Cristina Nita-Rotaru

7610: Distributed Systems

Distributed commit. 2PC. 3PC

Plan

- Distributed Commit
- Two-Phase Commit
- Three-Phase Commit
- CAP

Required reading for this topic…

- Non-Blocking Commit Protocols, D. Skeen, SIGMOD 1981

1: Distributed Commit

Distributed Commit Problem

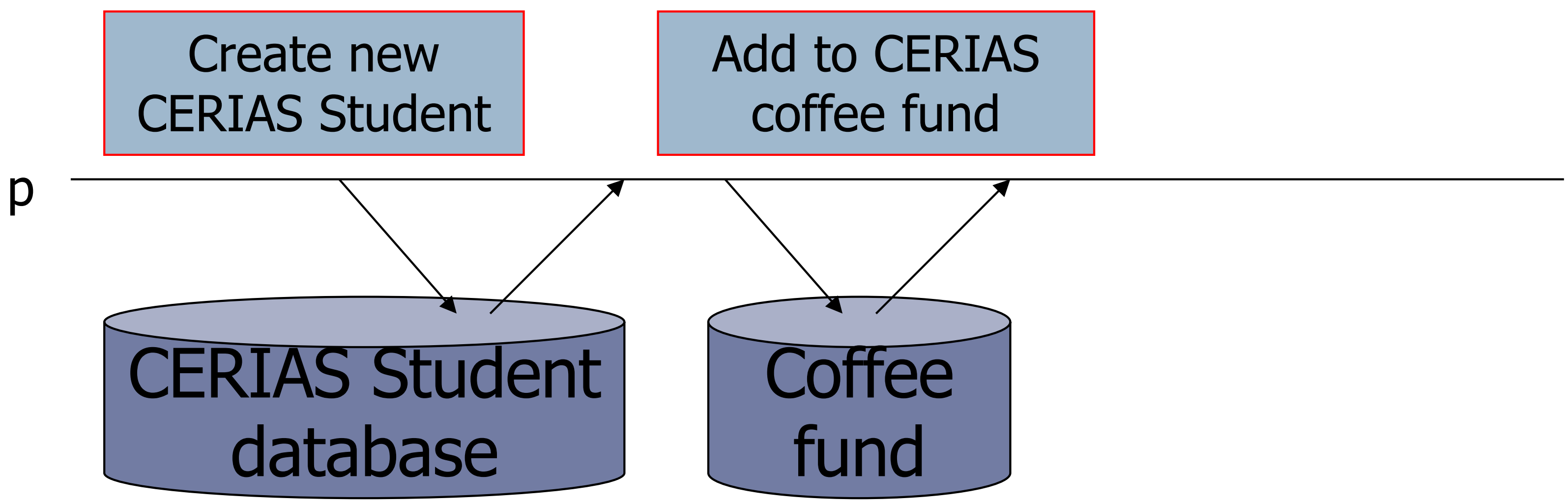
- Some applications perform operations on multiple databases
- We would like a guarantee that either all the databases get updated, or none does
- Distributed Commit Problem:
  - **Operation is committed when all participants can perform it**
  - **Once a commit decision is reached, this requirement holds even if some participants fail and later recover**

ACID Properties

- Transaction behaves 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 tables 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)

2: Two-Phase Commit (2PC)

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"
  - 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? Use logical clocks: run identifier can be process ID and the value of logical clock
- Messages 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

