

# CS603: Distributed Systems

## Lecture 9: Distributed Commit

# Distributed Commit Problem

---

- Some applications perform operations on multiple databases
- We would like a guarantee that either *all* the databases get updated, or *none* does
- 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

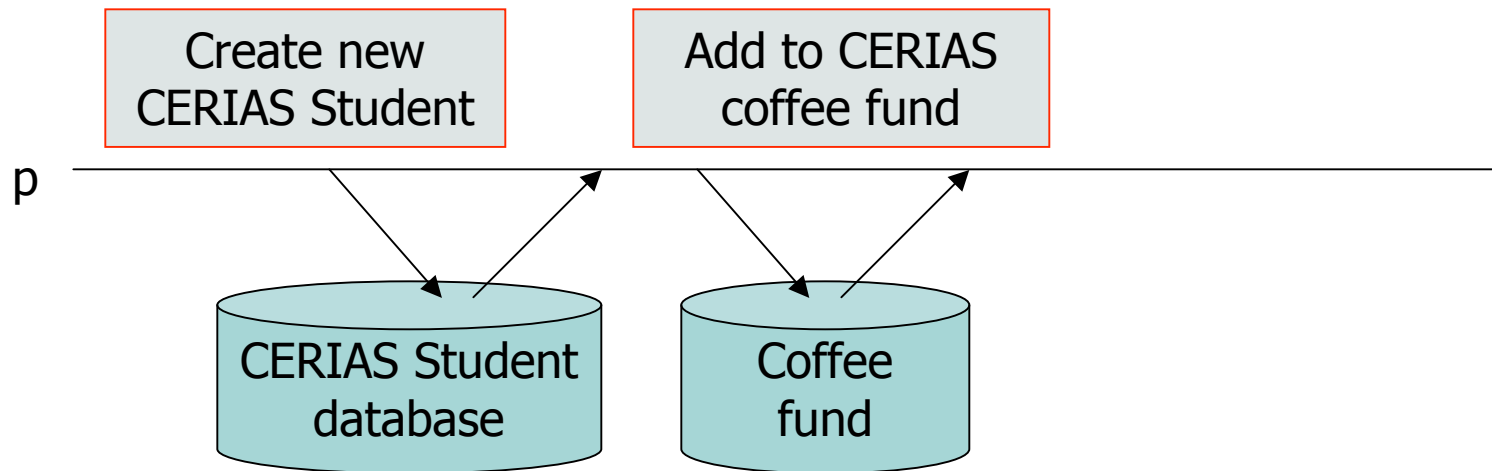
# ACID Properties

---

- Transaction behave as one operation
- *(Failure) Atomicity*: all or none, if transaction failed then no changes apply to the database
- *Consistency*: there is no violation of the database integrity constraints
- *Isolation (Atomicity)*: partial results are hidden
- *Durability*: the effects of transactions that were committed are permanent

# Example

---



- Either  $p$  succeeds, and both lists get updated, or something fails and neither does

# What can go wrong?

---

- Process  $p$  could crash during the execution
- ... a database could throw an exception, e.g. “invalid SSN” or “duplicate record”
- ... a database could crash, then restart, and may have “forgotten” uncommitted updates (presumed abort)

# 2PC Overview

---

- Assumes a coordinator that initiates the commit/abort
- Each database votes if it is ready to commit
  - Until the commit actually occurs, the update is considered temporary
  - Database is permitted to discard a pending update  
Until all servers vote “ok” a database can abort
- Coordinator decides outcome and informs all databases

**SOUNDS EASY!**

# 2PC: More Details

---

- Operates in rounds
- Coordinator assigns unique identifiers for each protocol run. How? It's time to use logical clocks: run identifier can be process ID and the value of logical clock
- Messages will carry the identifier of protocol run they are part of
- Since lots of messages must be stored, a garbage collection must be performed, the challenge is to determine when it is safe to remove the information

