CS 5450

Distributed Commit

Slides from Cristina Nita-Rotaru

Distributed Commit Problem

- Some applications perform operations on multiple databases
  - For example, replicated DBs for fault tolerance
- We would like a guarantee that either all databases get updated, or none do
- Distributed commit problem
  - Operation is committed when all participants can perform it
  - Once a commit decision is reached, this requirement holds even if some participants fail and later recover

ACID Properties

Transaction (group of operations) behaves as one operation

- **(Failure) Atomicity**: if transaction failed, then no changes apply to the database
- **Consistency**: database integrity constraints always hold
- **Isolation (Atomicity)**: partial results are hidden
- **Durability**: the effects of transactions that were committed are permanent

2PC Overview

- Assumes a coordinator
  - Initiates commit/abort
- Each database informs coordinator if it is ready to commit
  - Until the commit actually occurs, the update is considered temporary
  - A pending update can be discarded (ie, transaction aborts) until all databases say “ok”
- Coordinator decides outcome and informs all databases

SOUNDS EASY!

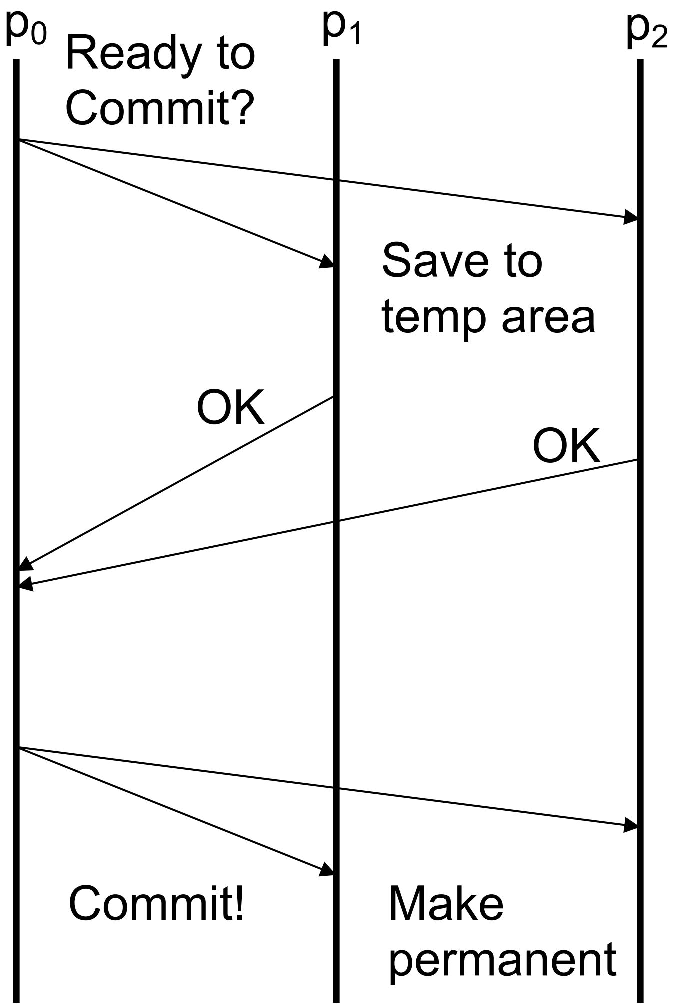
2PC (Simplified), No Failures

Coordinator:

- Multicast *ready_to_commit*
- Collect replies
  - All *Ok* => send *commit*
  - Else => send *abort*

