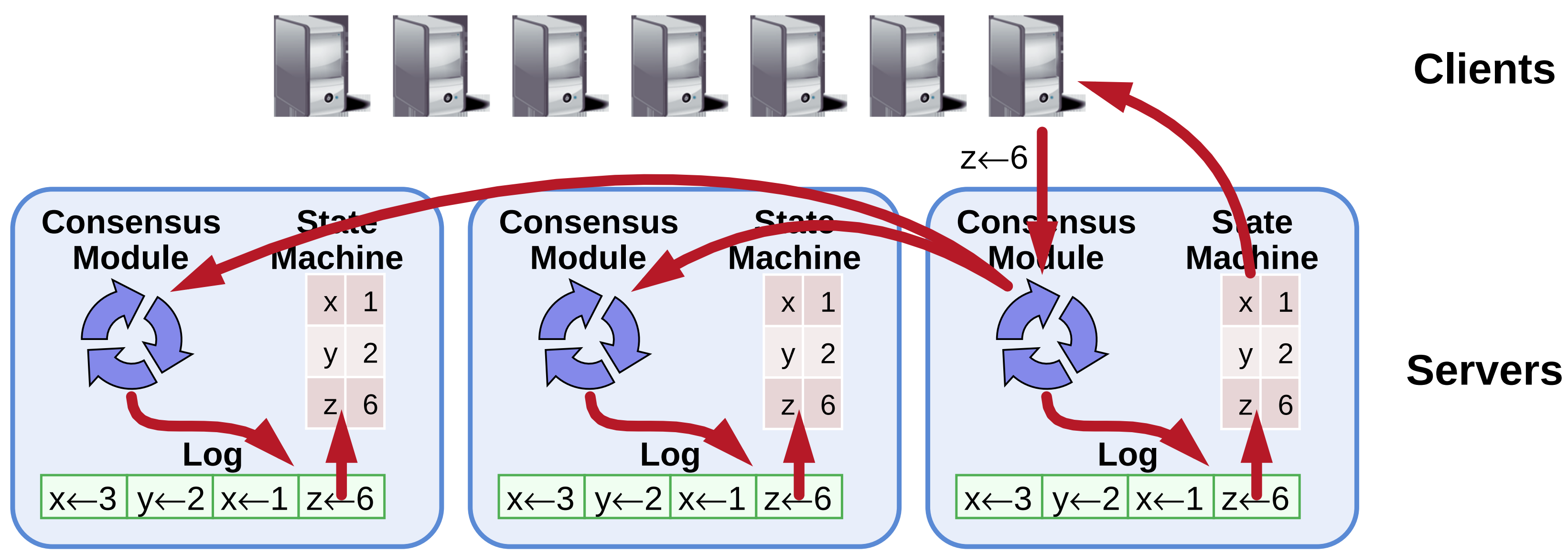
The Raft Consensus Algorithm and Implementing Raft in C++

Diego Ongaro, August 2015

Replicated State Machines

Typical architecture for consensus systems



- Replicated log ⇒ replicated state machine
  - All servers execute same commands in same order
- Consensus module ensures proper log replication

Raft

- Algorithm for implementing a replicated log
- System makes progress as long as any majority of servers up
- Failure model: fail-stop (not Byzantine), delayed/lost msgs
- Designed for understandability

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 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

just leader election today

Leader Election Review

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

LogCabin

- Started as research platform for Raft at Stanford
- Developed into production system at Scale Computing
- Network service running Raft replicated state machine
- Data model: hierarchical key-value store
- Written in gcc 4.4's C++0x (Rust was pre-0.1)

github.com/logcabin

C++ Wins

- Fast
- Easy to predict speed of language features
- No GC pauses
  - Raft election timeouts can be very low
- As low-level as you want
  - LogCabin forks a child process to write a consistent snapshot of its state machine
- Resource leaks are rarely an issue
  - Move semantics, `std::unique_ptr` in C++11
  - LogCabin has 47 calls to `new`, only 6 calls to `delete`
- All this is also true of Rust

Libraries in C++

- LogCabin is nearly* self-contained *protobuf and gtest libraries are great
  - Contains event loop (epoll), RPC system
  - Easier to debug, understand system end-to-end
  - Learned a lot
- Hard to depend on libraries
  - No standard packaging system
  - Libraries use different subsets of C++
    - Exceptions? Lambdas? `shared_ptr`?
  - Thread safety described in documentation (lol)
- Hard to extract LogCabin's Raft implementation as a library
- Rust: Cargo packaging, crates.io, rich type system