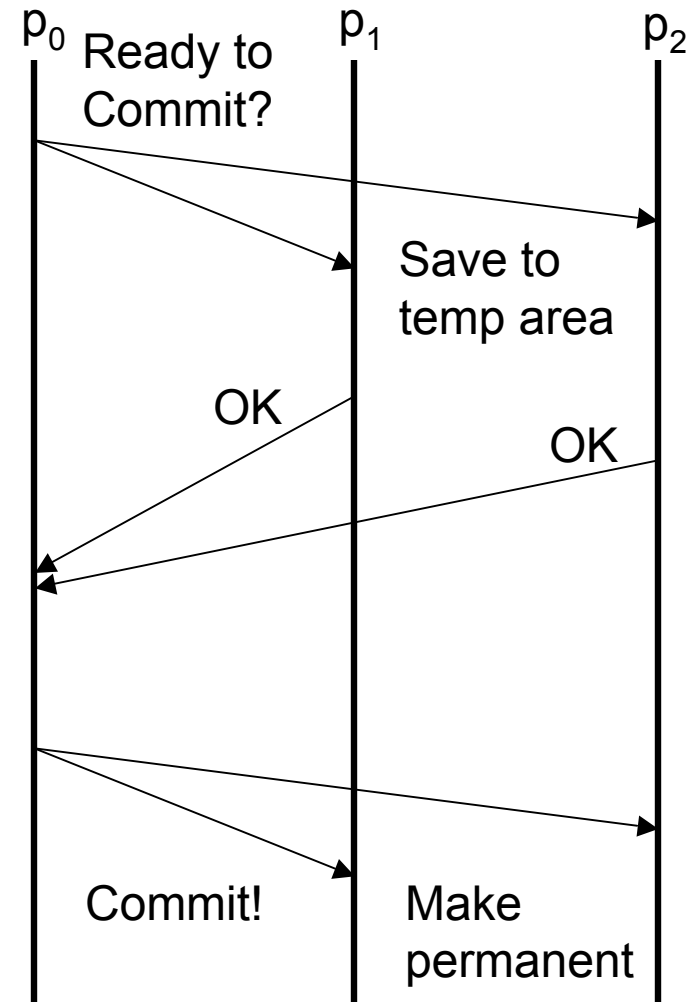
# 2PC: Simplified Version

Coordinator:

- Multicast: *ready\_to\_commit*
- Collect replies
  - All *Ok* => send *commit*
  - Else => send *abort*

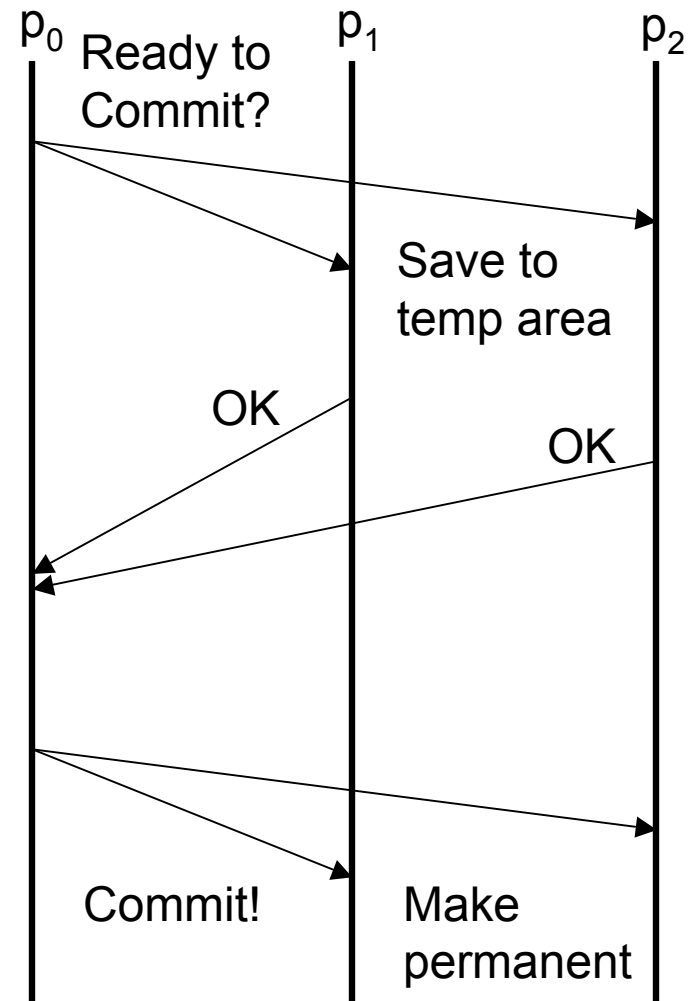
Participant receives:

