The Raft Consensus Algorithm and Implementing Raft in C++

Diego Ongaro, August 2015
Replicated State Machines

Typical architecture for consensus systems

- Replicated log $\Rightarrow$ 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)
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

```cpp
// 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

2PC
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

- Raft grade vs Paxos grade
- Points represent participants' choices: Raft then Paxos (+), Paxos then Raft (x)

Bar chart:
- Implement: Paxos much easier
- Explain: Paxos somewhat easier
- Roughly equal
- Raft somewhat easier
- Raft much easier

Comments:
- Paxos much easier
- Paxos somewhat easier
- Roughly equal
- Raft somewhat easier
- Raft much easier
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Primary Authors</th>
<th>Language</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>etcd/raft</td>
<td>Blake Mizerany, Xiang Li and Yicheng Qin (CoreOS)</td>
<td>Go</td>
<td>Apache 2.0</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>copycat</td>
<td>Jordan Halterman</td>
<td>Java</td>
<td>Apache2</td>
</tr>
<tr>
<td>LogCabin</td>
<td>Diego Ongaro (Stanford, Scale Computing)</td>
<td>C++</td>
<td>ISC</td>
</tr>
<tr>
<td>akka-raft</td>
<td>Konrad Malawski</td>
<td>Scala</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>rafter</td>
<td>Andrew Stone (Basho)</td>
<td>Erlang</td>
<td>Apache2</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>liferaft</td>
<td>Arnout Kazemier</td>
<td>Javascript</td>
<td>MIT</td>
</tr>
<tr>
<td>skiff</td>
<td>Pedro Teixeira</td>
<td>Javascript</td>
<td>ISC</td>
</tr>
<tr>
<td>ckite</td>
<td>Pablo Medina</td>
<td>Scala</td>
<td>Apache2</td>
</tr>
<tr>
<td>willemt/raft</td>
<td>Willem-Hendrik Thiart</td>
<td>C</td>
<td>BSD</td>
</tr>
</tbody>
</table>

Copied from Raft website, probably stale.