The Raft Consensus Algorithm

Diego Ongaro and John Ousterhout

November 2015
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
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
- System makes progress as long as any majority of servers up
- Failure model: fail-stop (not Byzantine), delayed/lost msgs
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
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

or https://raft.github.io/raftscope-replay/
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
Conclusion

- 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 dissertation)
  - Log compaction (expanded tech report/dissertation)
  - Client interaction (expanded tech report/dissertation)
  - Evaluation (dissertation: understandability, correctness, leader election & replication performance)
Questions

raft.github.io

raft-dev mailing list
How Is Consensus Used?

Top-level system configuration

Replicate entire database state

2PC
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 User Study

Raft grade vs. Paxos grade

- Raft then Paxos
- Paxos then Raft

Number of participants

- Paxos much easier
- Paxos somewhat easier
- Roughly equal
- Raft somewhat easier
- Raft much easier
Randomized Timeouts

• How much randomization is needed to avoid split votes?

Conservatively, use random range ~10x network latency
Raft Implementations
<table>
<thead>
<tr>
<th>Name</th>
<th>Primary Authors</th>
<th>Language</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>RethinkDB/clustering</td>
<td></td>
<td>C++</td>
<td>AGPL</td>
</tr>
<tr>
<td>etcd/raft</td>
<td>Blake Mizerany, Xiang Li and Yicheng Qin</td>
<td>Go</td>
<td>Apache2.0</td>
</tr>
<tr>
<td>LogCabin</td>
<td>Diego Ongaro (Stanford)</td>
<td>C++</td>
<td>ISC</td>
</tr>
<tr>
<td>go-raft</td>
<td>Ben Johnson (Sky) and Xiang Li (CMU, CoreOS)</td>
<td>Go</td>
<td>MIT</td>
</tr>
<tr>
<td>hashicorp/raft</td>
<td>Armon Dadgar (hashicorp)</td>
<td>Go</td>
<td>MPL-2.0</td>
</tr>
<tr>
<td>hoverbear/raft</td>
<td>Andrew Hobden, Dan Burkert</td>
<td>Rust</td>
<td>MIT</td>
</tr>
<tr>
<td>ckite</td>
<td>Pablo Medina</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>verdi/raft</td>
<td>James Wilcox, Doug Woos, Pavel Panchekha, Zach Tatlock, Xi Wang, Mike Ernst, and Tom Anderson (University of Washington)</td>
<td>Coq</td>
<td>BSD</td>
</tr>
<tr>
<td>OpenDaylight</td>
<td>Moiz Raja, Kamal Rameshan, Robert Varga (Cisco), Tom Pantelis (Brocade)</td>
<td>Java</td>
<td>Eclipse</td>
</tr>
<tr>
<td>zraft_lib</td>
<td>Gunin Alexander</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>kanaka/raft.js</td>
<td>Joel Martin</td>
<td>Javascript</td>
<td>MPL-2.0</td>
</tr>
<tr>
<td>akka-raft</td>
<td>Konrad Malawski</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>rafter</td>
<td>Andrew Stone (Basho)</td>
<td>Erlang</td>
<td>Apache2</td>
</tr>
<tr>
<td>floss</td>
<td>Alexander Flatter</td>
<td>Ruby</td>
<td>MIT</td>
</tr>
<tr>
<td>willemt/raft</td>
<td>Willem-Hendrik Thiart</td>
<td>C</td>
<td>BSD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Copied from Raft website, probably stale.
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, kept in memory
- Written in C++ (gcc 4.4's C++0x)