- *ready\_to\_commit* => save to temp area and reply *Ok*
- *commit* => make changes permanent
- *abort* => delete temp area



# Participant States

- **Initial state:**  $p_i$  is not aware that protocol started, ends when  $p_i$  received the *ready\_to\_commit* and it 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 state:**  $p_i$  knows the final decision, it must execute it



# Let's go back to what can go wrong!

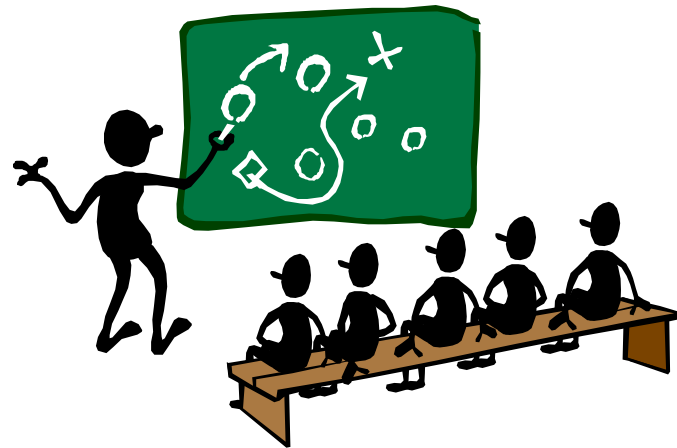
---

- **Initial state**: if  $p_i$  crashes before it received **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 learns the outcome of pending operations when it comes back: need service or logging system.
- **Commit or abort state**:  $p_i$  crashes before executing, it must complete the commit or abort repeatedly in spite of being interrupted by failures.

# What modifications are needed?

---

- A process must remember in what state it was before crashing
- A process must find out the outcome (by contacting the coordinator)
- The coordinator must find out when a process indeed completed the decision, since it can crash before executing it



# 2PC: Overcome Participant Failures

---

Coordinator:

- Multicast: *ready\_to\_commit*
- Collect replies
  - All *OK* => log 'commit' to 'outcomes' table and send *commit*
  - Else => send *abort*
- Collect acknowledgments
- Gargabe-collect protocol outcome information

Participant:

- *ready\_to\_commit* => save to temp area and reply *OK*
- *commit* => make changes permanent
- *abort* => delete temp area
- After failure:
  - For each pending protocol: contact coordinator to learn outcome

# What can go wrong - Part II

---

- **Coordinator can fail too**
- If coordinator crashed during first phase:
  - some participants will be ready to commit
  - others will not be able to (they voted on abort)
  - other process may not know what the state is
- If coordinator crashed during its decision or before sending it out:
  - some processes will be in prepare to commit state
  - some others will know the outcome

# Modifications ...

---

- If coordinator fails, processes are blocked waiting for it to recover
- After the coordinator recovers, there are pending protocols that must be finished
- Coordinator must remember its state before crashing (write commit or abort on permanent storage before sending commit or abort decision to other processes) and push these operations through
- Participant may see duplicated messages

# 2PC: Overcome Coordinator Failures (1)

---

Coordinator:

- Multicast: *ready\_to\_commit*
- Collect replies
  - All *OK* => log 'commit' to 'outcomes' table, wait until safe on persistent storage and send *commit*
  - Else => send *abort*
- Collect acknowledgments
- Garbage collect protocol outcome information

After failure:

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

# 2PC: Overcome Coordinator Failures (2)

---

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 (recovering coordinator)