Thread Safety Is Hard

- LogCabin uses Monitor style
  - One mutex per object
  - All public methods hold the mutex the entire time (except when blocked on a condition variable)
- No language support, not compiler-enforced

```
// occasionally hangs forever on shutdown
void threadMain() {
    while (!exiting) {
        std::unique_lock lockGuard(mutex);
        // ... do stuff ...
        condition.wait(lockGuard);
    }
}
```

Equivalent Rust code: `exiting` wouldn't be in scope

Conclusion

- Raft: designed for understandability
  - Randomized leader election approach
  - Videos of log replication and safety on Raft website
  - Paper/dissertation also include:
    - Cluster membership changes (simpler in dissertation)
    - Log compaction
    - Client interaction
    - Understandability, correctness, performance evaluation
- In LogCabin implementation, C++
  - Offers good and predictable performance
  - Is missing a healthy library ecosystem
  - Allows memory and thread safety bugs
- Excited to see Rust and raft-rs grow

Questions

raft.github.io

raft-dev mailing list

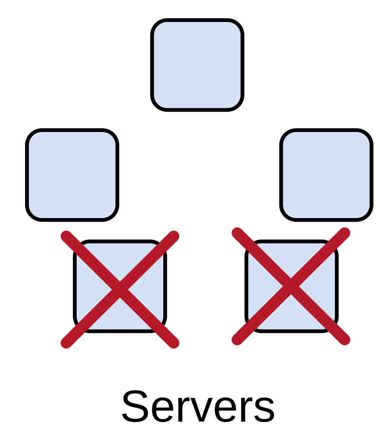
Backup Slides

Motivation

- Goal: shared key-value store (state machine)
- Host it on a single machine attached to network
  - Pros: easy, consistent
  - Cons: prone to failure
- With Raft, keep consistency yet deal with failures

What is consensus

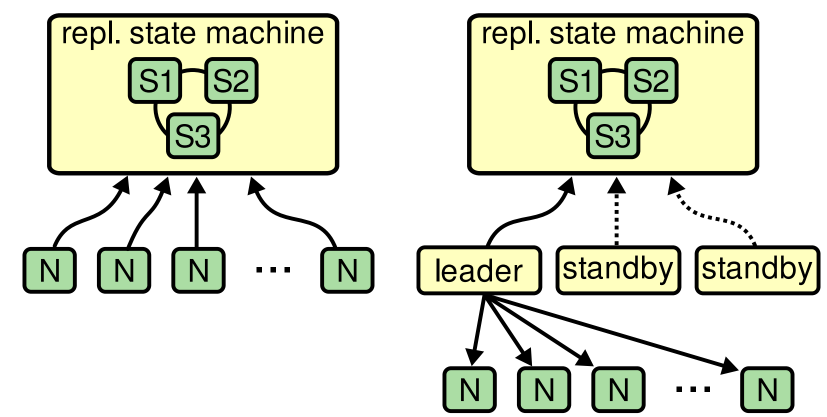
- Agreement on shared state (single system image)
- Recovers from server failures autonomously
  - Minority of servers fail: no problem
  - Majority fail: lose availability, retain consistency



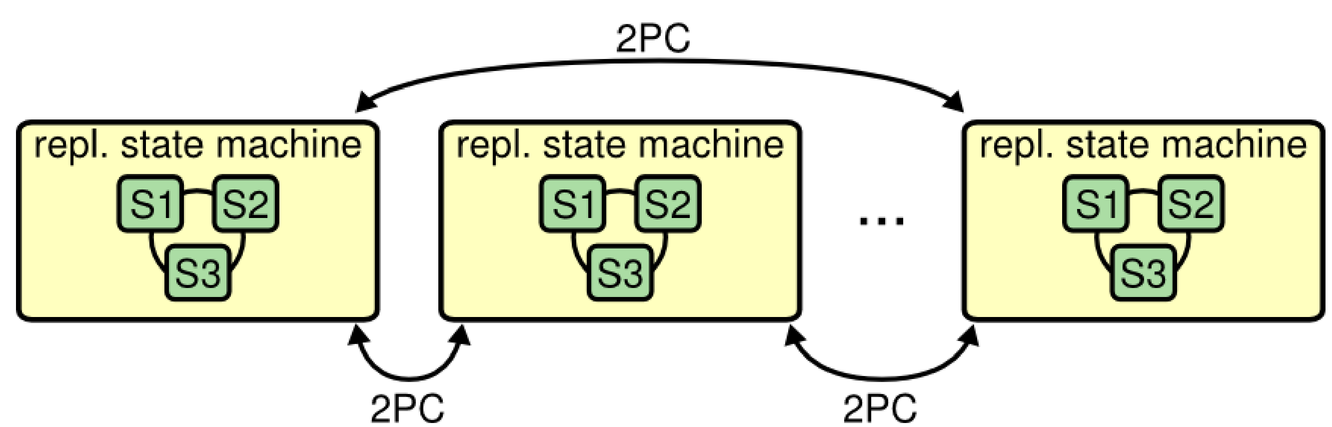
- Key to building consistent storage systems

How Is Consensus Used?

