Distributed systems

Week 6, Lecture 1

Logistics

- Remember: Prog. Assign. #2 delayed.

- Note on reading #3.
  - Don't get bogged down in the programming languages details. **Focus on content directly talking about transactions**. Ignore section 5. Skim the rest.
  - Goal of this reading: Expose you to a real context where transactions are having a real impact outside databases.
    - Transactions can be tedious. It's easy to blow them off as a topic of relevance to a small community, like the databases world.
    - STM is interesting because it is an example of transactions in a context that may impact all of us in the not-so-distant-future.
    - Don't get scared by the Haskell.

Before we dive into Ch. 15

- Two discussion points:

- 1. Programming assignment #2 Q and A.
- 2. Revisiting concurrency control

PA2 notes

- So far, some of you have turned in the code for the Linda example.
- The trickiest routine I've seen so far is the blocking in() method.
  - In: Attempt to read a tuple of a specific type, block until one appears of that type.
- There are three approaches to achieve this.
  - Busy waiting
  - Wait()/Notify()
  - Typed pending lists

Blocking in()

- Busy waiting
  - Spin in an infinite loop where the loop body is a synchronized block that scans the tuple space for a matching tuple.
    - This is bad w.r.t. performance. Serializes everything.
- Wait()/Notify()
  - A thread can voluntarily give up it's lock in a synchronized block by calling wait(). Another thread can cause it to come back to life by calling notify().
  - This is better, as the thread looking for a tuple will go to sleep until a new tuple comes along. No guarantee though that the new tuple is what it was looking for.
- Typed pending lists
  - Similar to wait()/notify(), but callers go into a blocked waiting state where they are awoken if and only if a tuple that matches the type they are looking for arrives.
- I did not specify which of these I wanted you to implement, so I will not penalize you if you choose an inefficient method. I would prefer you to attempt to try for the method that achieves the best performance though.

Wait and notify

- A thread in a synchronized block will call wait(), allowing it to go to sleep and other threads to acquire the lock it had.
- Another thread may call notify() to wake up threads that went to sleep on a wait() call.
- Threads that were wait()-ing are then able to attempt to reacquire the lock.
  - Other threads may acquire it first. There is no guarantee that the wait()ing thread will get the lock after a notify(). Some other thread may slip in. The awakened thread will just block in this case.
- Wait() and notify() are methods in Thread that are OK to use, but dangerous to overload.
- There is a version of wait() that allows threads to timeout on a wait call. This is not likely to be of use here.
  - Life can get complicated using this.

Annoyances

- RMI requires you to explicitly set up security policies to allow connections between clients and servers.
- For example, a server might say:

```
grant codeBase "file:/home/you/src/" {
    permission java.security.AllPermission;
};
```

- My only advice is to follow the guidance in the Java RMI tutorial. You can also turn up tips for making this work on different systems (like Windows) by Googling with the exception name. I searched for this and got good results:
  - java.security.AccessControlException RMI

Reviewing concurrency control

- Concurrency control is required when more than one unit of execution (thread, process, remote process, etc…) may access data in a manner in which order of accesses makes a difference in maintaining some correctness property of the system.

- An underlying assumption here is that the sequence of operations that we consider is large, far more than a single instruction.
  - In other words, coarser granularity than hardware can support atomicity for.

- Two methods discussed:
  - Critical sections and mutual exclusion.
  - Transactions.

Mutual exclusion

- The simplest case is enforcing mutual exclusion in a critical section. In other words, one and only one process in the CS at any given time.
- Critical sections don't only correspond to blocks of code. They correspond to the variables that the blocks are working on.
  - So, if "balance" is a sensitive data element to protect, all manipulations of it and computations based on it may need to fall within critical sections with mutually excusive access.
- Acquire lock at front of CS, release at end.
- **Benefit**: Easy to implement.
- **Problem**: Prone to deadlock, serialization of concurrent execution.
  - Serialization just means performance will be suboptimal.
  - Typically we want to avoid this, as serialization kills scalability.

Transactions

- Transactions are less aggressive.
- Surround the block of sensitive code with a transaction stating it should execute atomically.
- Acquire locks only when necessary during atomic section.
- Release at the end.
- Write data off to the side, write to main store only at end when you can show that doing so will not result in consistency problems with other threads.

- **Benefits**: More concurrency, less serialization.
  - Higher degree of concurrency = more scalable.
- **Problems**: Still prone to deadlock in some cases; more complex to implement.

Onwards

- Into Ch. 15.
- Replication
- This is a fairly digestible topic.
- I will deviate a little from the book here to give you examples in systems that you likely have really used.
  - I'll talk a little about Coda, but not the other two. Those you can get through the reading.

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.
  - V0(g) = {p};V1(g) = {p,p'};V2(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.