- Send acknowledgment

After failure:

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

# Allowing Progress...

---

- **WHAT IF THE COORDINATOR DOES NOT RECOVER?  
HOW CAN WE ALLOW PROGRESS?**
- One option instead of blocking is to allow the other 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.  
WHY?
- Timeout expired at a participant that is in the prepared-to-commit state:
  - The process can send out the first phase message, querying the state at other processes to learn outcome
  - Continue with second phase

# Allowing Progress (cont.)

---

- Can a process always determine the outcome?
- **Example: all processes are in prepared-to-commit state with the exception of one process let's say  $p_j$ , 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 learnt the outcome

# Garbage Collection

---

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

# 2PC: Final Version (Coordinator)

---

Coordinator:

- Multicast: *ready\_to\_commit*
- Collect replies
  - All *OK* => log 'commit' to 'outcomes' table, wait until safe on persistent storage and send *commit*
  - Else => send *abort*
- Collect acknowledgments

After failure:

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

Periodically

- Query each process: terminated protocols?
- For each coordinator: determine fully terminated protocols
- 2PC to garbage collect protocol outcome information

# 2PC: Final Version (Participant)

---

Participant: 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:
  - contact coordinator to learn outcome

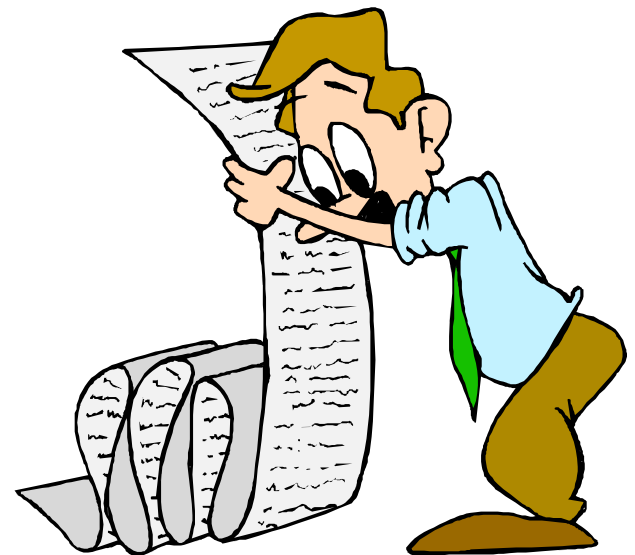
After timeout in prepare to commit state:

- 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: will be unable to terminate if both coordinator and a participant fail during the decision stage



---

# Three-Phase Commit

# 3 PC Overview

---

- Guarantees that the protocol will not block when only fail-stop failures occur
- A process fails only by crashing, crashes are accurately detectable
- Model is not realistic, but still interesting to look at
- Requires a fourth round for garbage collection
- Remember that 2 PC blocks when coordinator and one more participant fail
- Fundamental problem: coordinator will make a decision which will be known and acted upon for some process, while other processes will not know it

# 3 PC Key Idea

---

- Introduces an additional round of communication and delays to prepare-to-commit state to ensure that the state of the system can always be deduced by a subset of alive processes that can communicate with each other

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

# What happens in case of failures?

---

- Alive processes ( $P_i$ ) will select a new coordinator ( $P_j$ ) try to complete transaction, based on their current states
- New coordinator selection: membership is static, detection is accurate, alive process 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

---

Coordinator:

- Multicast: *ready\_to\_commit*
- Collect replies
  - All *OK* => log 'precommit' and send *precommit*
  - Else => send *abort*
- Collect acks from non-failed participants
  - All ack => log commit and send *commit*
- Collect acknowledgements
- Garbage collect protocol outcome information

