Logistics

- PA2 due today
- Last reading posted today
  - Kademlia

- Book assignment #2 posted by 5pm today
Maekawa voting revisited

- Responding to questions in office hours, let’s talk a bit more about how the Maekawa algorithm works and why we expect $\sqrt{N}$ messages to be required instead of $N$.

- Key: How do we come up with the subsets that:
  - Are of size $O(\sqrt{N})$
  - Guarantee a nonempty intersection with all others.
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
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.
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.