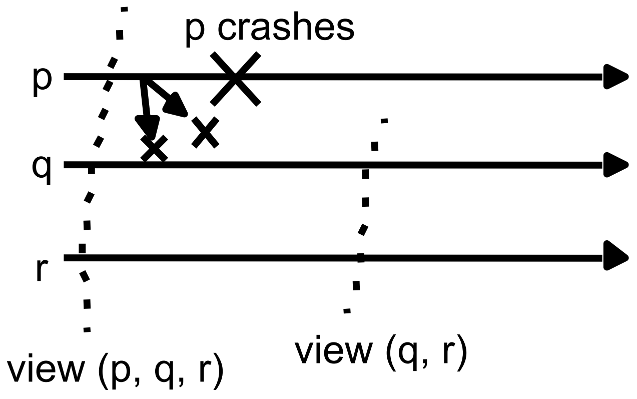
View-synchronous group comm.

- **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.

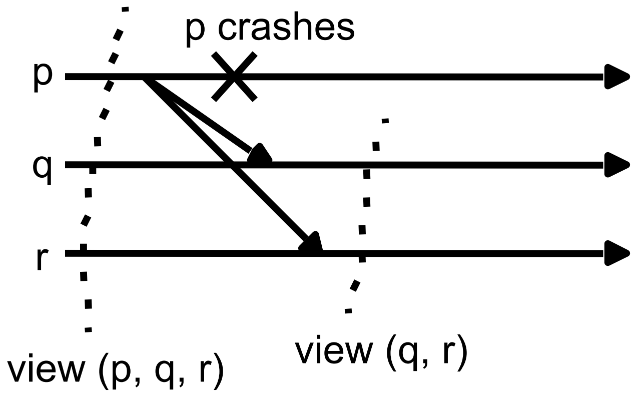
Delivering a view

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

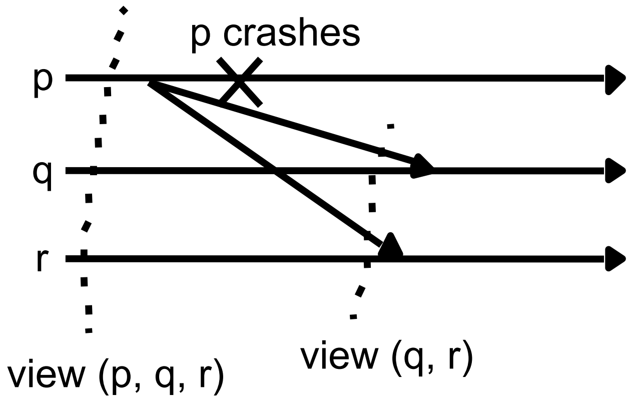
a (allowed).



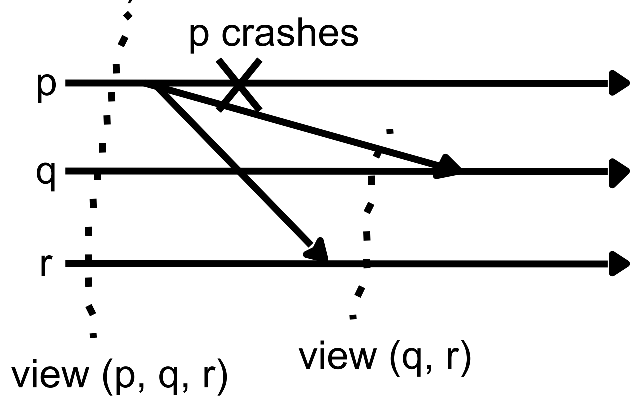
b (allowed).



c (disallowed).



d (disallowed).



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.

Linearizablity and sequential consistency

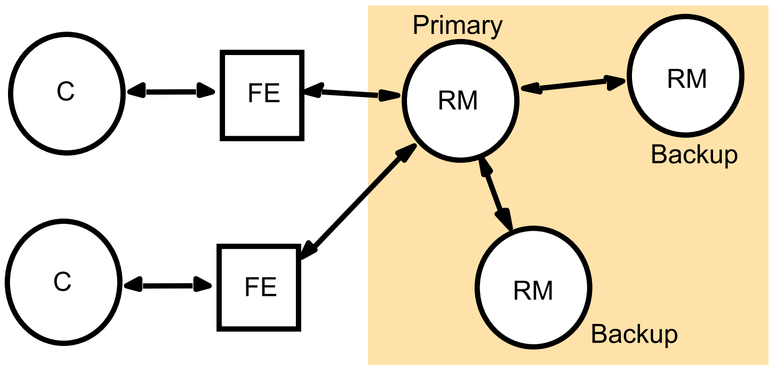
- Linearizability and SC are *properties* of a system.
- In both cases, we start with the constraint that:
  - An interleaved sequence of operations meets the specification of a (single) correct copy of the objects.
  - This means that the interleaved sequence yields an end result equivalent to only 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

