9.2. Commit coordination

Reliability: From a single node to a distributed system

- Singles node may cause significant availability problems: How long is the recovery time in ARIES?
- Single nodes are single points of failure ($\rightarrow$ D)
  - Loss of disk storage not very likely, but happening
- Even "centralized" systems need some "poor mans" distribution:
  - Periodic backup on a separate system ($\rightarrow$ periods between backups at risk)
  - Log shipping: write log to remote storage(s) ($\rightarrow$ how to ensure durability/stable storage, performance problem)
- Fully distributed setups need to coordinate (A,D)
  - Partitioning: split collections along a predicate or along attributes
  - Replication: keep multiple copies of the same data

Commit Coordination and Consensus

Problem setting

- A set of independent servers
  - storing data items
  - communicating by messages
- A transactions spanning several servers, i.e. subtransactions at some sites
- For a successfull commit, all substransactions have to be committed, but
  - Each subtransaction may fail (indepedently)
  - Even successfull substransactions may have to be aborted (atomicity)

Approach

- All nodes have to agree on the same outcome of the transaction
- Combination of local commit+agreement on commit decision
  - Local part: perform same steps in single-site commit (log), but wait for finalization until a consensus is achieved
  - Distribution: perform agreement to commit or abort, deal with failures
  - A single site cannot make a commit decision

Problem Modelling

- Nodes have local state
  - Needed for protocol
  - Also definition of Distributed Database
- Node exhibit fail-stop or fail-recover, extension to byzantine failures possible
- Asynchronous communication
  - Messages may take arbitrarily long
  - Message loss is indiscernible from arbitrary delay
  - We assume message integrity, though
  - (generalization of typical internet behavior)
- Network may see temporary interruptions and partitions

Requirements of commit/consensus

Formal Properties

- **Termination**: All correct processes *eventually* decide.
- **Agreement**: All correct processes select the same value (even if they fail later on)
- **Integrity/Validity**: All deciding processes select the "right" value (one that is proposed)

These are *safety* and *lifeness* properties

Practical Concern: Efficiency

- Number of messages (overall)
- Number of rounds/exchanges

Could you think of lower bounds of for the best case?

Challenges

Theoretical Impossibility

Our problem modelling clashes with Fisher-Lynch-Patterson (FLP) [1985]:
"No consensus can be guaranteed in an *asynchronous* communication system in the presence of any *failures*".

Inituition:
Is a process actually dead or will it come back and affect the consensus?

We cannot make systems synchronous and reliable
Possible workarounds:

- Fault masking: assume eventual recovery and keep waiting
- Failure detection - also affected by FLP, either
  - accurate but not live (possibly waiting forever)
  - live but not accurate: enforce synchronous behavior (e.g., timeouts) and restore/kill misdetected survivors

Overview on Commit Protocols

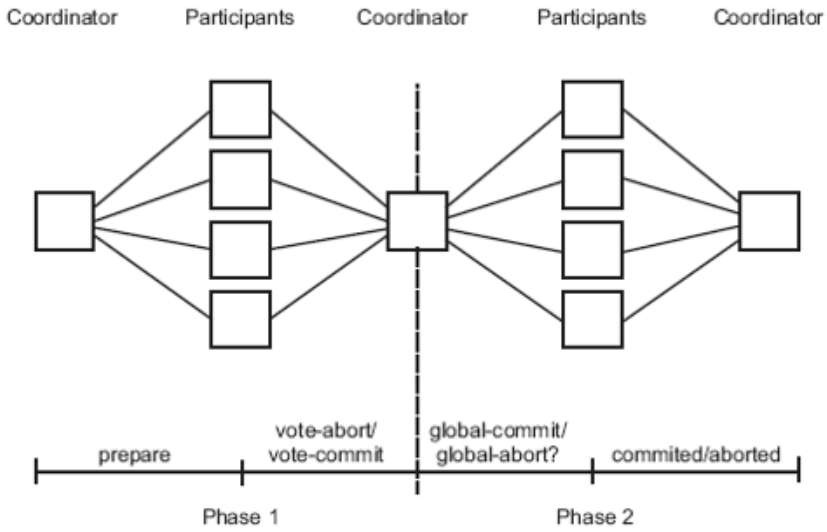
Fundamental approach

- Rely on a leader/coordinator
- A value is proposed by the leader or by a client talking to the leader
- Participants decide and inform the coordinator

Popular algorithms

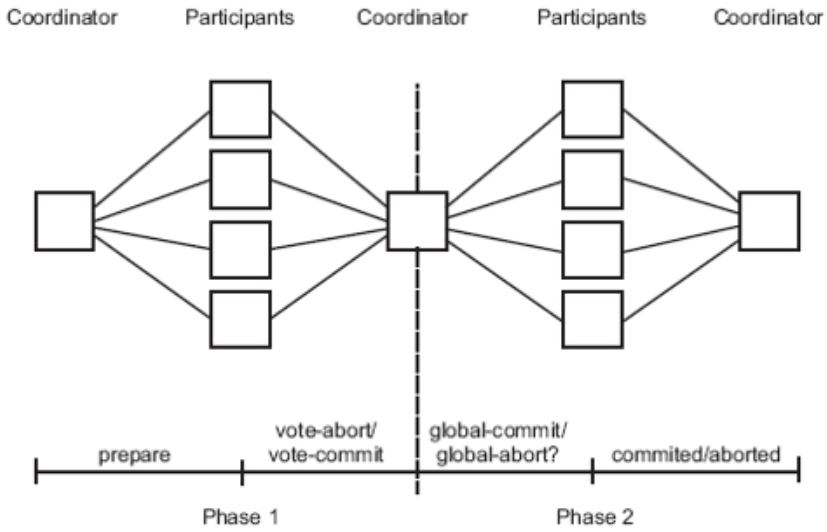
- 2PC: Simple and effective, but can degrade into blocking behavior
- 3PC: Add another phase to reduce period of vulnerability spread decision knowledge, may be unsafe in the presence of network splits
- Paxos, Multi-Paxos, Paxos commit: generalized, safe, nonblocking, may not terminate
- Raft: same goals as Paxos, supposedly simpler to understand and implement

9.2. 1 2-Phase-Commit (2PC)



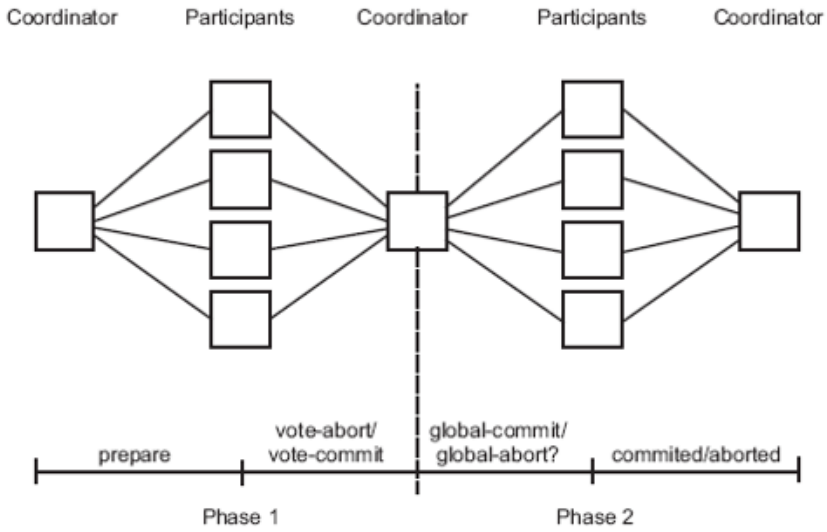
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly - discarding the result or making it permanent.