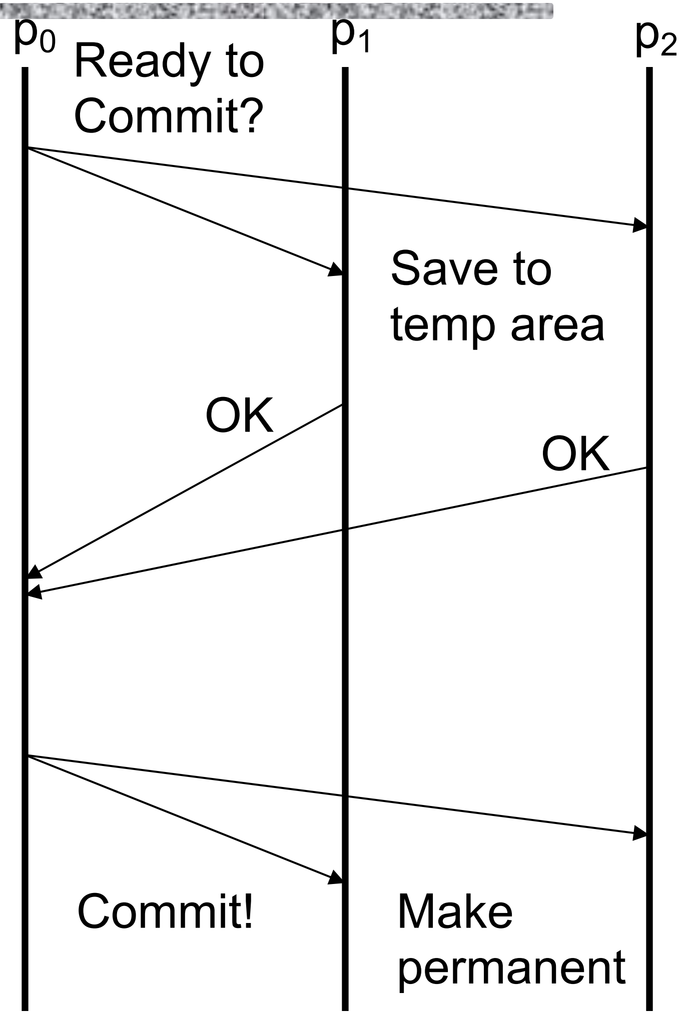
Participant receives:

- *ready_to_commit* => save to temp area and reply Ok
- *commit* => make changes permanent
- *abort* => delete temp area



Participant States

- **Initial state**: ends when p_i received *ready_to_commit* and is ready to send its *Ok*
- **Prepared to commit**: p_i sent its *Ok*, saves in temp area and waits for the final decision (*commit* or *abort*) from coordinator
- **Commit or abort**: p_i knows the final decision, must execute it



Participant Failures

- **Initial state**: if p_i crashes before receiving *ready_to_commit*, it does not send its *Ok* back, the coordinator will abort the protocol (not enough *Oks* are received)
- **Prepared to commit**: if p_i crashes before it learns the outcome, resources remained blocked. It is critical that a crashed participant learn the outcome of pending operations when it comes back (need logging system).
- **Commit or abort**: if p_i crashes before executing, it must complete the commit or abort repeatedly despite being interrupted by failures

Dealing with Participant Failures

- A participant that crashed and recovered:
  - Must remember in what state it was before crashing
  - Must find out the (global) outcome by contacting the coordinator
- The coordinator:
  - Must keep track of pending protocol runs
  - Must find out when a participant indeed completed the decision

Details

- Coordinator assigns unique identifiers for each protocol run
  - For example, use logical clocks + participant ID
- Messages carry the identifier of protocol run they are part of
- Lots of messages must be stored, need garbage collection, the challenge is to determine when it is safe to remove the information

2PC: Overcoming Participant Failures

Coordinator:

- Multicast *ready_to_commit*
- Collect replies
  - All *Ok* => log ‘commit’ to ‘outcomes’ table and send *commit*
  - Else => send *abort*
- Collect acknowledgments
- Garbage-collect protocol ‘outcomes’ information

Participant:

- Receives:
  - *ready_to_commit* => save to temp area and reply *Ok*
  - *commit* => make changes permanent, send *acknowledgment*
  - *abort* => delete temp area
- After recovering from failure:
  - For each pending protocol run: contact coordinator to learn outcome

Coordinator Failures

- If coordinator crashes when collecting *Oks*:
  - Some participants will be ready to commit (sent *Ok*)
  - Others decided to abort, did not send *Ok*
  - Others may not know the state
- If coordinator crashes during its decision or before sending it out:
  - Some participants will be in prepare-to-commit state
  - Others will not know the outcome

