8. Distributed DBMS Reliability

Chapter 12

Distributed DBMS Reliability

8. Distributed DBMS Reliability

- ☞ Concept of Reliability
- ❖ Local Reliability Protocols
- ❖ Distributed Reliability Protocols
- ❖ Brewer's CAP Theorem and Relevant Efforts

Reliability

Problem

How to maintain

- atomicity
- durability

properties of transactions?

Fundamental Definition - Reliability

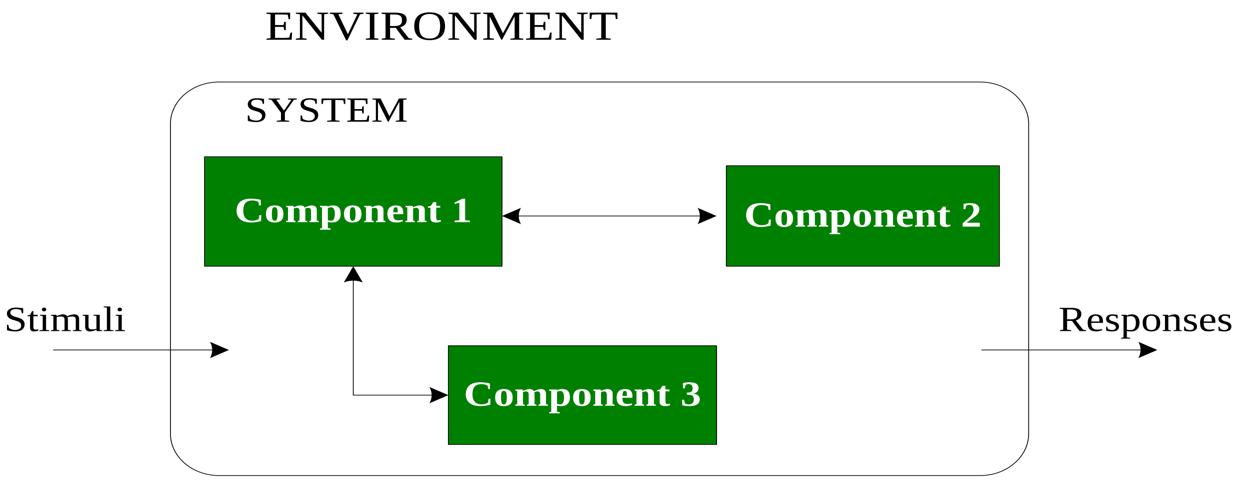
- A measure of success with which a system conforms to some authoritative specification of its behavior.
- Probability that the system has not experienced any failures within a given time period.
- Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.

Fundamental Definition - Availability

- The probability that the system is operational at a given time t.

- The fraction of the time that a system meets its specification.

Schematic of a System



External state of a system:

response that the system gives to an external stimulus

Internal state of a system:

union of the external states of the components that make up the system

From Fault to Failure

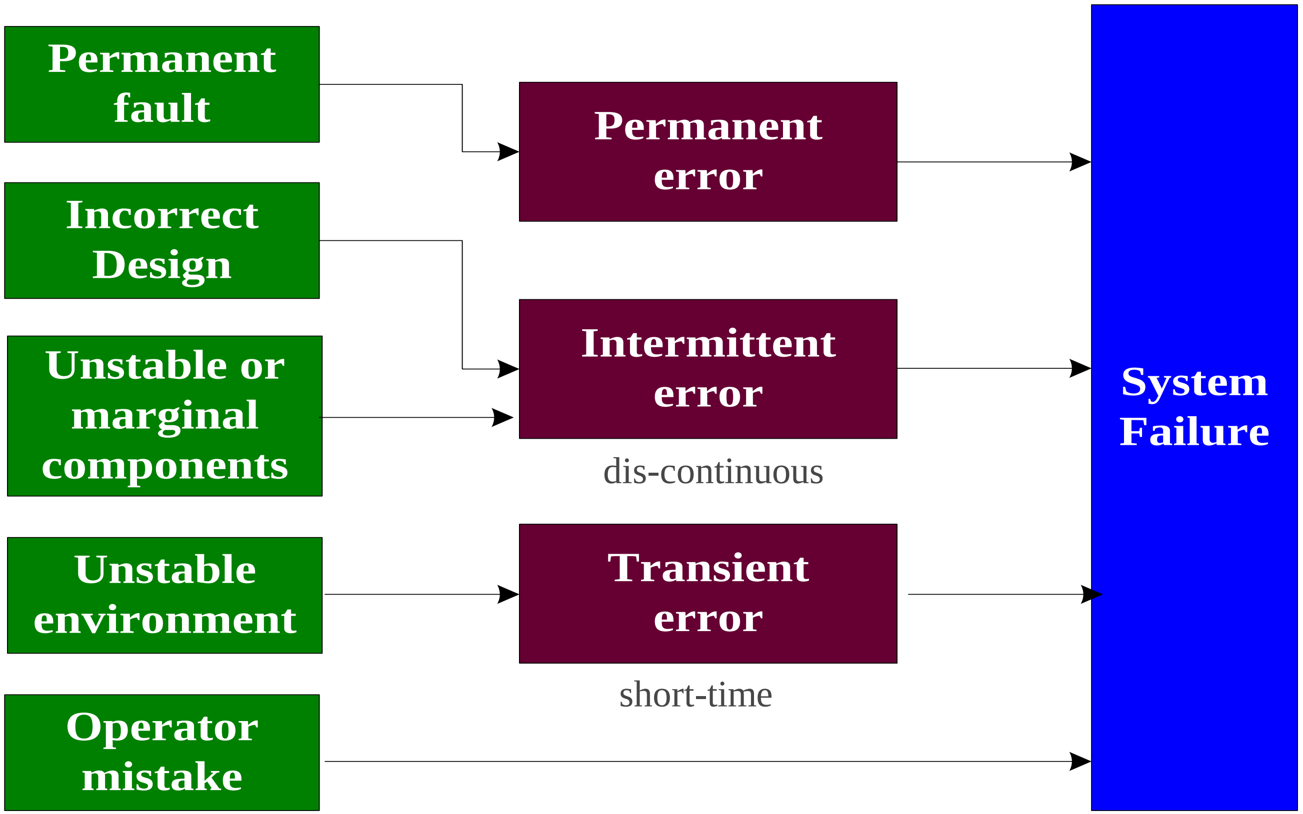
Fault $\xrightarrow{\text{causes}}$ **Error** $\xrightarrow{\text{results in}}$ **Failure**

- Fault: an error in the internal states of the components of a system or in the design of a system.
- Error: the part of the state which is incorrect.
- Erroneous state: the internal state of a system such that there exist circumstances in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.
- Failure: the deviation of a system from the behavior that is described in its specification.

Types of Faults

- Hard faults
  - Permanent (reflecting an irreversible change in the behavior of the system)
  - Resulting failures are called ***hard failures***
- Soft faults
  - Intermittent (not continuous) or transient (lasting for a short time) due to unstable states
  - Resulting failures are called ***soft failures***
  - Account for more than 90% of all failures

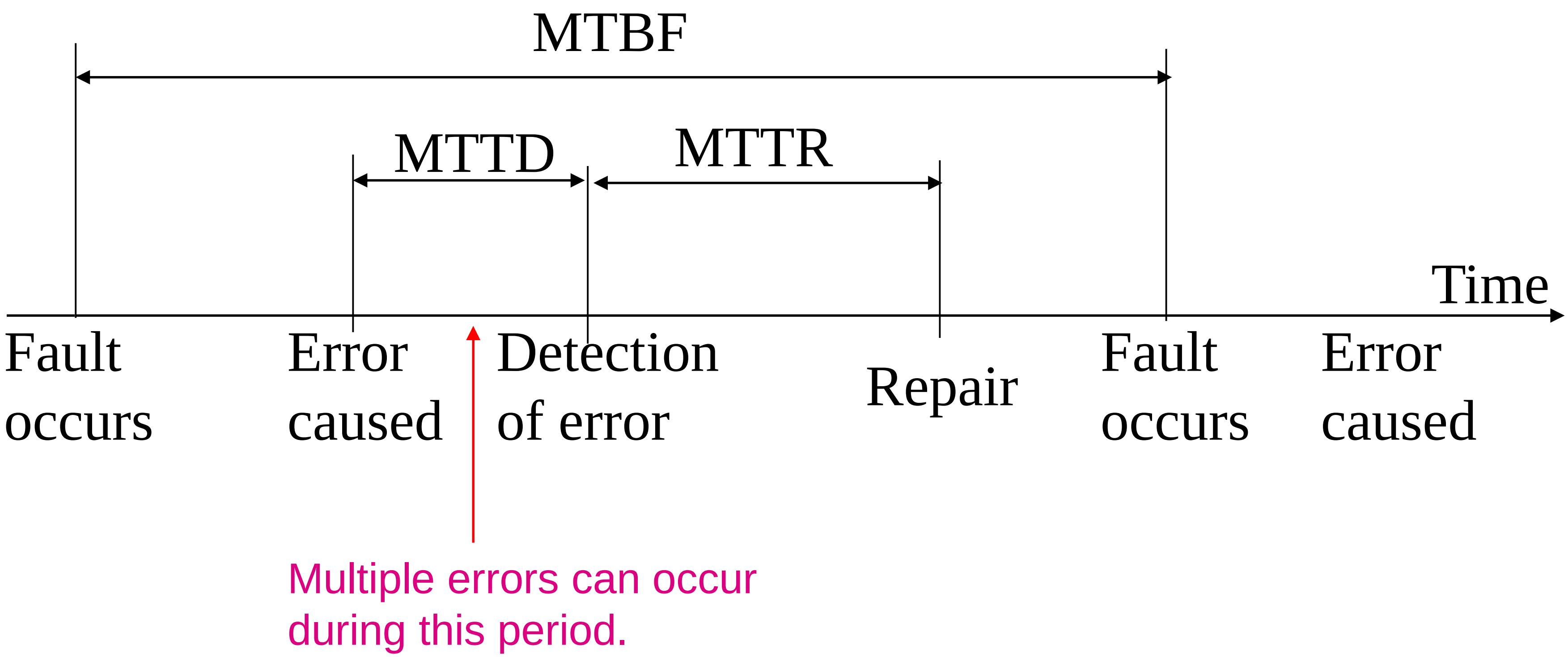
Fault Classification



Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- System (site) failures
  - Failure of processor, main memory, power supply, ...
  - Main memory contents are lost, but secondary storage contents are safe
- Media failures
  - Failure of secondary storage devices such that the stored data is lost
  - Head crash/controller failure
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

Failures

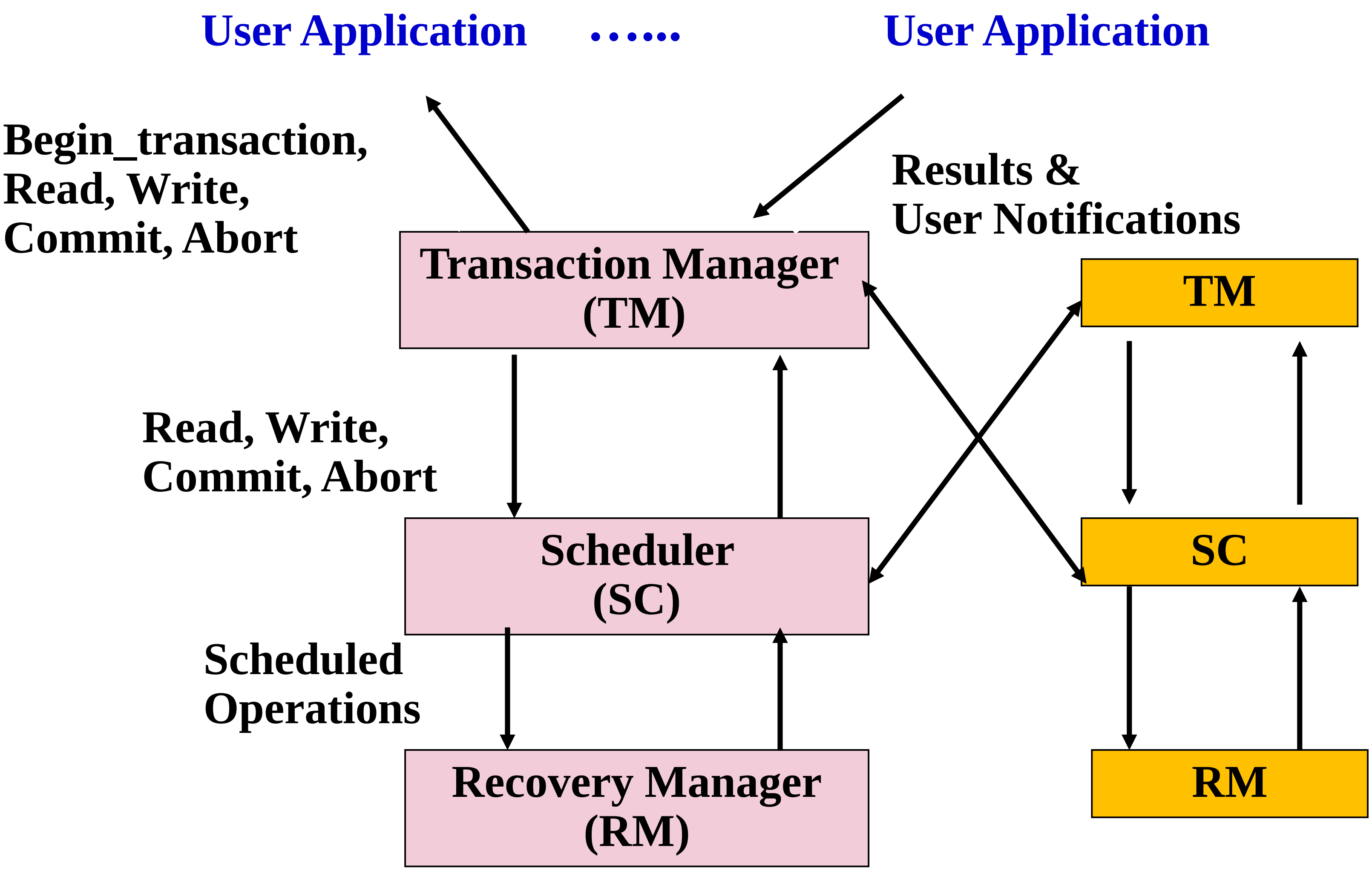


MTBF: *mean time between failures*
MTTD: *mean time to detect*
MTTR: *mean time to repair*

8. Distributed DBMS Reliability

- ❖ Concept of Reliability
- ☞ Local Reliability Protocols
- ❖ Distributed Reliability Protocols
- ❖ Brewer's CAP Theorem and Relevant Efforts

Centralized Transaction Execution

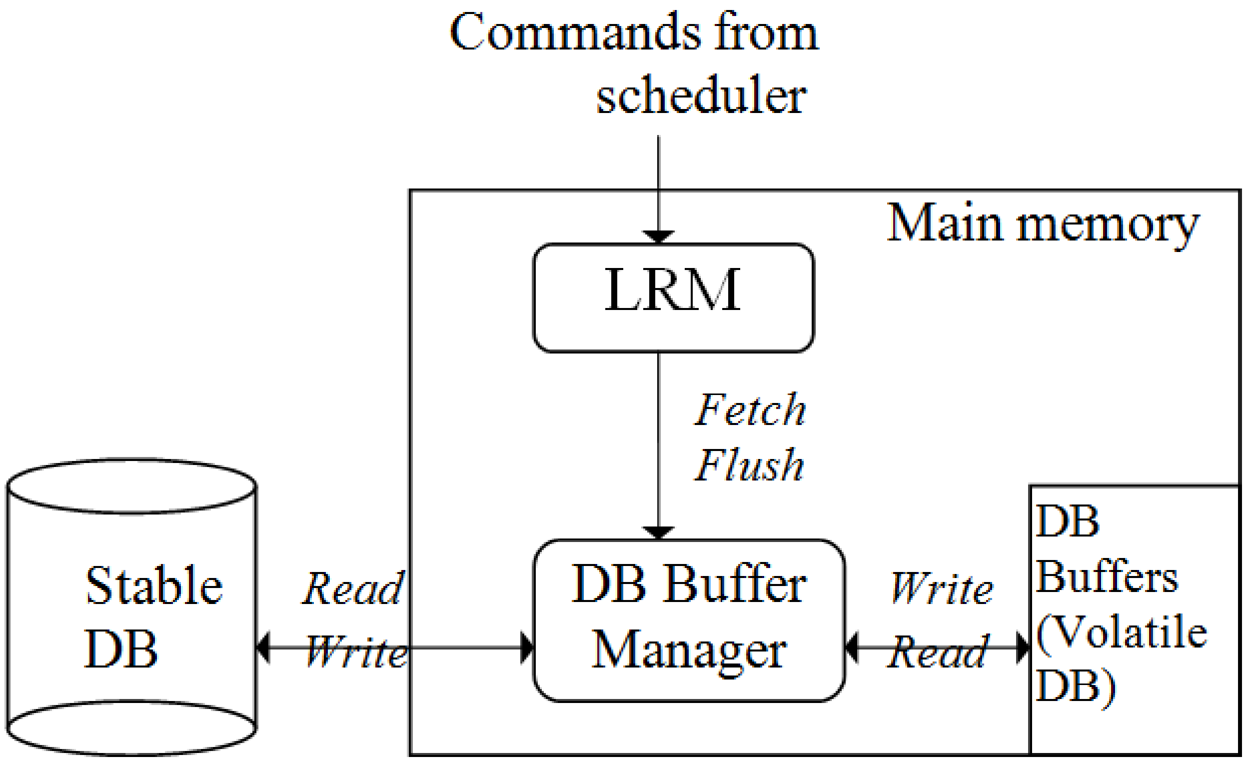


Local Reliability Protocols

- LRM (Local Recovery Manager) maintains the atomicity and durability properties of local transactions by performing some functions.

- Accepted commands
  - begin_transaction
  - read / write
  - commit / abort
  - recover

Architecture



Interfaces between LRM, Buffer and DB

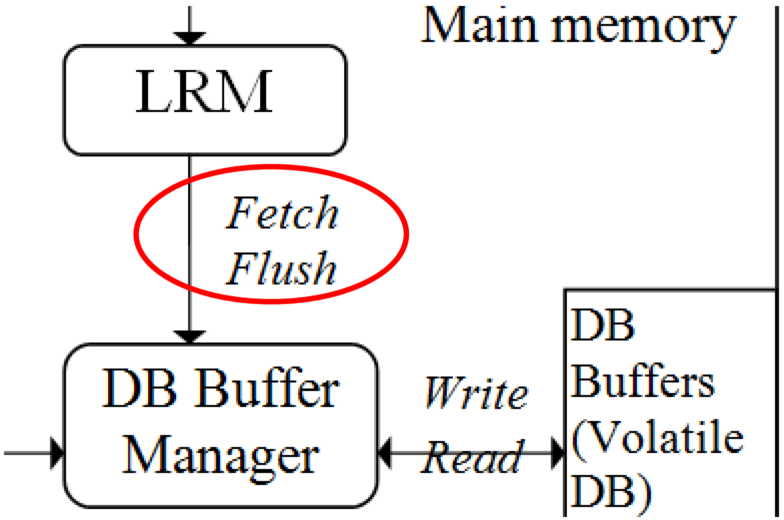
- LRM executes operations only on the volatile DB.
- Buffers are organized in pages

Volatile vs. Stable Storage

- Volatile storage
  - Consisting of the main memory of the computer system (RAM)
- Stable storage
  - Resilient to failures and losing its contents only in the presence of media failures (e.g., head crashes on disks)
  - Implemented via a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components.