Top-level system configuration



Replicate entire database state



Paxos Protocol

- Leslie Lamport, 1989
- Nearly synonymous with consensus

"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

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

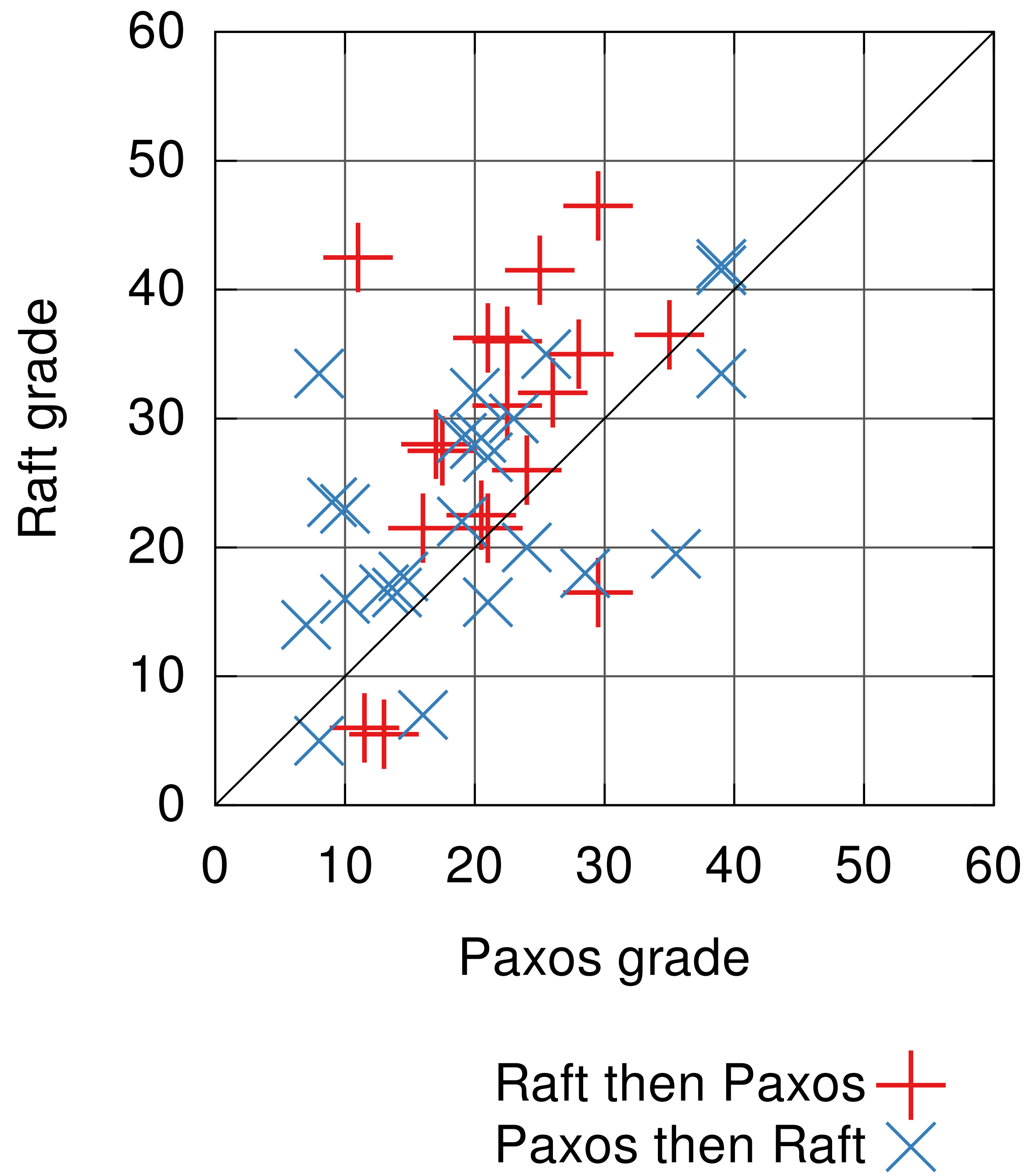
“What would be easier to understand or explain?”

