Today: Consistency and Replication 4/22/09

Sam Madden

Exam: Mean 55. If your grade was low -- don't sweat it. What matters is how you performed relative to the mean.

Last few lectures: seen how we can use logging and locking to ensure atomic operation, even in the face of failures.

But -- we might want to do better than just recover from a crash. We might want to be able to mask the failure of a system -- make the system more available.

Previously seen that replication is a powerful tool that help accomplish that. Today we will investigate replication in a bit more detail, and look some problems that arise with maintaining replicated systems.

Replication used in both the local area -- e.g., two replicas on the same rack of a machine room, as well as the wide area -- two replicas in different buildings, states, etc.

Wide area replication important for **disaster recovery.** "Half width of a hurricane."

Let's look at how we might apply replication to a system:

(Send requests to both; if one is down, just send to the other one.)

What could go wrong:

(Network partition. Some other client sends requests to other machine. Now they are out of sync.)

Solutions:

- Eventual consistency -- have some way to sync up after the fact (exposed inconsistency -- this could have never happened if we didn't have a replicated system.)

- Strict consistency -- "single-copy consistency" -- ensure that an external user can't tell the difference between one machine and several machines.

strict consistency and availability fundamentally at odds, esp. in the face of network partitions.

In above example, when the eventual consistency approach allowed either client to receive answers.

In strict consistency approach, best we can do is to designate one replica as the "master" (or authority) and allow clients talking to it to proceed. Other clients -- talking to other replica -- will not get service.

Eventual consistency

Easier to provide (usually) than strict consistency. Typical way that this works is to associate a time stamp (or version number) with each data item.

Have replicas periodically compare time stamps, use freshest. This is what Unison does.

DNS is an example of eventual consistency -- suppose I change:

db.csail.mit.edu from IP X to IP Y

Effect won't be visible immediately; instead, will propagate when the timeout on the DNS record expires. Remember -- DNS servers cache lookups for a fixed amount of time.

(Show slide)

This means that the machine with IP X may receive requests for db.csail.mit.edu for some period of time.

Inconsistency is considered acceptable here. Providing strict consistency here would be very hard: I'd have to contact every cache in the world and ensure they were

updated, or would have to have no caching and have every request come to csail.mit.edu on every lookup. Neither is OK.

One challenge with eventual consistency is what to do if there are concurrent updates to the same data. Not a problem in DNS because there is one master for each subdomain (e.g., csail.mit.edu) that totally orders updates. But in a system like Unison, we can have multiple updates to the same data item, which is problematic and requires some kind of manual conflict resolution.

Will see another example of eventual consistency next recitation, and see some of the additional pitfalls that concurrent updates to the same item can lead to.

Strict Consistency -- Replicated State Machines

Basic idea is to create *n* copies of system. Same starting state. Same inputs, in the same order. Deterministic operations. Ensures the same final state.

If one replica crashes, others have redundant copies of state and can cover for it. When that replica recovers, it can copy its state from the others.

But what about network partitions -- replica may have missed a request, and may not even know it!

If requests come from different clients on different sides of the partition, then replicas can see different requests. Diagram:

One option: create an ordering service that totally orders all requests. When a network partition happens, one replica may miss some requests, but it can detect that

and request those operations from other replicas before proceeding.
Diagram:

(Single point of failure -- will return to that in a minute.)

Degenerate version: Single Master Replication

Diagram:

All updates go to a single master, which is responsible for propagating updates to the replicas.

Seems like it might fix the problem because now there is only one master.

If master fails, switch over to one of the replicas. What's the problem here?

May have been writes which propagated to some of the replicas, but not others. How do replicas know what the current state is?

(They don't -- might think they could talk to each other and figure out the most update any of them had seen. But there might be something the crashed primary saw that they hadn't.)

Does two phase commit solve this problem?

(No. Suppose following

```
M                       R

WA
                        WA
prepare
                        prepare
                ok
commit
-----crash!-------
```

)

One approach is to simply tolerate some inconsistency -- e.g., accept that replicas may not know about the outcome of some transaction.

In practice, this is what is done for wide area replication -- e.g., in a database system.

How does the client know which server to talk to?

Ordering Service

How to build a reliable ordering service?

1 node -- low availability.

Use a replicated state machine. Diagram:

Still have problem with partitions. Typically solution is to accept answer
if a majority of nodes agree on it.
Challenge: network partitions, message re-orderings, dropped messages.

This problem is called agreement:

Agreement protocol (sketch)

Leader:	Worker
next(op)	
	agree(op)
accept(op)	
	accept_ok(op)
commit	

Phase 1: Leader node proposes next op to all others (propose)

Phase 2 : If majority agrees, it tells all others that this is the next op (accept)

Phase 3: Once a majority acknowledge accept,
Send out next op!

Reason for this protocol is we want to allow other nodes to continue if leader goes away.

Other nodes:

If after a random timeout can't contact leader, propose themselves as new leader. If majority agree, they are new leader.

(Nodes only respond to prepare / accept requests from their current leader.)

Once a new leader has been chosen, it checks to see if any node has accepted anything. If not, it restarts protocol.

If some nodes have agreed but no node has seen an accept, doesn't matter, because that transaction must not have committed yet, so leader can safely abort it.

Once a leader has received an accept_ok from from a majority, it knows that even if there is a network partition, where it is in the minority partition, majority partition will have at least one node that has committed.

Otherwise, majority partition might just have one node that is prepared, new node might assume that the transaction didn't commit and propose something else.

Nodes outside of the majority are inconsistent and need to recover, even if they have already sent an accept_ok message.

Commit point is when n/2+1th worker writes accept(ok).

Example 1 -- network partition

Example 2 -- failure of a leader

These agreement protocols are very tricky to get right, which is why I'm only sketching the idea here. Take 6.828 to learn more.

```
Protocol sketch:
        N :  number of replicas (odd, e.g., 3 or 5)
        votes = 0
        voters = {}
        R = 0 //must persist across reboots
        opQ = new Queue()  //must persist across reboots

        do(op):
                opQ.append(op)
                broadcast("next op is opQ[R], in round R");

        periodically:
                if (R < opQ.len)
                        broadcast("next op is opQ[R], in round R ");

        next(op,from, round):
                if (R < round):
                        wait to see if any more votes for R arrive
                        if (R < round):
                                recover
                                return

                if (op == opQ[R] && from not in voters):
                        votes++;
                voters = voters U from


                if (votes > N/2):
                        notifyWorkers (opQ[R])
                        R = R + 1
                        voters = {}
                        votes = 0

                if (N/2 - votes > N - len(voters)
                        deadlock
```