Architectural Considerations

- Fetch - Get a page
  - if the page is in DB buffers, then the Buffer Manager returns it; otherwise the buffer Manager reads it from the Stable DB and puts it in buffers. Buffers full (?)



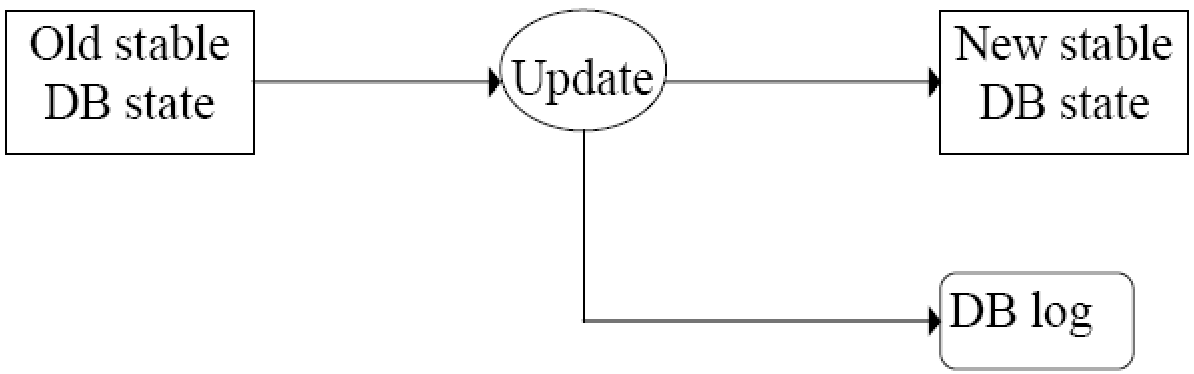
- Flush - Write pages
  - force pages to be written from buffers to the stable DB.

Recovery Information

- In-place update
  - Physically change the value of data items in stable DB. The previous values are lost.
- Out-of-place update
  - Do not change the value of data items in stable DB but maintain the new value separately.

In-Place Update Recovery Information

- Recovery information are kept in DB log. Each update not only changes DB, but also saves records in DB log.



Database Log

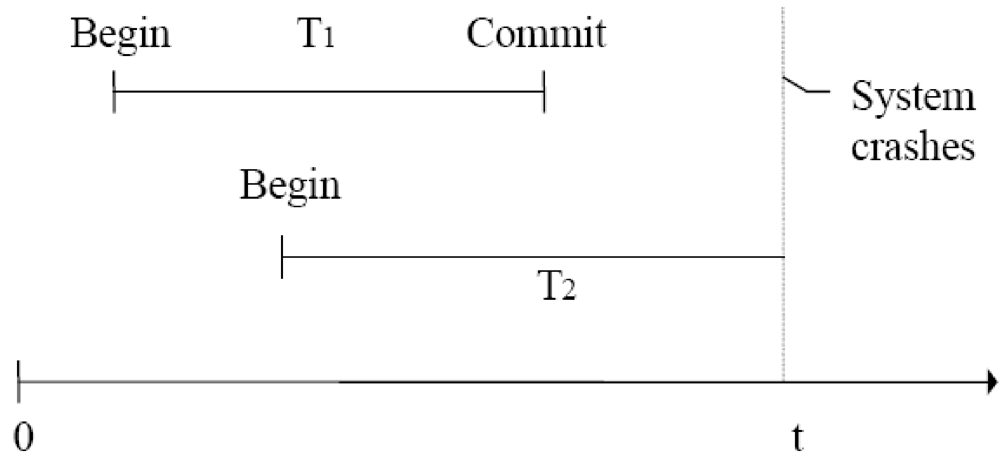
Every action of a transaction must not only perform the action, but also write a ***log*** record to an append-only file.

Logging

- The log contains information used by the recovery process to restore the consistency of a system.

- This information may include
  - transaction identifier
  - type of operation (action)
  - items accessed by the transaction to perform the action
  - old value (state) of item (before image)
  - new value (state) of item (after image), etc.

Why Logging?



- Assume buffer pages are written back to stable DB only when Buffer Manager needs new buffer space.
- T1: from user's viewpoint, it is committed. But updated buffer pages may get lost. Redo is needed.
- T2: not terminated, but some updated pages may have been written to stable DB. Undo is needed.

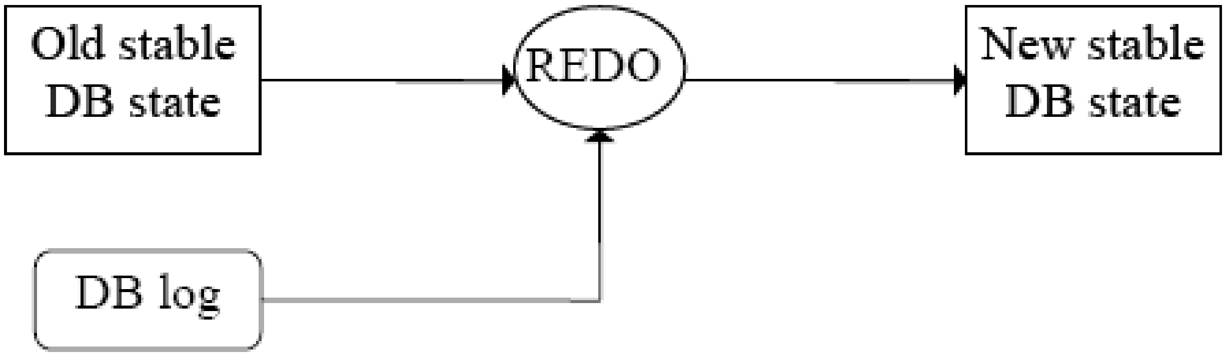
Failure Recovery

- If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (*undo portion* of the log).

- Once a transaction is committed, some of its actions might have to be redone. Need the after images (*redo portion* of the log).

REDO Protocol

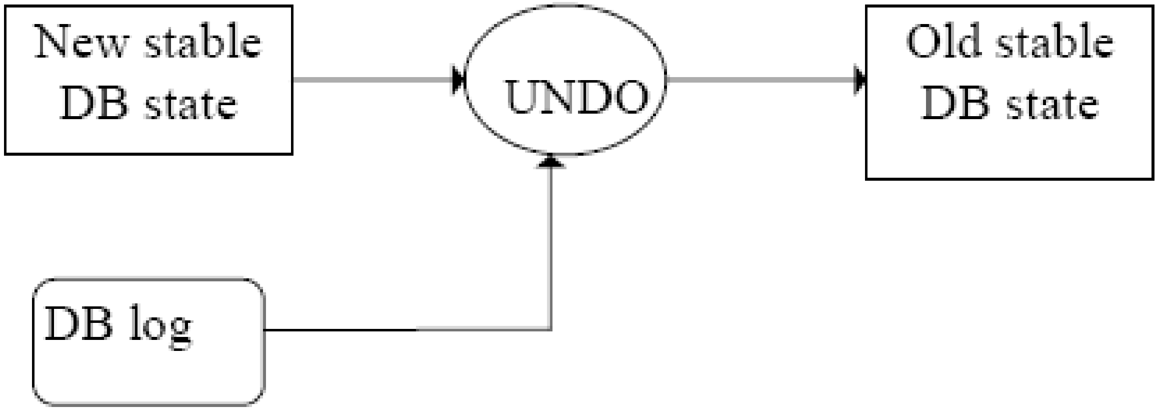
Redo T1



- REDO an action means performing it again.
- The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures.
- The REDO operation generates the new image.

UNDO Protocol

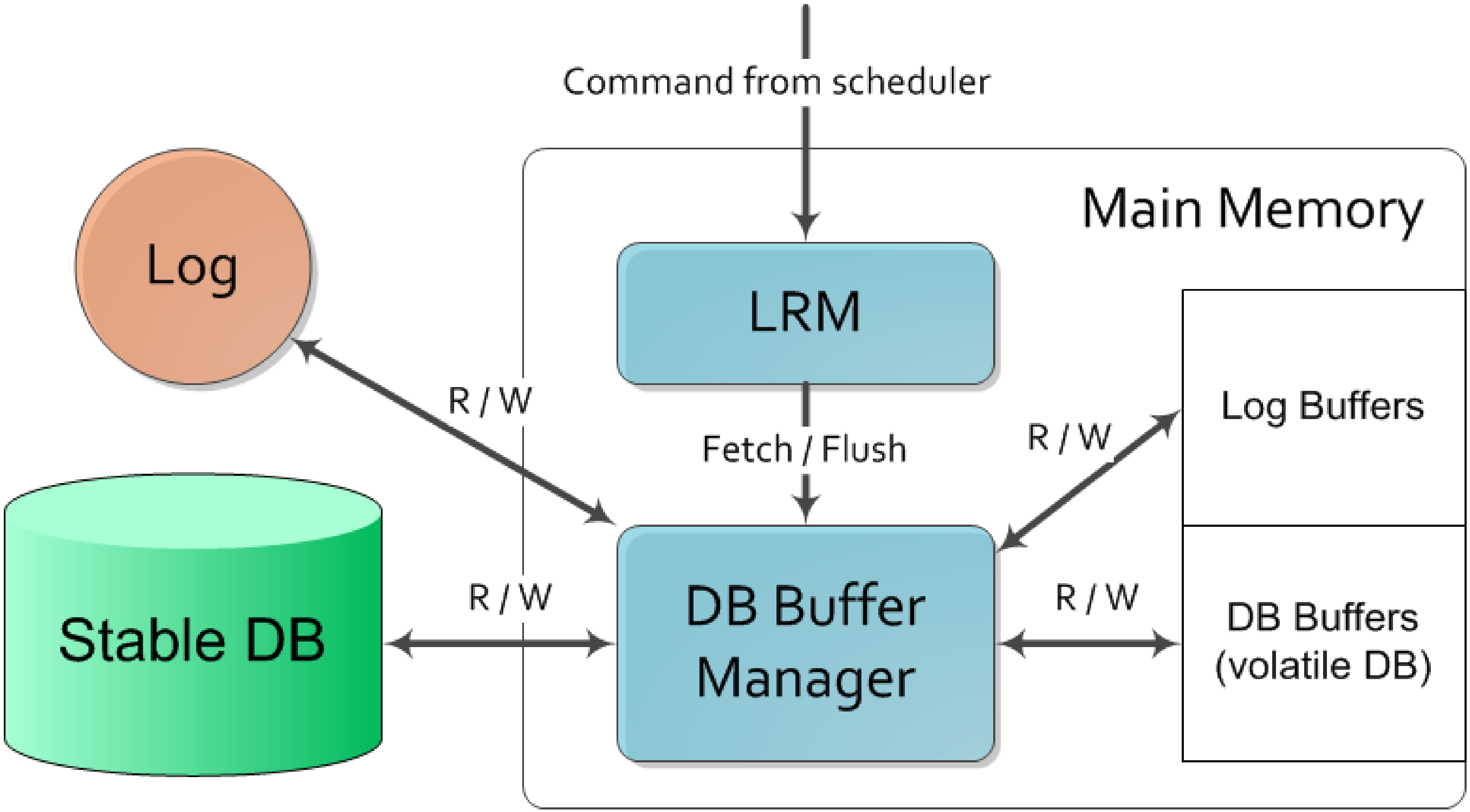
Undo T2



- UNDO an action means to restore the object to its before image.
- The UNDO operation uses the log information and restores the old value of the object.

Log File Maintenance

Interfaces between LRM, Buffer, Stable DB and Log file



When to Write Log Records Into Stable Store ?

Assume a transaction ***T*** updates a page *P*

- Fortunate case
  - System writes *P* in stable database
  - System updates stable log for this update
  - SYSTEM FAILURE OCCURS!... (before ***T*** commits)

We can recover (undo) by restoring *P* to its old state by using the log

When to Write Log Records Into Stable Store ? (*cont.*)

Assume a transaction ***T*** updates a page *P*

- Unfortunate case
  - System writes *P* in stable database
  - SYSTEM FAILURE OCCURS!... (before stable log is updated)

We cannot recover from this failure because there is no log record to restore the old value.

- Solution:
  - **Write-Ahead Log (WAL)** protocol

Write-Ahead Log (WAL) Protocol

- **Before** the stable DB is updated, the **before-image** should be stored in the stable log. This facilitates UNDO.

- When a transaction commits, the **after-images** have to be written in the stable log **prior to** the updating of the stable DB. This facilitates REDO.

Log File Maintenance

- Two ways to write log pages

1. Synchronous (forcing a log) – adding of each log record requires that the log be moved from main memory to the stable storage.

It's relatively easy to recover to a consistent state, but causes delay to the response time.

2. Asynchronous – the log is moved to stable storage either periodically or when the buffer fills up.

Out-of-Place Update Recovery Information - Shadowing

- When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database.
- Update the access paths so that subsequent accesses are to the new shadow page.
- The old page retained for recovery.

Out-of-Place Update Recovery Information - Differential files

- For each file F, maintain
  - a read only part FR
  - a differential file consisting of insertions part DF^+ and deletions part DF^-
  - Thus, $F = (FR \cup DF^+) - DF^-$

- Updates treated as delete old value, insert new value

LRM Commands

- begin_transaction
- read
- write
- abort
- commit
- recover

Independent of execution strategy for LRM

Command “begin_transaction”

- LRM writes a begin_transaction record in the log
  - This write may be delayed until first write command to reduce I/O.

Command “read(a data item)”

- LRM tries to read the data item from the buffer.
  If the data is not in the buffer, LRM issues a fetch command.

- LRM returns the data to scheduler.

Command “write(a data item)”

- If the data is in buffer, then update it; otherwise issue a fetch command to bring the data to the buffer first and then update it.
- Record before-image and after-image in the log.
- Inform the scheduler the write has been completed.

Execution Strategies for "commit, abort, recover" Commands

- Dependent upon
  - Whether the buffer manager may write the buffer pages updated by a transaction into stable storage during the execution of that transaction, or it waits for the LRM to instruct it to write them back?
    - no-fix / fix (not writing stable DB, only LRM makes a mark on updated pages)
  - Whether the buffer manager will be forced to flush the buffer pages updated by a transaction into stable storage at the end (commit point) of that transaction, or the buffer manager flushes them out whenever it needs to according to its buffer management algorithm?
    - no-flush / flush (explicitly commanded by LRM)

Possible Execution Strategies for "commit, abort, recover" Commands

- Four strategies

  1) no-fix / no-flush

  2) no-fix / flush

  3) fix / no-flush

  4) fix / flush

1) No-Fix / No-Flush

Updated data may/may not be written to stable storage before commit.

- “Abort” command
  - Buffer manager may have written some of the updated pages into the stable database.
  - LRM performs transaction undo (or partial undo)
- “Commit” command
  - LRM writes an “end_of_transaction” record into the log
- “Recover” command
  - For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, a partial redo is initiated by LRM
  - For those transactions that only have a “begin_transaction” in the log, a global undo is executed by LRM

2) No-Fix / Flush

Updated data may/may not be written to stable storage before commit.

- “Abort” command
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- “Commit” command
  - LRM issues a flush command to the buffer manager for all updated pages, which writes all updated pages into stable database
  - LRM writes an “end_of_transaction” record into the log
- “Recover” command
  - For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, no need to perform redo (since already flushed as instructed by LRM)
  - For those transactions that only have a “begin_transaction” in the log, a global undo is executed by LRM

3) Fix / No-Flush

Updated data were fixed and not written into stable storage before commit.

- **"Abort" command**
  - None of the updated pages have been written into stable database
  - Release (unfix) the fixed pages
- **"Commit" command**
  - LRM writes an "end_of_transaction" record into the log
  - LRM sends an unfix command to the buffer manager for all pages that were previously fixed
- **"Recover" command**
  - For those transactions that have both a "begin_transaction" and an "end_of_transaction" record in the log, perform partial redo
  - For those transactions that only have a "begin_transaction" in the log, no need to perform global undo

4) Fix / Flush

Updated data were fixed and not written into stable storage before commit.

- "Abort" command
  - None of the updated pages have been written into stable database
  - Release (unfix) the fixed pages
- "Commit" command (the following have to be done atomically)
  - LRM issues a flush command to the buffer manager for all updated pages
  - LRM sends an unfix command to the buffer manager for all pages that were previously fixed
  - LRM writes an "end_of_transaction" record into the log
- "Recover" command
  - For those transactions that have both a "begin_transaction" and an "end_of_transaction" record in the log, no need to perform partial redo.
  - For those transactions that only have a "begin_transaction" in the log, no need to perform global undo

Checkpointing

- Simplify the task of determining actions of transactions that need to be undone or redone when a failure occurs.
- Avoid the search of the **entire** log when recovery process is required.
  - The overhead can be reduced if it is possible to build a *wall* which signifies that the database at that point is up-to-date and consistent.
- The process of building the "wall" is called checkpointing.

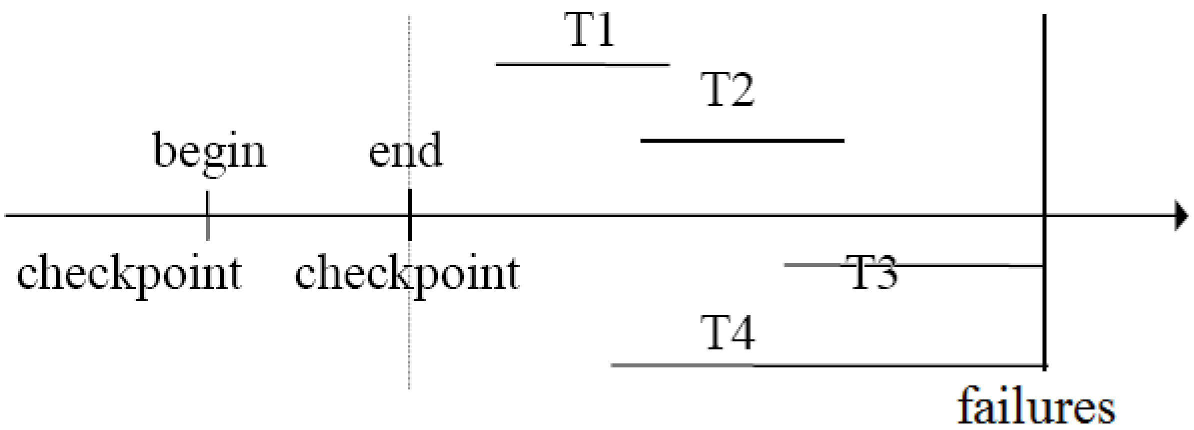
A Transaction-Consistent Checkpointing Implementation

1) Write the begin-checkpoint record in the log and stop accepting new transactions

2) Complete all active transactions and flush all updated pages to the stable DB

3) Write an end-of-checkpoint record in the log

Recovery based on Checkpointing



- Redo by starting from the earliest end-of-checkpoint. The sequence is T1, T2. Stop at the end of log.
- Undo by starting from the latest end-of-log. The sequence is T3, T4 (reverse order).

Coordinator vs. Participant Processes

- At the originating site of a transaction, there is a process that executes its operations. This process is called coordinator process.

- The coordinator communicates with participant processes at the other sites which assist in the execution of the transaction's operations.

