Paxos

Aurojit Panda

1 Paxos: Design Protocol so a Leader can learn about previous decisions

1.1 Contextual Note

When many people (including me) talk about Paxos without context what we are referring to is the Synod protocol, which was described by Lamport in 1998 in The Part-Time Parliament. The paper you read takes this primitive and uses it to build a RSM. The presentation here is designed to allow easy comparison to Raft above, but is not how one would traditionally present Paxos.

1.2 Plugging in other ways to appoint leaders

Raft depends on leader election preserving certain properties for safety. However, there are other reasons why a process might be chosen a leader. We will talk about one alternative next class.

What can we expect when using arbitrary procedures to appoint leaders:

- At any point in time there is **at most** one leader.
- We can assume access to a quantity similar to **terms** in Raft that allows us to totally order proposals for the same index.

What have we lost/what can we no longer assume: **anything** about the set of operations replicated at the leader. In particular this means that a **committed** operation **may not** be replicated at the leader.

Since we have no control over leadership, we must rely on the replication strategy for safety, which remember means preserving the invariant that **commmitted operations** remain committed.

1.3 Replication Strategy

Previously the leader just incremented and chose an index for each operation. This was because the leader always knew of all committed operations, and hence knew that its chosen index **cannot** conflict with a previously committed operation. We no longer have this luxury, and hence need to check before chosing an index. Do this at replication time.

For what follows assume we consider a single index (slot in the RvR paper) i, and leader p.

- Phase 1a: The leader sends a request to all other processes asking them about the state of slot i. This is a specialization of the $p1a$ message in the paper, you can think of it as $p1a(p, term, i)$ where p is the leader, and i is the slot.

- Phase 1b: On receiving the $p1a$ message each process checks to see whether (a) the leader term is correct, similar to what Raft is doing; and (b) whether it already has an operation replicated in slot i. If process q already has an operation o in slot i which was added in term t' it sends p a message of the form $p1b(q, index = i, op = o, term = t')$ otherwise (i.e., in the absence of such an operation) it sends p a message of the form $p1b(q, index = i, op = none)$.

- Phase 1b.5: The leader waits for $\frac{n}{2}$ responses to its $p1a$ messages. Using these responses it can now compute a set of operations recorded at different replicas for index i. There are a few possibilities here:

  - The set of operations is empty (i.e. everyone returned $p1b(q, index = i, op = none)$). In this case p can deduce that no operation has been committed for index i and hence it can use i to add a new operation.
  - The set of operations is of size one and looks like $\{(op, term)\}$. p **cannot** deduce whether or not operation op was committed, and for **safety** it needs to assume that op is a committed operation, and must chose to put op in index i
  - The set of operations is of size greater than one, i.e., there are several $\{(op_1, term_1), (op_2, term_2), \ldots\}$ pairs. Again, p **cannot** deduce whether or not an operation was committed. However, recursively it knows that no previous leader would have overwritten a committed operation. As a result it can safely pick the operation (let us say op_2) with the highest term.

- Phase 2a and 2b: In all cases above Phase 2a yields an operation that should be put in index i, and p can now replicate this operation. Use the same process as Raft.