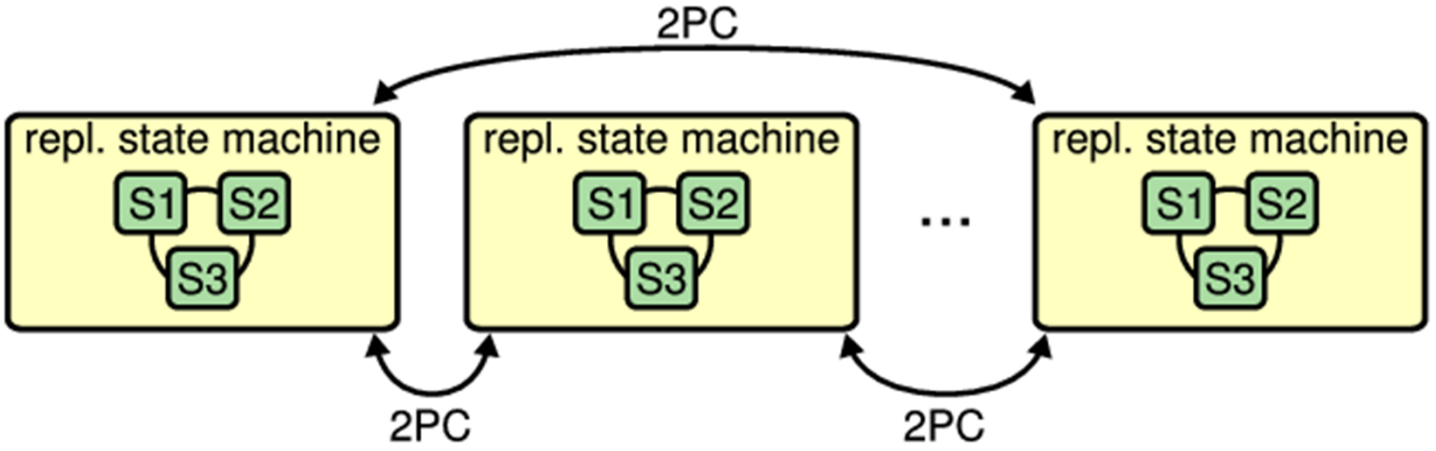
How Is Consensus Used?

- **Top-level system configuration**





- **Replicate entire database state**



Existing Consensus Algorithms

- **Paxos (Leslie Lamport)**

  "The dirty little secret of the NSDI community is that at most five people really, truly understand every part of Paxos ;-)." – NSDI reviewer

  "There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system…the final system will be based on an unproven protocol." – Chubby authors

- **Viewstamped Replication (Brian Oki, Barbara Liskov)**
  - Hadn't been revisited

