Egalitarian Paxos

Distributed consensus algorithms in a fault-tolerant environment

Alex Cappiello

Advised by David Andersen

*These slides were used to arrange a poster that was displayed at Carnegie Mellon's Computer Science Undergraduate Research Poster Session on 12/11/13.

Replicated State Machines and Consensus

Motivation: Replicating data across nodes increases the throughput for accessing that data and it remains reachable if one (or more) nodes fail.

Challenge: In a replicated state machine, executing a command must produce the same result regardless of which node handles the request. The role of the consensus algorithm is to resolve dependencies so that the system agrees on an ordering of commands.

Existing Paxos Variants

Canonical Paxos requires extensive optimization to be of any practical use, so many variants exist.

Many rely on having a coordinator (Multi-Paxos, Generalized Paxos). If this node fails, a new coordinator must be chosen, during which time the system is unavailable. Even worse, it can be a bottleneck, limiting throughput.

Mencius has no coordinator. Each replica handles a predefined partition of instances. This load balances effectively, but failures still halt availability while the failure is addressed.

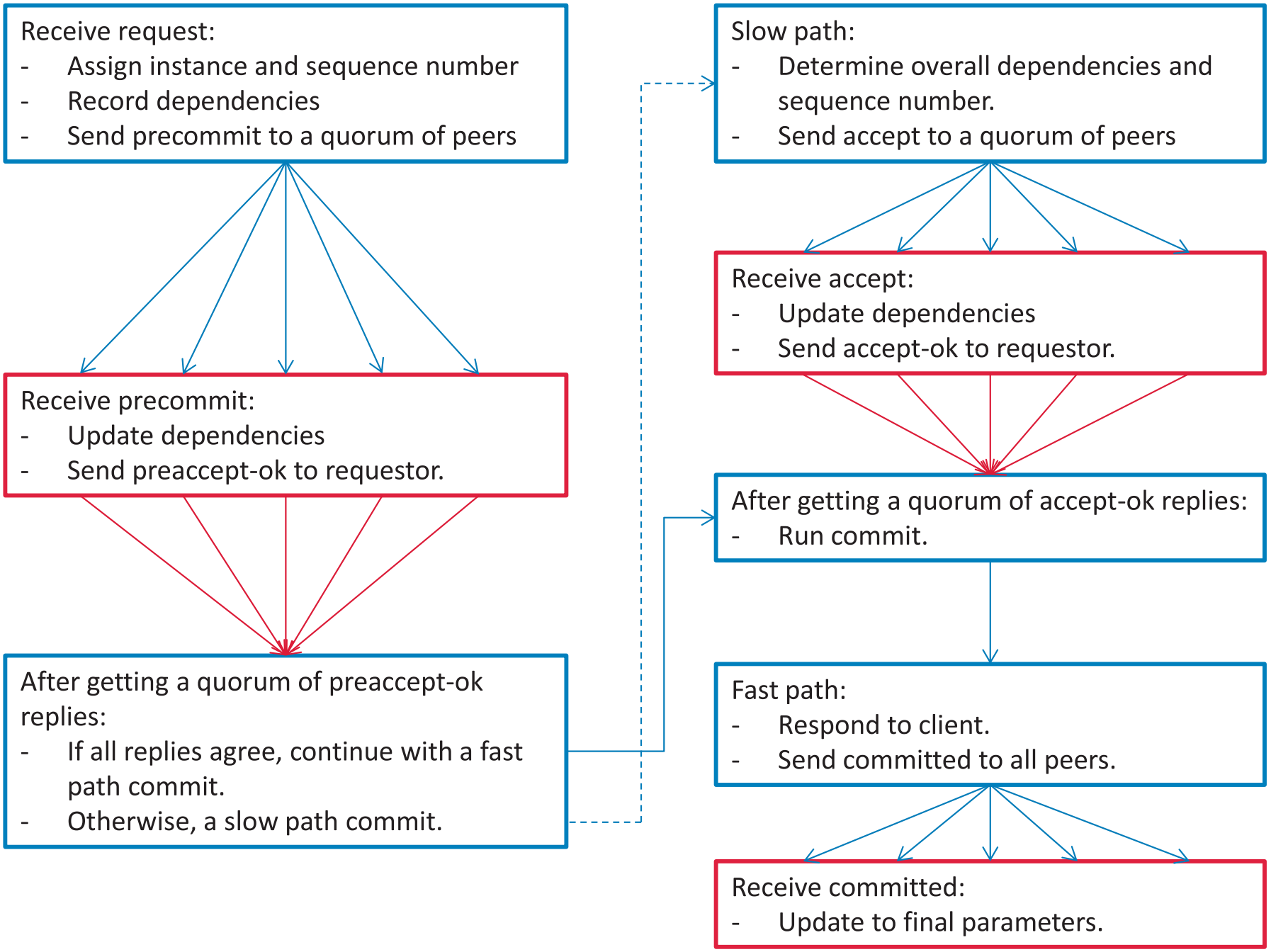
ePaxos

EPaxos takes the coordinator-free approach, where clients send commands directly to any replica, which becomes the leader for that command.

Goals

1. Optimal commit latency, even in the case of failures.
2. Uniform load balancing to maximize throughput.
3. Graceful performance degradation when replicas are slow or crash.

Commit Protocol



Existing Work

EPaxos represents recently published work

There is More Consensus in Egalitarian Parliaments
Moraru, Iulian and Andersen, David G. and Kaminsky, Michael
In Proc. 24th ACM Symposium on Operating Systems Principles (SOSP), Nov 2013

Preliminary findings have been promising.

Work to Date

I am writing an implementation of ePaxos in Go. My work so far has been centered around the commit protocol.

I'm currently working on a simplified algorithm. Optimizations exist which, reduce the quorum sizes.

One goal of my implementation is to keep the implementation relatively straightforward, following closely from the specification, while being robust enough to serve as a starting point for real-world usage.

Currently, the state of things just short of being able to look performance in a meaningful way.

Challenges

- Data structures! How should the state be structured? The commit protocol relies on being able to determine dependencies quickly.
- Minimizing communication. Commands can be batched to reduce overhead. It's also important to avoid sending unnecessary information.

Next Steps

After wrapping up the core implementation, I've identified a few areas to explore. I haven't decided which to focus on yet.

- Adding checkpointing for replica recovery. This is a necessity for real-world usage.
- Extensive optimization. Relatively straightforward–hunt for performance wherever possible. One direction is code optimization. Another is communication optimization, for example avoiding replicas with generally high latency.
- Evaluating performance under a wide range of conditions. Identifying relative strengths and weaknesses as well as providing a straightforward framework for testing.