Lecture 7: Raft and Paxos

Aurojit Panda

Contents

1 Replicated State Machines (RSM) .............................................. 1
   1.1 Making State Machines Fault Tolerance .............................. 2

2 Keeping Logs in Sync .......................................................... 2
   2.1 Failure Model .................................................................... 2
   2.2 What do we want .............................................................. 3
   2.3 How to decide on an order of operations ............................... 3

3 Raft: Select only safe leaders .................................................. 4
   3.1 Failure model ..................................................................... 4
   3.2 Replication Strategy .......................................................... 4
   3.3 Election Strategy .............................................................. 4

4 Paxos: Design Protocol so a Leader can learn about previous decisions .......................................................... 5
   4.1 Contextual Note ................................................................. 5
   4.2 Plugging in other ways to appoint leaders ............................ 5
   4.3 Replication Strategy .......................................................... 6
   4.4 How does this relate to Paxos Made Moderately Complex ....... 7

5 View change/Reconfiguration .................................................... 7

1 Replicated State Machines (RSM)

Consider a case where we are given a deterministic program whose semantics are entirely dictated by the order in which operations are invoked.

We have already seen examples of such programs in the class: for instance the linearizable queue we discussed in Lecture 4 meets this requirement if we disallow concurrent events (since any linearized schedule $< \text{linearized schedule}$ the realtime schedule the queues results must match commands
invoked in orders). In general lots of services including key-value stores can be modeled as such deterministic systems either by limiting concurrency or by finding ways to ensure determinism despite concurrency. Examples include most storage systems.

In the literature that we are currently reading such programs are referred to as state machines: the system can be modeled as a (possibly infinite) set of states, and each operation invocation moves the system from one state to another.

1.1 Making State Machines Fault Tolerance

State machines are widely used for storage, and availability is desirable. However, in a theme common to the rest of this class: machines, networks and other things fail. So the question becomes how to work around these failures? There are several answers, the one we are going to focus on today is Replicated State Machines.

The core idea here is simple: we have defined a state machine to be a deterministic program whose semantics are entirely dictated by the order in which operations are invoked. This in turn means that if we have 2-copies of the same state machine, and execute operations in the same order then both copies should exhibit the same semantics, i.e., they should be indistinguishable. This in turn means that either copy can stand in for the other: for example if one fails the other can take over. Voila, fault tolerance.

The question now becomes how to keep logs in sync.

2 Keeping Logs in Sync

We now turn to the problem of keeping logs in sync between these replicas.

2.1 Failure Model

Many of the protocols we are considering today do not work under arbitrary failure models, so in everything that follows we will only consider what the system does when it is within the failure model, e.g., for most of the class we will only consider behavior when fewer than a majority of processes have failed. In reality the protocols are designed to make the system unavailable when the failure model is violated.
2.2 What do we want

In replicating the log we want to:

- Decide on the order in which operations are performed.
- Replicate the order at a sufficient number of replicas.
- Respond with the result of the operation once it has been ordered and the ordering has has been replicated so that the ordering is stable. Refer to such operations as operations that have been committed, and associate both the operation’s log index (i.e., where in the order it occurs) with the operation in what follows/

The first ensures that all replicas will execute operations in the same order, the second ensures that the system can survive failures, and the third makes it so that the system as a whole appears to be linearizable.

Replicating operations is easy: just send messages between processes. The important questions are how to order events, and how to decide that an operation has been committed.

2.3 How to decide on an order of operations

Many possible strategies, for this lecture we are going to rely on a designated leader to order the events.

Important to remember that the leader is just a process and not special beyond being given the task of ordering events. In particular this means that the leader can fail, and this failure can occur at any time including:

- When no uncommitted operations are pending.
- Right after the leader receives an operation.
- When the leader is part way through replicating an operation and its order.
- Immediately after an operation is committed.

**Invariant** Committed operations remain committed despite failures, which means that they may not be lost nor their ordering change.

