Replication

- Replication is a common technique in distributed systems for addressing:

  - Performance
    - Workload balancing via multiple servers providing the same data; caching.
  - Availability
    - Multiple servers reduces probability of data being completely unavailable.
  - Fault tolerance
    - Dealing with failures
Replication

- Any system based on replication seeks transparency from the point of view of the client.

- Clients should not know that data is replicated.

- What makes this difficult?
  - For static data:
    - Nothing. This is easy.
  - For dynamic data:
    - Maintaining consistency in the replicated set.
    - This is especially difficult if updates from participants in the replicated set synchronize infrequently.
Replication examples

- We all (should) have had direct experience with replication.

- Where? Version control systems!
  - RCS, CVS, SVN, Git, Mercurial, etc…

- Recent developments in version control systems have definitely taken on a more sophisticated distributed flavor.
  - More about this later today.
Replication basics

- We have a set of objects we want to manage. The objects are *logical* objects.
- We maintain a set of instances of these objects stored on servers. These are the *physical* objects.
  - Physical copies are called *replicas*.

- A replication system allows clients to work with the logical objects by hiding the translation of the logical objects to the physical instances that they correspond to.
  - On the client side, we have a *front-end* that the client interacts with.
  - The server that manages the physical objects are called *replica managers*. The front-end talks to these on behalf of the client.
Failure model

- In this discussion, we only consider crash failures and network partitioning.
  - Crashes: A replica manager goes away.
  - Network partitioning: Components of the system stop being connected for some period of time.

- Some VC systems also address the arbitrary/byzantine failure case.

- An example of network partitioning in the version control case is your laptop.
  - You check out a version of the source repository onto your laptop and get onto a plane.
  - You interact with your local copy without being connected to the repository server.
Client/server interactions

- Clients interact with a front-end.
  - VC example: You are the client, SVN command line tool is the front end.

- The front-end interacts with replica managers via:
  - Direct interaction with one replica manager, which in turn interacts with other managers.
  - Multicast to a set of replica managers.

- Choice of interaction method is application dependent.
Interactions

- Interactions with the replica manager come in two flavors:

  - Read operations
    - Requesting an object.

  - Update operations
    - Providing a new version of an object to the replica manager.
    - Requires the set of replica managers to coordinate ordering when multiple updates occur simultaneously.
Coordination schemes

- How to coordinate multiple updates?

- Replica managers can choose from a few different ordering disciplines:
  - **FIFO**: Order updates in order that they appear at front-ends. If Front-end requests A before B, any replica manager that handles B must handle A before it.
  - **Causal**: Replica managers obey happened-before ordering of requests from front-ends.
  - **Total**: If a replica manager handles A before B, then all replica managers that handles B will handle A first.
Execution and agreement

- Replica managers will execute requests from front-ends.
- Tentative execution is possible if it is necessary to later undo the requests.
  - If replica managers obey a transactional discipline, this is necessary.
- Why?
  - **Remember:** Two phase commit protocol in distributed transaction requires tentative commits during first voting phase to store potential commit to nonvolatile store before second phase where, upon unanimous vote, actual commit occurs.

- Agreement amongst replica managers is consensus amongst the set on the effect of a request.
Group communication

- Replication is an inherently group-based operation.
  - Group is both client(s) and replica managers.

- A key concern is membership in the group.
  - Membership can change (participants come and go)
  - Failures may occur (participants may vanish)
  - Notification (participants may wish to be notified upon changes to replicated data)
    - Think subscription model discussed a couple weeks ago.
  - Address expansion (map logical group name to actual participant addresses)
IP multicast

- IP multicast (IPM) doesn’t quite cut it for this membership model.
  - IPM allows dynamic membership
  - IPM performs address expansion
  - IPM does not provide notification when members come and go
  - IPM does not help with delivery coordination when membership changes during messaging operations.