Raft's Design for Understandability

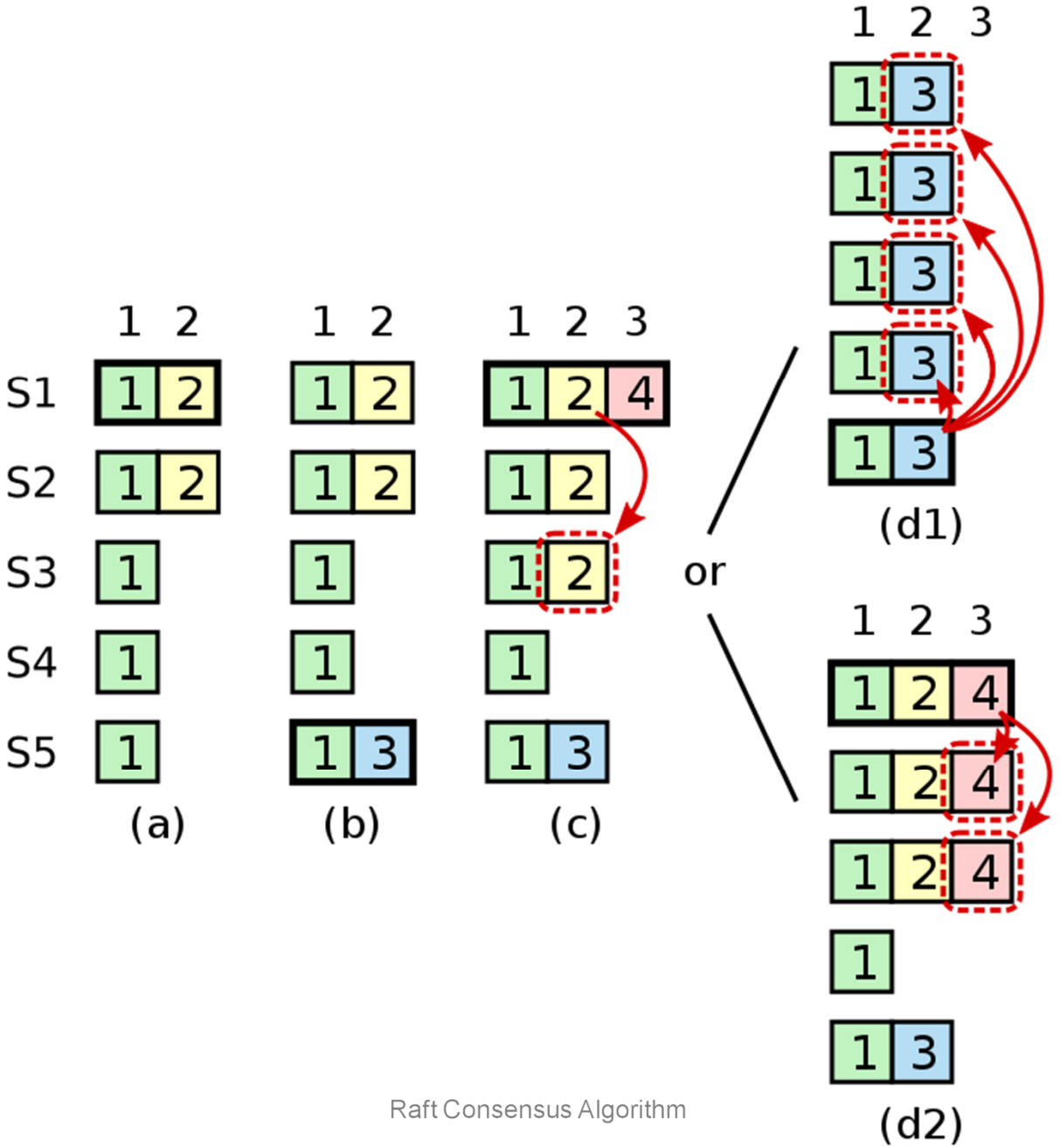
- **We wanted an algorithm optimized for building real systems**
  - Must be correct, complete, and perform well
  - Must also be understandable
- **"What would be easier to understand or explain?"**
  - Fundamentally different decomposition than Paxos
  - Less complexity in state space
  - Less mechanism

Raft Overview

1. **Leader election**
   - Select one of the servers to act as cluster leader
   - Detect crashes, choose new leader
2. **Log replication (normal operation)**
   - Leader takes commands from clients, appends them to its log
   - Leader replicates its log to other servers (overwriting inconsistencies)
3. **Safety**
   - Only a server with an up-to-date log can become leader

RaftScope Visualization

Deferred Commitment of Inherited Entries



Core Raft Review

1. Leader election

- Heartbeats and timeouts to detect crashes
- Randomized timeouts to avoid split votes
- Majority voting to guarantee at most one leader per term