8. Distributed DBMS Reliability

- Concept of Reliability
- Local Reliability Protocols
- ☞ Distributed Reliability Protocols
- Brewer's CAP Theorem and Relevant Efforts

Distributed Reliability Protocols

- The protocols address the distributed execution of the following commands
  - begin-transaction
  - read
  - write
  - abort
  - commit
  - recover

Distributed Reliability Protocols (*cont*.)

- “begin-transaction”

  (the same as the centralized case at the originating site)

  - execute bookkeep function
  - write a begin_transaction record in the log

- “read” and “write” are executed according to ROWA (Read One Write All) rule.

- Abort, commit, and recover are specific in the distribution case.

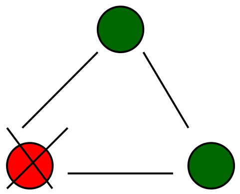
Three Components of Distributed Reliability Protocols

1) Commit protocols (different from centralized DB)

- How to execute commit command when more than one site are involved?
- Issue: how to ensure atomicity and durability?

2) Termination protocols

- If a failure occurs, how can the remaining operational sites deal with it?
- **Nonblocking**: the occurrence of failures should not force the sites to wait until the failure is repaired to terminate the transactions.

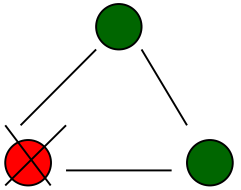


Three Components of Distributed Reliability Protocols (*cont.*)

3) Recover protocols (opposite to the termination protocols)

- When a failure occurs, how does the site where the failure occurred recover its state once the site is restarted?

- **Independent** : a failed site can determine the outcome of a transaction without having to obtain remote information.



Independent recovery ➔ nonblocking termination

Two-Phase Commit Protocol

- Global Commit Rule
  - The coordinator aborts a transaction if and only if at least one participant votes to abort it.
  - The coordinator commits a transaction if and only if all of the participants vote to commit it.

- 2PC ensures the atomic commitment of a distributed transaction.

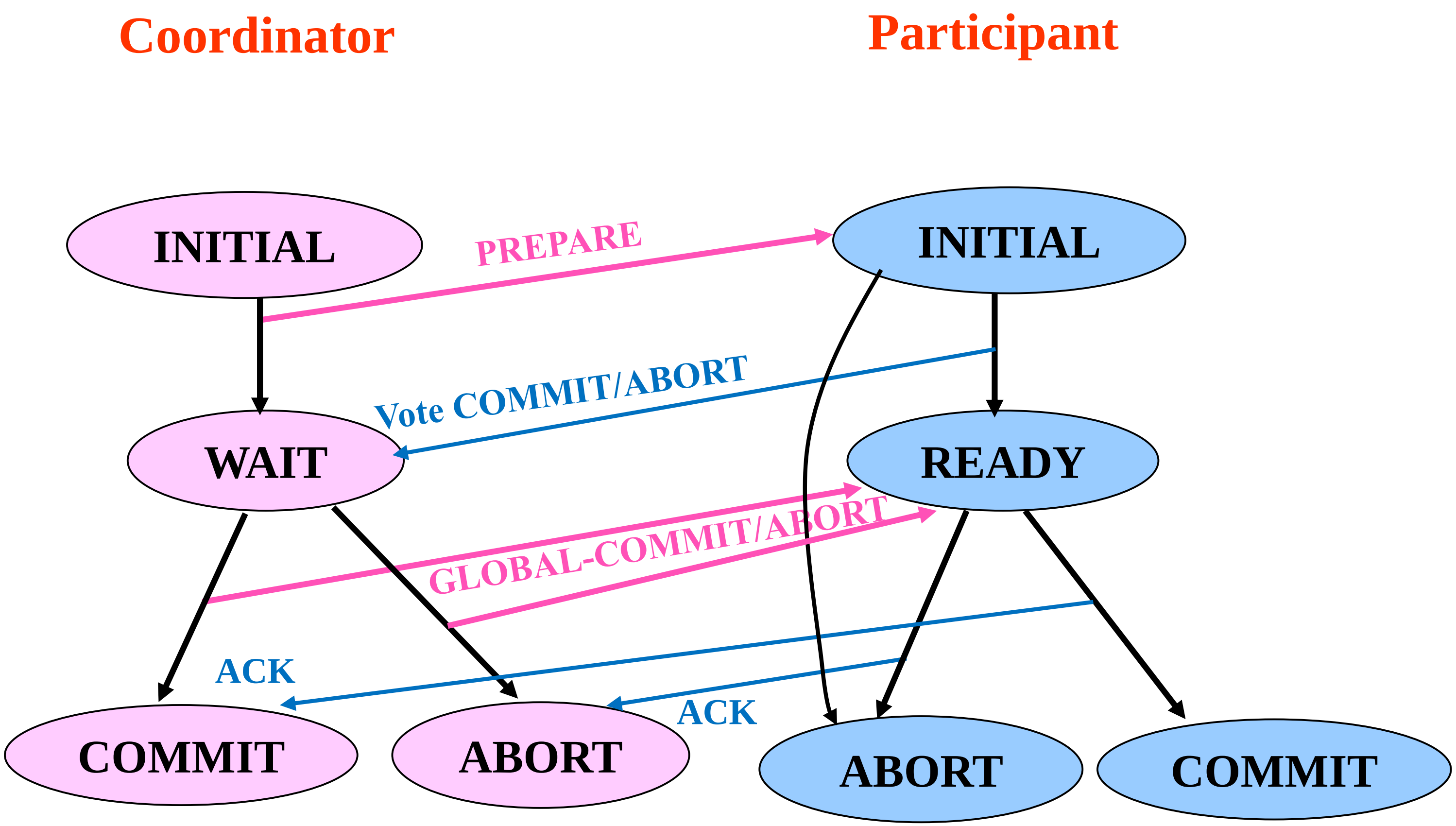
Phase 1

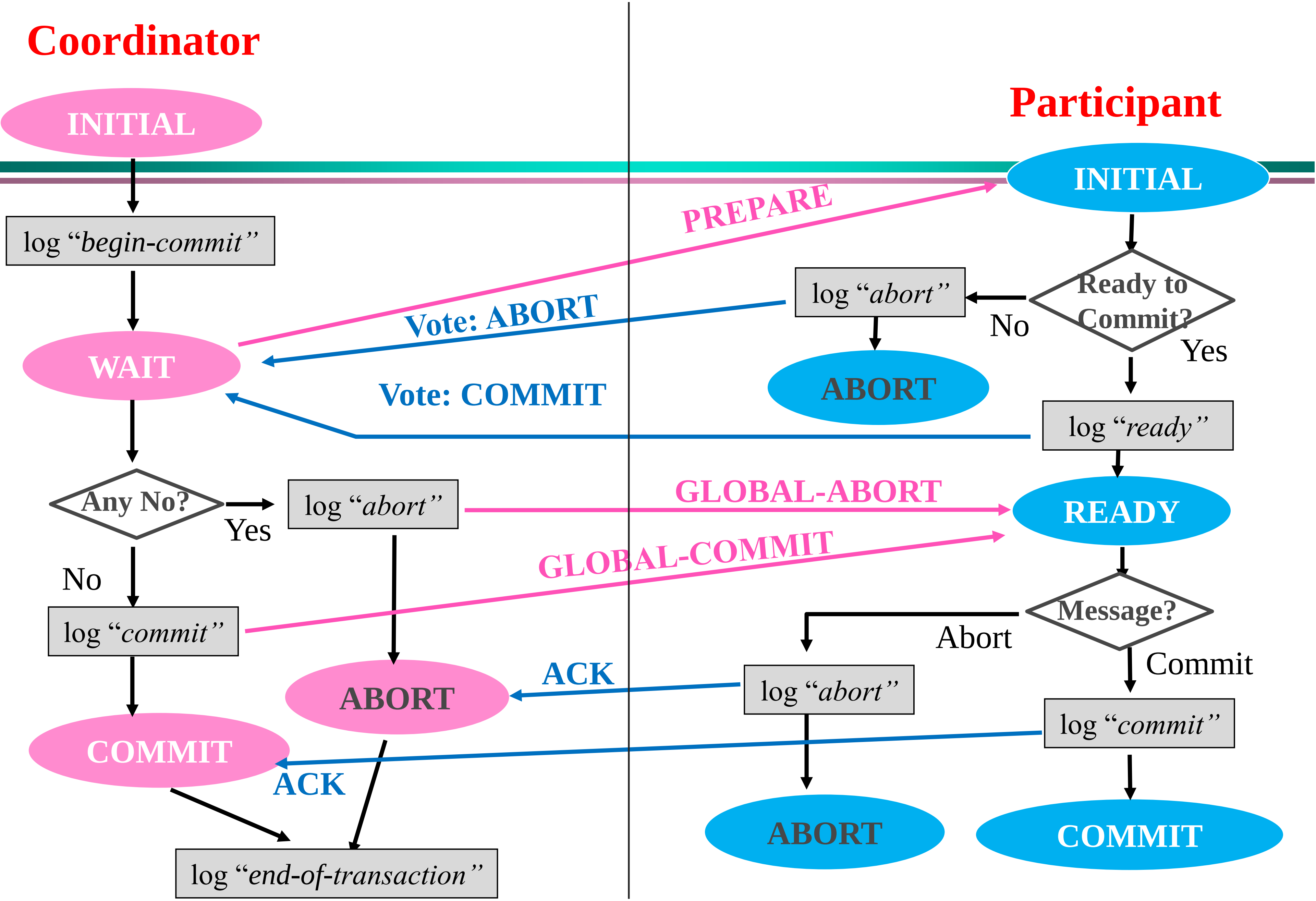
- The coordinator gets the participants ready to write the results into the database
  - The coordinator sends a message to all participants, asking if they are ready to commit, and
  - every participant answers “yes” if it's ready or “no” according to its own condition.

Phase 2

- Everybody writes the results into the database
  - The coordinator makes the final decision - global commit if all participants answer “yes” in phase 1; or global abort, otherwise.
  - It then informs all the participants its final decision.
  - All participants take actions accordingly.

A Simplified Version of 2PC

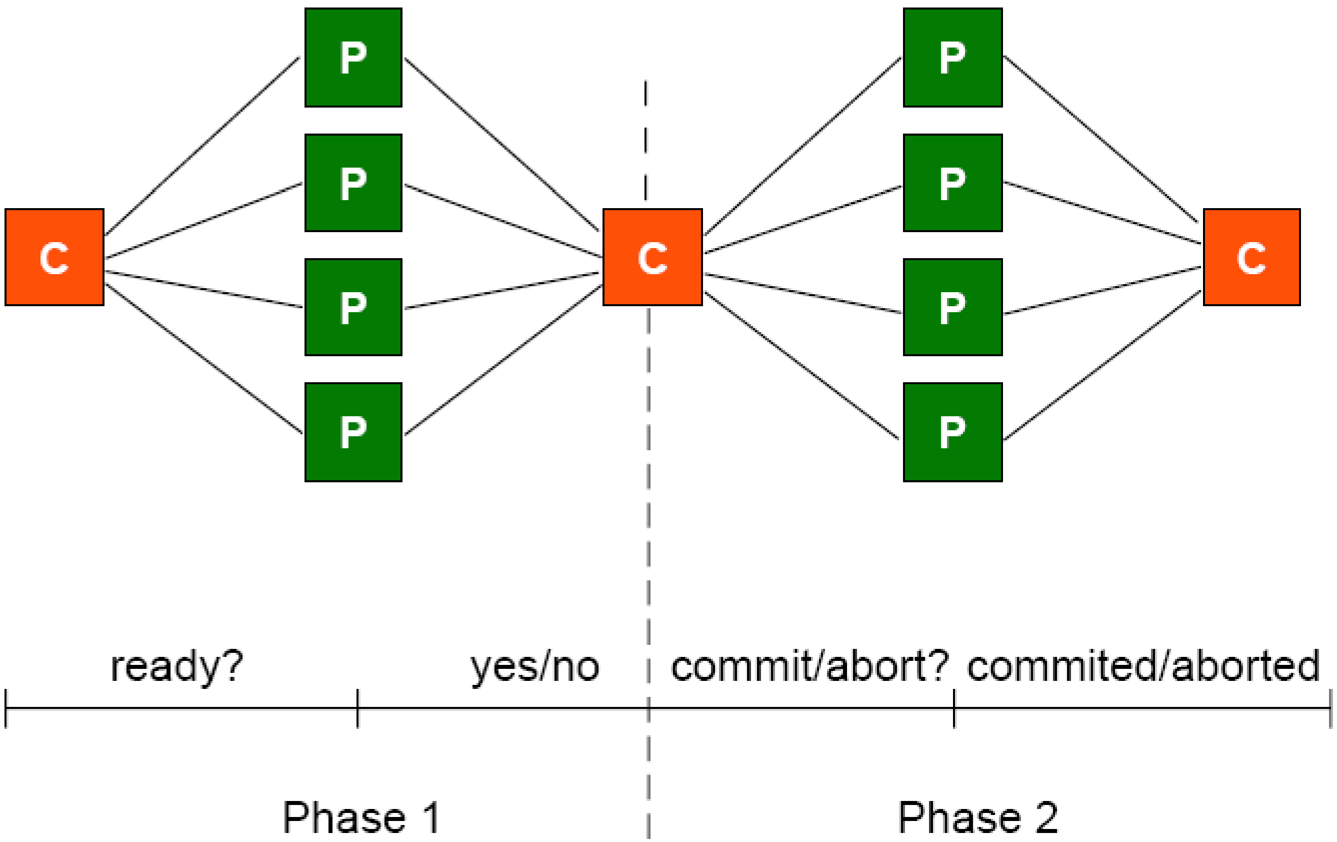



Coordinator
Participant
INITIAL
log "begin-commit"
WAIT
Any No?
Yes
No
log "abort"
log "commit"
ABORT
COMMIT
log "end-of-transaction"
PREPARE
Vote: ABORT
Vote: COMMIT
GLOBAL-ABORT
GLOBAL-COMMIT
ACK
ACK
INITIAL
Ready to Commit?
No
Yes
log "abort"
ABORT
log "ready"
READY
Message?
Abort
Commit
log "abort"
log "commit"
ABORT
COMMIT

Observations

1. A participant can unilaterally abort before giving an affirmative vote.
2. Once a participant answers "yes", it must prepare for commit and cannot change its vote.
3. While a participant is READY, it can either abort or commit, depending on the decision from the coordinator.
4. The global termination is commit if all participants vote "yes", or abort if any participant votes "no”.
5. The coordinator and participants may be in some waiting state, time-out method can be used to exit.

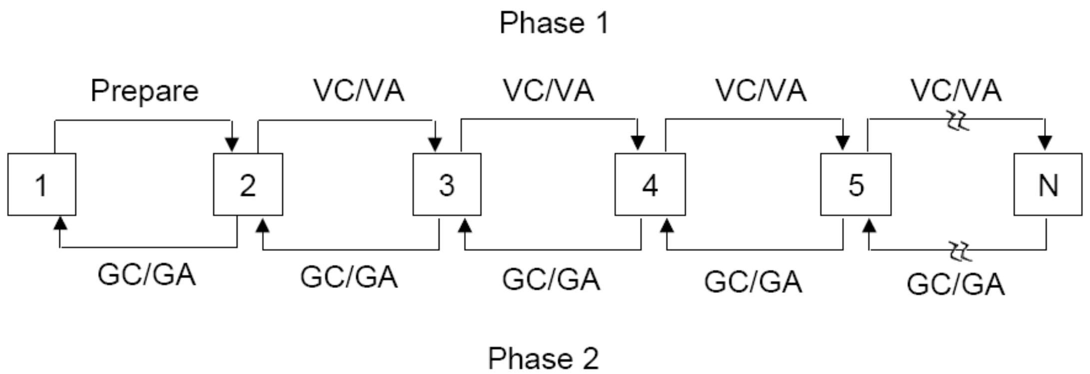
Centralized 2PC



no communication between participants

Linear 2PC

- Participants communicate with one another.

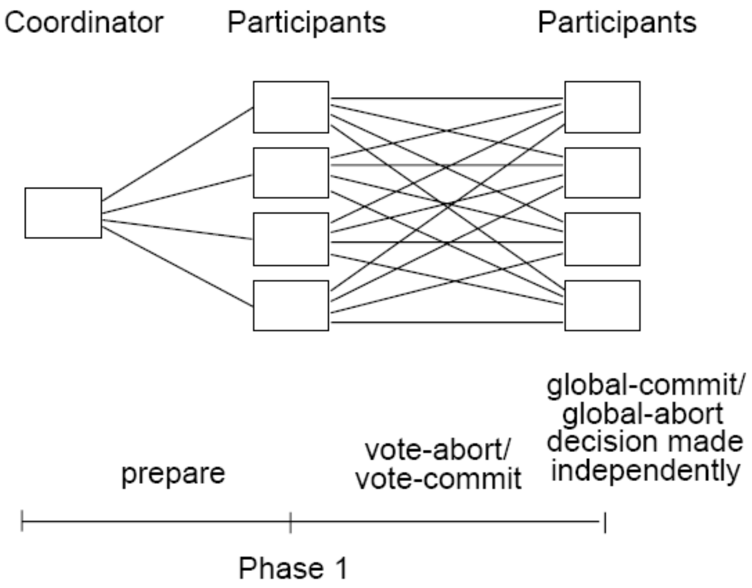


VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

- N participants are ordered from 1 (the coordinator) to N.
- Communications during the first phase is in forward fashion from 1 to N and in backward fashion during the second phase.
- Fewer messages but no parallelism

Distributed 2PC

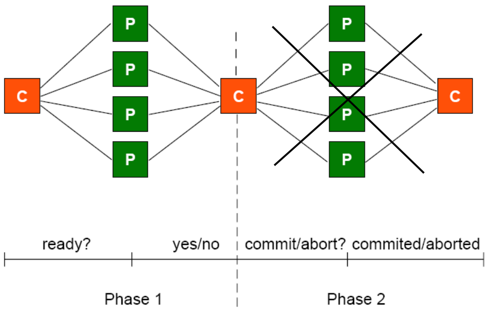
- Each participant broadcasts its vote to all participants.
- No need for the second phase (no ACK message is needed).
- Each participant needs to know all other participants.



Variants of 2PC

- Shortcomings of 2PC
  - Number of messages is big
  - Number of log-writing times is big

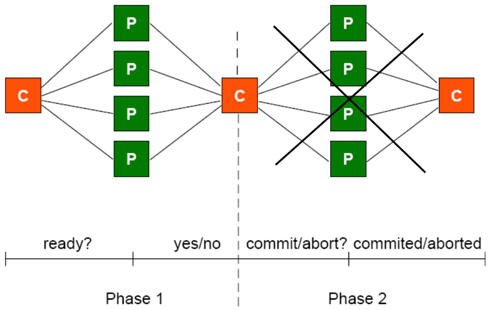
- Two variants of 2PC are proposed to improve performance
  - presumed abort 2PC
  - presumed commit 2PC



Presumed Abort 2PC Protocol

- Assumption
  - When a failed site recovers, the recovery routine will check the log and determine the transaction's outcome.
  - Whenever there is no information about the transaction's outcome ("commit" or "abort"), the outcome is abort.

- Commits have to be acknowledged, while aborts do not.

