Distributed Transactions

CS425/ECE428 – DISTRIBUTED SYSTEMS – SPRING 2019

Material derived from slides by I. Gupta, M. Harandi, J. Hou, S. Mitra, K. Nahrstedt, N. Vaidya
Client-Server Transactions

**Atomicity**: all-or-nothing
- Make updates on a shadow copy
- Real update on commit, discard on abort

**Consistency**: invariants satisfied
- Check and abort on violations

**Isolation**: concurrent transactions serially equivalent
- Two-phase locking (strict or otherwise)

**Durability**: results preserved after crashes
- Save committed updates to disk, recover state after crash
Distributed Transactions

A transaction that invokes operations at several servers.

Flat Distributed Transaction

Nested Distributed Transaction
Coordination in Distributed Transactions

Coordinator & Participants

The Coordination Process

Coordinator

Participant

Participant

Participant

Participant

Participant

Participant

Participant

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant

Coordinator

Participant

Participant

Participant
Distributed banking transaction

\[ T = \text{openTransaction} \]
\[ a.\text{withdraw}(4); \]
\[ c.\text{deposit}(4); \]
\[ b.\text{withdraw}(3); \]
\[ d.\text{deposit}(3); \]
\[ \text{closeTransaction} \]

Note: the coordinator is in one of the servers, e.g. BranchX
Distributed Transaction Challenges

**Atomicity**: all-or-nothing
- Must ensure atomicity across servers

**Consistency**: invariants satisfied
- Generally done locally

**Isolation**: concurrent transactions serially equivalent
- Locks at each server.

**Durability**: results preserved after crashes
- Each server keeps local recovery log
I. Locks in Distributed Transactions

Each server is responsible for applying concurrency control to objects it stores.

Servers are collectively responsible for serial equivalence of operations.

Locks are held locally, and cannot be released until all servers involved in a transaction have committed or aborted.

Locks are retained during 2PC protocol.

Since lock managers work independently, deadlocks are (very?) likely.
Distributed Deadlocks

The wait-for graph in a distributed set of transactions is distributed

Centralized detection

- Each server reports waits-for relationships to coordinator
- Coordinator constructs global graph, checks for cycles

Decentralized — edge chasing

- Forward “probe” messages to servers in the edges of wait-for graph, pushing the graph forward, until cycle is found.
Probes Transmitted to Detect Deadlock

Deadlock detected

Waits for

Held by

Waits for

Initiation

2019-03-28

NIKITA BORISOV - UIUC
Edge Chasing

**Initiation**: When a server $S_1$ notices that a transaction $T$ starts waiting for another transaction $U$, where $U$ is waiting to access an object at another server $S_2$, it initiates detection by sending $<T\rightarrow U>$ to $S_2$.

**Detection**: Servers receive probes and decide whether deadlock has occurred and whether to forward the probes.

**Resolution**: When a cycle is detected, one or more transactions in the cycle is/are aborted to break the deadlock.

Phantom deadlocks= false detection of deadlocks that don’t actually exist
- Edge chasing messages contain stale data (Edges may have disappeared in the meantime). So, all edges in a “detected” cycle may not have been present in the system all at the same time.
Transaction Priority

Transactions are given priorities
  ◦ E.g., inverse of timestamp
  ◦ Total order

When deadlock cycle is found, abort lowest priority transaction
  ◦ Only one aborted even if several simultaneous probes find cycle
II. Atomic Commit Problem

At some point, client executes closeTransaction()

- Result -> commit, abort

Atomicity requires all-or-nothing

- All operations on all servers are committed, or
- All operations on all servers are aborted

What problem statement is this?
Atomic Commit Protocols

**Consensus!**
- Impossible to be totally correct
- Possible to ensure safety, at the (possible) expense of liveness
- Plus, we already have a leader (coordinator)

First attempt: Coordinator decides
- Pick commit or abort
- Send message to all participants
- (Retransmit until acknowledged)

Problems?
- Participant crashes before receiving commit message
- Participant decides to abort (deadlock, other problems)
Two-phase Commit

Phase 1: all participants vote to commit or abort
  ◦ If you vote to commit, store partial results in permanent storage
  ◦ If crash after vote to commit, can restore transaction later

Phase 2:
  ◦ Save result of vote in permanent storage
  ◦ If all vote commit, multicast commit message
  ◦ If any vote abort, multicast abort message
RPCs for Two-Phase Commit Protocol

**Coordinator -> Participant**

- **canCommit?(trans)** -> **Yes / No**
  Ask whether participant can commit a transaction. Participant replies with its vote.