Modifications

- If coordinator fails, participants are blocked waiting for it to recover
- After the coordinator recovers, there are pending protocol runs that must be finished
- Coordinator must:
  - Remember its state before crashing
    - Write "commit" or "abort" into permanent storage before sending the decision to the other participants
  - Push pending operations through
- Participants may see duplicated messages

2PC Coordinator

- Multicast *ready_to_commit*
- Collect replies
  - All *OK* => log ‘commit’ to ‘outcomes’ table, wait until safely in persistent storage, then send commit
  - Else => send *abort*
- Collect *acknowledgments*
- Garbage collect protocol outcome information

After failure:

- For each pending protocol run in `outcomes’ table
  - Send outcome (*commit* or *abort*)
  - Wait for *acknowledgments*
  - Garbage collect outcome information

2PC Participant

- First time message received
  - *ready_to_commit*
    - save to temp area and reply *OK*
  - *commit*
    - make changes permanent
  - *abort*
    - delete temp area
- Message is a duplicate (because of a recovering coordinator)
  - Send *acknowledgment*
- After failure:
  - For each pending protocol run:
    - contact coordinator to learn outcome

What If Coordinator Never Recovers?

- Instead of blocking, can allow remaining participants to complete the protocol on their own
  - Caveat: any participant taking over will not be able to safely conclude that the coordinator actually failed
- If timeout expires at a participant that is in the prepare-to-commit state:
  - The participant can send out the first-phase message, querying the other participants to learn outcome
  - Continue with second phase

Allowing Progress

- Can a participant always determine the outcome?
  - Example: all participants are in prepared-to-commit state except p_j, which can not be reached
  - Only the coordinator and p_j can determine the outcome
- If the coordinator is itself a participant, only one failure blocks the protocol
- All participants must now maintain information about the outcome of the protocol until they are sure that all participants learned the outcome

Garbage Collection

- Add another phase from the coordinator to all participants, telling participants that it is safe to garbage-collect the protocol information
- If coordinator fails:
  - If a participant is in the final state but did not see the garbage collect message, it will send again the commit or abort message
  - All participants acknowledge after they execute
  - Once all participants acknowledged the message, garbage collection message can be sent out
- Garbage collection can be run periodically

2PC: Coordinator (Final)

Multicast: ready_to_commit

Collect replies

- All OK => log 'commit' to 'outcomes' table, wait until safe in persistent storage, send commit
- Else => send abort

Collect acknowledgments

After failure:

- For each pending protocol run in 'outcomes' table
  - Send outcome (commit or abort)
  - Wait for acknowledgments

Periodically

- Query each participant about terminated protocol runs
- Determine fully terminated protocol runs to garbage collect their outcome information

2PC: Participant (Final)

First time message received

- *ready_to_commit*
  - save to temp area and reply *OK*
- *commit*
  - Log outcome, make changes permanent
- *abort*
  - Log outcome, delete temp area

Message is a duplicate (recovering coordinator)

- Send *acknowledgment*

After failure:

- For each pending protocol run:
  - contact coordinator to learn outcome

After timeout in prepare to commit state:

- Query other participants about state
  - If outcome can be deduced: Run coordinator-recovery protocol
  - If outcome uncertain: must wait

2PC Summary

- Message complexity O(n^2)
- Worst case: network disrupts the communication in each phase
- Pure 2PC will always block if coordinator fails
- Final version provides increased availability but can still block if a failure occurs at a critical stage: both coordinator and a participant fail during the decision stage
    - Trades liveness for safety

3-Phase Commit

- Guarantees that the protocol will not block when only fail-stop failures occur
  - Assumption: a participant fails only by crashing, crashes are accurately detectable
- Fundamental problem: coordinator makes a decision which will be known and acted upon by some participant, while other participants will not know it

