Lecture 7: Raft and Paxos

Aurojit Panda

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 $<$ must be a superset of $<_H$ 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.