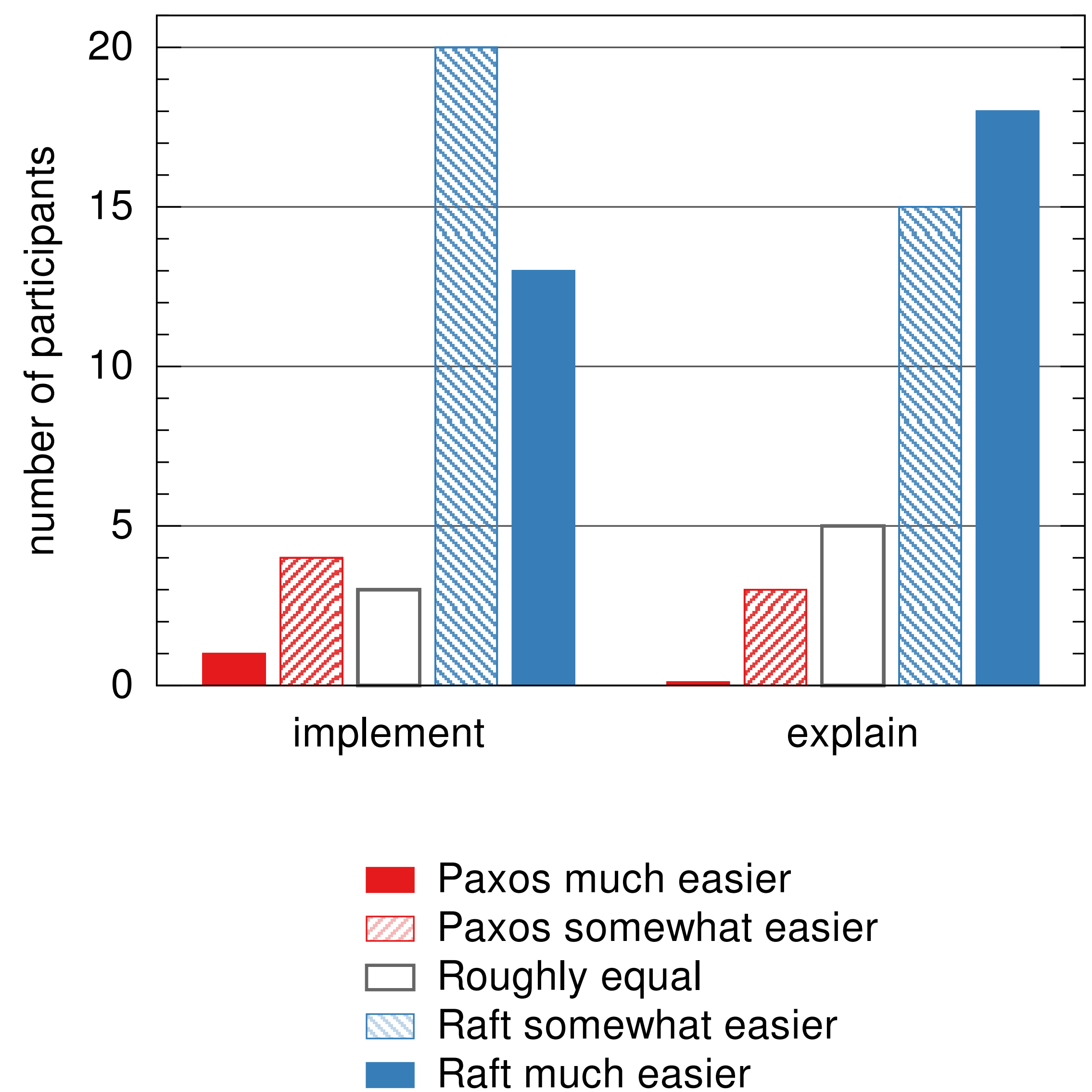
- Fundamentally different decomposition than Paxos
- Less complexity in state space
- Less mechanism

Safe Shutdown Is Hard

- `Globals` class constructs and destroys all major objects in correct order
  - (Config file, event loop, signal handlers, storage, Raft, state machine, RPC handlers)
- Still hard to get object lifetimes correct
  - RPCs, background threads
- Rust: compiler checks lifetimes, no dangling pointers