3PC Key Idea

Additional round of communication (and delays) to ensure that the state of the system can always be deduced by a subset of alive participants that can communicate with each other

before the commit, coordinator tells all participants that everyone sent OKs

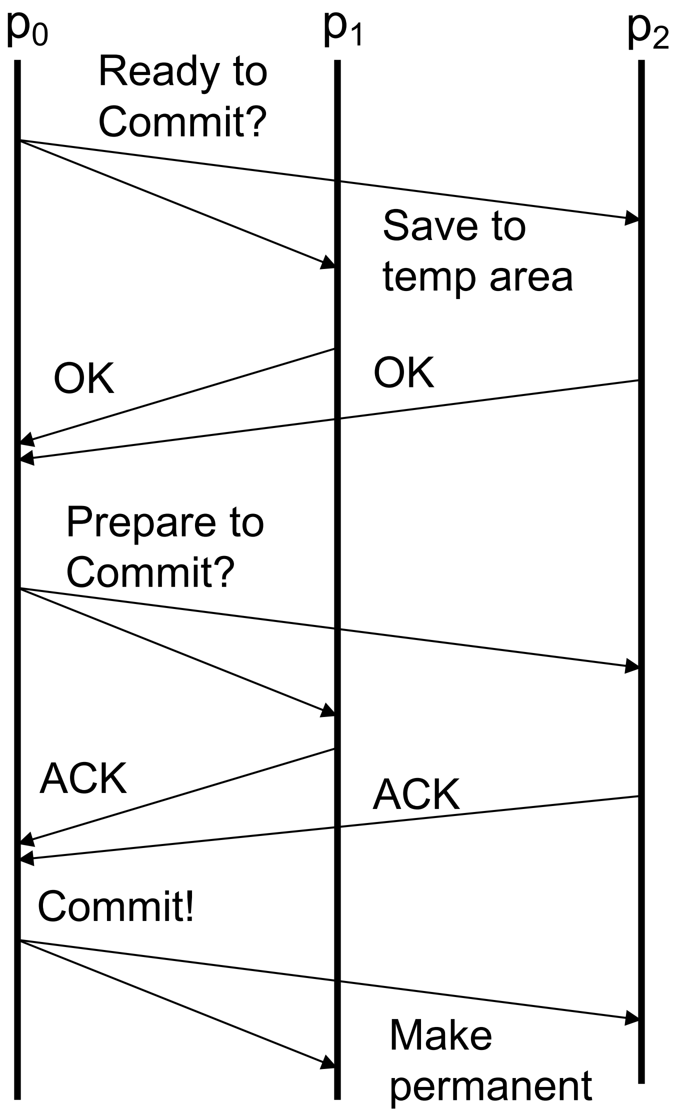
3PC, No Failures

Coordinator:

- Multicast *ready_to_commit*
- Collect OKs
  - All *Ok* => send pre*commit*
  - Else => send *abort*
- Collect ACKs
  - All *ACK* => send *commit*

Participant receives:

- *ready_to_commit* => save to temp area and reply Ok
- *precommit => send ACK*
- *commit* => make changes permanent
- *abort* => delete temp area



Dealing with Failures

- Alive participants select a new coordinator and try to complete transaction, based on their current states
  - Membership is static, detection is accurate, alive participant with lowest id is selected
- If crashed nodes committed or aborted, then survivors should not contradict; otherwise, survivors can do as they decide

