Logistics

- Any questions about last night’s discussion?
  - Slides will be posted with the main lecture slides.
- Programming assignment?
- Project?
- Paper?

**NOTE**: Programming assignment #2 due date pushed back two days until next Thursday at 5pm.
  - Originally was due Tuesday at 2pm.

- Research paper #3 posted.
  - On transactions for shared memory concurrency.
Today we do ch. 14.

Next week, we head into Ch. 15 and then back to Ch. 8.
Distributed Transactions

- In general, data items belonging to a service may be distributed among several servers
- Client transactions involve multiple servers
  - directly by requests made by a client
  - indirectly via requests made by servers
- Distributed transaction
  - any transaction whose activities involve multiple servers
- Client transactions that involve multiple servers indirectly may be modelled as nested transactions
Requirements

- **Atomicity**
  - either all of the servers involved commit
  - or all of them abort
  - coordinator ensures the same outcome
  - depends on protocol chosen
  - “two-phase commit protocol” is common

- **Concurrency control**
  - local control to ensure transactions are serializable
  - must be serialized globally
  - extension of concurrency control methods
Structuring of Distributed Transactions

- **Simple distributed transaction**
  - client makes requests to more than one server
  - each server carries out the client’s requests without invoking operations on other servers
  - each transaction accesses servers’ data items sequentially
  - when locking is used, a transaction can only be waiting for one data item at a time

- **Nested transaction**
  - server invokes operations on other servers
  - hierarchy of nested transactions
  - Hierarchical or flattened commit protocols.
Coordinator of a Distributed Transaction

- Distributed servers need to coordinate their actions when the transaction commits.
- Client sends *OpenTransaction* request to server.
- Server returns transaction ID.
  - must be unique within a distributed system.
  - server ID + unique ID within server.
- First server in the transaction become *coordinator*.
  - responsible for committing or aborting.
  - responsible for adding other servers (*workers*).
  - records list of worker, coordinator ID.
**AddServer** Transactional Service Function

- **AddServer**(*Trans, Server ID of coordinator*)
  - informs server involved in transaction *Trans*

- **AddServer** must be used by the client before any operations are requested in a server not yet joined
  - supplies transaction ID
  - supplies transaction coordinator ID

- Receipt of **AddServer**
  - initializes local transaction
  - sends **NewServer** request to coordinator
    - **NewServer**(*Trans, Server ID of worker*)
Coordination and Transaction Completion

- Coordinator and workers knowing each other enables them to collect information needed at commit time
- Distribution of servers in a transaction can be made transparent to user-level programs
  - record ID of server that opens transaction
  - issue AddServer when new server joins with ID
- **CloseTransaction or AbortTransaction**
  - called when transaction ends
Atomic Commit Protocols

- Transaction end when client requests that the transaction should be committed or aborted
- One-phase atomic commit protocol
  - coordinator communicates the commit or abort request to all the servers in the transaction
  - continue repeating request until all had acknowledged
- One-phase atomic commit is inadequate
  - client requests a commit
  - does not allow server to unilaterally abort
  - servers must be able to abort in certain situations
Two-Phase Commit Protocol

- Designed to allow any server to abort its part of the transaction
- Due to atomicity, if one part of a transaction is aborted, the whole transaction must be aborted

First phase
- each server votes for transaction to be committed or aborted
- once a server votes commit, it cannot abort
- server must ensure it can commit before voting
- transaction is said to be a *prepared* state
Two-Phase Commit Protocol (continued)

- **Second phase**
  - every server carries out the joint decision
  - if any one server votes to abort, then the decision must be to abort
    - Think of the majority function as being boolean AND.
  - if all servers vote to commit, then the decision is to commit the transaction

- **The problem is to ensure that all the servers vote and that they all reach the same decision**
  - simple with no errors
  - protocol must work correctly in face of failures, lost messages, temporary loss of communication
More Two-Phase Commit Protocol

- A client’s request to commit/abort directed to coordinator
- Client abort or server transaction abort
  - coordinator informs workers immediately
- Two-phase commit protocol comes into play when client asks coordinator to commit
- First phase (commit)
  - coordinator asks workers if they are prepared
  - coordinator tells workers to commit (abort)
  - server-to-server operations
More Two-Phase Commit Protocol

- **Voting phase and completion phase**
- Apparently straightforward protocol could fail due to one or more of the servers failing or due to a breakdown in communication
- Each server saves information relating to the two-phase commit protocol in permanent storage
  - Permanent storage here is non-volatile, temporary space essentially.
- Timeout actions are included in the protocol
  - various stages at which a server cannot progress its part of the protocol until it receives another request or reply from one of the other servers
Timeouts

- Worker votes *Yes* and waits for coordinator to report on the outcome.
- Worker is uncertain of the outcome and cannot proceed.
- Worker makes *GetDecision* request:
  - get reply to continue protocol
  - wait for reply
- Worker could obtain decision cooperatively:
  - distributed agreement algorithm
  - useful when coordinator has failed
  - still need to get out of uncertain states
Timeouts (continued)

