Replicated State Machines

Aurojit Panda

1 Plan

- Failure Models
- What and why RSMs?
- Desirable properties?
- Accomplishing what is deirable
- Reconfiguration/View Change

2 Failure Model

So far in this class we have used the term failure quite loosely. We have generally looked at scenarios where any process can fail, and not really thought about what happens when a process fails. This is going to be problematic going forward, because as we will see many of the protocols we discuss don't work under arbitrary failures. You have already seen some of this in the work you have done so far. So to make this easier to discuss let us briefly talk about how we characterize failures and describe what is possible. This is commonly referred to as a failure model.

Broadly a failure model needs to address two questions:

- How many processes can fail?
- How do failed processes behave.

The first of these is easy, so let us talk about the types of failures one can expect. We will only encounter a subset of these types this semester:

- Fail-stop This is what we have largely assumed thus far. A failed process stops sending or receiving messages and processing. Moreover failed processes do not recover and rejoin without an explicit step for this. Can you think of why this last bit might be important?

- Fail-recover Same as above except that processes can eventually rejoin.
- Byzantine Failed processes can exhibit arbitrary behavior. They can send incorrect messages, at the wrong time. Used to mean processes could have bugs, increasingly means processes could be a security risk.
- Things we ignored in this list
  - Errors which cause message processing to be delayed. [Ignored because we are largely sticking to the asynchronous model.]
  - Failure models which require the use of stable storage [Reduce the burden on recovery logic]
  - Many many more things.

3 What & why state machines

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.

3.1 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 order the requests and how to keep logs synchronized.

4 Desirable properties

- All non-faulty processes agree on the contents and order of the log.
- The log contains values/commands proposed by clients.
- Commands from a single client are in the order they were issued.
- How to order commands from different clients?
  - Linearizability?
  - Causal consistency?
  - Something else?

5 How to meet these properties

5.1 Start with keeping logs synchronized across replicas

5.1.1 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.

Let us do this more formally:

Validity: Any command in the log must have been proposed by a correct client.

Agreement: For any slot in the log that has already been decided, all clients agree on the contents of the slot.

5.1.2 The challenge with meeting this goal

Next week: assuming fail-stop behavior and asynchronous networks there is no protocol that is guaranteed to finish in bounded time that also achieves these two requirements.

So what option do we have? Relax the asynchronous network assumption to one around partially synchronous networks. You see this in more detail next week, but this is similar to the assumption made by Lamport: there is an upper bound on the communication latency between any two correct processes. While we might not be given this upper bound we can potentially compute it (how?). Why might this help?

5.2 How to order logs for a single client

Remember we want to ensure client commands are in order. How do we ensure that this is the case?

5.3 How to order logs from different clients

One thing to observe here is that to do this we either need to ensure that clients are involved or make stronger assumptions about the system. Why?

So now the question turns to how do we use clients to achieve any of the versions we discussed above?

- Linearizability?
- Causal consistency?
- Others?

6 View Change or Reconfiguration

So far we have not considered the question of how to add new processes to the distributed system. But most of the protocols we will be studying assume fail-stop failures or byzantine failures. In neither case can we assume that failed processes will eventually recover. Easy to see this is a problem (why?).

So need to be able to add new processes. This requires two things

1. Makes sure new process has the latest state.

2. Make sure that all other processes know about new process.

But need to accomplish this while running on an asynchronous network, etc. Ideally without needing to make any additional assumptions. One useful observation here is that this problem itself is quite similar to the agreement problem we had to solve above to keep logs synchronized across replicas (why?).