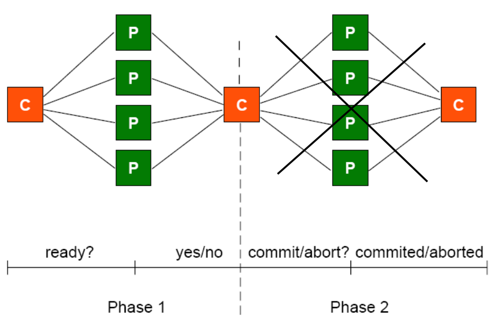


Presumed Abort 2PC Protocol (*cont.*)

- In case of "**abort**" transactions
  - The coordinator can forget abort the transaction immediately after it decides to abort it.
    - It writes an abort record directly in the log and not expect the participants to acknowledge the abort command.
    - It does not need to write an end-of-transaction in the log after an abort record.
    - It does not have to force the abort record to stable storage.
  - The participants also do not need to force the abort record

    → **Presumed Abort**

It saves some message transmission between the coordinator and the participants in case of aborted transactions, and is thus more efficient.



Presumed Abort 2PC Protocol (*cont.*)

- In case of "**commit**" transactions
  - The same as regular 2PC
  - Commits have to be acknowledged (while aborts do not)

- When a site fails before receiving the decision and recovers later, it can
  - find the "commit" and "end_transaction" in the log of the coordinator, or
  - find or may not find the "abort" record in the log of the coordinator

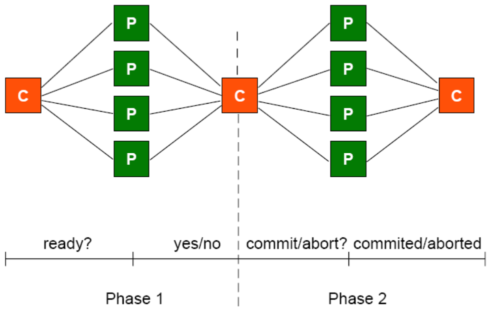
  and take the corresponding action.

- More efficient for "abort" transactions
  - Save some message transmission between the coordinator and the participants

Presumed Commit 2PC Protocol

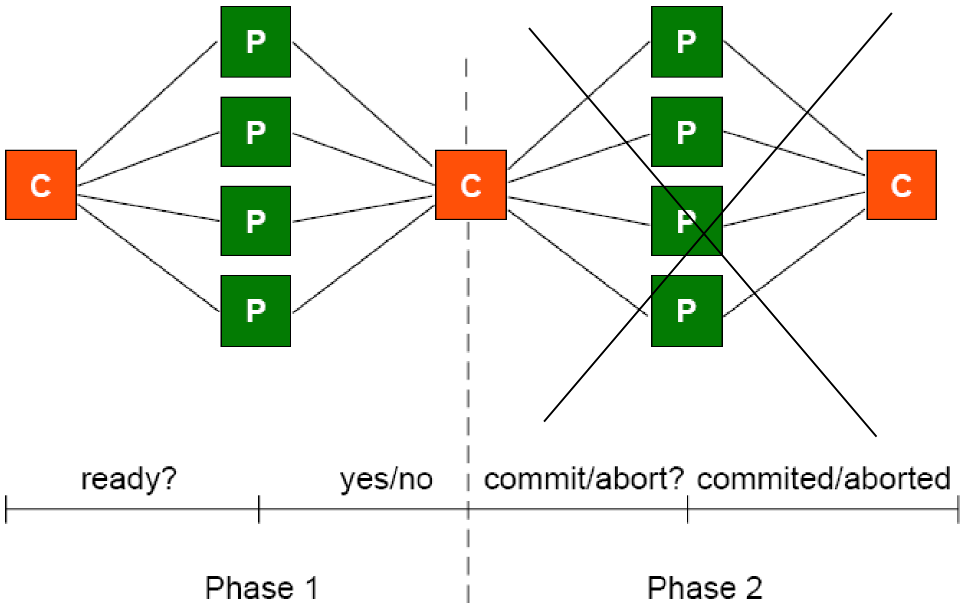
- Assumption
  - When a failed site recovers, the recovery routine will check the log and determine the transaction's outcome.
  - No information available to the recovery process from the coordinator is equivalent to a "commit".

- Aborts have to be acknowledged, while commits do not.



Presumed Commit 2PC Protocol (*cont.*)

- An exact dual of Presumed Abort 2PC will look like:
  - The coordinator forgets about the transaction after it decides to commit.
  - The commit record of the coordinator (also the ready record of the participants) needs not be forced.
  - The commit command needs not be acknowledged.



Presumed Commit 2PC Protocol (*cont.*)

- However, it does not work correctly in the following case.
  - The coordinator fails after sending the prepare message for vote-collection, but before collecting all votes from the participants.
  - In recovery process:
    - The coordinator will undo the transaction since no global agreement had been achieved. But all participants will commit by assumption.

      ➔ **causing inconsistency**

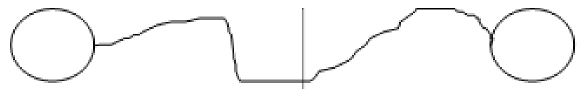
Presumed Commit 2PC Protocol (*cont.*)

- Correction to overcome the above case
  - The coordinator, prior to sending the "prepare" message, **force-writes a "collecting" record** containing the names of all participants in the log.
  - The participants then enter "COLLECTING" state.
  - The coordinator then sends the "prepare" message and enters the WAIT state.

Presumed Commit 2PC Protocol (*cont.*)

- The coordinator decides “global abort” or “global commit”
  - If “abort”, the coordinator writes an abort record, enters the ABORT state, and sends a “global-abort” message.
  - If “commit”, the coordinator writes a commit record, sends a “global-commit” command, and **forgets the transaction**.
- When the participants receive a
  - “global-abort” message, they write an abort record and **acknowledge**.
  - “global-commit” message, they write a commit record and update the DB.

Independent and Non-blocking



Failed site

Recovery protocol required

Independent recovery wanted!

Operational site

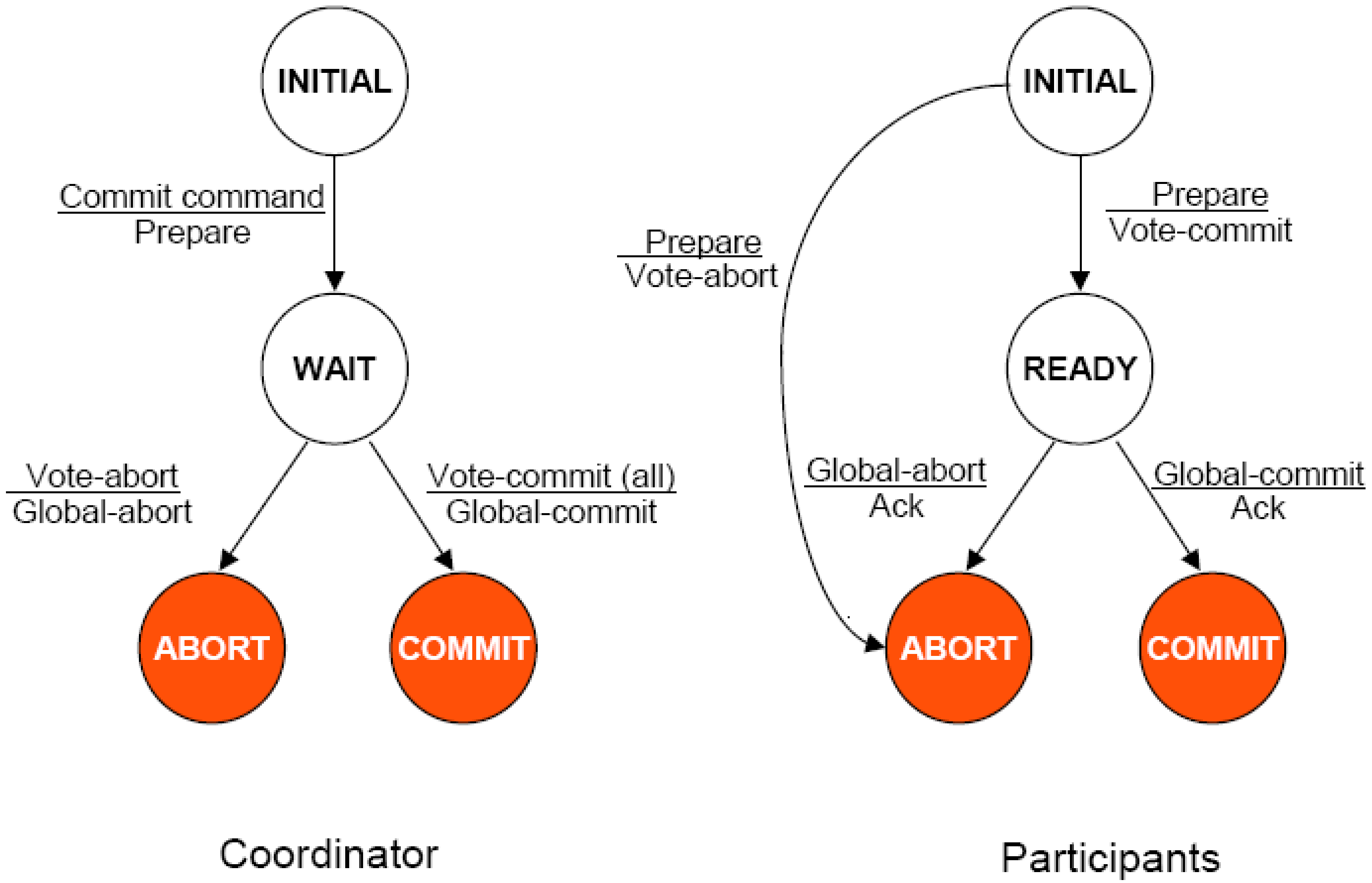
Termination protocol required

Non-blocking wanted!

Independent recovery and non-blocking protocols exist only for single-site failure, and not possible when multiple sites fail.

2PC is inherently blocking !

State Transition in 2PC Protocol



Labels on the edge

Top: the reason for the state transition (a received message)

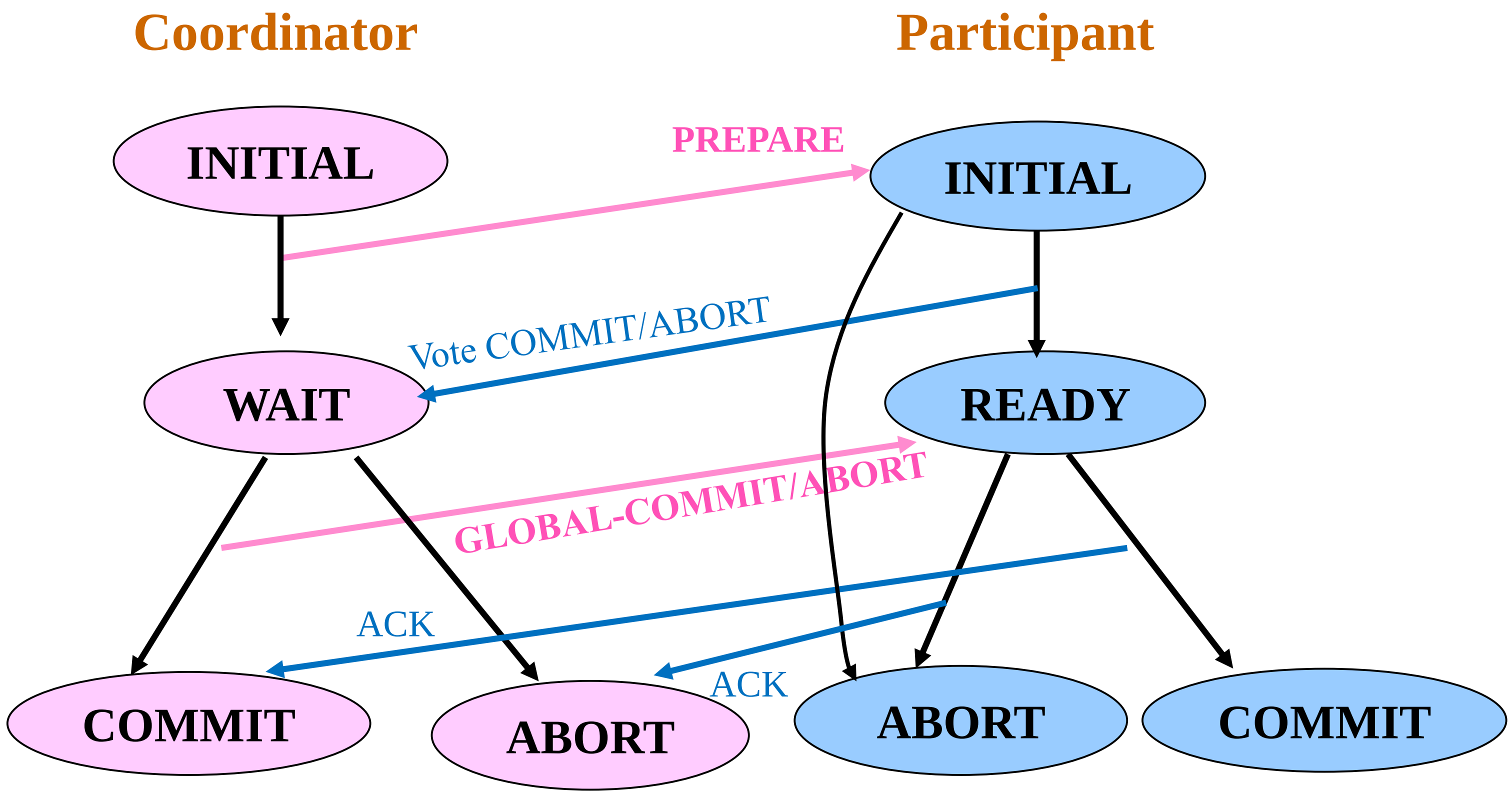
Bottom: the message sent as a result of the state transition

Termination

- A timeout occurs at a destination site when it cannot get an expected message from a source site within the expected time period.

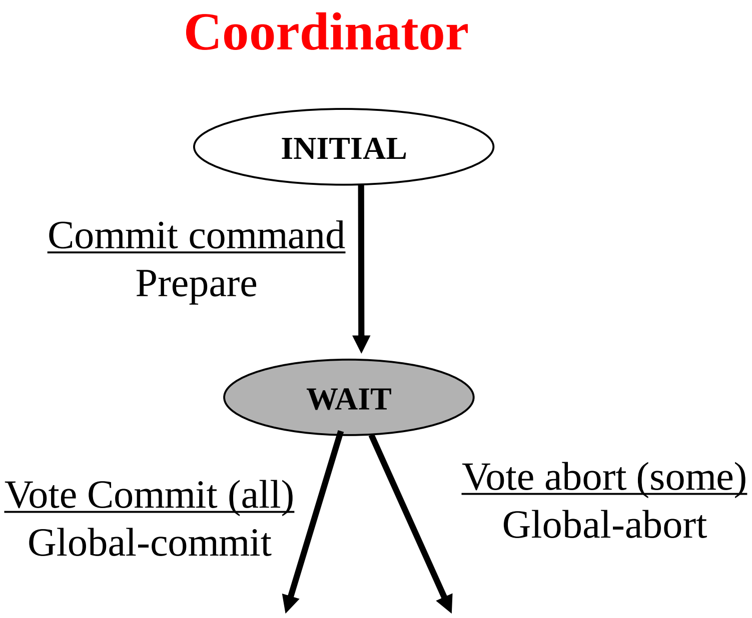
Coordinator Timeouts

- The coordinator can time-out in WAIT, ABORT, and COMMIT states.



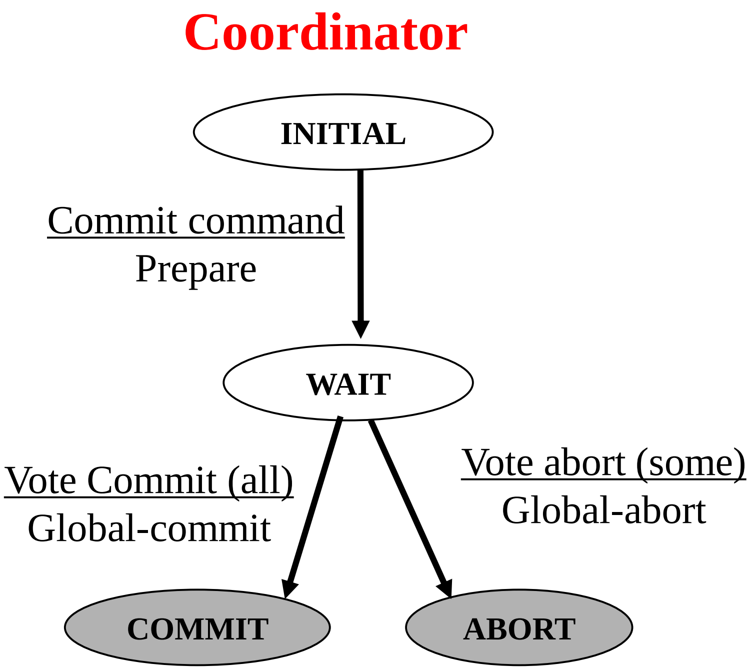
Coordinator Timeouts (*cont.*)

- "WAIT"
  - The coordinator is waiting for the local decisions from the participants.
  - Solution: the coordinator decides to globally abort the transaction by writing an abort record in the log, and sending a global abort to all participants.



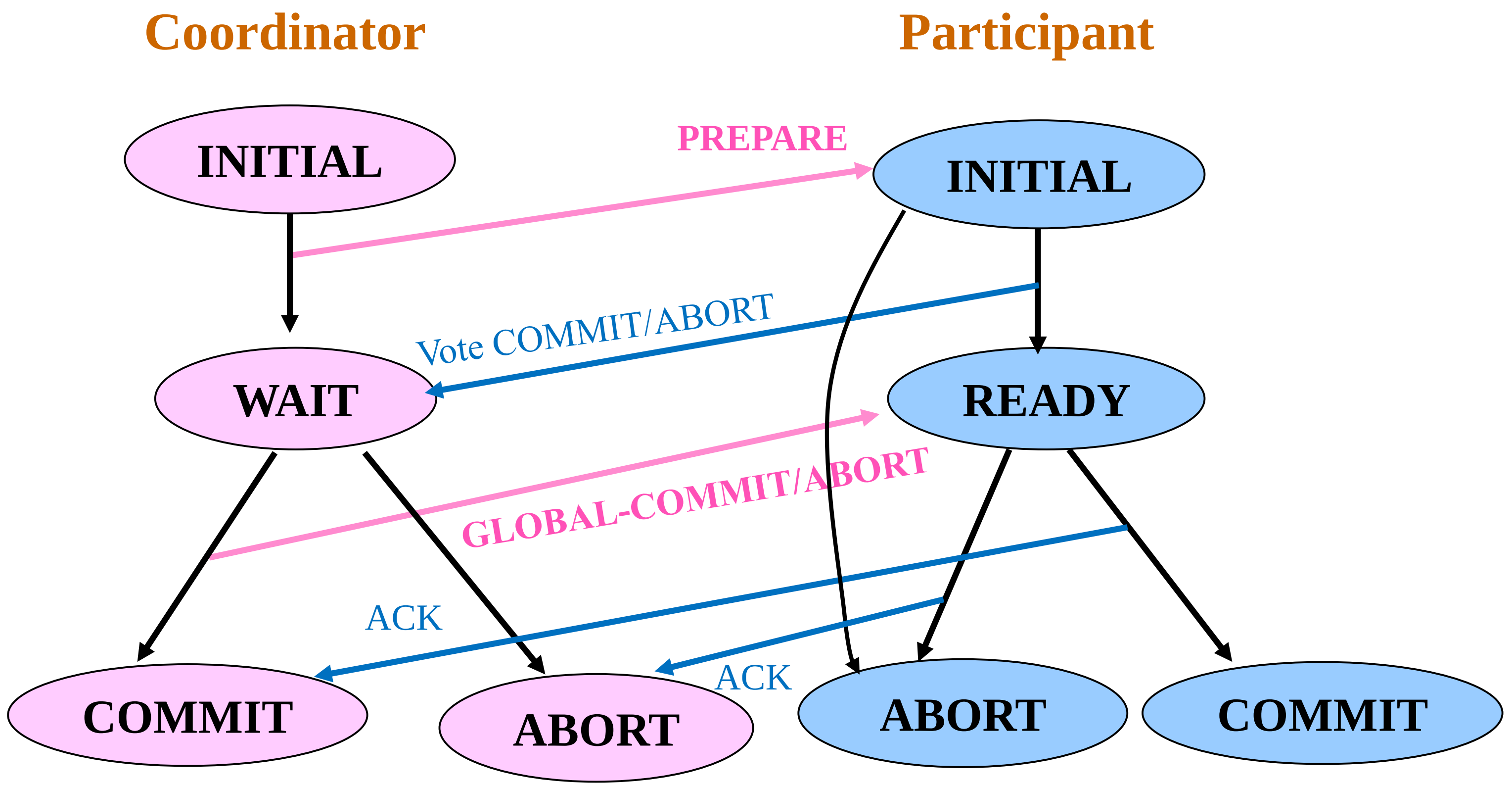
Coordinator Timeouts (*cont.*)

- "COMMIT" or "ABORT"
  - The coordinator is not certain if the commit or abort procedures have been completed by all the participants.
  - Solution: re-send the "global-commit" or "global abort" to the site that have not acknowledged.