2. Log replication (normal operation)

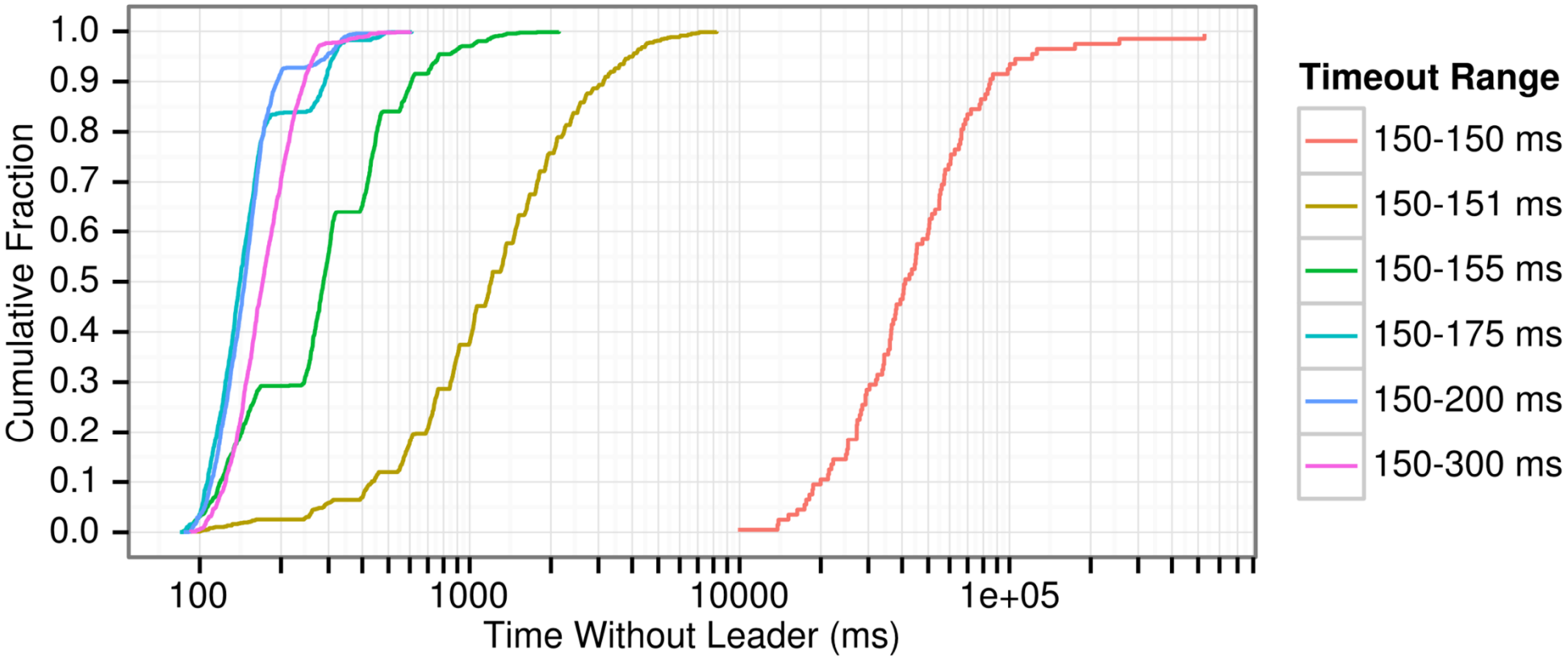
- Leader takes commands from clients, appends them to its log
- Leader replicates its log to other servers (overwriting inconsistencies)
- Built-in consistency check simplifies how logs may differ

3. Safety

- Only elect leaders with all committed entries in their logs
- New leader defers committing entries from prior terms

