6.033 Lecture 19: Replicated State Machines
4/16/2014

Where are we?

Learned about transactions
- Provide ability to recover from crashes via logging
- Isolate concurrent transactions via 2PL

Last time, as how to extend this to multiple nodes with two-phase commit.

[Show 2PC recap slide]

[demo?]

Problem: failures of nodes can make system unavailable.

Might want a long running service to be able to continue to work in the presence of a node failure.

How to continue despite failures?
General plan: multiple servers, replication.
Already seen some cases: DNS, RAID, ..

This week: how to handle harder cases.
E.g., replicated storage (e.g., mail server, file server, etc.)
E.g., replicated master for 2PC
..

Often shows up in data center infrastructure at Google, Facebook, Microsoft, etc.

Ideal goal for replicated system: **single-copy consistency**.

Property of the externally-visible behavior of a replicated system.

Operations appear to execute as if there's only a single copy of the data.
Internally, there may be failures or disagreement, which we have to mask.
[show slide]

Similar to how we defined serializability goal ("as if executed serially").

Many systems give up on this ideal semantics, but some need it
E.g., Expensive to achieve in wide-area networks
DNS and PNUTS settle for weaker semantics

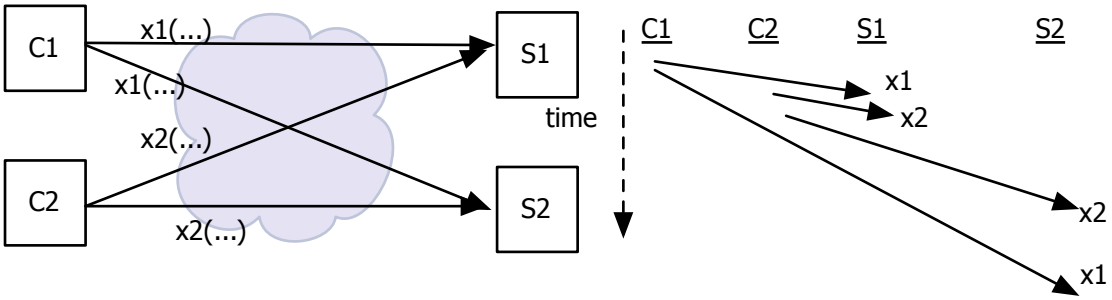
Possible for different clients to observe different states of the system

Replicating a server (e.g., a bank account server).
[diagram: two clients, two servers]

Strawman: clients send requests to both servers.

Tolerating faults: if one server is down, clients send to the other.
Limit ourselves to two servers for now

Problem: replicas can become inconsistent.
Issue: clients' requests to different servers can arrive in different order.



x1 is delayed from c1—>s2, arrives in different order on two replicas

How do we ensure the servers remain consistent?

Solution: **Replicated state machines [show slide]**

- Start with the same initial state on each server.
- Provide each replica with the same input operations, in the same order.
- Ensure all operations are deterministic.
  E.g., no randomness, no reading of current time, etc.

These rules ensure each server will end up in the same final state.

Assumption: Independent failure of replicas
But:
- Replicated bugs
- Shared network
...
Easy to build non-fault-tolerant fault tolerant systems
E.g., Amazon EC2 failure, Gmail failure, etc.
http://research.microsoft.com/pubs/191008/FailureRecoveryBeEvil.pdf

RSM in practice — Plan:

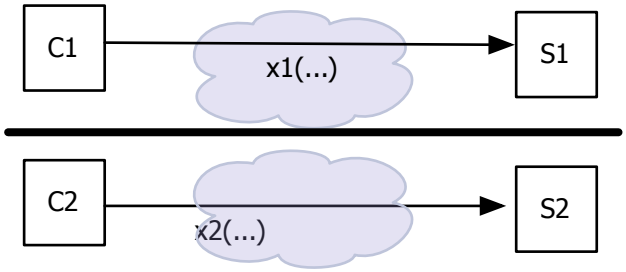
client —> primary DB —> backup DB
log writes

- primary ensures that it sends all updates to backup before ack'ing client
- primary chooses an ordering for all operations, so primary and backup agree
- primary decides all non-deterministic values (e.g., random(), time(), send to backup)

What if primary fails?