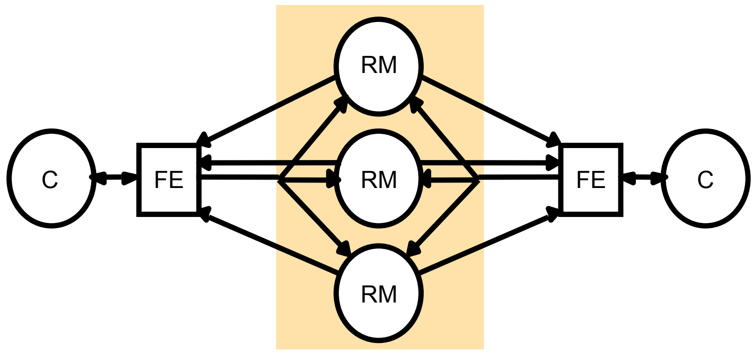
- Linearizability is stricter than sequential consistency. Why?
  - Linearizability speaks of consistency with respect to the **real time** that operations occurred in actual execution.
  - This is hard to implement practically due to issues we already saw a few weeks ago related to tightly synchronizing 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 it's 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: SVN, Mercurial (Hg), Git, Darcs, Arch

SVN

- Basic client-server model.
- Doesn’t facilitate distributed, replicated objects.
- Clients acquire copies from a master server, but can only see modifications from other clients when those clients synchronize with the server.
- Big improvement over CVS with respect to multiple clients:
  - Atomic commits

- SVN wasn’t a very aggressive change, but others were…

Mercurial

- Mercurial is an example of a true distributed version control system.
- Each client has a full copy of the repository.
  - Developers can commit to their local copy.
  - Clients can merge their repositories with each other to synchronize their set of objects.
  - More of a peer-to-peer style model.
- The "merge" operation is the fundamental method of synchronizing clients.
- Instead of "check in" to a centralized repository, any participants can merge each other's repositories.
- Note that merging is not necessarily automatic. Conflicts may arise that need to be resolved. Tools assist with this.

Git

- Similar to mercurial.
- Maintains cryptographic signatures for histories.
  - So corruptions of a history can be detected.
- Based on append-only database containing blobs of data and metadata to describe what files/directories the contents correspond to, digital signatures, and versioning information.
- Interestingly, we see in Git something similar to what we described last time as the *recovery manager*.
  - Git implements a recovery manager that manages the history of changes and updates.
  - The update manager deals with aborting commits and rolling back history to undo updates.
  - This is one complaint about Git – it periodically must run some maintenance of the database to clean up for efficiency (time/space).

Notes

- Other systems
  - Darcs
    - Haskell-based distributed version control. Based on pull/push model of update and synchronization.
  - Monotone
    - One of the original distributed vc systems.

- A consistent theme in these systems is the use of hash signatures (eg: SHA-1) to identify versions of files.
  - Unique signature for a file.
  - If you made changes based on a file with signature A, and merge with another repo with the same signature file, you can determine the history from a common point.

Coda, Intermezzo

- Coda is a distributed filesystem.
  - Anyone here build their own Linux kernels? Coda is an option.
  - Replicated servers.
  - Disconnected operation (ie: laptops can unplug for a little while).
  - Based on local caches of the filesystem.
  - When a system reconnects to the network, the client sends it's log of operations on it's local cache to the server(s) in order to resynchronize.
- Coda uses vector timestamps on each file.
  - Used for version history, conflict detection and resolution, and stale replica update.

Coda

- Support for disconnected operation only makes sense if you actually have the files to operate on BEFORE disconnection.
  - Not safe to rely on cache. What if a directory contains A, B, and C, but you haven't accessed C before disconnect. You might want it after disconnect.
- Uses notion of "hoarding". You provide a list of files and directories to have offline, and manually hoard them with a command.
  - "spy" command helps analyze your usage patterns while connected to determine what files to hoard without manually writing them all down.

Coda

- Disconnection and reconnection manually performed with the “cfs disconnect” and “cfs reconnect” commands.
- Reconnect attempts to merge changes made while offline with the replica managers.
- “repair” command used when conflicts arise that cannot be automatically resolved.

- Coda led to a follow-on project called InterMezzo.
- InterMezzo didn’t go very far, but led to a project that has been commercialized called “Lustre”.
  - Lustre is used for parallel filesystems in clusters.
  - Interesting note: The latest big machines in DOE are moving to Lustre or Panasas (another parallel FS) for their filesystems.
    - Nobody uses NFS at scale anymore.

Perfect place to leave off

- Next time:
  - Distributed filesystems.
  - This week's paper: Google File System (GFS)