- So, IPM may be a protocol to build upon for replication, but additional higher-level work may need to be built to provide all of these functions.
Exclusion

- Group membership services may choose to exclude members when they become suspected of failure.
- This may require a member to rejoin as a new member if it becomes excluded.
- This can be very disruptive. Requires failure detection mechanisms to be careful to avoid false exclusions when possible.
  - Failure detector must take care to avoid false exclusions.
- Partitioning of networks leads to tricky situations.
  - Sometimes it is OK to run with a partial set of members.
  - Depends on application and nature of replicated data.
View-synchronous group comm.

- A view is a set of processes at a given point in time that are associated with a group that can be reached from processes within that group.
  - $V_0(g) = \{p\}$; $V_1(g) = \{p, p'\}$; $V_2(g) = \{p\}$

- The point of views is to:
  - Formalize the dynamic membership of groups.
  - Be able to say things about legality of certain interactions under changing membership conditions.
**View-synchronous group comm.**

- **Agreement:** Correct processes deliver the same sequence of views and the same set of messages in any given view.
  - In other words, if a process delivers $m$ in view $v(g)$, then all other correct processes that deliver $m$ do so in view $v(g)$.

- **Integrity:** If $p$ delivers $m$, then it will not deliver $m$ again. $P$ must be in the group the message $m$ was targeted to, and the sender of $m$ must also be in the view in which $p$ delivers $m$.

- **Validity:** If the system fails to deliver a message to a process, the members of the group that succeeded to deliver the message will deliver a new view with the failed process excluded immediately after the view in which the message was delivered.
Delivering a view

a (allowed).

p crashes

view (p, q, r)

view (q, r)

b (allowed).

p crashes

view (p, q, r)

view (q, r)

c (disallowed).

p crashes

view (p, q, r)

view (q, r)

d (disallowed).

p crashes

view (p, q, r)

view (q, r)

These diagrams should look familiar from our discussion of cuts back when we talked about global states.
Correctness criteria

- Say we have a set of clients executing operations on a single server.
- The single server will serialize the client operations.
- We can establish a canonical execution of the set of operations against a single ‘virtual’ single image of the shared objects.
- Clients see a view of the shared objects that is consistent with this virtual view.

- What we wish to do is establish criteria for allowed interleavings that preserve correctness properties assumed on the clients.
  - Think of the bank account examples earlier in the term.
Linearizability and sequential consistency

- Linearizability and SC are *properties* of a system.
- In both cases, we start with the constraint that:
  - An interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
  - This means that the interleaved sequence yields an end result equivalent to only one process executing the sequence of operations.
- Linearizability:
  - Order of operations in interleaving is consistent with *real times at which the operations occurred in actual execution*.
- Sequential consistency:
  - Order of operations in interleaving is consistent with the *program order in which each individual client executed them*. 
SC vs Linearizability

- Linearizability is stricter than sequential consistency. Why?
  - Linearizability speaks of consistency with respect to the real time that operations occurred in actual execution.
  - This is hard to implement practically due to issues we already saw a few weeks ago related to tightly synchronizing clocks to come to a globally consistent view of time.

- SC relaxes the constraint a bit, and instead simply says that the operations must occur in interleavings that preserve program order w.r.t. each client.

- Practical constraints make SC more common in practice. In some cases, even SC isn’t preserved if the system can tolerate the impact it would have on the data.
Passive vs. active replication

Passive:
- Front-ends interact with a primary replica manager.
- Primary RM interacts with backup RMs.
- Failure of primary RM results in promotion of a backup RM to take on its role.

Active:
- Replica managers form a group.
- Front-ends communicate with group of RMs.
Version control systems

- VC systems are an everyday example of a distributed replication system that we all are (or should be) familiar with.

- In the last 10 years or so, there has been an interesting proliferation in VC systems with a significant focus on:
  - Large, distributed development efforts.
  - Version control management in the presence of temporary disconnection from the main repository servers.

- Examples: Mercurial (Hg), Git, Darcs, Arch