Participant Timeouts

- A participant can time-out in INITIAL or READY states.



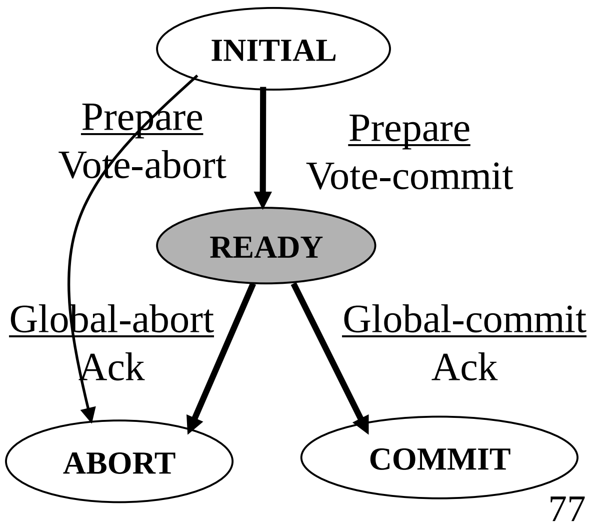
Participant Timeouts (*cont.*)

- "INITIAL"
  - The participant is waiting for a "prepare" message.
  - The coordinator must have failed in INITIAL state.
  - Solution: The participant unilaterally aborts the transaction. If the "prepare" message arrives later, it can be responded by
    - voting abort, or
    - just ignoring the message
      This causes the time-out of the coordinator in the WAIT state (abort and re-send global abort to participants)

Participant Timeouts (*cont.*)

- “READY”
  - The participant must have "voted commit" and therefore cannot change it and unilaterally abort it.
  - Solution: ***blocked*** until it can learn (from the coordinator or other participants) the ultimate fate of the transaction.

Participants



In centralized communication structure,

a participant has to ask the coordinator for its decision.

If the coordinator failed, the participant will remain blocked.

Can Blocking Problem be Overcomed?

- No!
- 2PC is an inherently blocking protocol.

Another Distributed Termination Protocol

- Assume participants can communicate with each other.

- Let *Pi* be the participant that timeouts in the READY state, and *Pj* be the participant to be asked.

All the Cases that *Pj* Can Respond

1. *Pj* is in the INITIAL state. This means *Pj* has not voted yet. *Pj* can unilaterally abort the transaction and reply to *Pi* with a “vote-abort” message.
2. *Pj* is in the READY state. *Pj* does not know the global decision and cannot help.
3. *Pj* is in COMMIT or ABORT state. *Pj* can send “global-commit” or “global-abort” to *Pi*.

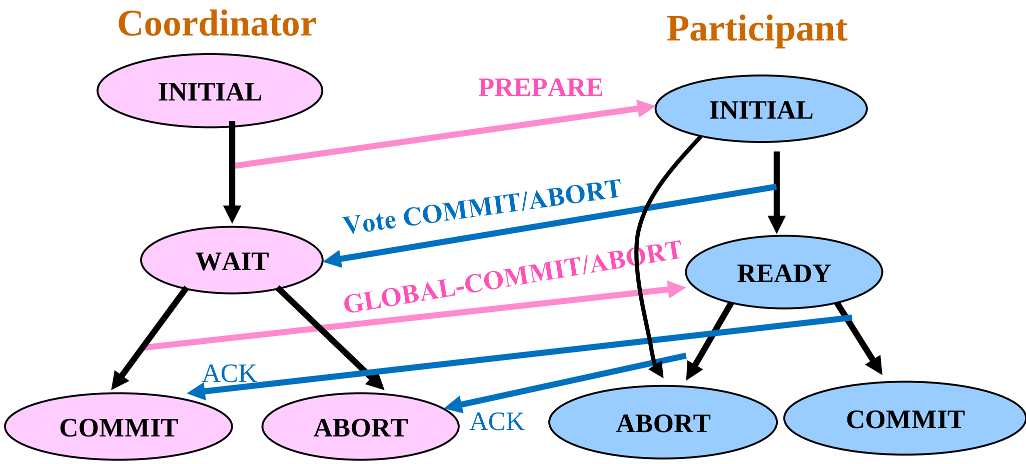
How *Pi* interprets these responses?

All the alternatives that the termination protocol needs to handle.

1. *Pi* receives “vote-abort” from all *Pj*s. *Pi* just proceeds to abort the transaction.
2. *Pi* receives "vote-abort" from some *Pj*, but some other participants are in READY state. *Pi* goes ahead and aborts the transaction.
3. *Pi* receives the information that all *Pjs* are READY. *Pi* **is *blocked***, since it has no knowledge about the global decision.
4. *Pi* receives either “global-abort” or “global-commit” messages from all *Pj*s. *Pi* can go ahead and terminate the transaction according to the message.
5. *Pi* receives either “global-abort” or “global-commit” messages from some *Pj*, but others are in READY. *Pi* takes action same as (4).

Recovery

- A failed coordinator or participant recovers when it restarts.
- Assuming
  1. Writing log and sending messages are in an atomic action;
  2. The state transition occurs after message sending.



Coordinator Site Failure

- The coordinator fails while in INITIAL state.
  - Action: restart the transaction.
- The coordinator fails while in WAIT state.
  - Action: restart the commit process by sending the “prepare” message once more.
- The coordinator fails while in COMMIT / ABORT state.
  - Action: If all ACK messages have been received, then no action is needed; otherwise follow the termination protocols (re-send “global-commit/abort” message to participant sites).

Participant Site Failure

- A participant fails while in INITIAL.
  - Action: Upon recovery, the participant should abort the transaction unilaterally.
- A participant fails while in READY.
  - Action: Same as time-out in the READY state and follow its termination protocols (ask for help).
- A participant fails while in ABORT/COMMIT.
  - Action: No action.

Problem with 2PC

- Blocking
  - "Ready" implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery
  - Blocking reduces availability

Problem with 2PC (*cont.*)

- Independent recovery is not possible
- It is known that
  - Independent recovery protocols exist only for single site failures;
  - No independent recovery protocol exists which is resilient to multiple-site failures.

- So we search for these protocols – 3PC

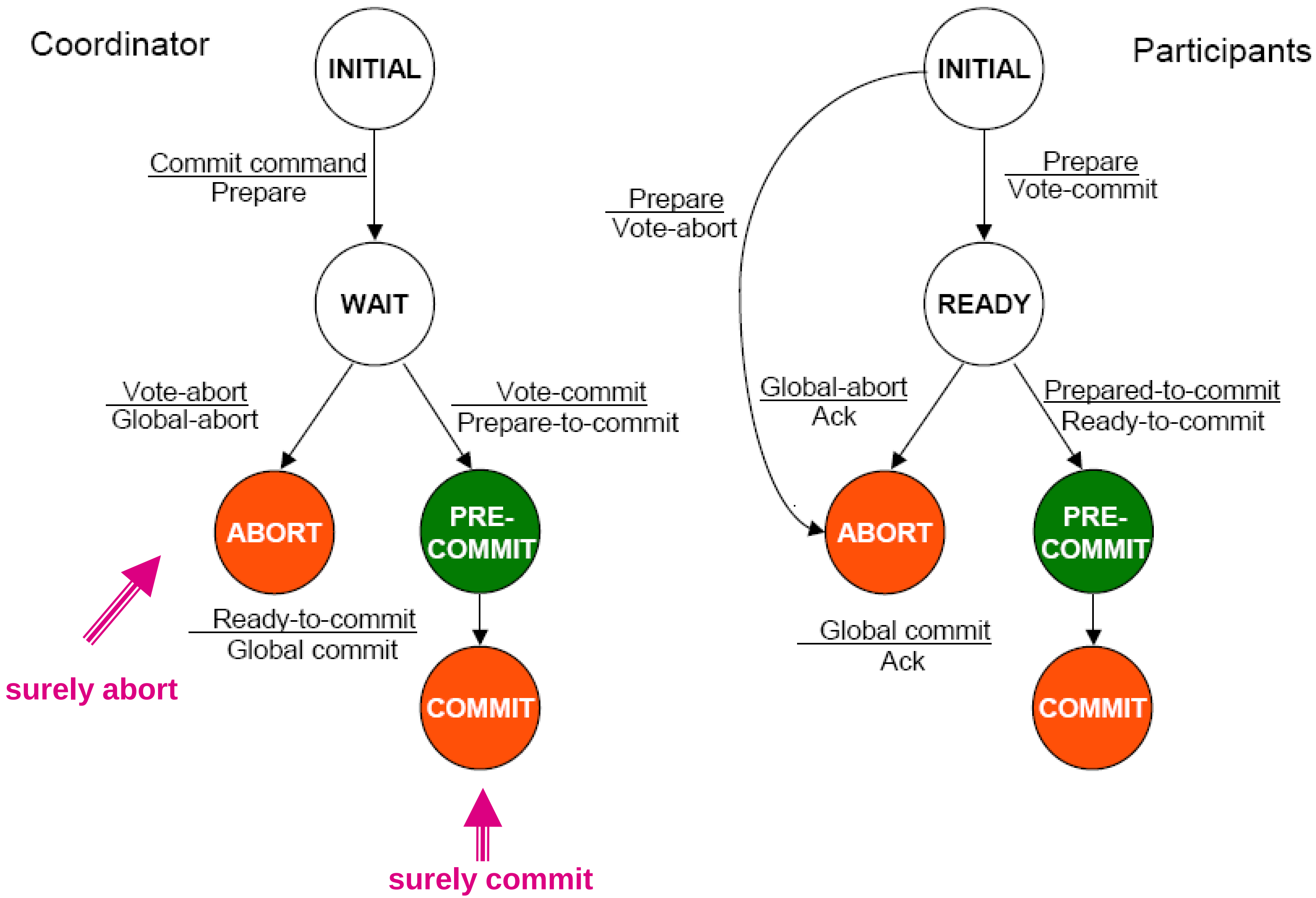
Three-Phase Commit (3PC)

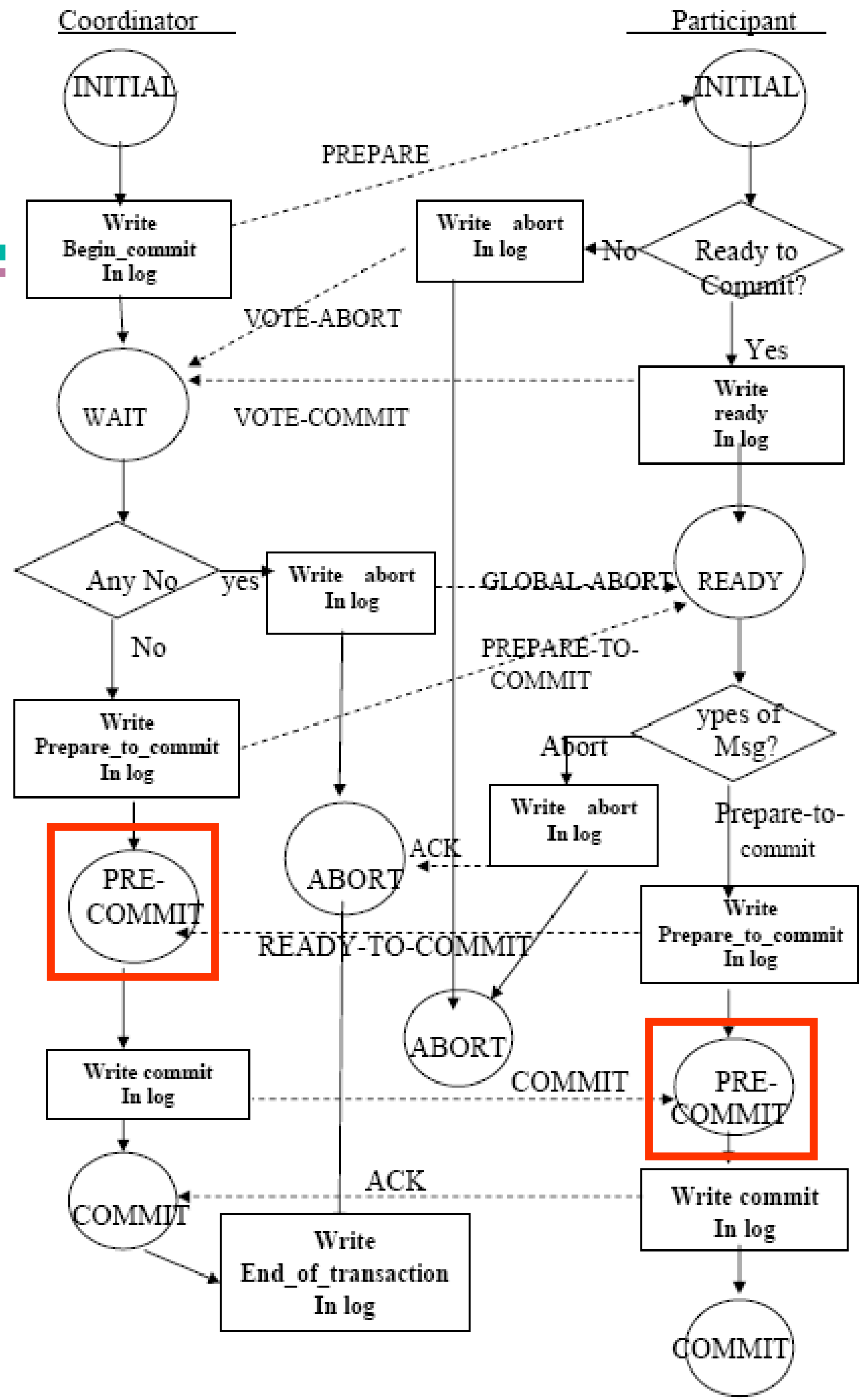
- 3PC is non-blocking
- A commit protocol is non-blocking iff
  - it is synchronous within one state transition, and
  - its state transition diagram contains no state which is "adjacent" to both a commit and an abort state, and
  - no non-committable state which is "adjacent" to a commit state
- "Adjacent" - possible to go from one state to another with a single state transition

3PC (*cont.*)

- Committable: all sites have voted to commit a transaction
  - COMMIT – commitable state
  - WAIT, READY – non-commitable state

Add a “Pre-Commit” State





Coordinator
Participant
INITIAL
INITIAL
PREPARE
Write Begin_commit In log
Write abort In log
No
Ready to Commit?
VOTE-ABORT
Yes
WAIT
VOTE-COMMIT
Write ready In log
Any No
yes
Write abort In log
GLOBAL-ABORT
READY
No
PREPARE-TO-COMMIT
Write Prepare_to_commit In log
ypes of Msg?
Abort
Write abort In log
Prepare-to-commit
PRE-COMMIT
ABORT
ACK
READY-TO-COMMIT
Write Prepare_to_commit In log
ABORT
Write commit In log
COMMIT
PRE-COMMIT
ACK
COMMIT
Write commit In log
Write End_of_transaction In log
COMMIT

3PC Termination Protocol

- Coordinator timeouts
  1. In the WAIT state
     - Same as in 2PC (The coordinator unilaterally aborts the transaction and send a “global abort” message to all participants).
  2. In the PRE-COMMIT state
     - All participants must at least be in READY state (have voted to commit).
     - The coordinator globally commits the transaction and sends “pre-commit” message to all operational participants.
  3. In the COMMIT (or ABORT) state
     - Just ignore and treat the transaction as completed
     - Participants are either in PRE-COMMIT or READY state and can follow their termination protocols.

3PC Termination Protocol (*cont.*)

- Participants timeout

1. In the INTIAL state
   - Same as 2PC (coordinator must have failed, and thus unilaterally aborts the transaction).
2. In the READY state
   - Have voted to commit, but does not know the coordinator's global decision.
   - Elect a new coordinator and terminate using a special protocol (*to be discussed below*).
3. In the PRE-COMMIT state
   - Wait for the "global-commit" message from the coordinator.
   - Handle it the same as timeout in READY state (above).

3PC Termination Protocol Upon Coordinator Election

- The new elected coordinator can be in WAIT (READY), PRE-COMMIT, COMMIT, or ABORT state.

3PC Termination Protocol Upon Coordinator Election (*cont*.)

- The new coordinator then guides the participants towards termination
  - If the new coordinator is in WAIT (READY) state
    - Participants can be in INITIAL, READY, PRE-COMMIT, or ABORT states.
    - New coordinator **globally aborts** the transaction.
  - If the new coordinator is in PRE-COMMIT state
    - Participants can be in READY, PRE-COMMIT or COMMIT states.
    - The new coordinator globally commits the transaction.
  - If the new coordinator is in COMMIT state
    - The new coordinator globally commits the transaction
  - If the new coordinator is in ABORT state
    - The new coordinator globally aborts the transaction

3PC Recovery Protocols

- The coordinator fails in WAIT
  - This causes participants timeout, which have elected a new coordinator and terminated the transaction
  - The new coordinator could be in WAIT or ABORT state, leading to the **aborted** transaction
  - Ask around upon recovery.
- The coordinator fails in PRE-COMMIT
  - Ask around upon recovery.
- The coordinator fails in COMMIT or ABORT
  - Nothing special if all the acknowledgements have received; otherwise the termination protocol is involved.

