Consensus: Bridging Theory and Practice

Thesis Proposal

Diego Ongaro

June 14, 2013

While working on RAMCloud, quickly found that building correct fault-tolerant systems is hard

- Building blocks not obvious (which and how?)

Consensus is necessary for consistent systems

- Fundamental, well-defined problem

1. Consensus algorithms are not widely understood by systems-builders
2. Many details needed to build a complete replicated state machine are missing or unprincipled
3. Few implementations of consensus are available, complete, reliable, maintainable, and usable
4. Where to apply consensus is not widely understood by systems-builders

Outline

Current status / existing work

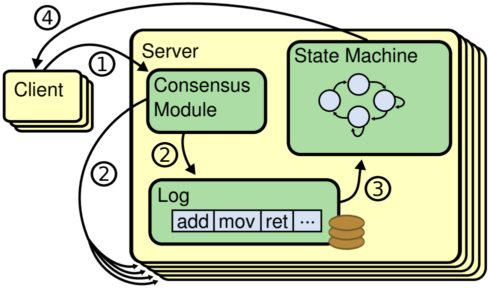
Plan

Future work

Current status

- Raft consensus algorithm
  - Implementation (nearly) done
  - Formal spec in TLA+, English proof of safety
- Reconfiguration
  - Implementation (nearly) done
- Log compaction
  - Implementation in progress (snapshotting)
- User study
  - Demonstrated Raft easier to learn than Paxos
- Implementation artifacts
  - Raft embedded as part of LogCabin
  - LogCabin's data model still in flux
  - Only unit tests

Context: replicated state machines



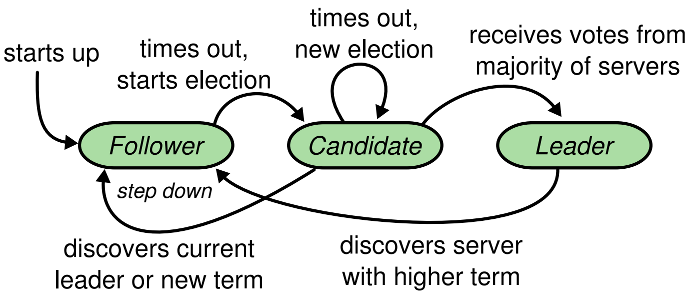
- State machine defines data structure
  - Interface is application-specific
- Replicated log feeds commands to state machine
- Same log ⇒ same sequence of states, outputs
- Raft and Multi-Paxos are two consensus algorithms to manage the replicated log

Raft overview

- Design for understandability
  - Problem decomposition, state space reduction
- Strong leader
  - Only leaders (and candidates) issue requests
  - Servers never pull data

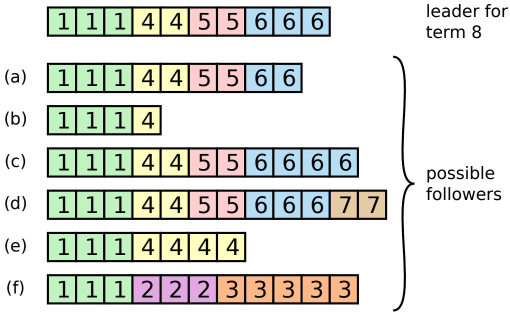
1. Leader election
2. Log replication
3. Safety

1. Leader election



- Each server may vote once per term
- Majority requirement ensures one leader per term
- Wait for next timeout in case of split vote
- Randomized timeouts (e.g., 150-300ms) for liveness

2. Log replication



- Leader assumes its log is "the truth"
- Sends entries to followers
- Consistency check: follower rejects entry unless the preceding entry's term matches
- On reject, step back and try again
- Followers discard conflicting log suffixes

3. Safety

Leader election rigged:

- Server only grants vote if candidate's log is at least as current as its own, *i.e.*, if

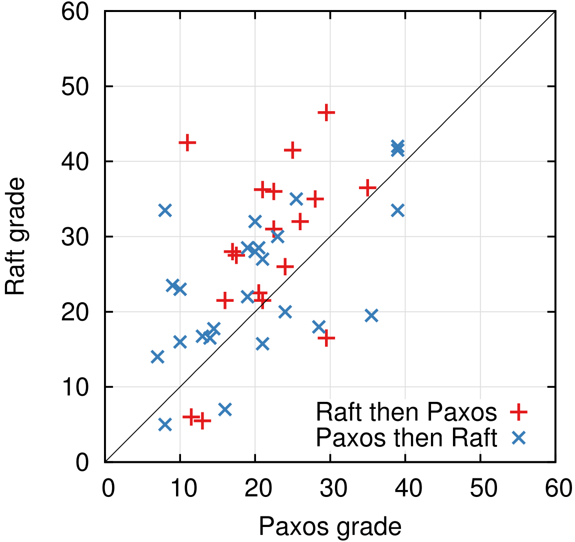
  candidate's (last log term, last log index) $\geq$ voter's

A leader marks a log prefix committed when:

1. It is stored on a majority of servers, and
2. This leader created the last entry

Together, guarantees that every future leader has all committed entries.