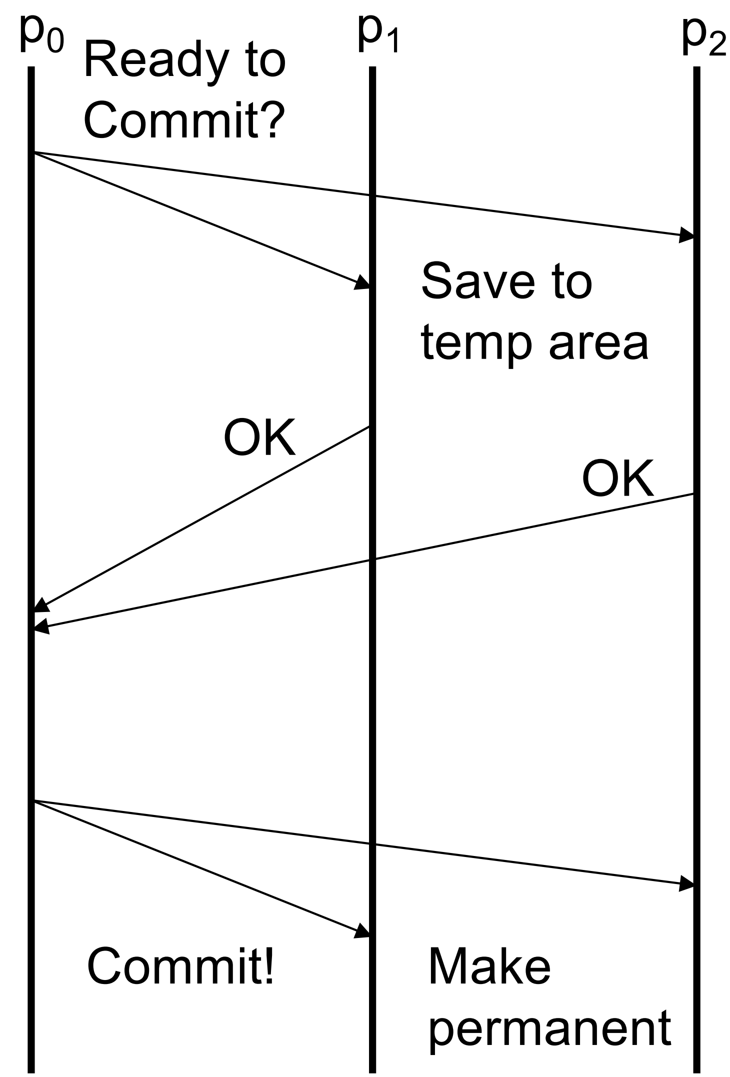
2PC Simplified Version: No Failures

Coordinator:

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

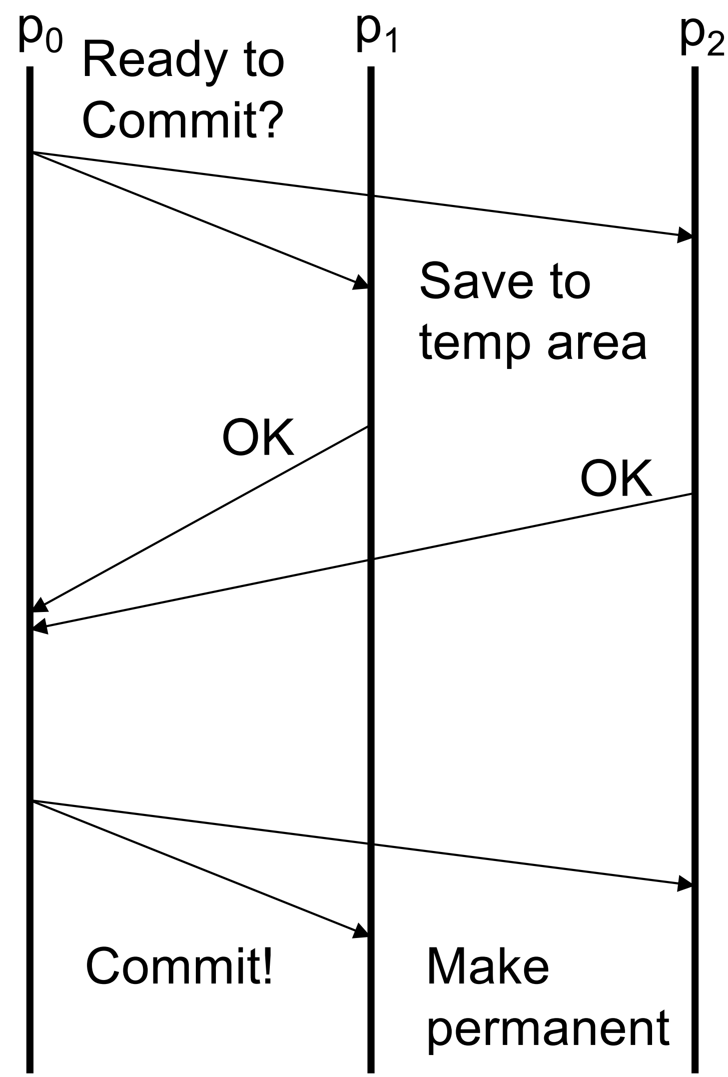
Participant receives:

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



Participant States

- **Initial state**: p_i is not aware that protocol started, ends when p_i received *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**: p_i knows the final decision, it must execute it

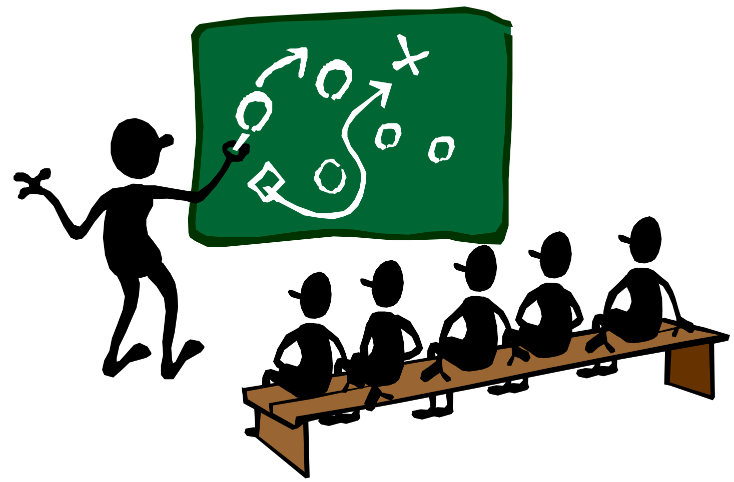


Failures: Participant

- **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. <u>It is critical that a crashed participant learns the outcome of pending operations when it comes back</u>: need logging system.
- **Commit or abort**: p_i crashes before executing, it must complete the commit or abort repeatedly in spite of being interrupted by failures.

How to Fix It?

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



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: contact coordinator to learn outcome

Failures: Coordinator

- If coordinator crashed during first phase when collecting *Oks*:
  - Some participants will be ready to commit (they sent *Ok*)
  - Others will not be able to (they voted on abort)
  - Others may not know the state
- If coordinator crashed during its decision or before sending it out:
  - Some processes will be in prepare to commit state
  - 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)
  - push pending operations through
- Participants may see duplicated messages

2PC Overcoming Coordinator Failures: 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 Overcoming Coordinator Failures: 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:
  - 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 prepare-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

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

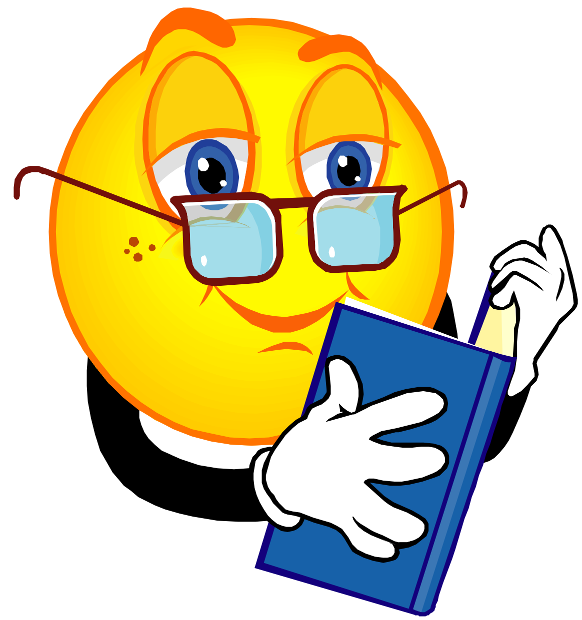
- Query each process: terminated protocols?
- Determine fully terminated protocols to garbage collect protocol outcome information

2PC Final Version: 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(n2)
- 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



3: Three-Phase Commit (3PC)