3PC Recovery Protocols (*cont.*)

- The participants fail in INITIAL
  - Unilaterally abort upon recovery
- The participants fail in READY
  - The coordinator has been informed about the local decision
  - Upon recovery, ask around
- The participants fail in PRE-COMMIT
  - Upon recovery, ask around to determine how the other participants have terminated the transaction
- The participants fail in COMMIT or ABORT
  - No need to do anything

More about 3PC

- Advantage
  - Non-blocking
- Disadvantages
  - Fewer independent recovery cases
  - More messages

Network Partitioning

- Simple partitioning
  - The network is partitioned into two parts
- Multiple partitioning
  - More than two parts

Network Partitioning (*cont.*)

- Formal bounds
  - There exist non-blocking protocols
    - which are resilient to a single network partition if all undeliverable messages are returned to sender
  - There exists no non-blocking protocol
    - that is resilient to a network partition if messages are lost when partition occurs.
    - that is resilient to a multiple partition.

Design Decisions

- Allow partitions to continue their operations and compromise database consistency;

  or

- Guarantee the consistency by permitting operations in one partition, while the sites in other partitions remain blocked.

8. Distributed DBMS Reliability

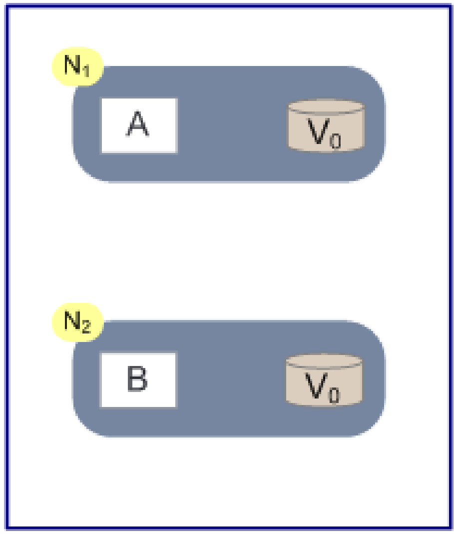
- ❖ Concept of Reliability
- ❖ Local Reliability Protocols
- ❖ Distributed Reliability Protocols
- ☞ Brewer's CAP Theorem and Relevant Efforts

Eric A. Brewer's CAP Theorem for Availability

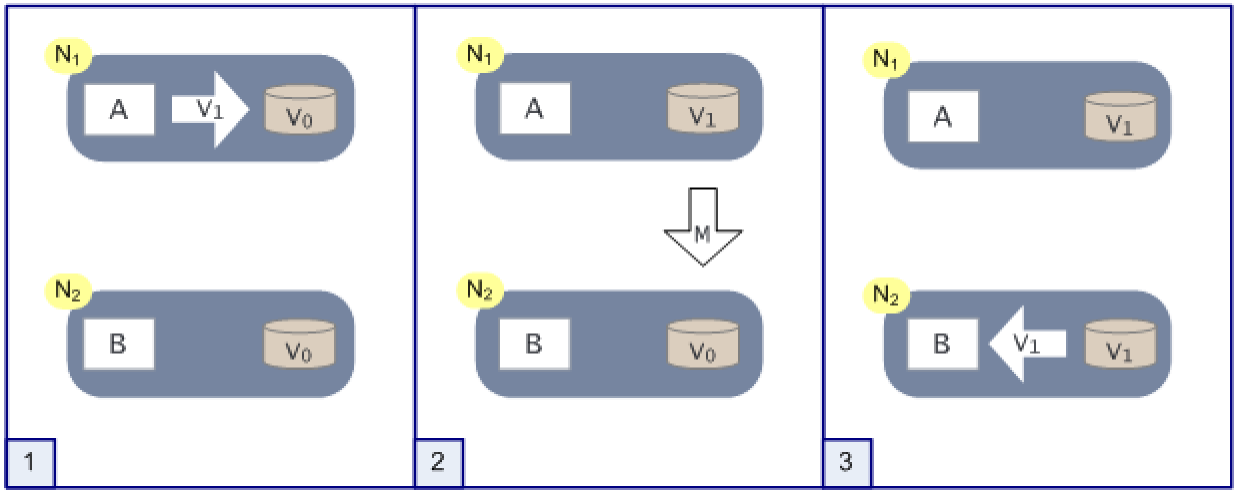
- Traditionally, thought of as the server/process available five 9′s (99.999 %).
- However, for large node systems, at almost any point in time there′s a good chance that a node is either down or there is a network disruption among the nodes.
  - Want a system that is resilient in the face of network disruption

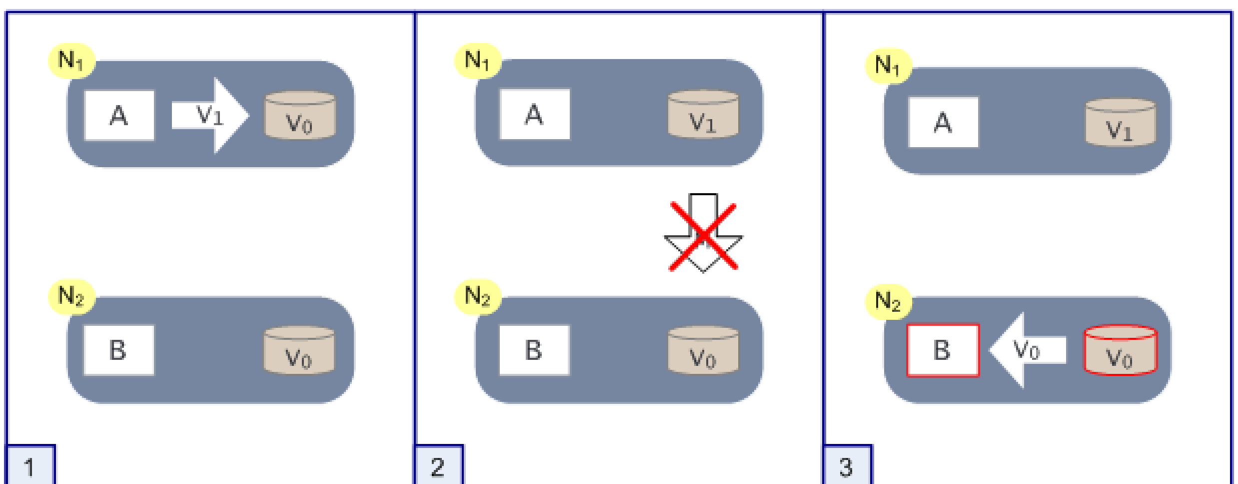
Consistency Problem

- For example:
  - Row V_0 is replicated on nodes N1 and N2
  - Client A writes row V_0 to node N1
  - Some period of time t elapses.
  - Client B reads row V_0 from node N2
  - Does client B see the write from client A?



Consistency Problem (*cont.*)





Consistency

- A consistency model determines rules for visibility and apparent order of updates
  - Locking-based or Timestamp order-based
  - For Distributed DBMS we learned, the answer is: yes
- Consistency is a continuum with tradeoffs
- CAP Theorem states that
  - **Strict Consistency** can't be achieved at the same time as availability and partition-tolerance.

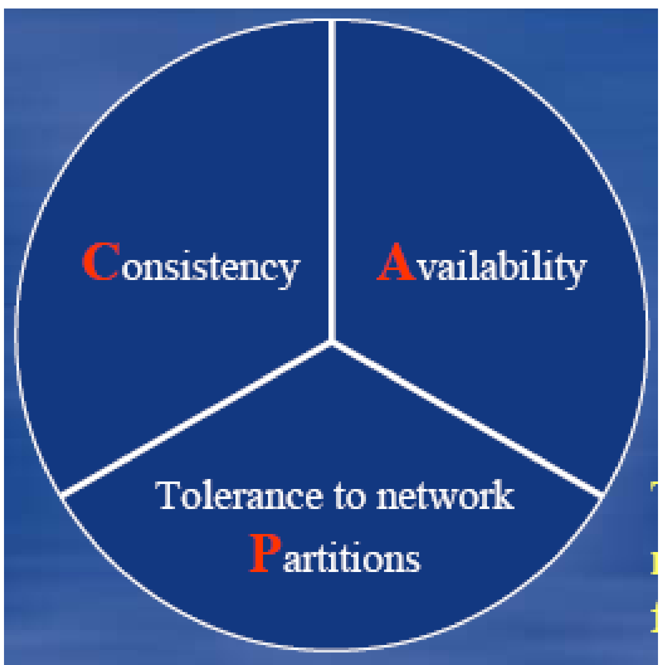
Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent.

- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service.

Brewer's CAP Theorem

- Born at the talk on Principles of Distributed Computing (PODS) Conference, July 2000
- Three properties of a system -
  - availability, consistency, and partitions



Theorem: You can have at most two of these three properties for any shared-data system.

Brewer's CAP Theorem (*cont*.)

- To scale out, you have to partition. That leaves either consistency or availability to choose from
  - In almost all cases, you would choose availability over consistency
  - It is impossible to achieve all three.

ACID vs. BASE

- DBMS research is about ACID (mostly)
- But we loss “C” and “I” for availability, graceful degradation, and performance

This tradeoff is fundamental.

BASE:

- Basically Available (system seems to work all the time)
- Soft-state (it doesn't have to be consistent all the time)
- Eventual consistency (it becomes consistent at some later time)

ACID vs. BASE (*cont.*)

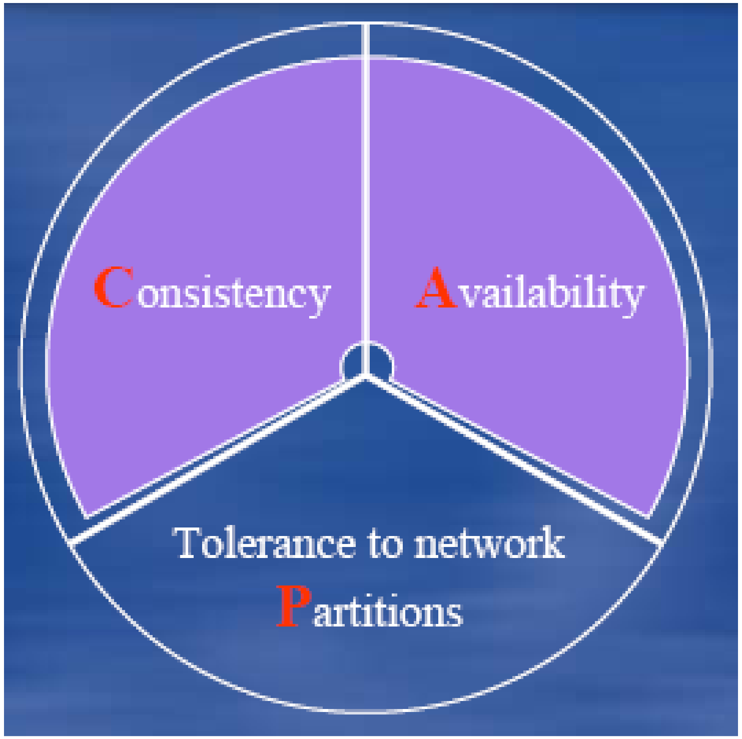
ACID

- Strong consistency
- Isolation
- Focus on "commit"
- Nested transaction
- Availability?
- Conservative (pessimistic)
- Difficult evolution (e.g., schema)

BASE

- Weak consistency
  - Stale data OK
- Availability first
- Approximate answers OK
- Aggressive (optimistic)
- Simpler!
- Faster
- Easier evolution

Forfeit Partitions in CAP Theorem



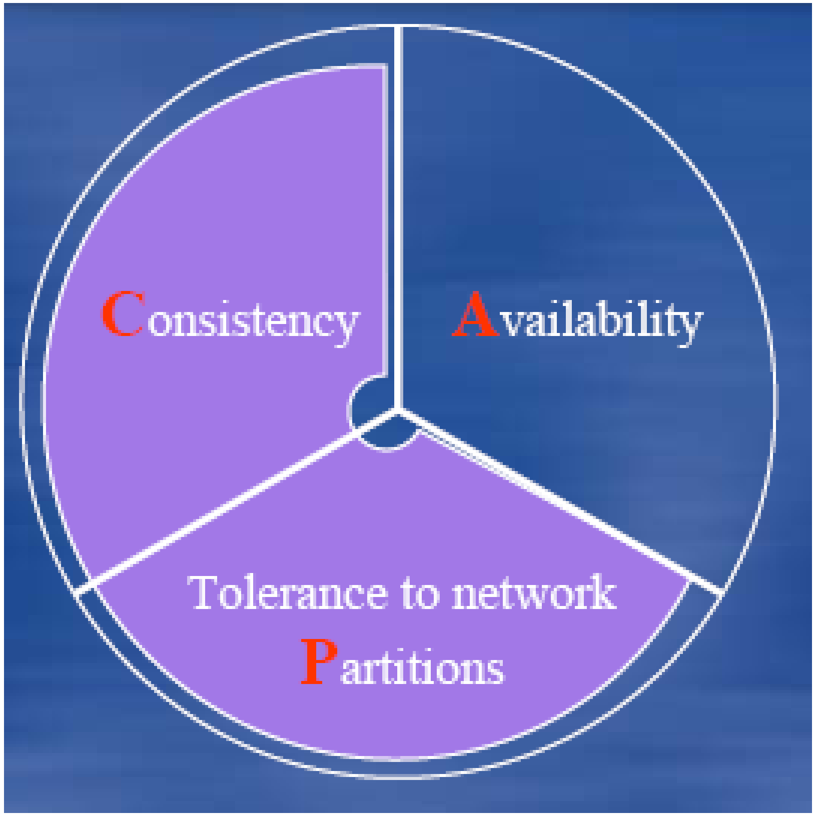
Examples:

- single site databases
- cluster databases
- LDAP
- xFS file systems

Traits:

- 2-phase commit
- Cache validation protocols

Forfeit Availability in CAP Theorem



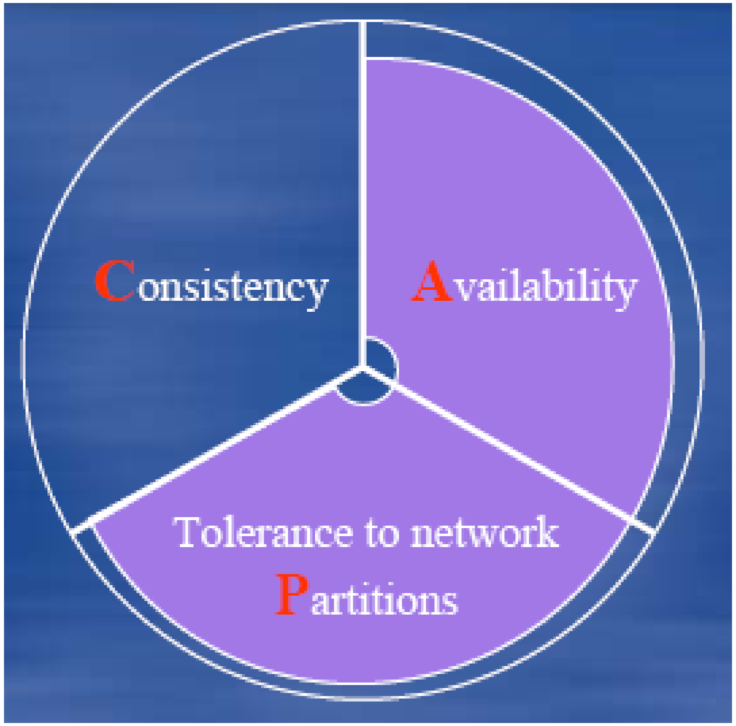
Examples:

- Distributed databases
- Distributed locking

Traits:

- Pessimistic locking
- Making minority partitions unavailable

Forfeit Consistency in CAP Theorem



Examples:

- Web Caching
- DNS (Domain Name System)
- Coda file systems

Traits:

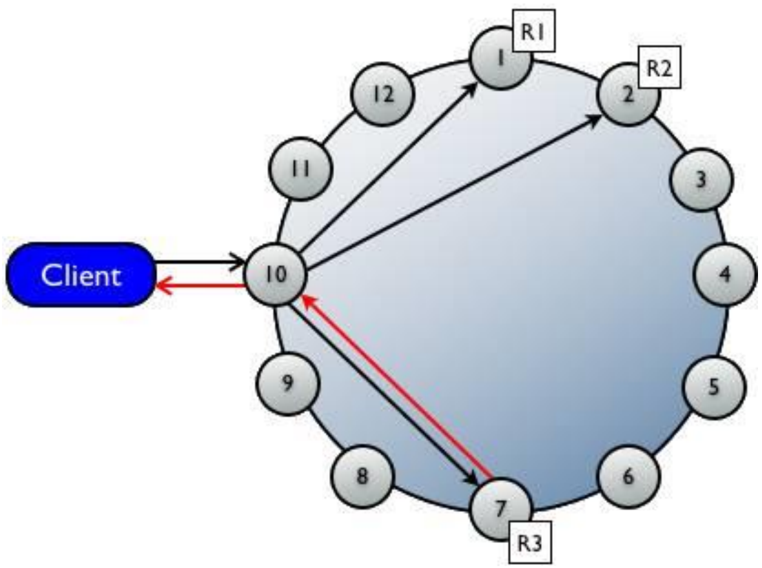
- Expirations/leases
- Conflict resolution
- Optimistic

Tradeoffs in Reality

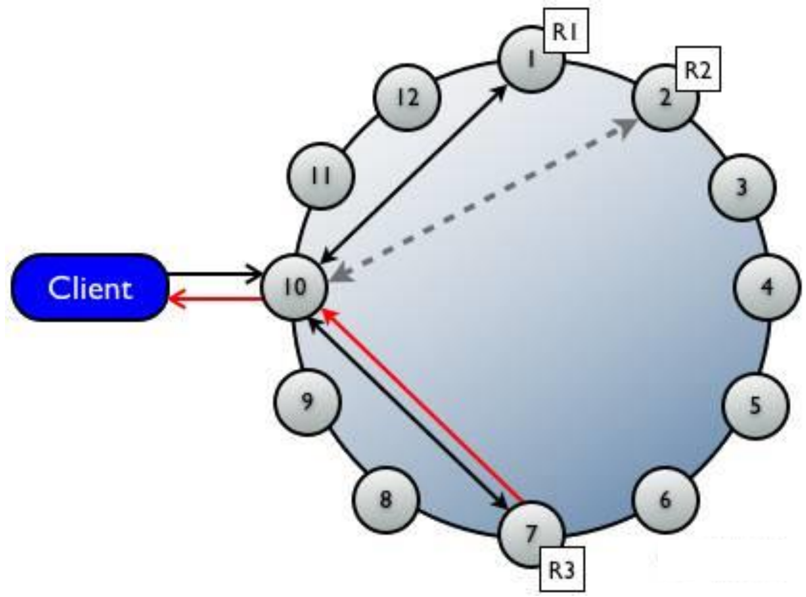
- The whole space is useful
- Real internet systems are a careful mixture of ACID and BASE
  - Use ACID for user profiles and logging (for revenue)
- Symptom of a deeper problem: systems and database communities are separate but overlapping (with distinct vocabularies)
- Big applications like Google, Yahoo, Facebook, Amazon, eBay, etc. adopt CAP and BASE

How Cassandra Reads and Writes?