3PC: Coordinator

Multicast *ready_to_commit*

Collect replies

- All *OK* => log 'precommit' and send precommit
- Else => send *abort*

Collect acks from alive participants

- All *ack* => log commit and send commit

Collect acknowledgements that operation was finished

Garbage-collect protocol outcome information

3PC: Participant

Participant logs its state after each message

ready_to_commit

- save to temp area and reply *OK*

precommit

- Enter precommit state, send *ack*

commit

- make changes permanent

abort

- delete temp area

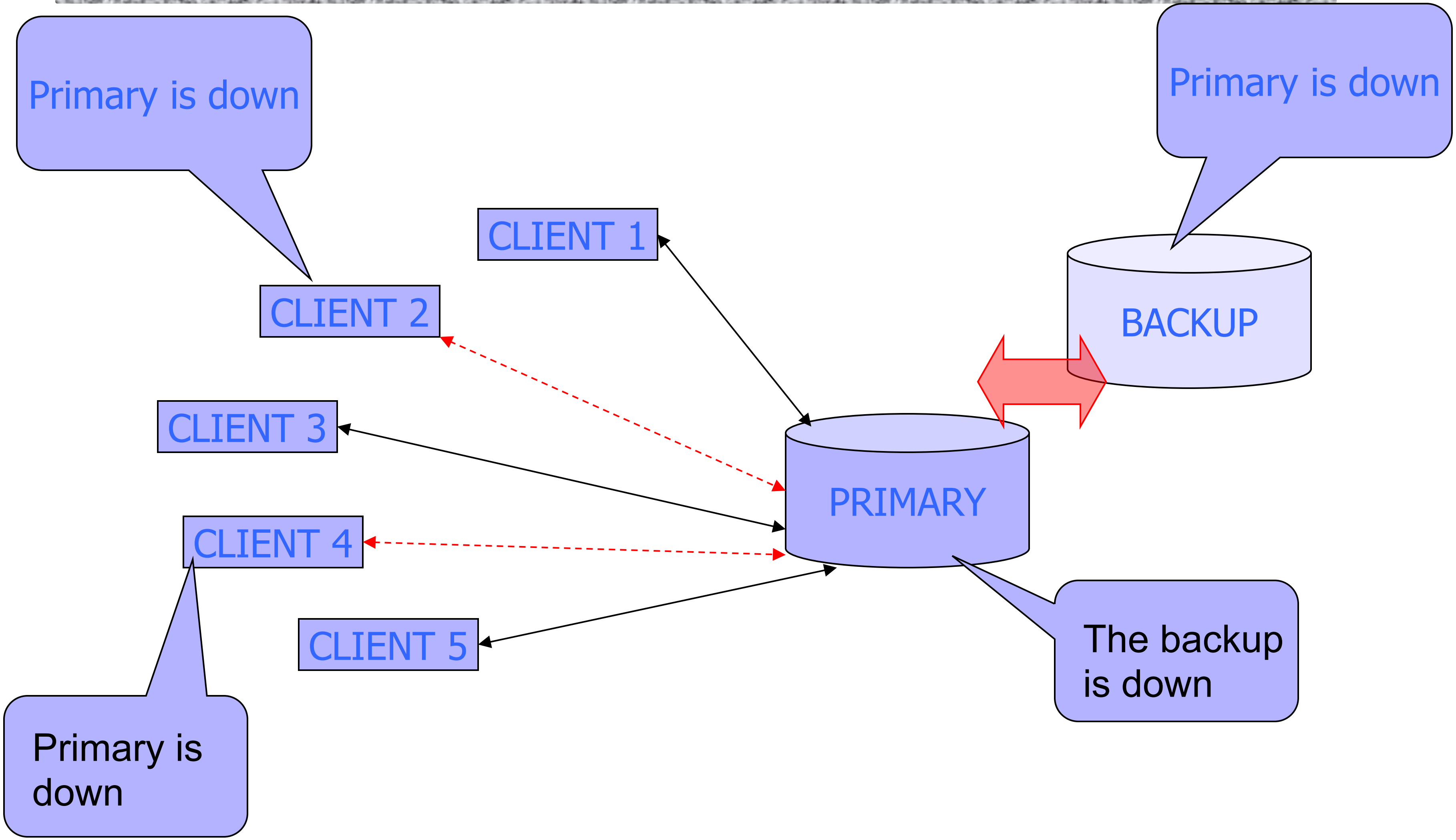
After failure:

- Collect participant state information
- All *precommit* or any *commit*, push forward the commit
- Else, push back the abort

3PC and Network Partitions

- Suppose a network partition separates the participants in two groups:
  - One group has received precommit, so they all terminate the protocol by commit
  - The other group sees a state that is ok to commit but they have not received precommit, so they all abort upon timeout
- 3PC does not work in case of network partitions

Things Goes Wrong…



3PC Summary

- Requires 3 phases
  - 4 with garbage collection
- Works only under fail-stop failures
- Does not work (produces an inconsistent global state) if network partitions happen
  - 3PC trades safety for liveness
  - Remember CAP theorem?

- Can we design a protocol that is safe and live?
  - FLP impossibility result