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

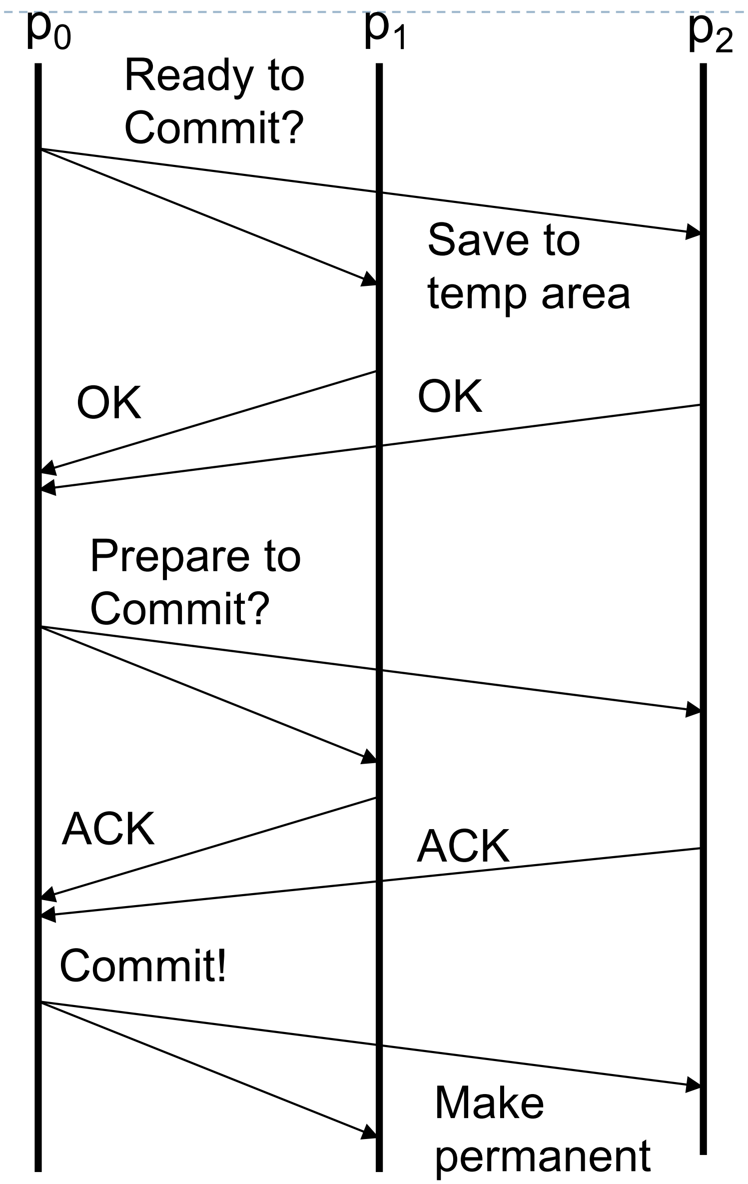
3PC Simplified Version: No Failures

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



What happens in case of failures?

- Alive processes (p_i) will select a new coordinator and 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

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 that operation was finished

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

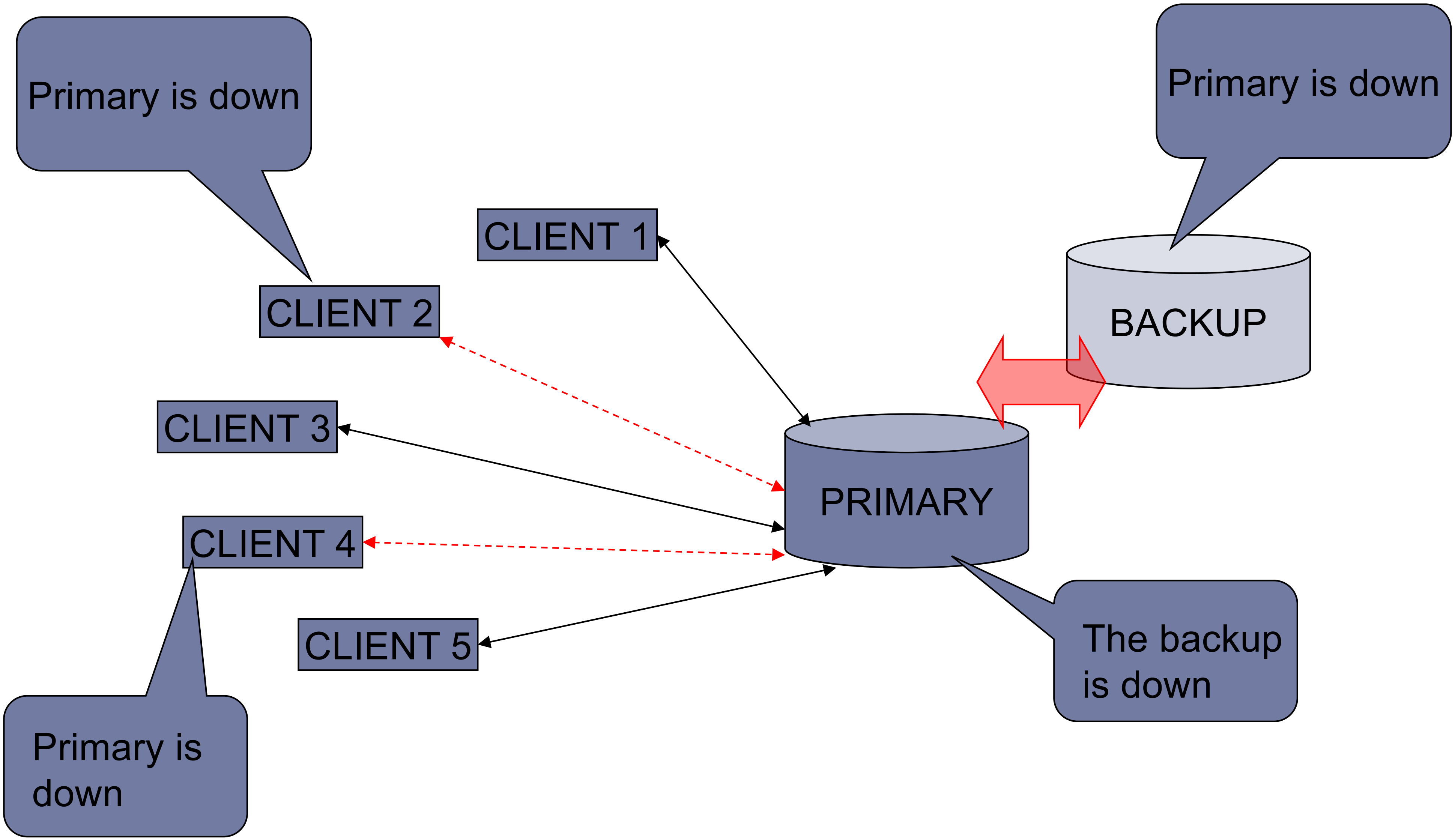
All *precommit* or any *commit*, push forward the commit