- A masterless design where all nodes are the same, which provides operational simplicity.
- Linear scale performance



Write Request



Read Request
(direct read; read repair)

Cassandra’s Eventually Consistency

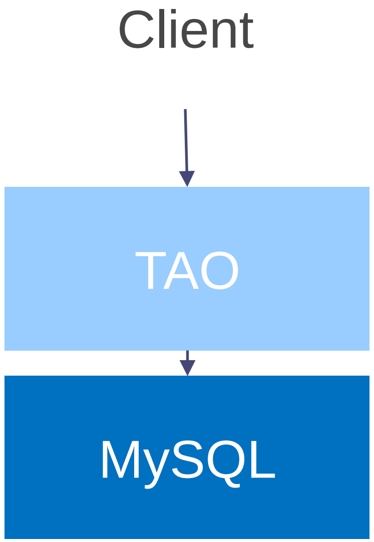
- Data is replicated across a cluster for data availability. Nevertheless, for a very short interval, some replicas may have the latest data while other replica are still saving/synchronizing the new value.
- Cassandra provides choices of requesting different levels of consistency on each read or write operation at the cost of latency
  - Low consistency request may return older copy of data (or stale data) but promise a faster response. Higher consistency request will reduce the chance of stale data but will have longer latency and more vulnerable to replica outage.

Cassandra's Eventually Consistency

- The consistency setting allows developers to decide how many replicas in the cluster must acknowledge a write operation or respond to a read operation before considering the operation successful.
- Cassandra also runs a read repair for the queried key to bring the key/column back to consistency. For a low consistency level, read repair will run in background. Otherwise, it is done before returning the data.

Facebook's Consistency

- TAO: Facebook's Distributed Data Store for the Social Graph
- Originally storing the social graph in MySQL, querying it from PHP, and caching results in memcache
- Later replacing memcache with TAO
- Features
  - Data model: *objects* and *associations*
  - Architecture: *leaders* and *followers*
  - API: simple API for clients to use



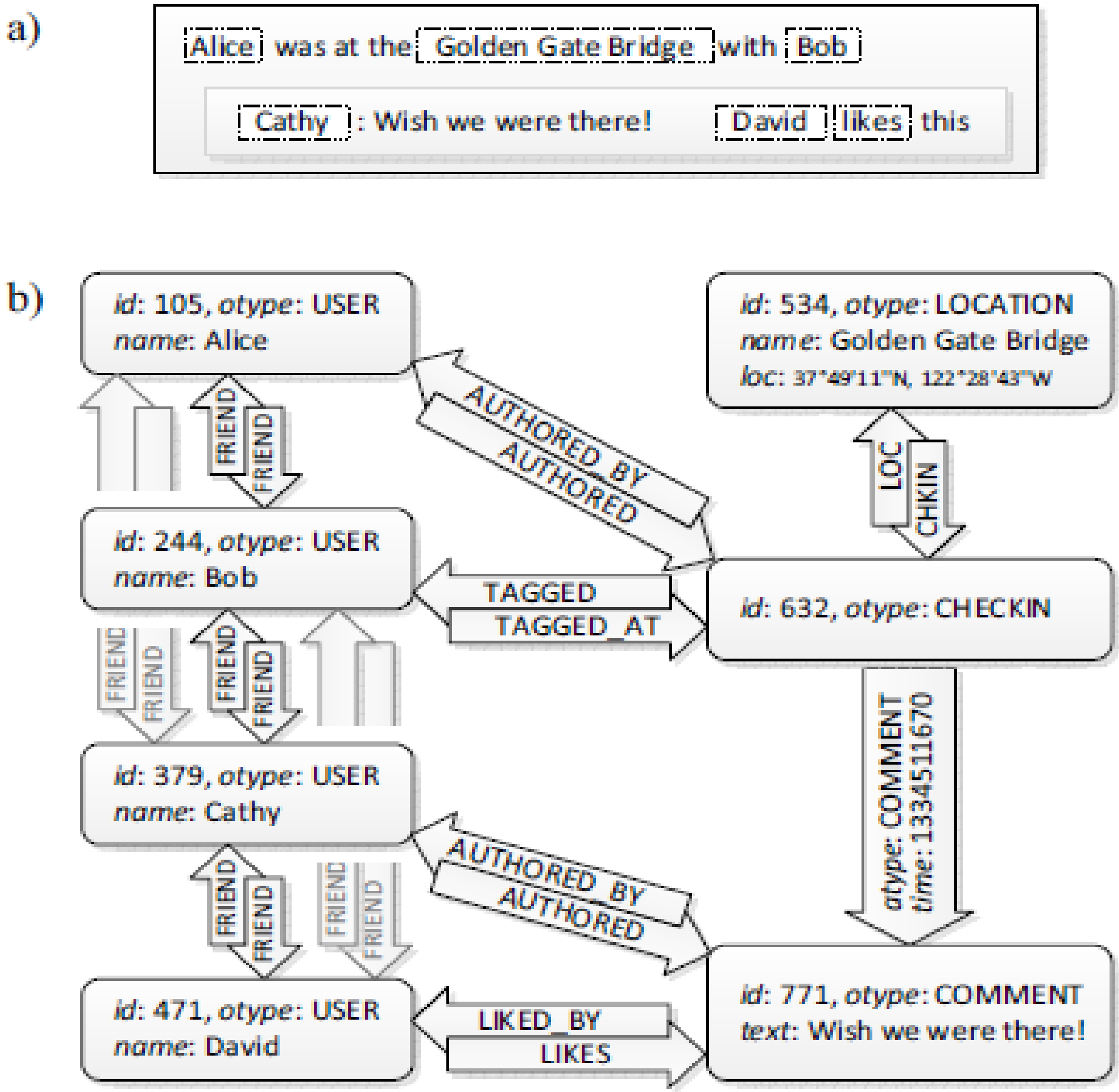


Figure 1: A running example of how a user's checkin might be mapped to objects and associations.

Facebook TAO's API

Object

```
create
retrieve
update
delete
```

Association

```
assoc_add
assoc_del
assoc_change_type
```

Query

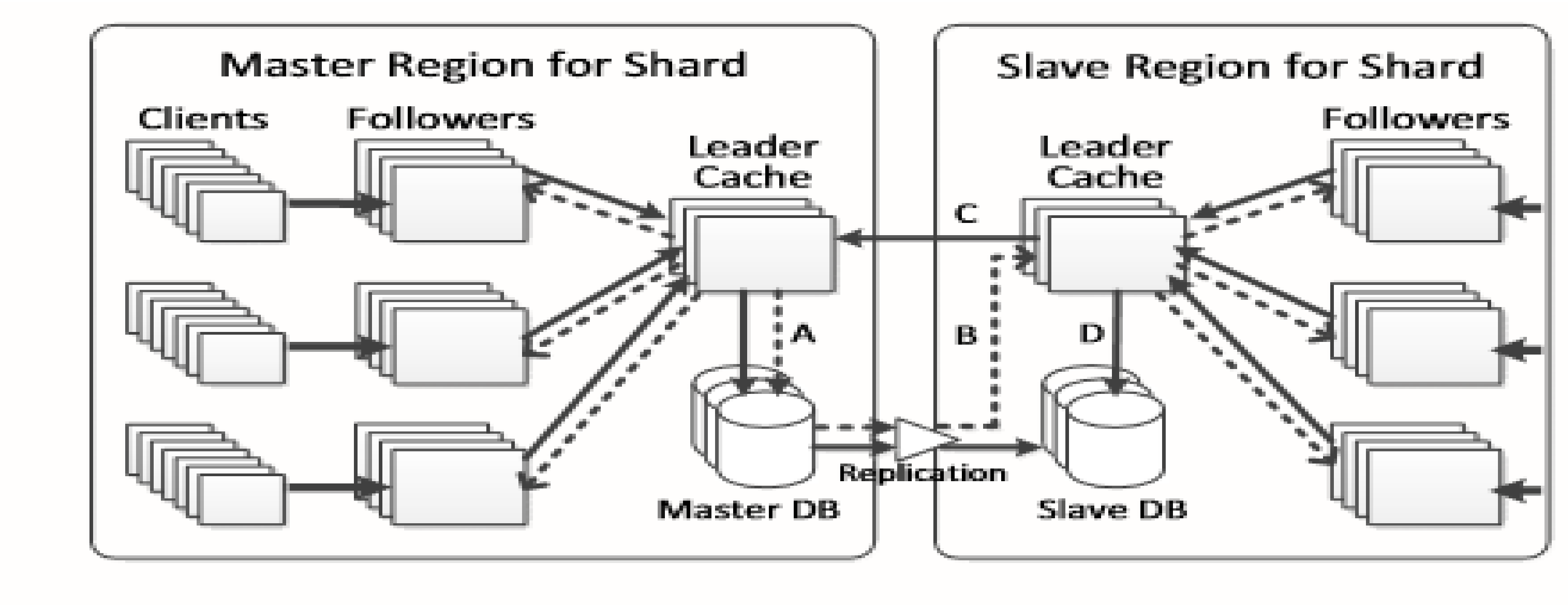
```
assoc_get
assoc_count
assoc_range
assoc_time_range
```

- Simple to use
- Any client (not just a PHP client) can use

- Two query examples
  - "50 most recent comments on Alice's checkin" ⇒ assoc_range(632, COMMENT, 0, 50)
  - "How many checkins at the GG Bridge?" ⇒ assoc_count(534, CHECKIN)

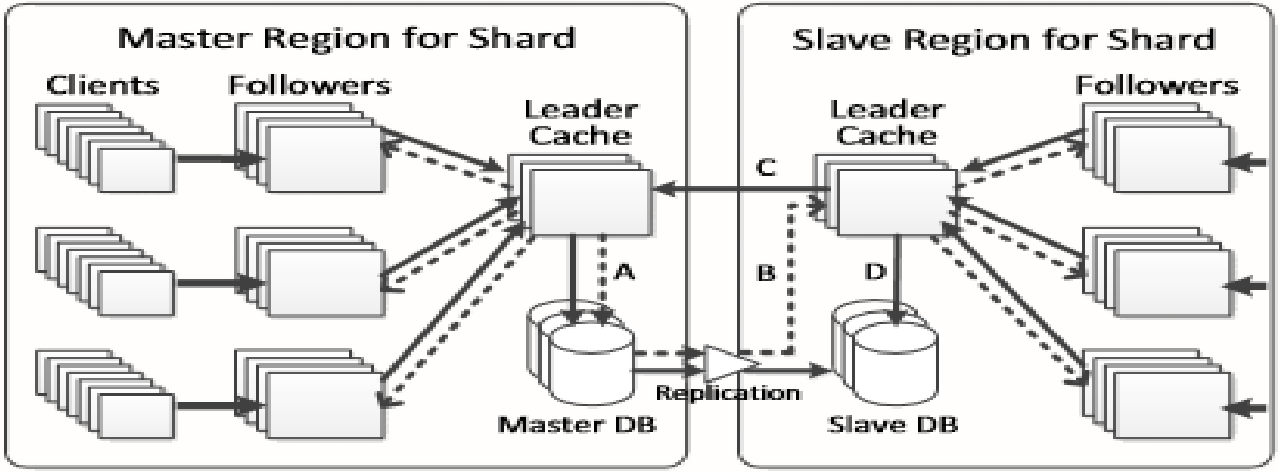
Facebook TAO's Eventually Consistency

- Successful writes (by leaders) return a *changeset* to slave leader/follower tiers
- Followers may invalidate stale data, then query leaders for fresh data

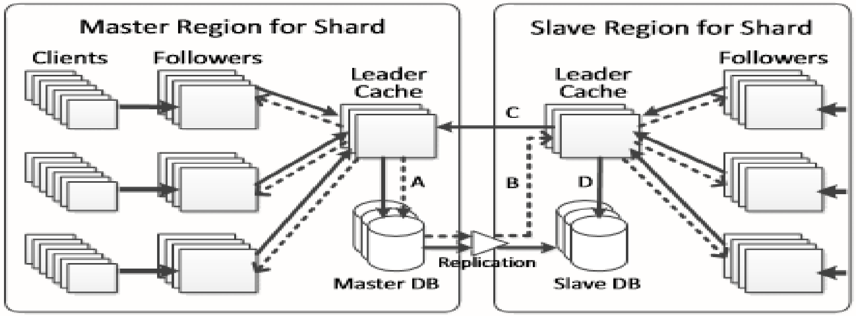


Facebook TAO's Failure Detection

- Relies on network timeouts
- Handles
  - Database failures
  - Leader failures
  - Refill and invalidation failures
  - Follower failures

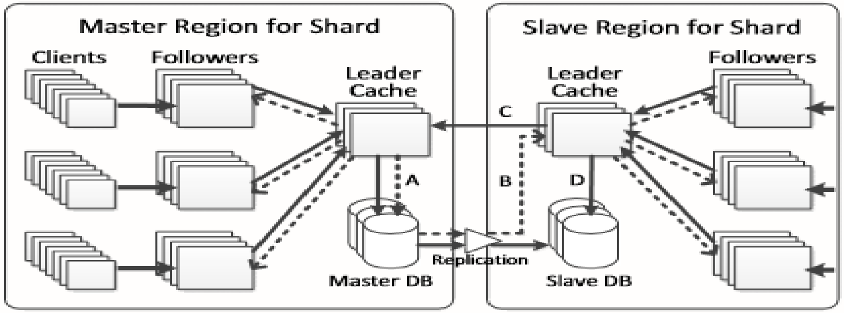


Database Failures



- When a **master DB** is down, one of its slaves is automatically promoted to be the new master.
- When a **slave DB** is down,
  - cache misses are redirected to the TAO leaders in the master region hosting the master DB.
  - an additional binlog tailer runs on the master DB, and the refills and invalidates are delivered inter-regionally.
  - When the slave DB comes back up, invalidation and refill messages from the outage will be delivered again.

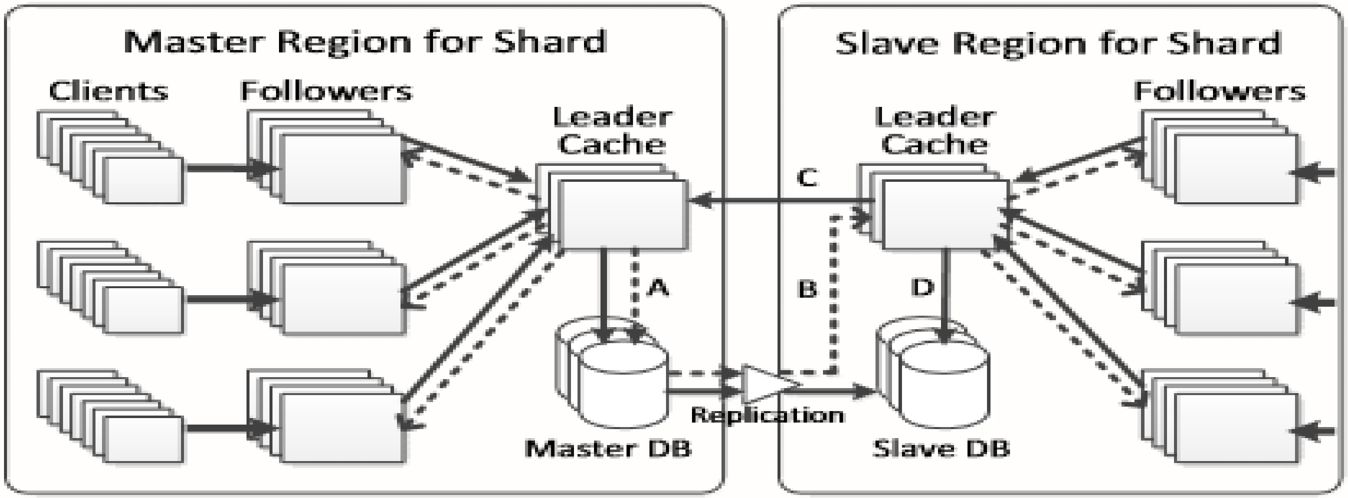
Leader Failures



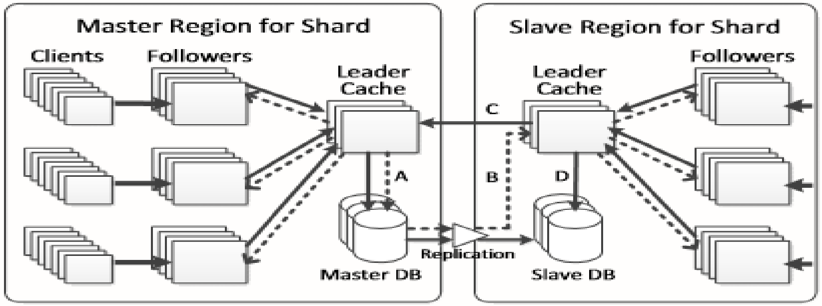
- When a leader cache server fails, followers automatically route read and write requests around it.
- Followers reroute read misses directly to the database.
  Writes to a failed leader, in contrast, are rerouted to a random member of the leader's tier as the replacement leader.
- The replacement leader also enqueues an asynchronous invalidation to the original leader that will restore its consistency.

Refill and Invalidation Failures

- Leader may send invalidation messages occasionally to inform followers they have stale data.
- If a follower is unreachable, the leader queues the message to disk to be delivered at a later time.



Follower Failures



- Each TAO client is configured with **a primary and backup follower tier**.
- In normal operations requests are sent only to the primary.
- If the server that hosts the shard for a particular request has been marked down due to timeouts, then the request is sent instead to that shard's server in the backup tier.

NewSQL Solutions

- SQL as the primary mechanism for application interaction.
- ACID support for transactions
- Enterprises cannot afford to lose the ACID properties
- Most current enterprise applications require SQL support

What we are talking about in the course is still valuable!

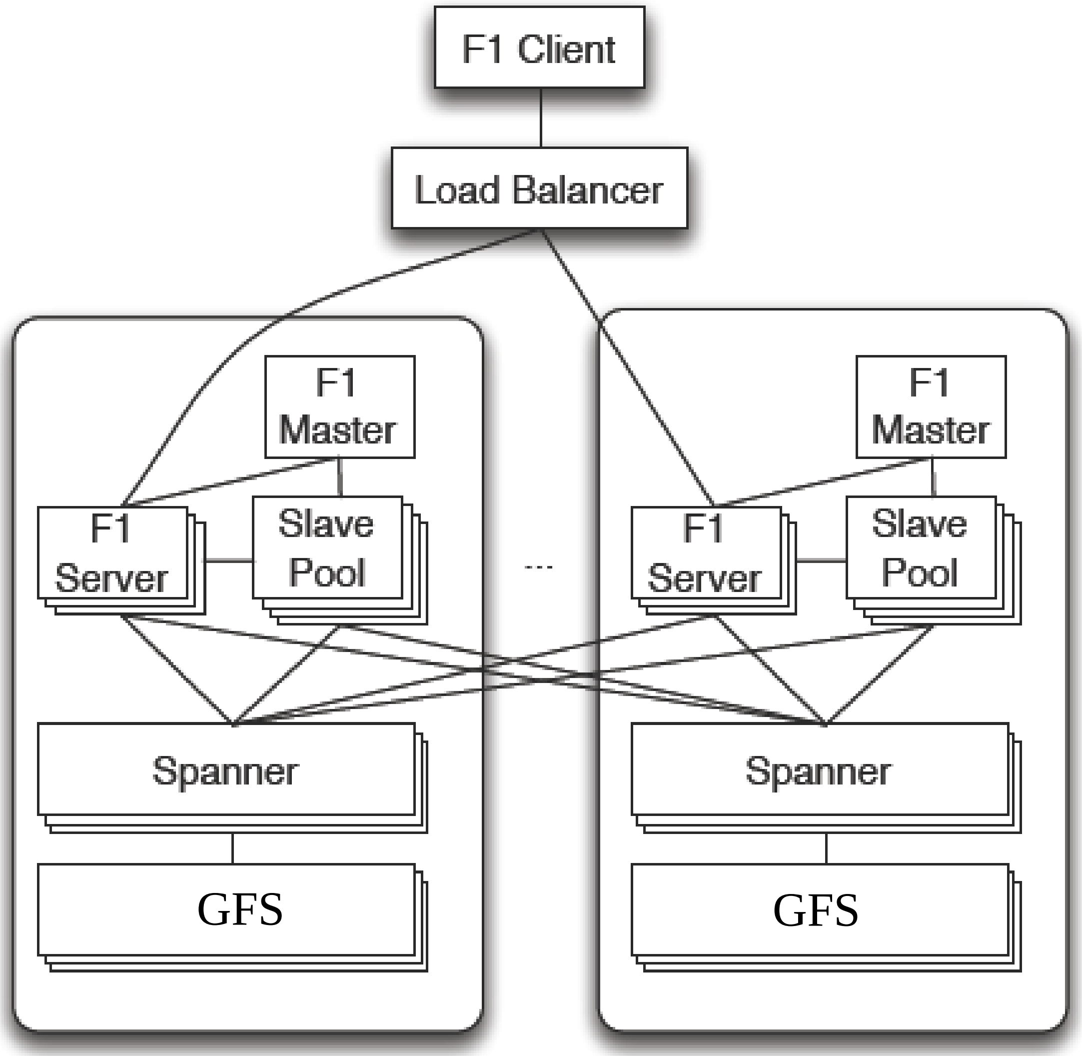
NewSQL Solutions (*cont.*)

- All systems are probabilistic
  - no such thing as a 100% working system
  - no such thing as 100% fault tolerance
  - partial results are often OK (and better than none)
- A non-locking concurrency control mechanism, so real-time reads will not conflict with writes.
- An architecture providing much higher per-node performance than available from traditional RDBMS solutions.
- A scale-out, shared-nothing architecture, capable of running on a large number of nodes without suffering bottlenecks.