9.2. 1 2-Phase-Commit (2PC)



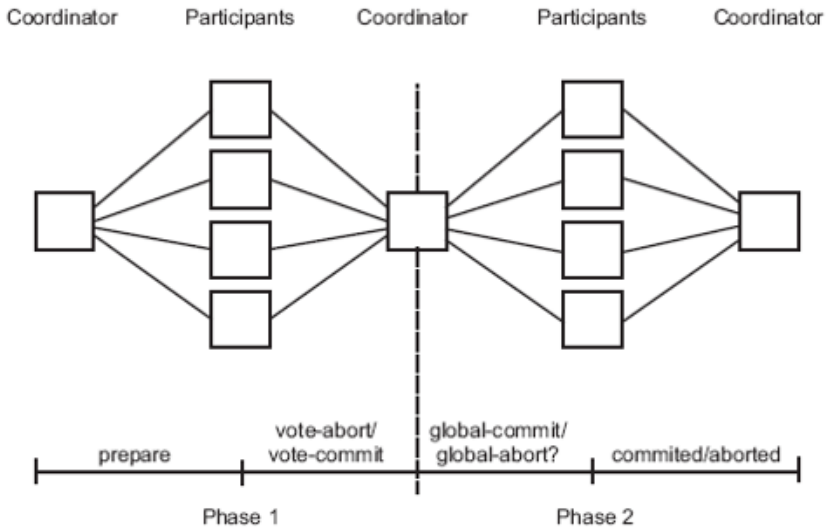
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly - discarding the result or making it permanent.

9.2. 1 2-Phase-Commit (2PC)



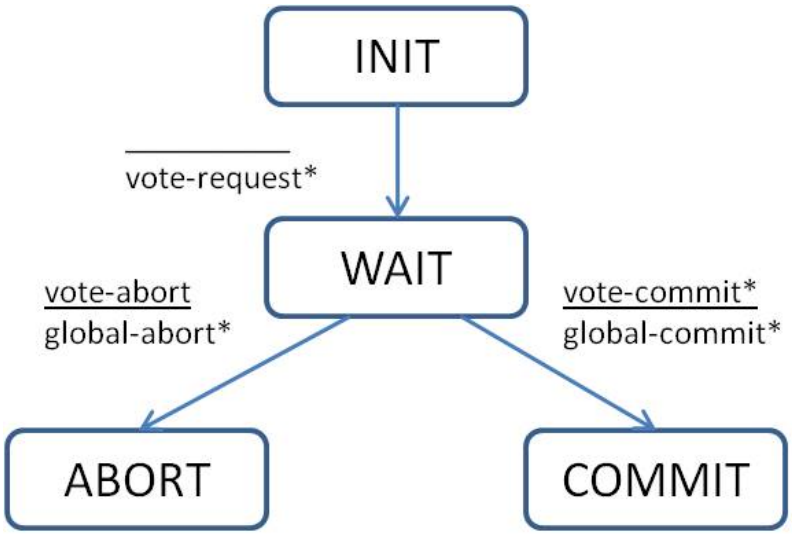
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly - discarding the result or making it permanent.

9.2. 1 2-Phase-Commit (2PC)

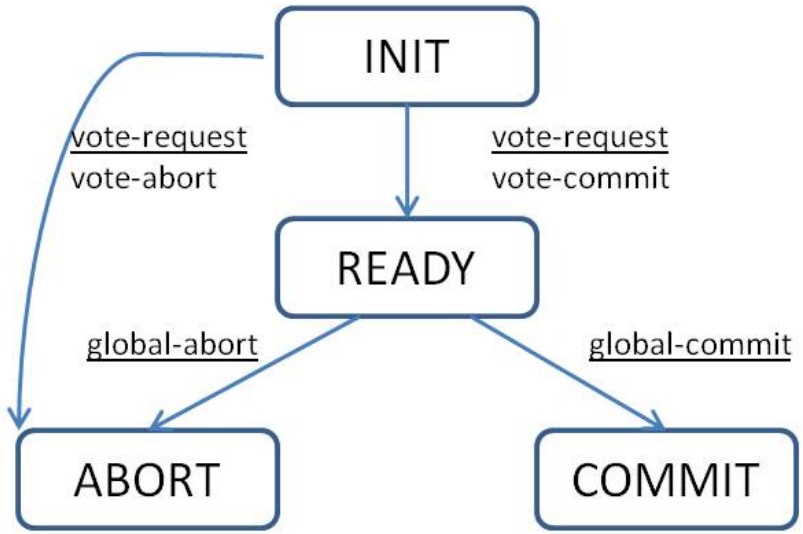


- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly - discarding the result or making it permanent.

Protocal automata



Coordinator



Participant

Notation: $\frac{message\ \ received}{message\ \ sent}$

msg^*: message sent-to/received-from all

Distributed Transaction Log: Supporting fail-recover

Log operations

(1) When the coordinator sends *vote-request*, it writes a *start-2PC* record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

(2) If a participant replies *vote-commit*, it writes a *vote-commit* record in the DT log, before sending *vote-commit* to the coordinator. This record contains the name of the coordinator and a list of the other participants. If the participant votes no, it writes an *abort* record either before or after the participant sends *vote-abort* to the coordinator.

(3) Before the coordinator sends *global-commit* to the participants, it writes a *commit* record in the DT log.

(4) When the coordinator sends *global-abort* to the participants, it writes an *abort* record in the DT log. The record may be written before or after sending the messages.

(5) After receiving *global-commit* (or *global-abort*), a participant writes a *commit* (or *abort*) record in the DT log.

Distributed Transaction Log: Supporting fail-recover

Log operations

(1) When the coordinator sends *vote-request*, it writes a *start-2PC* record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

(2) If a participant replies *vote-commit*, it writes a *vote-commit* record in the DT log, before sending *vote-commit* to the coordinator. This record contains the name of the coordinator and a list of the other participants. If the participant votes no, it writes an *abort* record either before or after the participant sends *vote-abort* to the coordinator.