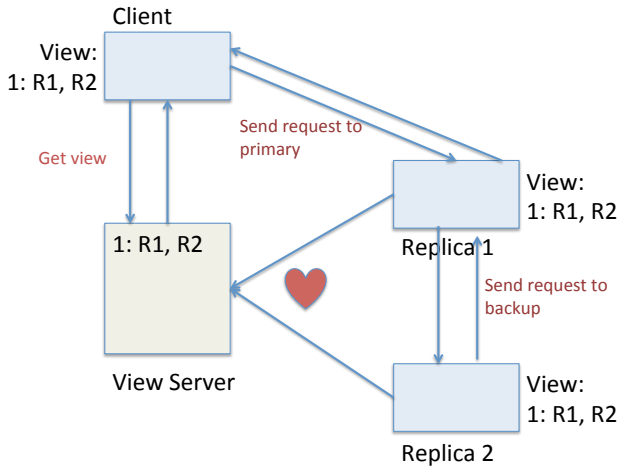
client could know about both primary and backup, and decide which to use

problem: multiple clients / split brain, e.g., C1 might think S1 is primary, but C2 might think S2 is primary, if there are network partitions



One solution to this is to have a human decide when to make switch from primary to backup. Common in small web services, etc with database server.

Automated solution
Introduce "view server", that maintains current primary — provides a *consensus mechansims*



Clients contact view server, so they all agree on primary

Workers also contact view server to learn which is primary

For now, won't worry about how to make view server itself fault tolerant

View server can recruit a new, idle server to act as backup, by asking it to copy state from current primary

If network is partitioned, some server may have stale view, but view server is a single place where decision about current view is held.

view server
  maintains a sequence of "views"

```
    view #, primary, backup
    1: S1 S2
    2: S2 --
    3: S2 S3  <— recruit new server S3
```

View server monitors server liveness
    each server periodically sends a Ping RPC
        "dead" if missed N Pings in a row
        "live" after single Ping

  Can be more than two servers Pinging view server
    if more than two, "idle" servers

  if primary is dead
    new view with previous backup as primary

  if backup is dead, or no backup
    new view with previously idle server as backup

How to ensure only one server acts as primary?
(Even though more than one may *think* it is primary! — due to network partitions)

  "acts as" == executes and responds to client requests

  the basic idea:
    1: S1 S2
    2: S2 --
    S1 still thinks it is primary — and clients may talk to it
    S1 must forward ops to S2

    S2 thinks S2 is primary
    so S2 can reject S1's forwarded ops

Key Rule: primary not allowed to respond until it gets ack from backup

The rules: [show slide]

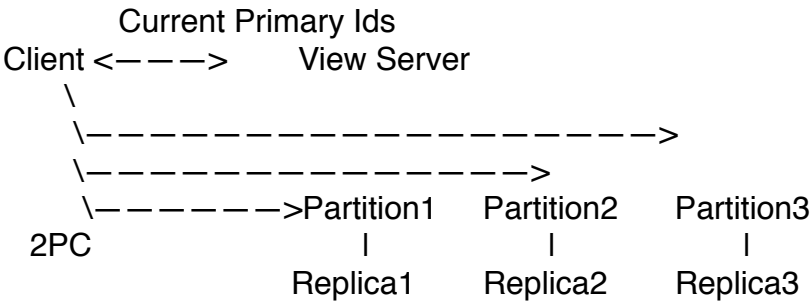
1. Primary must wait for backup to accept each request
2. Non-backup must reject forwarded requests
3. Primary in view i must have been primary or backup in view i-1
4. Non-primary must reject direct client requests

example: [show animations]

1: S1, S2
   viewserver stops hearing Pings from S1
2: S2, --
it may be a while before S2 hears about view #2
before S2 hears about view #2
  S1 can process ops from clients, S2 will accept forwarded requests
  S2 will reject ops from clients who have heard about view #2
after S2 hears
  if S1 receives client request, it will forward, S2 will reject
    so S1 can no longer act as primary
  S1 will send error to client, client will ask vs for new
    view, client will re-send to S2
the true moment of switch-over occurs when S2 hears about view #2

Before switch over, all clients either get no response, or see view of S1
After switch over, all clients get view of S2

Ok, so how does this stuff all fit together:



Provides: distributed transactions across many nodes

Tolerance to faults of any one node

Next time — how to deal with faults in view server

[Show recap slide]