Google’s NewSQL Solution

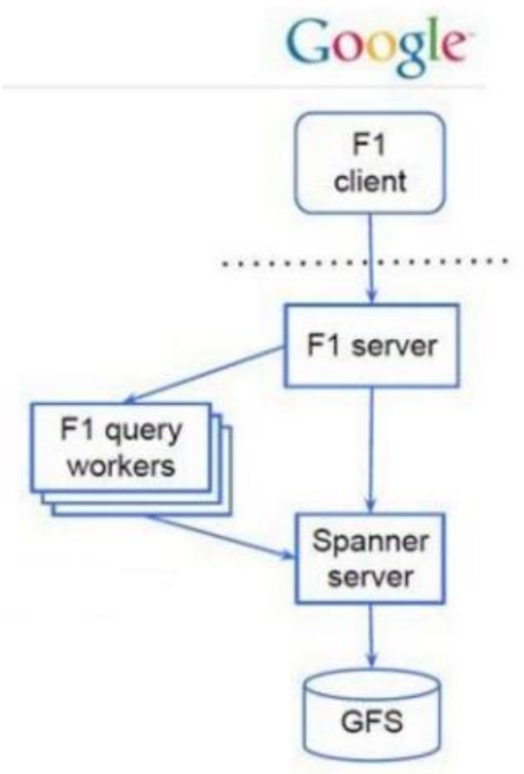
- Google’s BigTable supports NoSQL and BASE
- Goolge also needs to support ACID
- Google’s F1 is a hybrid database that combines high availability, the scalability of NoSQL systems like Bigtable, and the consistency and usability of traditional SQL databases.

Basic Architecture of F1

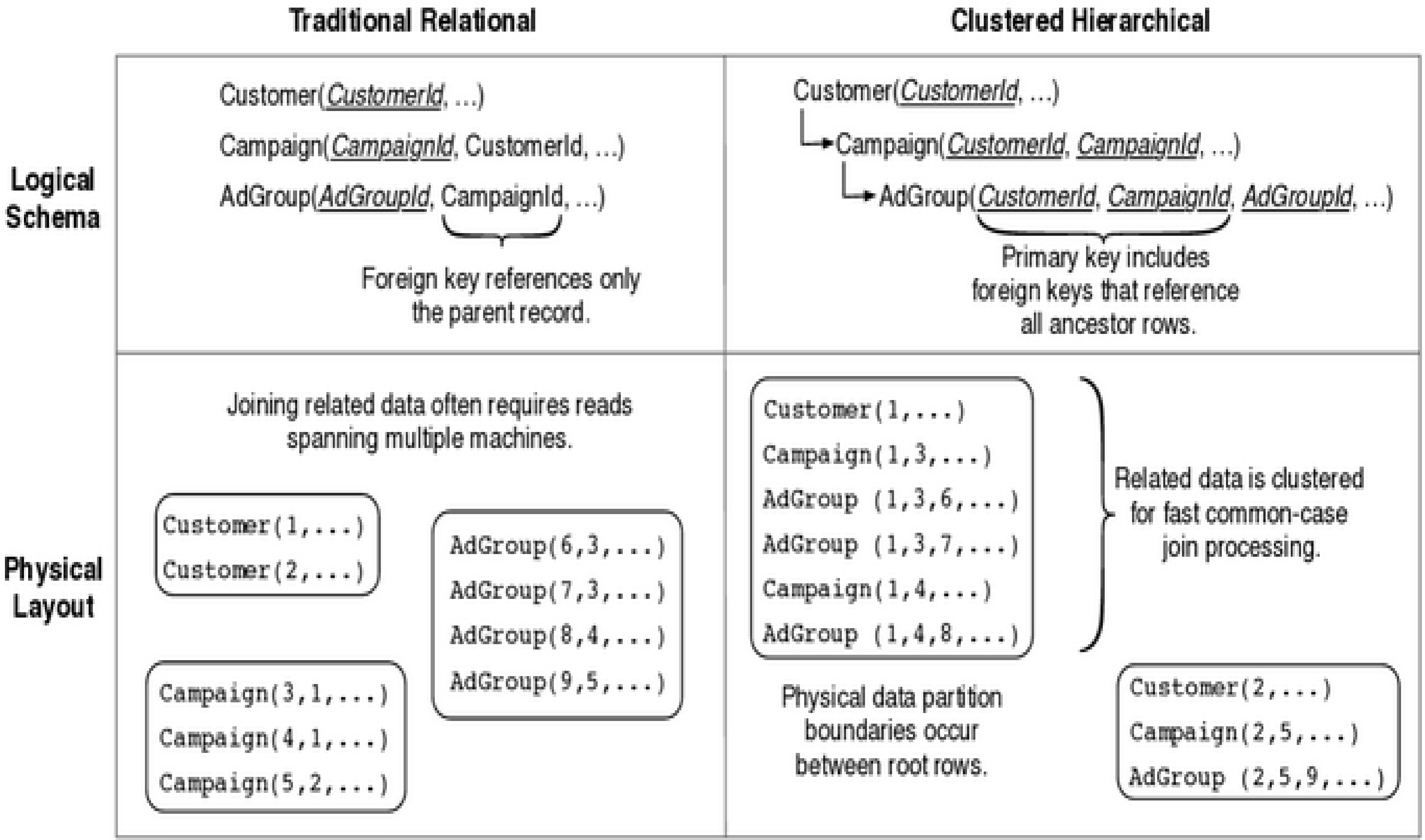


Google F1 and Spanner

- F1 is built on Spanner, which provides synchronous cross-data center replication and strong consistency.
  - Spanner: Google's Globally-Distributed Database
  - Synchronous replication implies higher commit latency
  - Spanner mitigates this latency by using a hierarchical schema model with structured data types and through smart application design.

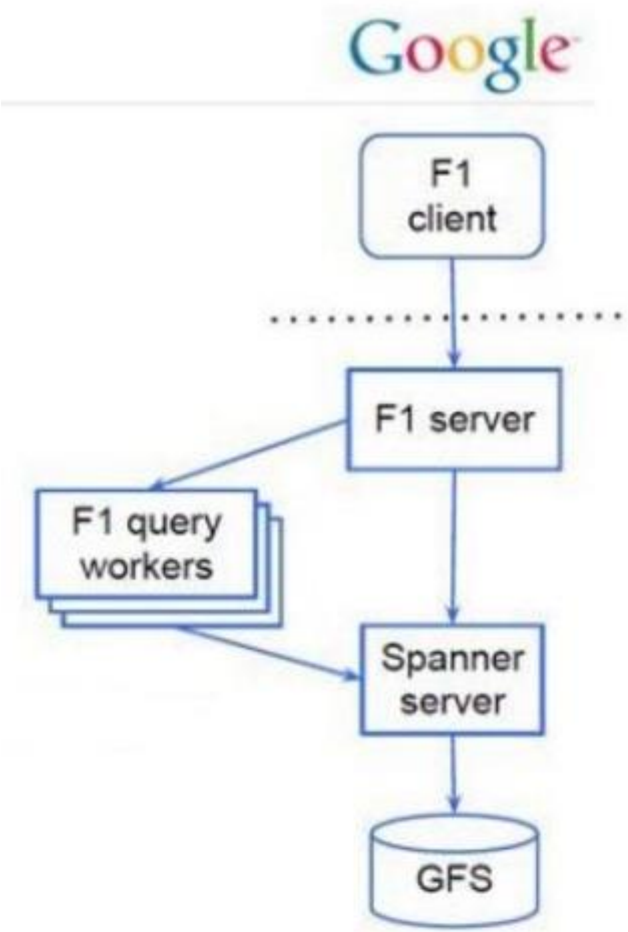


F1's clustered hierarchical schema



Besides

- F1 also includes a fully functional distributed SQL query engine and automatic change tracking and publishing.



Google File System (GFS)

F1 supports three types of transactions

- Each F1 transaction consists of multiple reads, optionally followed by a single write that commits the transaction.
- F1 implements three types of transactions, all built on top of Spanner's transaction support
  - Snapshot transaction
  - Pessimistic transaction
  - Optimistic transaction

F1’s Snapshot Transaction

- Read-only transaction with snapshot semantics
- Multiple client servers can see consistent views of the entire database at the same timestamp

F1's Pessimistic Transaction

- The same as Spanner transactions
- Use a stateful communications protocol to require holding locks, so all requests in a single pessimistic transaction get directed to the same F1 server.
  - If the F1 server restarts, the pessimistic transaction aborts.
  - Reads in pessimistic transactions can request either shared or exclusive locks.

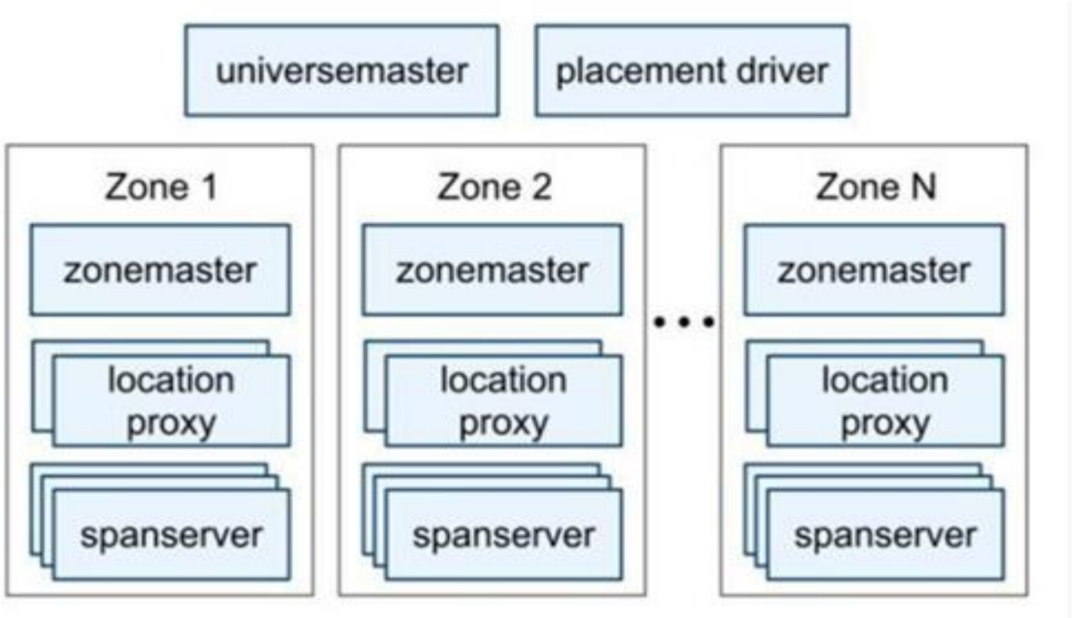
F1's Optimistic Transaction

- Consist of an arbitrarily long read phase, which never takes Spanner locks, and then a short write phase.
- To detect row-level conflicts, F1 returns with each row its last modification timestamp, which is stored in a hidden lock column in that row.
- The new commit timestamp is automatically written into the lock column whenever the corresponding data is updated (in either pessimistic or optimistic transactions).

F1's Optimistic Transaction (*cont*.)

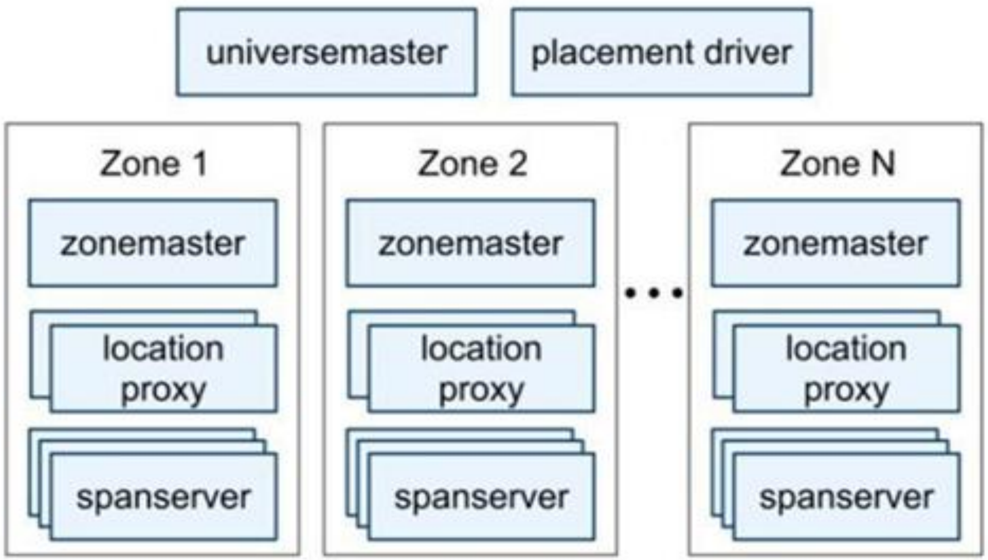
- The client library collects these timestamps, and passes them back to an F1 server with the write that commits the transaction.
- The F1 server creates a short-lived Spanner pessimistic transaction and re-reads the last modification timestamps for all read rows.
  - If any of the re-read timestamps differ from what was passed in by the client, there was a conflicting update, and F1 aborts the transaction;
  - Otherwise, F1 sends the writes on to Spanner to finish the commit.

Architecture of Spanner



- A zone has one zonemaster and hundreds of spanservers.
  - zonemaster assigns data to spanservers
  - Spanservers serve data to clients.
  - The per-zone location proxies are used by clients to locate the spanservers for the requested data.

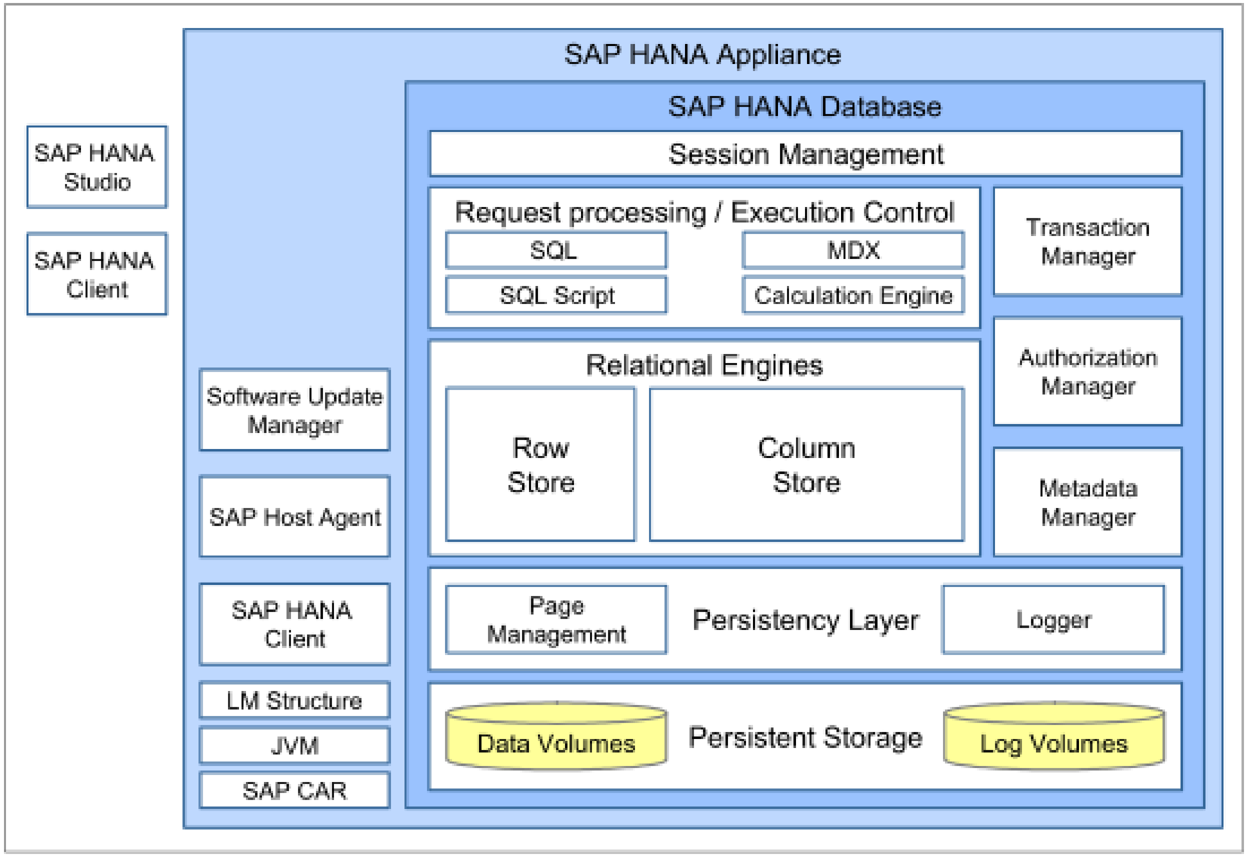
Architecture of Spanner (*cont.*)



- The universe master displays status information about all the zones for interactive debugging.
- The placement driver handles automated movement of data across zones on the timescale of minutes.
  - It periodically communicates with the spanservers to find data that needs to be moved, either to meet updated replication constraints or to balance load.

SAP Hana

- In-memory DB
- Combines row, column and object-oriented technology at the table level



Conclusion

- Can have consistency & availability within a cluster, but it is still hard in practice
- OS/Networking good at BASE/Availability, but terrible at consistency
- Databases better at Consistency than Availability
- Wide-area databases can’t have both
- Disconnected clients can’t have both

Question & Answer