Raft User Study



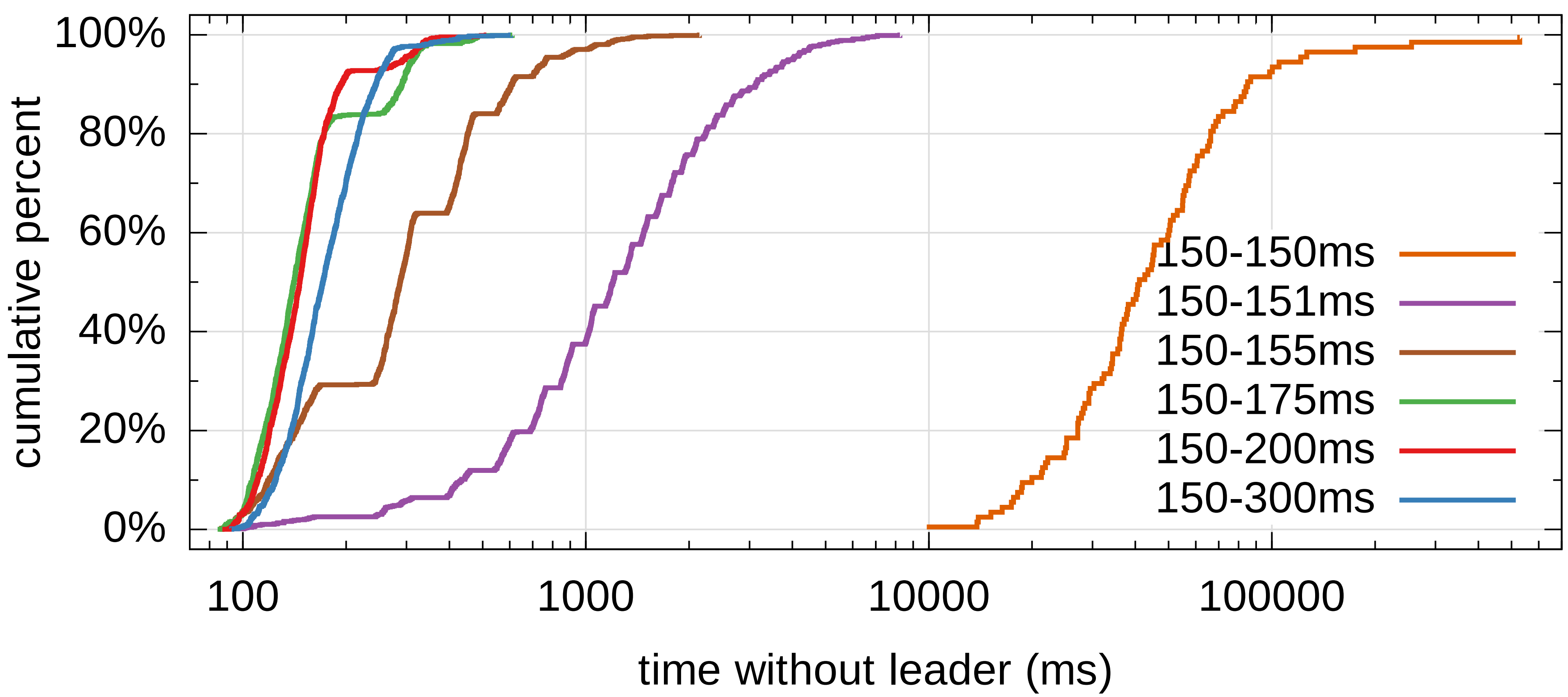


Core Raft Review

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 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

Raft Implementations

Name	Primary Authors	Language	License
etcd/raft	Blake Mizerany, Xiang Li and Yicheng Qin (CoreOS)	Go	Apache 2.0
go-raft	Ben Johnson (Sky) and Xiang Li (CMU, CoreOS)	Go	MIT
hashicorp/raft	Armon Dadgar (hashicorp)	Go	MPL-2.0
copycat	Jordan Halterman	Java	Apache2
LogCabin	Diego Ongaro (Stanford, Scale Computing)	C++	ISC
akka-raft	Konrad Malawski	Scala	Apache2
kanaka/raft.js	Joel Martin	Javascript	MPL-2.0
rafter	Andrew Stone (Basho)	Erlang	Apache2
OpenDaylight	Moiz Raja, Kamal Rameshan, Robert Varga (Cisco), Tom Pantelis (Brocade)	Java	Eclipse
liferaft	Arnout Kazemier	Javascript	MIT
skiff	Pedro Teixeira	Javascript	ISC
ckite	Pablo Medina	Scala	Apache2
willemt/raft	Willem-Hendrik Thiart	C	BSD

Copied from Raft website, probably stale.