(3) Before the coordinator sends *global-commit* to the participants, it writes a *commit* record in the DT log.

(4) When the coordinator sends *global-abort* to the participants, it writes an *abort* record in the DT log. The record may be written before or after sending the messages.

(5) After receiving *global-commit* (or *global-abort*), a participant writes a *commit* (or *abort*) record in the DT log.

Distributed Transaction Log: Supporting fail-recover

Log operations

(1) When the coordinator sends *vote-request*, it writes a *start-2PC* record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

(2) If a participant replies *vote-commit*, it writes a *vote-commit* record in the DT log, before sending *vote-commit* to the coordinator. This record contains the name of the coordinator and a list of the other participants. If the participant votes no, it writes an *abort* record either before or after the participant sends *vote-abort* to the coordinator.

(3) Before the coordinator sends *global-commit* to the participants, it writes a *commit* record in the DT log.

(4) When the coordinator sends *global-abort* to the participants, it writes an *abort* record in the DT log. The record may be written before or after sending the messages.

(5) After receiving *global-commit* (or *global-abort*), a participant writes a *commit* (or *abort*) record in the DT log.

Distributed Transaction Log: Supporting fail-recover

Log operations

(1) When the coordinator sends *vote-request*, it writes a *start-2PC* record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

(2) If a participant replies *vote-commit*, it writes a *vote-commit* record in the DT log, before sending *vote-commit* to the coordinator. This record contains the name of the coordinator and a list of the other participants. If the participant votes no, it writes an *abort* record either before or after the participant sends *vote-abort* to the coordinator.

(3) Before the coordinator sends *global-commit* to the participants, it writes a *commit* record in the DT log.

(4) When the coordinator sends *global-abort* to the participants, it writes an *abort* record in the DT log. The record may be written before or after sending the messages.

(5) After receiving *global-commit* (or *global-abort*), a participant writes a *commit* (or *abort*) record in the DT log.

Distributed Transaction Log: Supporting fail-recover

Log operations

(1) When the coordinator sends *vote-request*, it writes a *start-2PC* record in the DT log. This record contains the identities of the participants, and may be written before or after sending the messages.

(2) If a participant replies *vote-commit*, it writes a *vote-commit* record in the DT log, before sending *vote-commit* to the coordinator. This record contains the name of the coordinator and a list of the other participants. If the participant votes no, it writes an *abort* record either before or after the participant sends *vote-abort* to the coordinator.

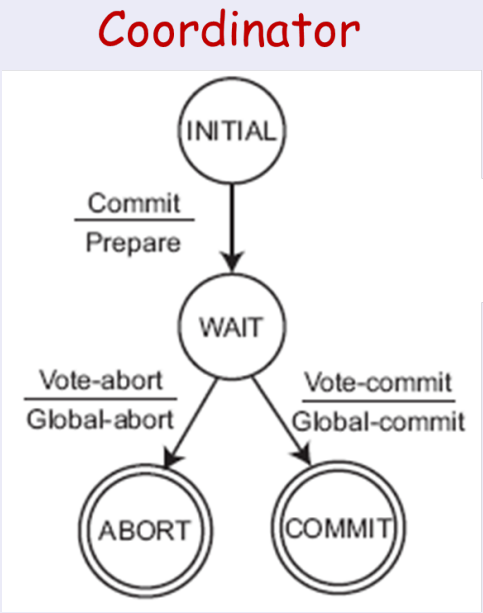
(3) Before the coordinator sends *global-commit* to the participants, it writes a *commit* record in the DT log.

(4) When the coordinator sends *global-abort* to the participants, it writes an *abort* record in the DT log. The record may be written before or after sending the messages.

(5) After receiving *global-commit* (or *global-abort*), a participant writes a *commit* (or *abort*) record in the DT log.

Termination Protocol: Coordinator Timeouts

- Timeout @ WAIT
  - Can not unilaterally commit.
  - Can abort and send Global-abort, since no global commit has been made
- Timeout @ ABORT / COMMIT
  - Repeatedly send Global-abort / Global-commit to the unresponsive participants.
  - **Stay blocked and wait for their ACK messages**.

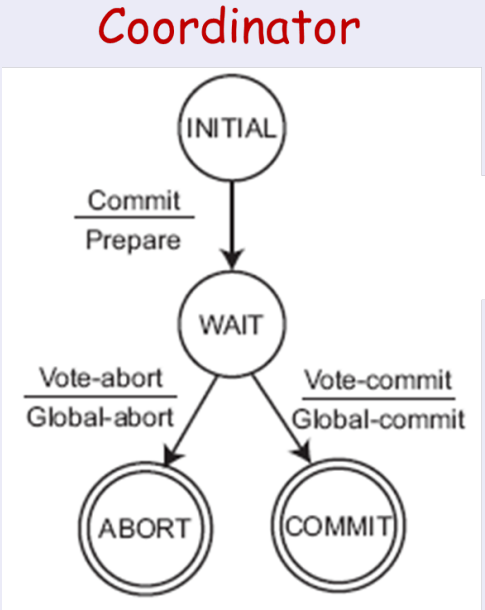


Termination Protocol: Participant Timeouts

- Timeout @ INITIAL
  - Coordinator must have failed at INITIAL.
  - Can abort.
  - If Prepare arrives later, can either Vote-abort or ignore it (i.e., let the coordinator timeout @WAIT).
- Timeout @ READY
  - Can not unilaterally commit or change its decision to an abort.
  - **Stay blocked**.

Recovery Protocol: Coordinator Failures

- Failure @ INITIAL
  - Start the commit process upon recovery.
- Failure @ WAIT
  - Restart the commit process upon recovery.
- Failure @ ABORT / COMMIT
  - If all ACKs have been received, nothing to do.
  - Else, invoke the termination protocol.



Recovery Protocol: Participant Failures

- Failure @ INITIAL
  - Abort upon recovery.
- Timeout @ READY
  - The coordinator has already been informed about the local decision.
  - Treat as Timeout @ READY and invoke the termination protocol.
- Timeout @ ABORT/COMMIT
  - Nothing to do

Recovery Protocol at Log Level

- If the DT log contains a *start-2PC* record, then S was the host of the coordinator. If it also contains a *commit* or *abort* record, then the coordinator had decided before the failure and it can resend its decision. If neither record is found, the coordinator can now unilaterally decide Abort by inserting an *abort* record in the DT log.
- If the DT log doesn't contain a *start-2PC* record, then S was the host of a participant. There are three cases to consider:
  (1) The DT log contains a *commit* or *abort* record. Then the participant had reached its decision before the failure.
  (2) The DT log does not contain a *vote-commit* record. Then either the participant failed before voting or voted *vote-abort* (but did not write an *abort* record before failing). It can therefore unilaterally abort by inserting an *abort* record in the DT log.
  (3) The DT log contains a *vote-commit* but no *commit* or *abort* record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the cooperative termination protocol.