- **doCommit(trans)**
  Tell participant to commit its part of a transaction.

- **doAbort(trans)**
  Tell participant to abort its part of a transaction.

**Participant -> Coordinator**

- **haveCommitted(trans, participant)**
  Confirm that participant has committed the transaction. (May not be required if getDecision() is used – see below)

- **getDecision(trans)** -> **Yes / No**
  Ask for the decision on a transaction after participant has voted **Yes** but has still had no reply after some delay. Used to recover from server crash or delayed messages.
2PC – Coordinator

Phase 1:
- Send `canCommit?` to all participants, tabulate replies

Phase 2:
- If all votes are yes, send `doCommit` to all participants
- If any votes are no, or any participant doesn’t reply after timeout, send `doAbort` to all participants [who said yes]
- Store commit decision to stable storage to support recovery

Recovery after crash
- If commit decision in stable storage, confirm with participants (push) or wait for `getDecision` (pull)
- If `getDecision` called on commit not in log, reply `No`
2PC - Participant

Phase 1: receive canCommit?
- If OK to commit, reply Yes and store transaction in permanent storage
- If not OK, reply No and abort immediately

Phase 2
- If receive doCommit, commit transaction
- If receive doAbort, abort transaction
- If timeout, call getDecision

Recovery after crash
- If crashed after a Yes in Phase 1, call getDecision
- If should commit, recover transaction from permanent storage and commit
The two-phase commit protocol

Phase 1 (voting phase):
- The coordinator sends a `canCommit?` request to each of the participants in the transaction.
- A participant receives a `canCommit?` request and replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If its vote is No, the participant aborts immediately.

Phase 2 (completion according to outcome of vote):
- 3. The coordinator collects the votes (including its own).
  - (a) If there are no failures and all the votes are Yes, the coordinator decides to commit the transaction and sends a `doCommit` request to each of the participants.
  - (b) Otherwise, the coordinator decides to abort the transaction and sends `doAbort` requests to all participants that voted Yes.
- 4. Participants that voted Yes are waiting for a `doCommit` or `doAbort` request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a `haveCommitted` call as confirmation to the coordinator.
To deal with server crashes

- Each participant saves tentative updates into permanent storage, right before replying yes/no in first phase. Retrievable after crash recovery.

To deal with canCommit? loss

- The participant may decide to abort unilaterally after a timeout (coordinator will eventually abort)

To deal with Yes/No loss, the coordinator aborts the transaction after a timeout (pessimistic!). It must announce doAbort to those who sent in their votes.

To deal with doCommit loss

- The participant may wait for a timeout, send a getDecision request (retries until reply received) – cannot unilaterally abort after having voted Yes but before receiving doCommit/doAbort!
Two Phase Commit (2PC) Protocol

**Coordinator**
- **Execute**
  - Precommit
- **Uncertain**
  - Send request to each participant
  - Wait for replies (time out possible)
- **Abort**
  - Send ABORT to each participant

**Participant**
- **Execute**
  - Precommit
  - Send YES to coordinator
  - Wait for decision
  - Commit
    - Send COMMIT to each participant
    - Make transaction visible
  - Commit decision
  - ABORT decision
- **Abort**
  - Send NO to coordinator
  - Not ready
  - Ready
- **Abort**
  - CloseTrans()
Summary

Distributed Transactions

- More than one server process (each managing different set of objects)
- One server process marked out as coordinator
- Atomic Commit: 2PC
- Deadlock detection: Edge chasing