- Worker can be delayed when carried out all client requests, but not yet received `CanCommit?` from coordinator
  - worker can decide to `Abort` unilaterally
- Coordinator may be delayed waiting for votes from the workers
  - may decide to abort the transaction
  - announce `AbortTransaction` to the workers who have already sent their votes
  - tardy workers voting `Yes` will be ignored
Performance of Two-Phase Commit Protocol

- All goes well ($N$ servers)
  - $N-1$ CanCommit? messages and replies
  - $N-1$ DoCommit messages
    - proportional to $3N$
  - time cost: three rounds of messages
  - HaveCommitted not counted

- Worst case
  - arbitrarily many server and communication failures
  - can tolerate succession of failures
  - guarantees to complete eventually
Performance (continued)

- Considerable delay to workers in uncertain states
- Occurs when the coordinator has failed and cannot reply to *GetDecision* requests from workers
- Three-phase commit protocols have been designed to alleviate delays
Distributed Concurrency Control

- Collection of servers of distributed transactions
  - jointly responsible for ensuring transaction performed in serial equivalent manner
- T before U at one server, it must be in that order at all servers
- Mechanisms
  - locking
  - timestamp ordering
  - optimistic concurrency control
Distributed Deadlocks

- A global wait-for graph can in theory be constructed from local ones
- There can be a cycle in the global wait-for graph that is not in any single local one
  - distributed deadlock
  - deadlock iff there is a cycle in the wait-for graph
- Detection of distributed deadlock requires a cycle to be found in global transaction wait-for graph distributed among the servers
  - local wait-for graphs
  - communication required between servers
Distributed Deadlock Solutions

- Centralize deadlock detection
  - one server is global deadlock detector
  - collects local wait-for graphs
  - builds global wait-for graph and finds cycles
  - decides how to resolve deadlock
  - inform servers as to the transactions to be aborted

- Issues
  - centralized approach has poor reliability
  - transmitting local wait-for graphs is high
Phantom Deadlocks

- Deadlock detected but not really a deadlock
- Information about wait-for relationships between transactions eventually collected in one place
- Chance that transaction holding a lock will release it during deadlock detection algorithm and no deadlock will actually exist
- Simple phantom deadlocks will not arise if two-phase locks are used
  - Recall: two phase locking involves grow then shrink phase, with no releases followed by more lock acquisitions.
- A phantom deadlock could be detected if a waiting transaction in a deadlock cycle aborts during the deadlock detection procedure
Edge Chasing (Path Pushing)

- Global wait-for graph not constructed
  - servers involved each know some edges
- Servers attempt to find cycles by forwarding messages called **probes**
  - follow edges of the graph throughout system
  - contains transaction wait-for relationships representing a path in the global wait-for graph
- When should a server send out a probe?
- At any point, a transaction can be either active or waiting at just one of these servers
Edge Chasing (Path Pushing) (continued)

- Coordinator records active or waiting for a data item and workers can get this information
  - lock managers inform coordinators when transactions start waiting or become active
- Coordinator informs workers when transaction is aborted and locks can be released and edges removed in local wait-for graphs
- Edge chasing has three steps:
  - initiation: sending out probes on waiting events
  - detection: receiving probes and detecting cycles
  - resolution: aborting transactions to break deadlock
Edge Chasing (Path Pushing) (continued)

- **Initiation**
  - $T$ waits for $U$ where $U$ is waiting to access a data item at another server
  - send probe containing edge $<T \rightarrow U>$ to server where $U$ is blocked
  - if $U$ sharing a lock, probes sent to holders of lock

- **Detection**
  - receive $<T \rightarrow U>$ : check to see if $U$ also waiting
  - if so, transaction it waits for is added to the probe $<T \rightarrow U \rightarrow V>$ and probe is forward if necessary
Before a server transmits a probe to another server, it consults the coordinator of the last transaction in the path to find out whether the latter is waiting for another data item elsewhere.

Most often servers send probes to transaction coordinators which then forward them to the server of the data item the transaction is waiting for.

Deadlocks should be found provided waiting transactions do not abort and there are no failures.

- $2(N-1)$ messages sent for a cycle involving $N$ transactions.
Recovery

- Recovery necessary for failure atomicity and durability of transactions.
- Recovery manager helps make this happen.
  - Saves objects in permanent store for committed transactions.
  - Restores server objects after a crash.
  - Manage layout of permanent store to improve performance of recoveries.
  - Clean up and optimize space usage of permanent store.
Recovery and permanent store

- In distributed transactions, we have a two (or more) phase protocol.
- Before actual commit occurs, distributed servers agree that they are prepared to commit.
  - This must be recorded to permanent store before sending their response to the coordinator.
- The recovery manager also maintains a list of objects and corresponding values created by active transactions.
  - “Tentative versions” of objects.
  - Commitment causes tentative versions to replace committed versions.
Common technique: Logging

- Historical record of transactions performed by a server.
  - Values, transaction statuses, intention lists.
  - Ordered by order in which transactions occurred (started, committed, aborted).

- Think of the log as a sophisticated version history repository.
  - Maintain historical record of changes and operations.
  - Allow restoration of the most recent valid snapshot of the system when recovering from crash.