Recovery Protocol at Log Level

- If the DT log contains a *start-2PC* record, then S was the host of the coordinator. If it also contains a *commit* or *abort* record, then the coordinator had decided before the failure and it can resend its decision. If neither record is found, the coordinator can now unilaterally decide Abort by inserting an *abort* record in the DT log.
- If the DT log doesn't contain a *start-2PC* record, then S was the host of a participant. There are three cases to consider:

  (1) The DT log contains a *commit* or *abort* record. Then the participant had reached its decision before the failure.

  (2) The DT log does not contain a *vote-commit* record. Then either the participant failed before voting or voted *vote-abort* (but did not write an *abort* record before failing). It can therefore unilaterally abort by inserting an *abort* record in the DT log.

  (3) The DT log contains a *vote-commit* but no *commit* or *abort* record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the cooperative termination protocol.

Recovery Protocol at Log Level

- If the DT log contains a *start-2PC* record, then S was the host of the coordinator. If it also contains a *commit* or *abort* record, then the coordinator had decided before the failure and it can resend its decision. If neither record is found, the coordinator can now unilaterally decide Abort by inserting an *abort* record in the DT log.
- If the DT log doesn't contain a *start-2PC* record, then S was the host of a participant. There are three cases to consider:

  (1) The DT log contains a *commit* or *abort* record. Then the participant had reached its decision before the failure.
  (2) The DT log does not contain a *vote-commit* record. Then either the participant failed before voting or voted *vote-abort* (but did not write an *abort* record before failing). It can therefore unilaterally abort by inserting an *abort* record in the DT log.
  (3) The DT log contains a *vote-commit* but no *commit* or *abort* record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the cooperative termination protocol.

Recovery Protocol at Log Level

- If the DT log contains a *start-2PC* record, then S was the host of the coordinator. If it also contains a *commit* or *abort* record, then the coordinator had decided before the failure and it can resend its decision. If neither record is found, the coordinator can now unilaterally decide Abort by inserting an *abort* record in the DT log.
- If the DT log doesn't contain a *start-2PC* record, then S was the host of a participant. There are three cases to consider:

  (1) The DT log contains a *commit* or *abort* record. Then the participant had reached its decision before the failure.
  (2) The DT log does not contain a *vote-commit* record. Then either the participant failed before voting or voted *vote-abort* (but did not write an *abort* record before failing). It can therefore unilaterally abort by inserting an *abort* record in the DT log.
  (3) The DT log contains a *vote-commit* but no *commit* or *abort* record. Then the participant failed while in its uncertainty period. It can try to reach a decision using the cooperative termination protocol.

DT log garbage collection

- A site cannot delete log records of a transaction T from its DT log before its recovery manager has processed Commit or Abort.
- The coordinator should not delete the records of transaction T from its DT log until it has received messages indicating that Commit or Abort has been processed at all other sites where T executed. To this end participants may send a final *ACK*-message when moving in their commit-state.

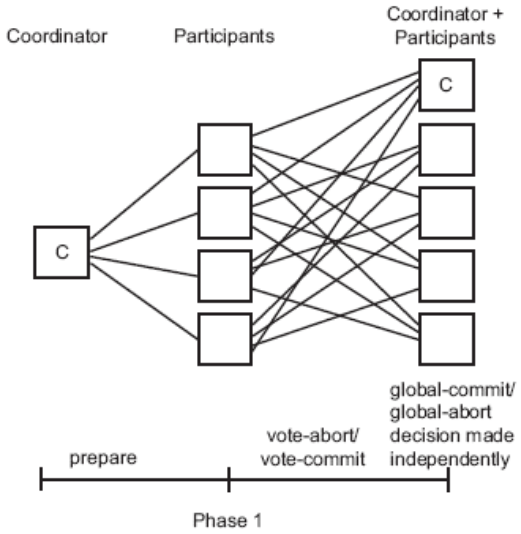
In the literature there are many optimizations described for 2PC - have a look into the Weikum-Vossen book, for example!

DT log garbage collection

- A site cannot delete log records of a transaction T from its DT log before its recovery manager has processed Commit or Abort.
- The coordinator should not delete the records of transaction T from its DT log until it has received messages indicating that Commit or Abort has been processed at all other sites where T executed. To this end participants may send a final *ACK*-message when moving in their commit-state.

In the literature there are many optimizations described for 2PC - have a look into the Weikum-Vossen book, for example!

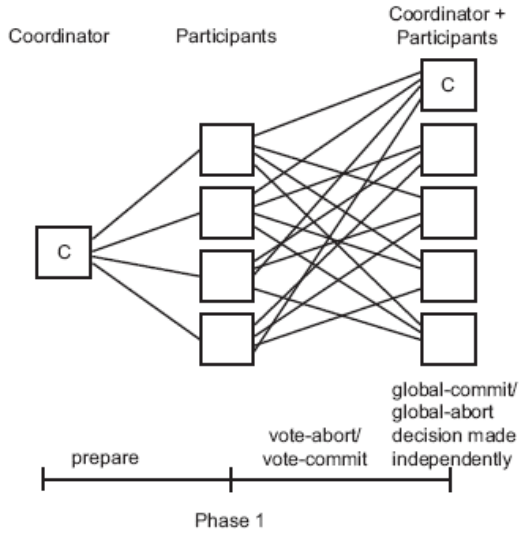
2-Phase-Commit Variants



decentralized 2PC

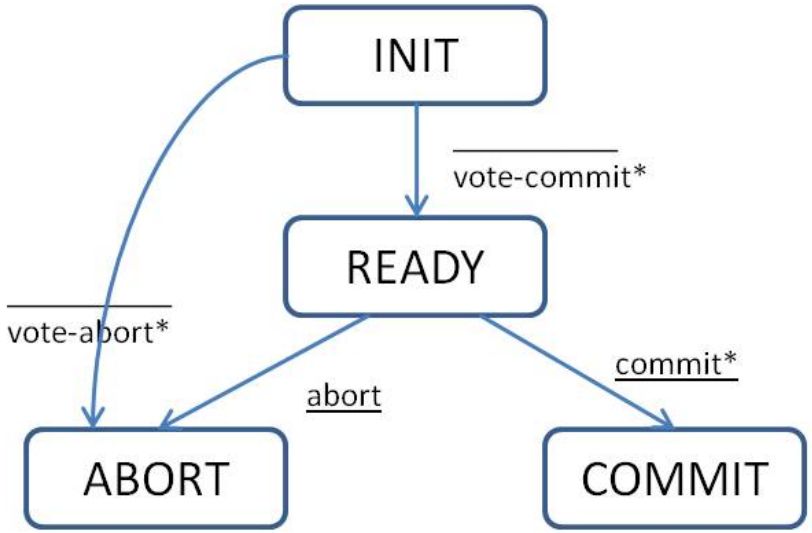
- Phase 1: Coordinator sends, depending on its vote, *vote-commit* or *vote-abort* to all participants.
- Phase 2a: When a participant receives *vote-abort* from the coordinator, it simply aborts. Otherwise it has received *vote-commit* and returns either *commit* or *abort* to coordinator and to all other participants. If it sends *abort*, it aborts its local computation.
- Phase 2b: After having received all votes, the coordinator and all participants have all votes available; if all are *commit*, they commit and otherwise abort.