User study


60
50
40
30
20
10
0
Raft grade
0
10
20
30
40
50
60
Paxos grade
Raft then Paxos
Paxos then Raft

Reconfiguration (cluster membership changes)

- Leverage the log to order configuration changes
- Lamport's approach (α) won't work with strong leadership
  - Servers in new group need to be up-to-date to become leader
- Raft's approach: switch to a transitional configuration, then to the new configuration
- Serial log operation enables concurrent client requests (without limit)

Log compaction

- Need to compact log somehow (cleaning, snapshotting, or write-ahead log)
- Despite strong leader, simplest approach is that servers compact independently
  - *E.g.*, no need to worry about leader changes
- Snapshotting is sane for in-memory state machines
  - $10\times$ memory-sized log $\Rightarrow$ 10% b/w overhead
  - Recovery time mitigated by majority approach
- Leverage OS fork() to take consistent snapshot without complicating state machine code

Related work

- Paxos
  - Most popular consensus algorithm today
  - Problem decomposition makes it less suitable for understanding and systems-building
- Leader-based consensus algorithms: Viewstamped Replication (Revisited) and Zab
  - Raft is similar in structure but “sweeter” in the details
  - VR not implemented?; VRR partially implemented; Zab built into ZooKeeper

Outline

Current status / existing work

Plan

Future work

Plan

Graduate June 2014

1. Finish implementing snapshotting
2. Address paper reviews
3. Extract libraft from LogCabin
4. Do system-level testing of libraft/LogCabin

Concurrently:

- Support growing user community
- Give more talks
- Write thesis

Thesis outline: Raft algorithm

1. Core Raft consensus algorithm
   - Algorithm (main body of paper)
   - Discussion of design choices
2. Cluster membership changes
   - Basic algorithm and catch-up, etc
   - Summary of and comparison with Lamport, DM, Zab, ... approaches
3. Log compaction
   - Discussion of solution space
   - Description of algorithm used in LogCabin
4. Client interaction
   - Providing linearizability
   - How clients find the leader

Thesis outline: Raft algorithm (continued)

5. Raft user study
   - Methodology, results, conclusions
6. Proof of core Raft algorithm
   - Formal spec (TLA) and proof (English)
7. Related work: Paxos
   - Detailed explanation of algorithm presented in DM and John's talks
   - What's wrong with Paxos
8. Related work: Leader-based
   - Summary of original VR, changes in VR Revisited, Zab
   - What's wrong with these

Thesis outline: Implementation

9. libraft: Raft library
   - API
   - Internals (threading model, etc.)
   - Testing
10. LogCabin service
    - Old API
    - New API
    - Experience from RAMCloud
11. 3rd-party Raft implementations
    - Brief summary of the important ones and any significant deviations in implementation

Outline

Current status / existing work

Plan

Future work

Randomized testing

Goal: increase confidence about:

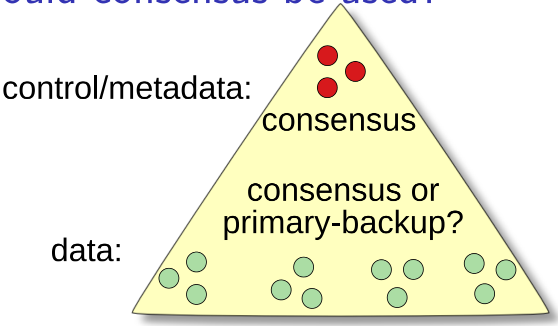
- Basic Raft implementation
- Reconfiguration
- Snapshotting
- Writing to disk

Randomly kill, unplug, delay, reconfigure
Clients shouldn't notice a thing

What should LogCabin's data model be?

- Started as a log interface, got sucky
  - Garbage collection annoying
  - Ordering not desired
- Chubby, ZooKeeper provide:
  - Random access to data (hierarchical KV store)
  - Conditional writes
  - Mutual exclusion (leases or ephemeral nodes)
  - Notification mechanism (blocking or wait-free)
  - Massive pipelining & stale reads (ZooKeeper)
- Important question but hard to evaluate

Where should consensus be used?



Consensus	**Primary-Backup**
Reacts to failures autonomously	Failures depend on external system
Needs $2f + 1$ machines	Needs $f + 1$ machines
Masks latency in minority	Allows even numbers of machines

It seems large systems should use primary-backup for data, but Megastore and Spanner use consensus?

Summary

- Bridging the gap between theory and practice for consensus
- New consensus algorithm, user study showed easier to understand than Paxos
- Working out details needed for a complete system
- Implementation progressing; need to expose as library and service, do randomized system testing
- Some interesting open research questions
- Planning to graduate June 2014