Randomized Timeouts

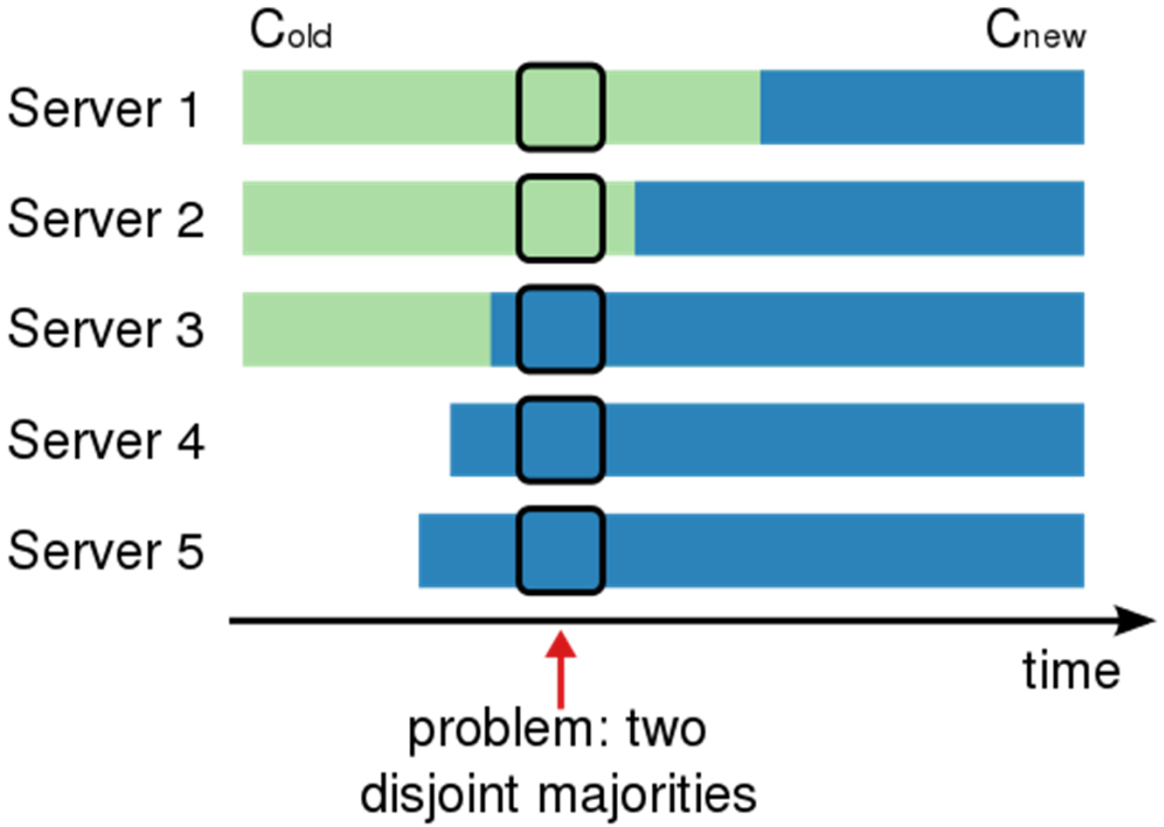
- **How much randomization is needed to avoid split votes?**



- **Conservatively, use random range ~10x network latency**

Membership Changes

- **Problem: changing from one cluster configuration directly to another can be unsafe**



Simplification for Safety

- **Paper describes *joint consensus* approach that uses an intermediate phase to avoid such problems**
- **Better approach: restrict to single-server additions and removals**
  - Majorities still overlap
  - Described in thesis (paper was already published)

Membership Change Approach

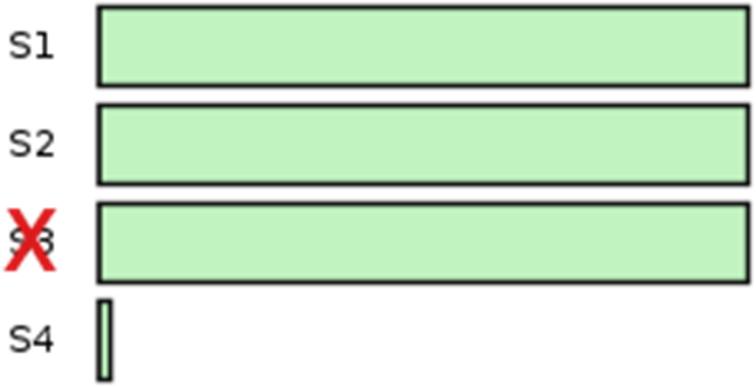
1. **Leader appends Cnew entry to log, replicates it**
2. **Cnew takes effect on a server as soon as it is added to that server's log**
   - Each server always uses the latest configuration entry in its log (regardless of whether that entry is committed)
   - Leader uses Cnew to determine commitment of the Cnew entry
3. **Once Cnew is committed, the configuration change is complete**
   - Further changes can then be started