2-Phase-Commit Variants

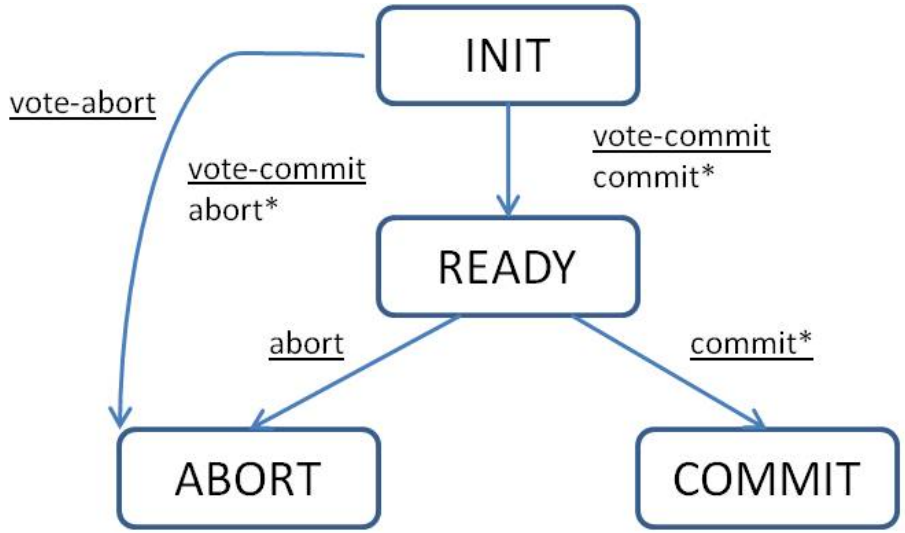


decentralized 2PC

- Phase 1: Coordinator sends, depending on its vote, *vote-commit* or *vote-abort* to all participants.
- Phase 2a: When a participant receives *vote-abort* from the coordinator, it simply aborts. Otherwise it has received *vote-commit* and returns either *commit* or *abort* to coordinator and to all other participants. If it sends *abort*, it aborts its local computation.
- Phase 2b: After having received all votes, the coordinator and all participants have all votes available; if all are *commit*, they commit and otherwise abort.



Coordinator

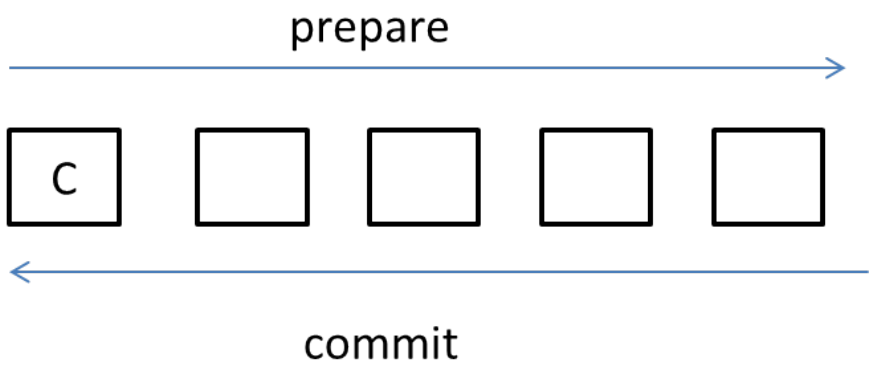


Participant

Notation: $\frac{message\ \ received}{message\ \ sent}$

msg*: message sent-to/received-from all

State transitions during decentralized 2PC.



linear 2PC: outer nodes

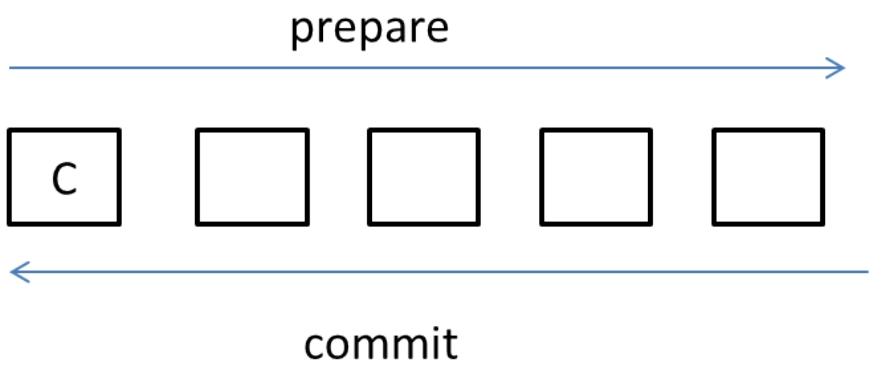
All processes are linearly ordered, w.l.o.g. $P_0, P_1, P_2, \ldots, P_n$, where P_0 is the coordinator. Communication is possible between neighbors.

(S1) When the protocol starts, P_0 sends message *vote-request* to its right neighbor.

(S4) If process P_n receives a message from its left neighbor:

- (1) If message is *vote-request*, then
  - (i) if its own vote is commit, it sends *commit* to its left neighbor and commit.
  - (ii) if its own vote is abort, it sends *abort* to its left neighbor and aborts.
- (2) If message is *abort*, then it aborts.

(S5) If process P_0 receives message *commit* from its right neighbor, it commits; if it receives message *abort*, it aborts.



linear 2PC: outer nodes

All processes are linearly ordered, w.l.o.g. $P_0, P_1, P_2, \ldots, P_n$, where P_0 is the coordinator. Communication is possible between neighbors.

(S1) When the protocol starts, P_0 sends message *vote-request* to its right neighbor.

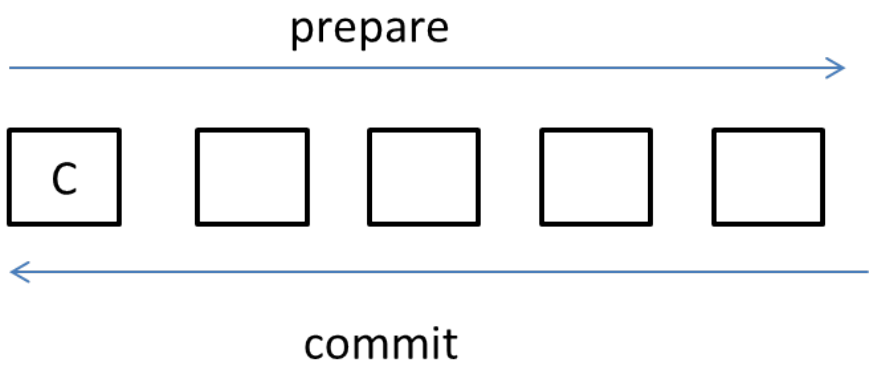
(S4) If process P_n receives a message from its left neighbor:

(1) If message is *vote-request*, then

(i) if its own vote is commit, it sends *commit* to its left neighbor and commit.
(ii) if its own vote is abort, it sends *abort* to its left neighbor and aborts.