Else, push back the abort

3PC and Network Partitions

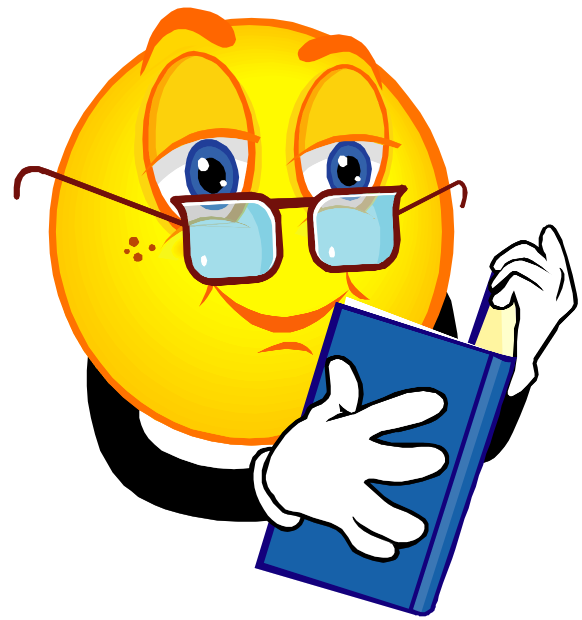
- Consider the case when a network partition separates the processes in two groups:
  - One group sees that they are prepared to commit and go and terminate the protocol by commit
  - The other group sees a state that is ok to commit and would consider the safe decision to be abort
- 3PC does not work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

Things go wrong…



3PC

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



4: CAP and PACELC

CAP Theorem

- States that any networked shared-data system can have at most two of three desirable properties:
  - consistency (C) equivalent to having a single up-to-date copy of the data;
  - high availability (A) of that data (for updates);
  - tolerance to network partitions (P).
- During a network partition and recovery from partition one can not have perfect availability and consistency
- Modern CAP goal should be to maximize combinations of consistency and availability that make sense for a specific application

CAP Twelve Years Later: How the "Rules" Have Changed, E. Brewer

Beyond CAP: PACELC Theorem

- States that:
  - In case of network partitioning (P) in one has to choose between availability (A) and consistency (C)
  - but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).
- Address the fact that CAP does not capture the consistency/latency tradeoff of replicated systems present at all times during system operation
- Example: Dynamo, Cassandra, and Riak are PA/EL systems if a partition occurs, they give up consistency for availability, and under normal operation they give up consistency for lower latency.