Lecture 6: FLP and Partial Synchrony

Aurojit Panda

1 Plan

1.1 Recap Replicated State Machine

1.2 Consensus

1.3 FLP

1.4 Partial synchrony

2 Recap Replicated State Machines

Goal: Replicate state at a server so that system remains available despite some set of failures.

Broad strokes of how we achieved this: assume server is deterministic, maintain a consistent log of commands at all servers, and profit.

Consensus, the topic of today's class is all about how to maintain consistent logs. But rather than talk about the entire log we will focus on a single slot.

3 Consensus

As we shall see for the next several classes, getting multiple processes to agree on a value is a core primitive in building distributed systems Here are some simple examples where consensus is useful:

- Deciding which process holds a lock.
- Deciding on the next operation to handle.
- Deciding on the current configuration.

Consensus is often modeled as each process having an input (either provided as an initial message or in process state), and being able to produce a

special message (we will call it commit) that indicates that the process has decided on a value.

Consensus ideally meets three requirements:

3.0.1 Agreement

If one process decides on value v, then all processes will decide on value v. In particular, this means that we cannot have a case where process p decides v, and q decides v' such that $v \neq v'$.

3.0.2 Validity

The value decided must be input to one of the processhttps://www.ups.com/track?loc=null&tracknum= This is important to rule out trivial algorithms such as ones where all processes just choose 0. This also leads to strong unanimity in the Dwork paper.

3.0.3 Termination

Ideally, we want consensus protocols that terminate: i.e., there is a finite amount of time after which all correct processes have decided on a value.

4 FLP

4.1 History

From `http://www.podc.org/influential/2001-influential-paper/`: Lots of efforts in the early 80s to come up with a consensus protocol. Danny Dolev and Nancy Lynch were looking at results in the synchronous setting, while others including Butler Lampson at Xerox Park (who talked to Fischer) were looking at asynchronous protocols. Fischer, Lynch and Patterson who all happen to be on the east coast decide to simultaneously look for a consensus protocol and prove that no such protocol exists. The former failed, the results of the later effort are in front of you.

4.2 What does it say?

If you consider an asynchronous setting, no protocol can solve consensus in a setting where communication is reliable, but one process can fail by crashing.

4.2.1 Why this model?

Generality: The failure model chosen is in a strict sense "simpler" than others: a very small number of processes can fail, behavior under failure is controlled, etc. Showing that consensus is impossible under this model covers others.

4.2.2 Why the proof works?

What the proof is asking is whether there exists a consensus protocol that can work with a failed process. This in turn means that we want a consensus protocol that **terminates** when a process fails. But in an asynchronous network a process p might not be heard from either because it has failed or because of how the network schedules message deliveries. This in turn means that the protocol cannot distinguish between these two conditions, and a message delayed sufficiently would require that the consensus protocol treat a process as failed. However, due to validity if that message is then delivered the consensus protocol needs to make a different decision. Repeatedly applying this construction allows the authors to show that there are sequences of events which would cause a protocol to not terminate.

4.3 Proof and Preliminaries

4.3.1 Binary Consensus

Going to focus on the task of choosing 0 vs 1.

4.3.2 Process

- Input: Each process has some input, can consider this to either be a message sent from the environment or a value embedded in its configuration.
- Decision/Output: Each process can commit to a single value: done by either producing a commit message or writing to a piece of state.

4.3.3 Configuration

State at all processes + pending messages: The network is modelled as a buffer, sending a message adds to the buffer, receiving a message removes from the buffer. Events: the act of receiving a message moves the system from one configuration to another.

4.3.4 Enabled Events

In any configuration the set of messages waiting in the network are enabled events: the schedule can pick one of them to deliver.

4.3.5 Diamond Lemma [Lemma 1]

See paper + presentation.

4.3.6 0-valent, 1-valent and bivalent configurations

A 0-valent (1-valent) configuration is one where regardless of the order in which enabled events are applied all processes will decide on 0. A bivalent configuration is one where either choice is possible.

4.3.7 Proof sketch:

1. All 1-failure tolerant protocols have a bivalent initial configuration [See paper and presentation] Show by a diagonalization like argument: by validity we know that there is a 0-valent configuration and a 1-valent configuration. Must exist a configuration when we flip from 0-valence to 1-valence. But protocol is 1-fault tolerant. What happens if the process that flipped is silent during an execution: by indistinguishability we can find a bivalent initial configuration.

2. Given a bivalent configuration C and an event e can find C' such that:

   - $C \Longrightarrow C'$
   - $e(C')$ is bivalent

   How: [See paper + presentation]

3. Combine both to keep extending schedule indefinitely.

5 Partial Synchrony

5.1 What is Partial Synchrony

The basic idea is that messages cannot be arbitrarily delayed forever. Question becomes how to model this. A few ways:

- Unknown but finite bound on message delay.

- A global stabilization time (GST) after which message bounds are delayed
- Unknown but finite bound on process delays.

5.2 Practical implication

The core idea here is that networks and machines don't change very frequently. While we cannot assume that after GST no changes happen, since the protocol itself terminates just need to make sure no changes happen for long enough.

5.3 Round Based Algorithm and Quorums

[See presentation]