(2) If message is *abort*, then it aborts.

(S5) If process P_0 receives message *commit* from its right neighbor, it commits; if it receives message *abort*, it aborts.



linear 2PC: outer nodes

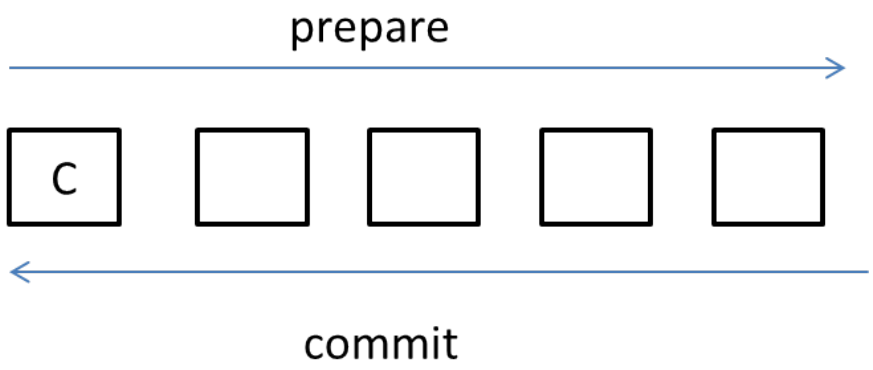
All processes are linearly ordered, w.l.o.g. $P_0, P_1, P_2, \ldots, P_n$, where P_0 is the coordinator. Communication is possible between neighbors.

(S1) When the protocol starts, P_0 sends message *vote-request* to its right neighbor.

(S4) If process P_n receives a message from its left neighbor:

- (1) If message is *vote-request*, then
  - (i) if its own vote is commit, it sends *commit* to its left neighbor and commit.
  - (ii) if its own vote is abort, it sends *abort* to its left neighbor and aborts.
- (2) If message is *abort*, then it aborts.

(S5) If process P_0 receives message *commit* from its right neighbor, it commits; if it receives message *abort*, it aborts.



linear 2PC: outer nodes

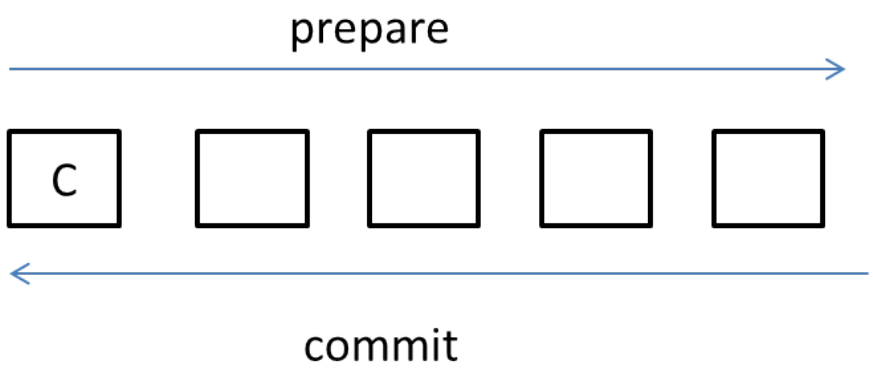
All processes are linearly ordered, w.l.o.g. $P_0, P_1, P_2, \ldots, P_n$, where P_0 is the coordinator. Communication is possible between neighbors.

(S1) When the protocol starts, P_0 sends message *vote-request* to its right neighbor.

(S4) If process P_n receives a message from its left neighbor:

- (1) If message is *vote-request*, then
  - (i) if its own vote is commit, it sends *commit* to its left neighbor and commit.
  - (ii) if its own vote is abort, it sends *abort* to its left neighbor and aborts.
- (2) If message is *abort*, then it aborts.

(S5) If process P_0 receives message *commit* from its right neighbor, it commits; if it receives message *abort*, it aborts.



linear 2PC - inner nodes

(S2) If process P_i, $1 \leq i < n$, receives a message from its left neighbor:

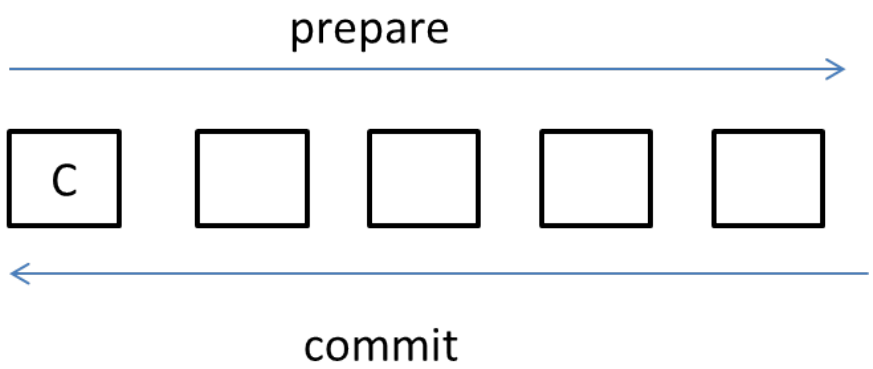
(1) If message is *vote-request*, then

(i) if its own vote is commit, it sends *vote-request* to its right neighbor.
(ii) otherwise, it sends *abort* to its left and right neighbors and aborts.

(2) If message is *abort*, then it sends *abort* to its right neighbor and aborts.

(S3) If process $P_i, 1 \leq i < n$, receives a message from its right neighbor:

(1) If message is *commit*, then it sends *commit* to its left neighbor and commits.
(2) If message is *abort*, then it sends *abort* to its left neighbor and aborts.



linear 2PC - inner nodes

(S2) If process P_i, $1 \leq i < n$, receives a message from its left neighbor:

(1) If message is *vote-request*, then

(i) if its own vote is commit, it sends *vote-request* to its right neighbor.