- **Cluster continues servicing requests throughout change**

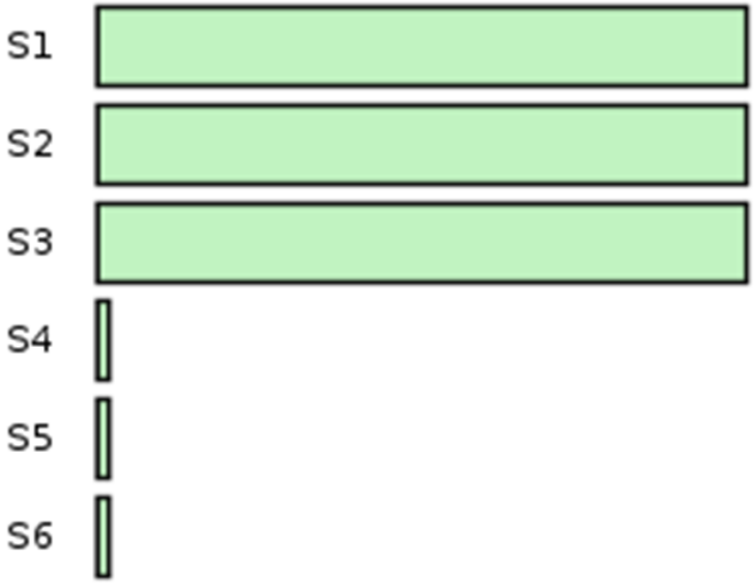
Availability Issues

1. Catching up new servers
2. Removing the current leader
3. Disruptive servers

Catching up new servers

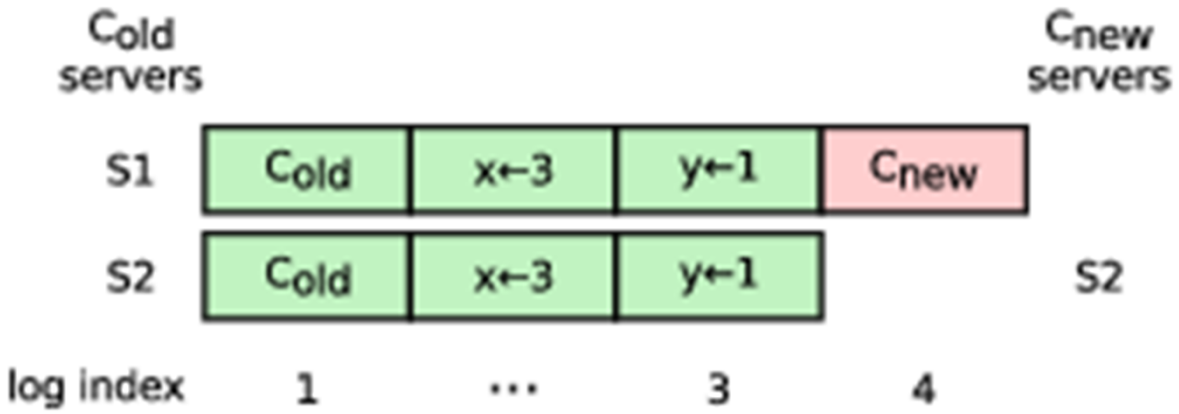


Failure of S3 while adding S4

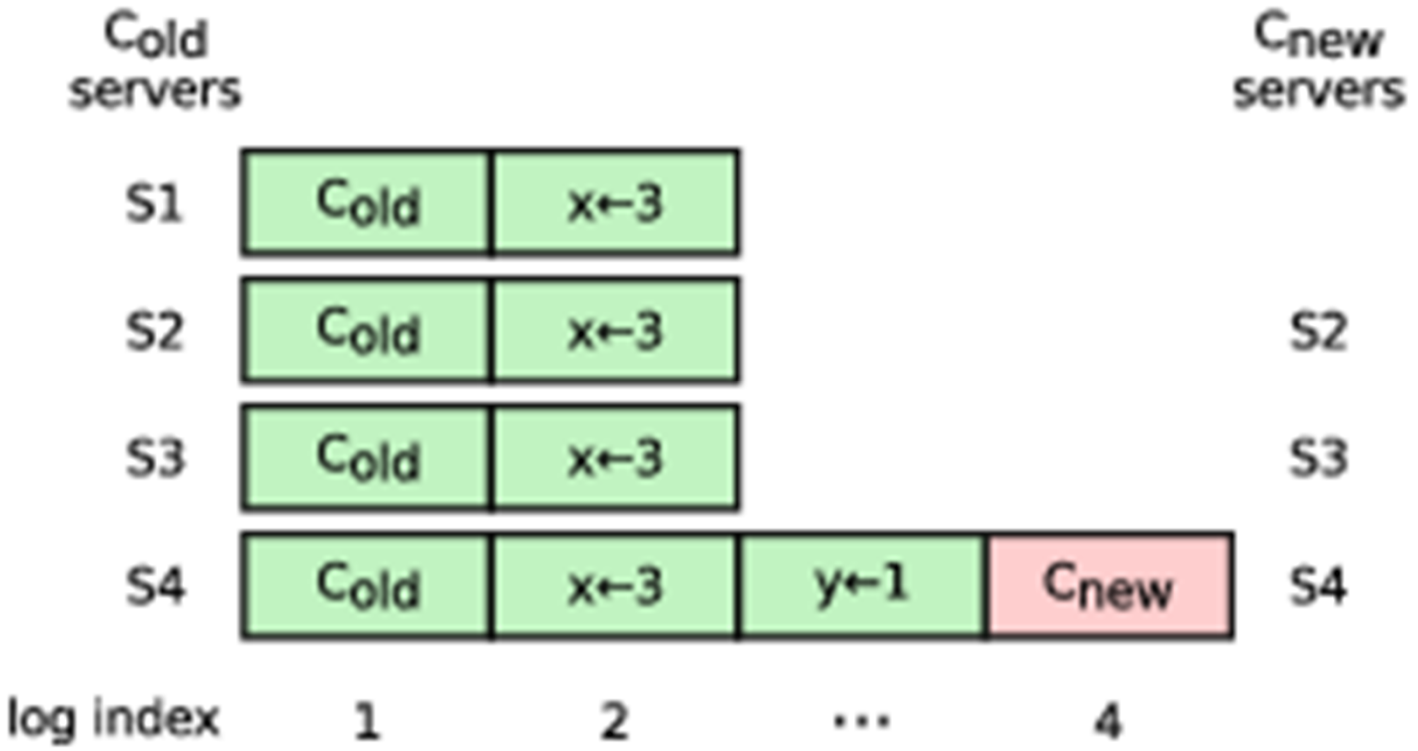


Adding S4-S6 in quick succession

Removing the current leader



Disruptive Servers



Conclusions

- **Consensus widely regarded as difficult**
- **Raft designed for understandability**
  - Easier to teach in classrooms
  - Better foundation for building practical systems
- **Pieces needed for a practical system:**
  - Cluster membership changes (simpler in thesis)
  - Log compaction (expanded tech report/thesis)
  - Client interaction (expanded tech report/thesis)
  - Evaluation (thesis: understandability, correctness, leader election & replication performance)