**Core Challenge**: How to maintain this invariant as leaders fail.

The two protocols take different approaches to this. Surprisingly, we will see that this does not change when operations are committed, that bit is dictated by the failure model.
3 Raft: Select only safe leaders

Remember the problem from above: the leader orders operations (decides on the operation’s index) and then replicates the operation. The leader can fail at any point, including while replicating the operation. We want to make sure that despite failures committed operations remain committed.

One approach to do this is to set up leader election so that only processes where all committed operations are replicated can be leaders. The challenge is how to identify committed operations?

3.1 Failure model

For both Raft and Paxos we are going to start by assuming the that no more than \( f \) processes can fail in a system with \( 2f + 1 \) processes.

3.2 Replication Strategy

- The leader replicates operations by first assigning an index (an order), and sending messages to all processes and waiting for a majority (\( \frac{n}{2} \) implicitly 1 due to the leader being a replica) to respond. Observe, that this is the maximum number of responses that the leader can safely wait for since \( f \) processes could have failed. We say that an operation has committed once the leader hears back from \( \frac{n}{2} \) processes, which means the operation is replicated at \( \frac{n}{2} + 1 \) processes.

Leaders choose indices so that each proposed operation has a unique index.

- Each replication request includes the operation to be replicated, its index, and the leader (or rather the term, but observe that there is at most one leader in any term) which replicated the previous operation. This check ensure that there are no holes in the log. [This is largely an optimization to reduce message sizes, you should try and see why that is the case]

3.3 Election Strategy

Observe that in the description above, only the leader knows whether or not an operation has committed: the leader counts number of acknowledgements to determine whether an operation has committed, but no other process sees the acknowledgement.
Goal: leader election should only elect leaders who have a log of all committed entries.

How?

• To become a leader the process must receive “votes” from \( \frac{n}{2} \) other processes.

• A process \( p \) will grant a vote to another process \( q \) only if \( q \)’s log is at least as up to date as \( p \)’s. Equivalently this means \( q \) must know of all of the operations replicated by \( p \).

Taken together this means that if process \( p \) can become a leader it knows all of the operations replicated at \( \frac{n}{2} + 1 \) processes. Since there are only \( n \) processes total, and a committed operation is replicated at \( \frac{n}{2} + 1 \) processes, the leader must (by pigeonhole principle) contain replicas of all committed operations.

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

4.1 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.

4.2 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 committed operations remain committed.

### 4.3 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 choosing 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 \( p_{1a} \) message in the paper, you can think of it as \( p_{1a}(p, \text{term}, i) \) where \( p \) is the leader, and \( i \) is the slot.

- **Phase 1b:** On receiving the \( p_{1a} \) 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 \( p_{1b}(q, \text{index} = i, \text{op} = o, \text{term} = t') \) otherwise (i.e., in the absence of such an operation) it sends \( p \) a message of the form \( p_{1b}(q, \text{index} = i, \text{op} = \text{none}) \).

- **Phase 1b.5:** The leader waits for \( \frac{n-1}{2} \) responses to its \( p_{1a} \) 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 \( p_{1b}(q, \text{index} = i, \text{op} = \text{none}) \)). 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 \( \{(\text{op}, \text{term})\} \). \( p \) cannot deduce whether or not operation \( \text{op} \) was committed, and for safety it needs to assume that \( \text{op} \) is a committed operation, and must chose to put \( \text{op} \) 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.

4.4 How does this relate to Paxos Made Moderately Complex

The above description might leave you confused about the relation to Paxos Made Complex. Here we briefly discuss the relationship:

- Ballots serve two purposes: first, they act as terms; second, they provide a mechanism for leader election. Since we are fine with any mechanism for leader election, this one seems fine too.

- In the description above we describe running Phase 1 (a and b) for each index. However, we can do this in bulk when a process first becomes a leader. This allows the process to both discover the next index to assign to a new operation and saves on the number of messages. The Scout is responsible for executing Phase 1a and b in bulk in that paper.

5 View change/Reconfiguration

[Will cover if time, see video]