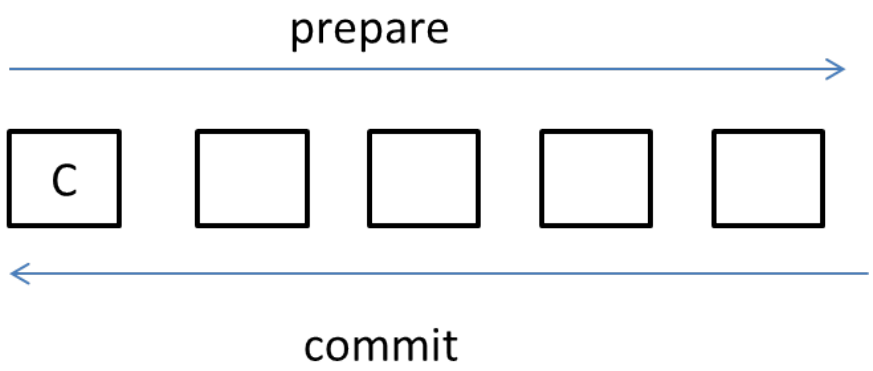
(ii) otherwise, it sends *abort* to its left and right neighbors and aborts.

(2) If message is *abort*, then it sends *abort* to its right neighbor and aborts.

(S3) If process $P_i, 1 \leq i < n$, receives a message from its right neighbor:

(1) If message is *commit*, then it sends *commit* to its left neighbor and commits.

(2) If message is *abort*, then it sends *abort* to its left neighbor and aborts.



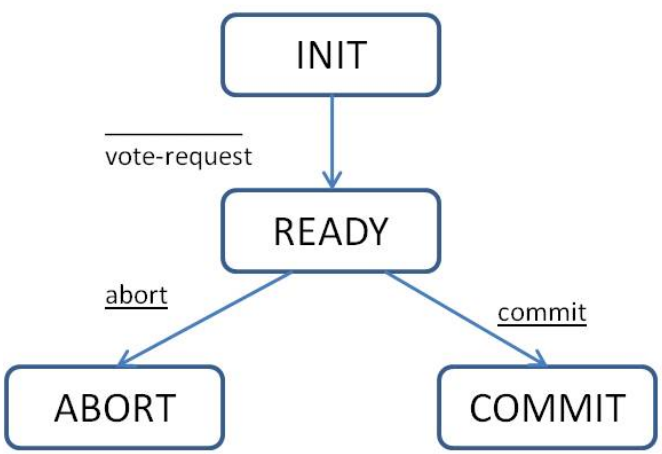
linear 2PC - inner nodes

(S2) If process P_i, $1 \leq i < n$, receives a message from its left neighbor:

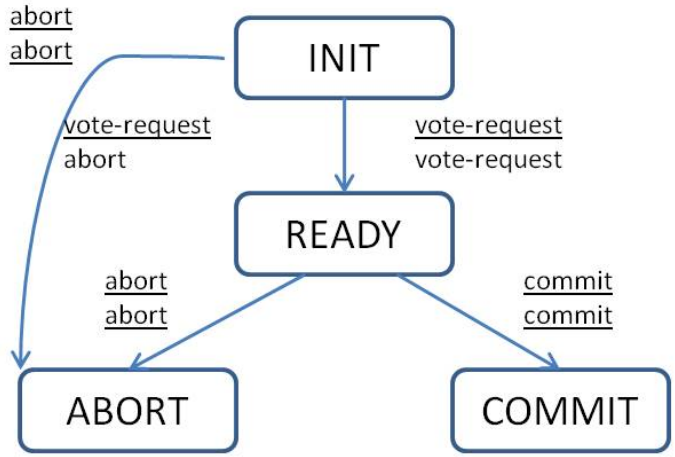
- (1) If message is *vote-request*, then
  - (i) if its own vote is commit, it sends *vote-request* to its right neighbor.
  - (ii) otherwise, it sends *abort* to its left and right neighbors and aborts.
- (2) If message is *abort*, then it sends *abort* to its right neighbor and aborts.

(S3) If process $P_i, 1 \leq i < n$, receives a message from its right neighbor:

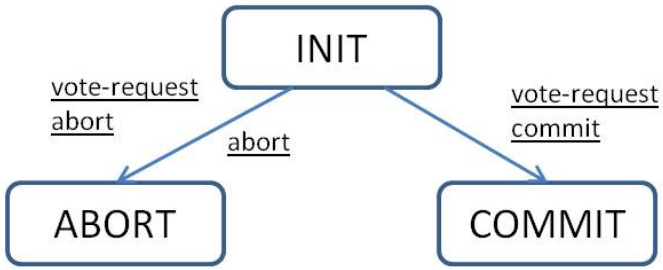
- (1) If message is *commit*, then it sends *commit* to its left neighbor and commits.
- (2) If message is *abort*, then it sends *abort* to its left neighbor and aborts.



Leftmost participant



Participant



Rightmost participant

Notation: $\frac{\textit{message received}}{\textit{message sent}}$

State transitions during linear 2PC.

Analysis of 2PC

Correctness of 2 PC

- **Agreement**: Every node agrees on the value proposed by the coordinator if and only if it is told by it. The coordinator sends the same value to everybody.
- **Validity**: A value is chosen that is proposed by at least one participant
- **Termination**: If nodes never fail, the protocol with eventually terminate (even under asynchronous semantics).

2 PC and failures

2PC may be blocking even in case of only partial failures.

Analysis of 2PC

Correctness of 2 PC

- **Agreement**: Every node agrees on the value proposed by the coordinator if and only if it is told by it. The coordinator sends the same value to everybody.
- **Validity**: A value is chosen that is proposed by at least one participant
- **Termination**: If nodes never fail, the protocol with eventually terminate (even under asynchronous semantics).

2 PC and failures

2PC may be blocking even in case of only partial failures.

Efficiency Comparison

Message Complexity: How many messages are exchanged to reach a decision?
Time Complexity: How long does it take to reach the decision? As several messages can be send in parallel, the number of message exchange *rounds* is counted.

	Number of messages	Rounds of communication
centralized 2PC	$3n$	3
decentralized 2PC		
linear 2PC		

n participants.

How far is this from an optimal solution

Efficiency Comparison

Message Complexity: How many messages are exchanged to reach a decision?
Time Complexity: How long does it take to reach the decision? As several messages can be send in parallel, the number of message exchange *rounds* is counted.

	Number of messages	Rounds of communication
centralized 2PC decentralized 2PC linear 2PC	$3n$	3

n participants.

How far is this from an optimal solution

9.2. 2 3-Phase-Commit (3PC)

3PC: Unblock by 2PC by spreading decision knowledge

- The period between the moment a process votes `Yes` for commit and the moment it has received sufficient information to know the decision is called *uncertainty period*. During its uncertainty period a process is called *uncertain*.

NB: If any operational process is uncertain, then no process (whether operational or failed) can have decided to commit.

- As a consequence, if the operational sites discover that they all are uncertain, they can decide to abort, as the other failed process cannot have decided commit before.
- 3PL splits the commit/abort phase in two steps
  - First communicate the outcome to everyone (but not force them to commit)
  - Let them commit only after everyone knows the outcome
- In case of coordinator failure, participants know the outcome

9.2. 2 3-Phase-Commit (3PC)

3PC: Unblock by 2PC by spreading decision knowledge

- The period between the moment a process votes `Yes` for commit and the moment it has received sufficient information to know the decision is called *uncertainty period*. During its uncertainty period a process is called *uncertain*.