Questions

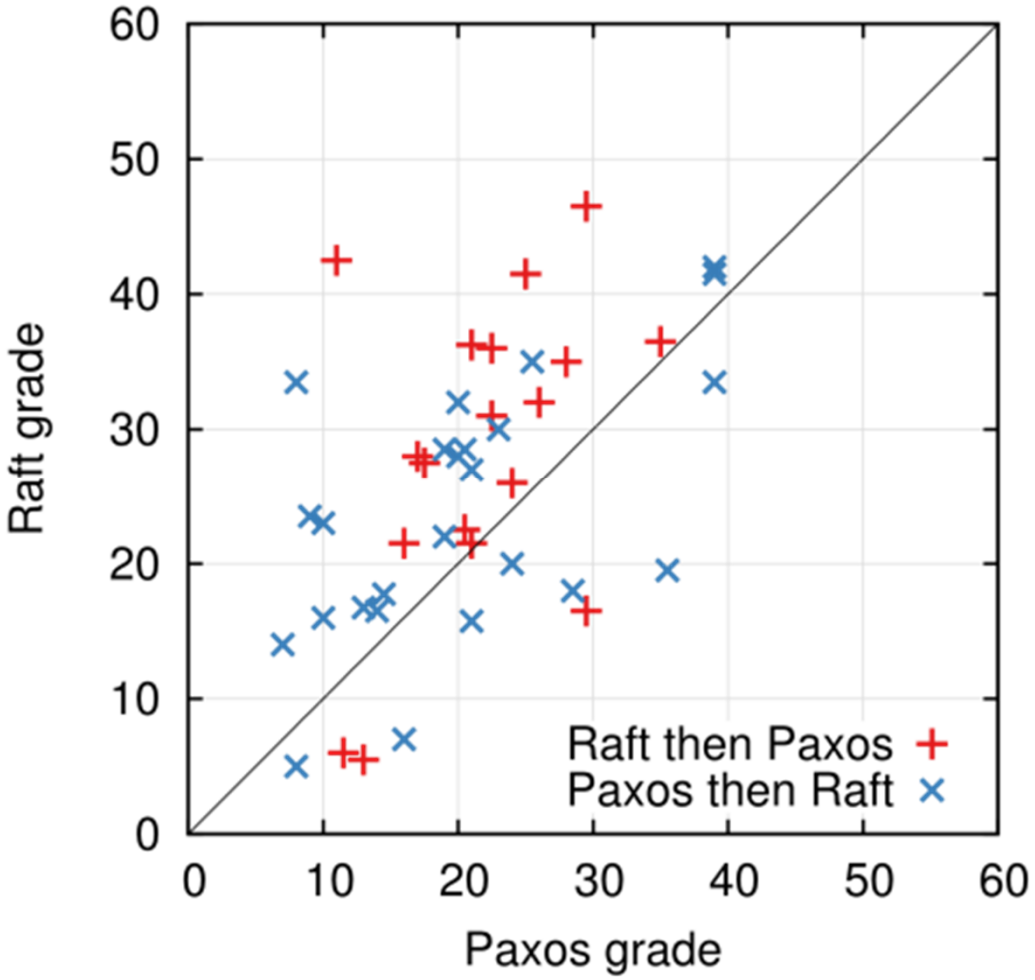
raftconsensus.github.io

raft-dev mailing list, or ongaro@cs if you're shy

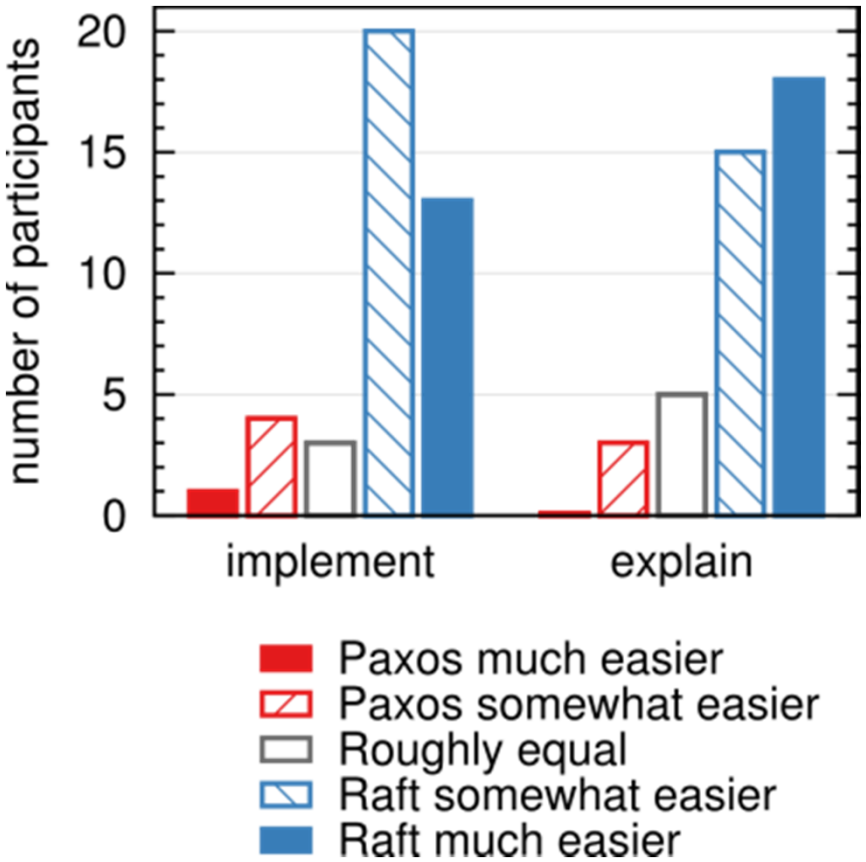
Extra Slides

Raft User Study

Quiz Grades



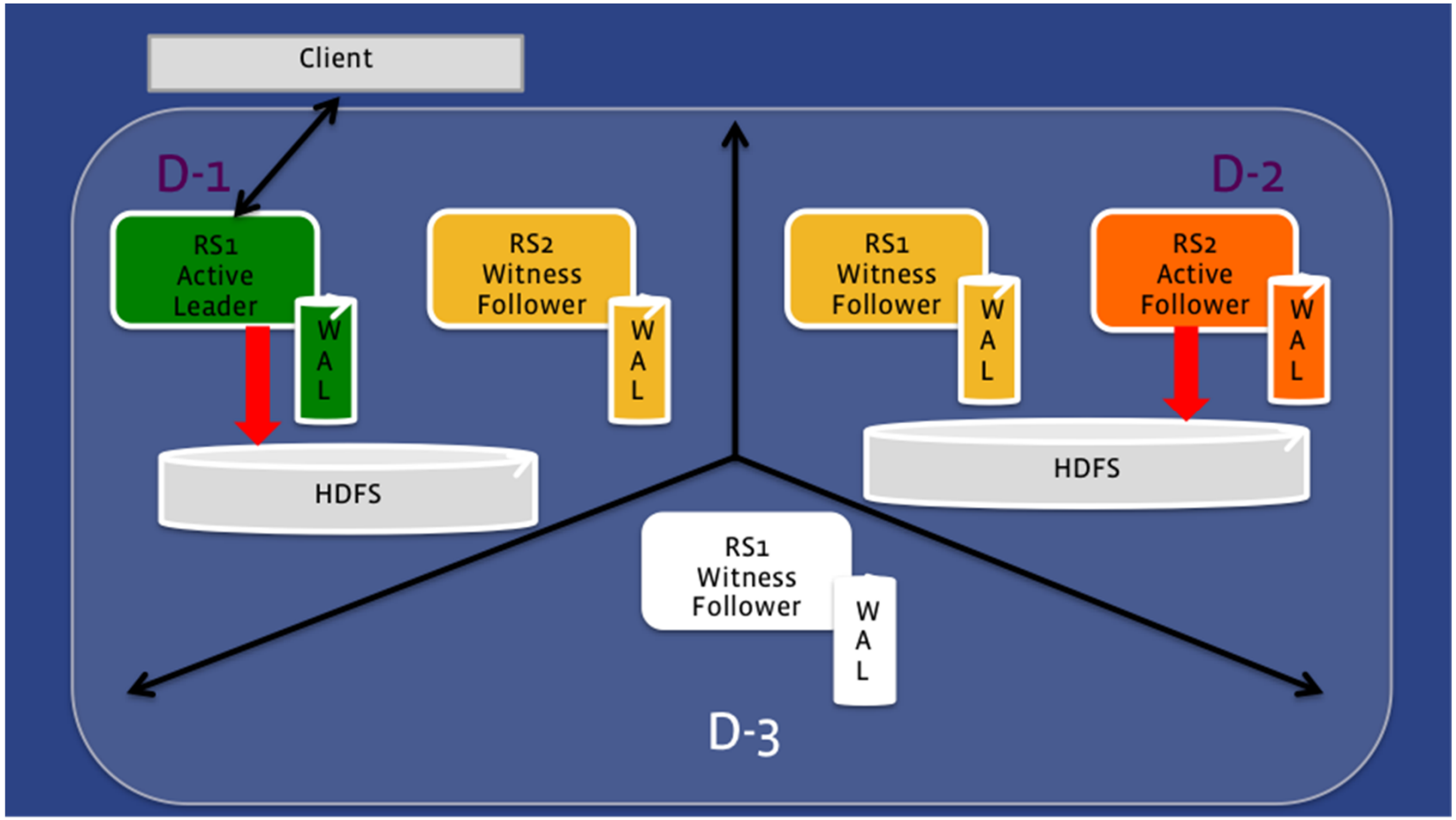
Survey Results



Raft Implementations (Stale)

go-raft	Go	Ben Johnson (Sky) and Xiang Li (CoreOS)
kanaka/raft.js	JS	Joel Martin
hashicorp/raft	Go	Armon Dadgar (HashiCorp)
rafter	Erlang	Andrew Stone (Basho)
ckite	Scala	Pablo Medina
kontiki	Haskell	Nicolas Trangez
LogCabin	C++	Diego Ongaro (Stanford)
akka-raft	Scala	Konrad Malawski
floss	Ruby	Alexander Flatten
CRaft	C	Willem-Hendrik Thiart
barge	Java	Dave Rusek
harryw/raft	Ruby	Harry Wilkinson
py-raft	Python	Toby Burress
...		

Facebook HydraBase Example



https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/