# 3PC: Participant

---

Participant: logs state on each message

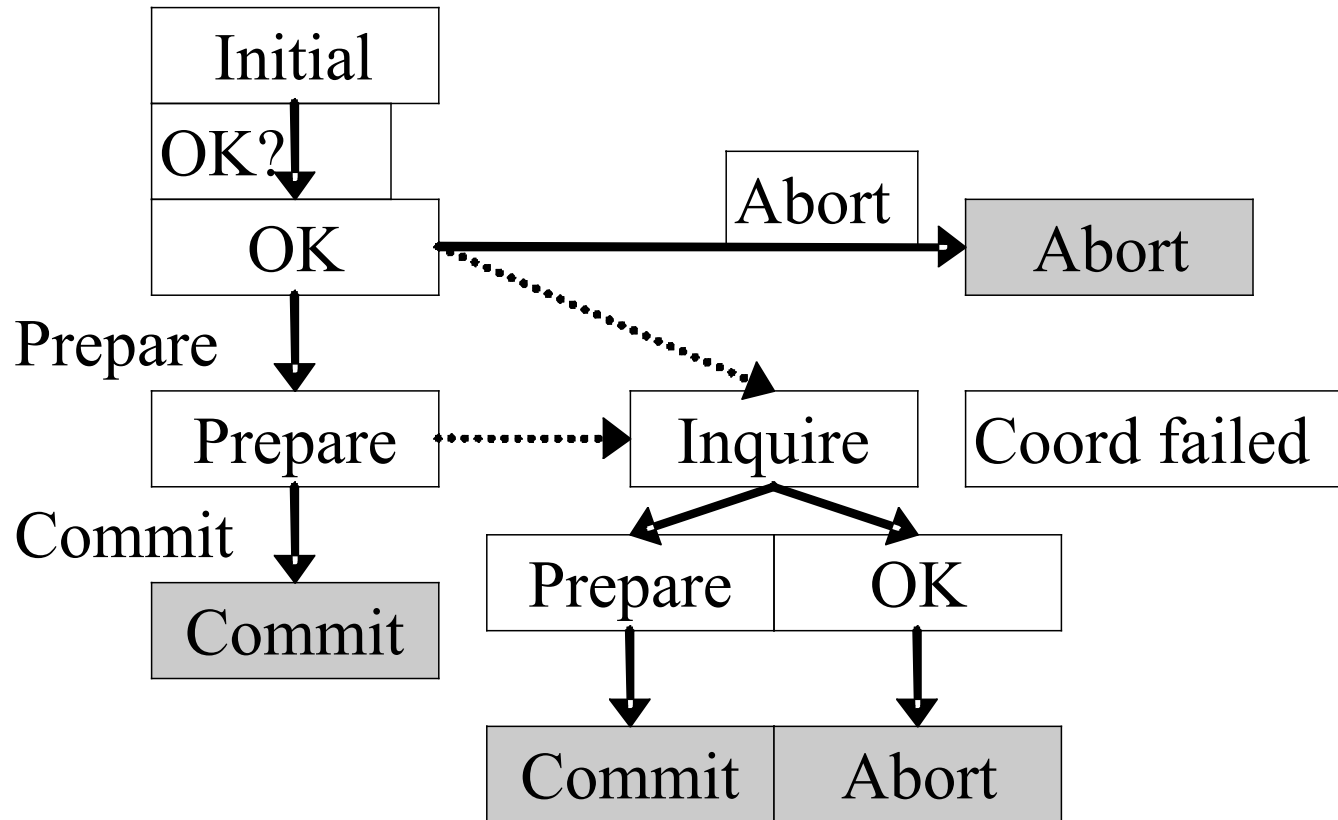
- *ready\_to\_commit*
  - save to temp area and reply *OK*
- *precommit*
  - Enter precommit state, send *ack*
- *commit*
  - make changes permanent
- *abort*
  - delete temp area

After failure:

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

# States for a nonfaulty participant in 3PC

---



# 3PC and Network Partitions

---

- Consider the case when a network partition separates the processes in two groups:
  - One group sees that they are prepared to commit and go and terminate the protocol by commit
  - The other group sees a state that is ok to commit and would consider the safe decision to be abort
- **3PC does not work in case of network partitions**

# 3PC

---

- Requires 3 phases (4 with garbage collection)
- Works only under fail-stop (model unrealistic)
- Does not work if network partitions happen

