CS 44800: Introduction To Relational Database Systems

Distributed Databases
Prof. Chris Clifton
14 April 2021

Distributed Database

Data
Lafayette
Westville
Data

Data
Fort Wayne
Beijing
Data
Distributed Database: Why?

- Performance
  - Put the data close to the users
  - Parallelism
- Resilience
  - Fewer failures that can stop users from reaching the data
- Redundancy
  - Copies of data to handle media failure
  - Continue running when one machine fails

Distributed Database: Challenges

- Where’s the data?
  - Easy – Block ID includes location
  - Hard – What if we want to move the data?
  - Harder – What about multiple copies of the data?
- Query Processing
  - Query may access multiple sites
- Concurrency Control
  - Distributed transactions
- Failure/recovery
  - Is the fail-stop model still appropriate?
Failure Model

• Fail-stop: Entire system stops when anything fails
  – Defeats the purpose
• Individual sites fail-stop
  – Challenge: Multi-site transactions
• New problem: Link Failure
  – Both machines still running
  – But can’t communicate

Link Failure Model: *Partition*
Solution: Fail-Stop Model

• One partition continues
  – The other stops
• Which one?
  – Partition that has majority
    • Can be slow to determine majority
  – “Leader”
    • If leader not in partition, elect a new leader
    • Requires majority vote
    • Leader must ensure its partition has a majority before other partition could elect a new leader

Limitation: Transient Partition

• One side checks, can’t communicate
  – The other side is able to when it checks
• No “perfect” solution
  – See “Byzantine Generals” problem
• Requires some single point of failure
  – Make that single point extremely reliable
Distributed Failure/Recovery

- We’re back to fail-stop:
  - Does everything work as before?
- Problem: Distributed Transactions

What is a Distributed Transaction?

[Diagram showing data and transaction components distributed among multiple locations]
Why are Distributed Transactions Hard?

• **Atomic**
  – Different parts of a transaction may be at different sites
  – How do we ensure all or none committed?

• **Consistent**
  – Failure may affect only part of transaction

• **Isolated**
  – Commitment must occur “simultaneously” at all sites

• **Durable**
  – Not much different when other problems solved
  – Makes “delayed commit” difficult

Distributed Failure/Recovery

• We’re back to fail-stop:
  – Does everything work as before?

• Problem: Distributed Transactions

• Simplifying assumption: No data replication
  – Locking handled at local site
  – Transaction ensures 2-phase locking

• *Concurrency control still works*
  – Ignore the difficulty of deadlock detection/prevention
Committing a Distributed Transaction

Atomic Commit Protocols

- The steps in an Atomic Commit Protocol (ACP) are as follows:
  - TM gets a commit operation from the txn.
  - ACP needs to arrive at a **single, consistent decision** to commit or abort based upon the state of the txn at each site i.e.
    - Scheduler
    - DM (ensure that redo rule is satisfied) if there were only read operations at a site, ACP doesn't need to consult DM
  - Can do this by polling all sites.
  - Send the decision to each site.
ACP Requirements

• AC1: All processes that reach a decision reach the same one.
• AC2: A process cannot reverse its decision after it has reached one.
• AC3: The Commit decision can only be reached if all processes voted Yes.
• AC4: If there are no failures and all processes voted yes, then the decision will be to commit.
• AC5: Consider any execution containing only failures that the ACP is designed to tolerate. At any point in this execution, if all existing failures are repaired and no new failures occur for sufficiently long, then all processes will eventually reach a decision.

Key Issues

• Commitment
  – Standard techniques preserve properties when commit occurs
  – Distributed systems need commit protocols so we know when commit has occurred

• Failures
  – Standard techniques support durability for commit/abort
  – What happens if a site fails during commitment?
Two-Phase Commit  
(Lamport ’76, Gray ’79)

- Assumes central coordinator  
  - Coordinator initiates protocol  
  - Participants: entities with actions to be committed/aborted  

- Phase 1:  
  - Coordinator asks if participants can commit  
  - Participants respond yes/no  

- Phase 2:  
  - If all votes yes, coordinator sends Commit  
    • Otherwise send Abort  
  - Participants send Have Committed / Have Aborted

---

2 Phase Commit Protocol  
(Lamport ’76, Gray ’79)

1. Coord sends VOTE_REQ to all participants.
2. Each P sends a msg back with its vote: YES or NO. If it votes NO, it decides ABORT and stops.
3. The Coord collects all votes.  
   • If all are YES and its own vote is YES, it decides COMMIT and sends COMMIT msgs to each participant. Stop  
   • Otherwise, it decides ABORT and send ABORT msgs to all participants that voted YES. Stop.
4. Each participant that voted YES waits for the coord’s decision, decides accordingly and stops.
Two-Phase Commit

Complications

- If no failures take place this ACP works fine.
- However, if there are failures, we need to specify what happens when:
  - There is a timeout while waiting for a message; or
  - A site crashes and then recovers during the ACP?
- Timeout actions:
  - Participant waiting for a VOTE_REQ: unilaterally abort.
  - Coord waiting for a vote: decide ABORT and send msg to all sites that voted yes.
Cooperative Termination Protocol

- Process $P$ sends a *decision_REQ* message to every participant, $Q$. $P$ learns of the other participants from the *VOTE_REQ* message sent by the Coord.
- $Q$ does the following:
  - If $Q$ has already decided, then it sends its *decision* to $P$
  - If $Q$ has not yet voted, then it can *unilaterally abort* and send *ABORT* to $P$.
  - If $Q$ is also uncertain then it cannot help $P$ – both are *blocked*.

Handling site failure in 2PC

- We use a *distributed transaction log* to record necessary information about termination protocols, in order to recover correctly.
- The DT log can be a part of the regular log too.
- It works as follows:
  - When *Coord* sends a *VOTE_REQ*, it writes a *start-2PC* record (before or after sending message).
  - If a participant votes yes, it writes a *yes* record *before* sending the vote. This record contains the identities of the coordinator and other participants (as given by the initial message of the coord).
DT Log

- If the participant votes *no*, it writes an *abort* record, either before or after sending the vote.
- Before the *Coord* sends a commit decision, it writes a *commit* record.
- When the *Coord* sends *abort*, it writes the *abort* record to the log.
- After receiving *commit*(*abort*), a participant writes a *commit*(*abort*) record to its log.

Recovery

- When a site recovers, the fate of a distributed txn is determined as follows.
- If the DT log contains a *start-2PC* record, then the recovering site, *s*, was the coordinator
  - if it also contains a *commit* or *abort* record, then the coord had reached a decision before failure.
  - if neither is found, the coord can now unilaterally *decide* ABORT.
- If the DT log doesn’t contain the *start-2PC* record, then the site was a participant. There are three cases:
Recovery (contd.)

- The DT log contains a *commit* or *abort* record – i.e. participant had reached a decision.
- The DT log does not contain a *yes* record: either the participant failed before voting, or voted NO. It can therefore unilaterally decide to *ABORT*.
- The DT log contains a *yes* record, but no *commit* or *abort* record: participant failed during the uncertainty period – use the termination protocol to determine fate.

3PC

- The problem with 2PC is that the coordinator sends *Commit* messages while the participants are uncertain.
- Thus participants can *decide* commit while some other participants are uncertain.
- 3PC avoids this by sending *pre-Commit* messages instead of *Commit* messages, thereby moving every participant out of the uncertainty period before any participant commits.
- After coord receives ack for *pre-Commits*, it sends commit, allowing participants to commit.