NB: If any operational process is uncertain, then no process (whether operational or failed) can have decided to commit.

- As a consequence, if the operational sites discover that they all are uncertain, they can decide to abort, as the other failed process cannot have decided commit before.
- 3PL splits the commit/abort phase in two steps
  - First communicate the outcome to everyone (but not force them to commit)
  - Let them commit only after everyone knows the outcome
- In case of coordinator failure, participants know the outcome

9.2. 2 3-Phase-Commit (3PC)

3PC: Unblock by 2PC by spreading decision knowledge

- The period between the moment a process votes `Yes` for commit and the moment it has received sufficient information to know the decision is called *uncertainty period*. During its uncertainty period a process is called *uncertain*.

NB: If any operational process is uncertain, then no process (whether operational or failed) can have decided to commit.

- As a consequence, if the operational sites discover that they all are uncertain, they can decide to abort, as the other failed process cannot have decided commit before.
- 3PL splits the commit/abort phase in two steps
  - First communicate the outcome to everyone (but not force them to commit)
  - Let them commit only after everyone knows the outcome
- In case of coordinator failure, participants know the outcome

9.2. 2 3-Phase-Commit (3PC)

3PC: Unblock by 2PC by spreading decision knowledge

- The period between the moment a process votes `Yes` for commit and the moment it has received sufficient information to know the decision is called *uncertainty period*. During its uncertainty period a process is called *uncertain*.

NB: If any operational process is uncertain, then no process (whether operational or failed) can have decided to commit.

- As a consequence, if the operational sites discover that they all are uncertain, they can decide to abort, as the other failed process cannot have decided commit before.
- 3PL splits the commit/abort phase in two steps
  - First communicate the outcome to everyone (but not force them to commit)
  - Let them commit only after everyone knows the outcome
- In case of coordinator failure, participants know the outcome

3-phase commit (3PC) protocol

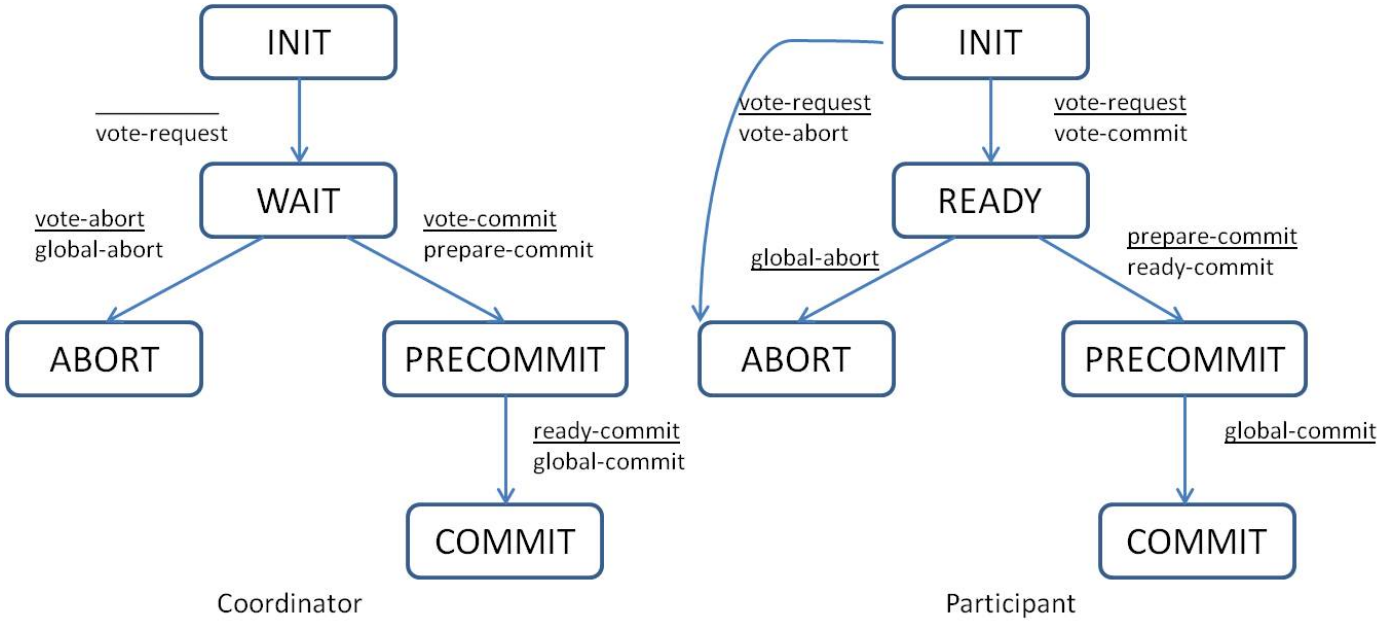
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *prepare-commit* to all participants, otherwise it sends *global-abort*, and halts.
- Phase 2b: Each participant that voted *vote-commit* waits for *prepare-commit*, or waits for *global-abort* after which it halts. If *prepare-commit* is received, the process replies *ready-commit* and therefore the coordinator knows that this process is no longer uncertain.
- Phase 3a: (Prepare to commit) Coordinator waits until all participants have sent *ready-commit*, and then sends *global-commit* to all.
- Phase 3b: (Prepare to commit) Participant waits for *global-commit* and then commits. It knows that no other process is uncertain and thus commits without violating NB.

3-phase commit (3PC) protocol

- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *prepare-commit* to all participants, otherwise it sends *global-abort*, and halts.
- Phase 2b: Each participant that voted *vote-commit* waits for *prepare-commit*, or waits for *global-abort* after which it halts. If *prepare-commit* is received, the process replies *ready-commit* and therefore the coordinator knows that this process is no longer uncertain.
- Phase 3a: (Prepare to commit) Coordinator waits until all participants have sent *ready-commit*, and then sends *global-commit* to all.
- Phase 3b: (Prepare to commit) Participant waits for *global-commit* and then commits. It knows that no other process is uncertain and thus commits without violating NB.

3-phase commit (3PC) protocol

- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *prepare-commit* to all participants, otherwise it sends *global-abort*, and halts.
- Phase 2b: Each participant that voted *vote-commit* waits for *prepare-commit*, or waits for *global-abort* after which it halts. If *prepare-commit* is received, the process replies *ready-commit* and therefore the coordinator knows that this process is no longer uncertain.
- Phase 3a: (Prepare to commit) Coordinator waits until all participants have sent *ready-commit*, and then sends *global-commit* to all.
- Phase 3b: (Prepare to commit) Participant waits for *global-commit* and then commits. It knows that no other process is uncertain and thus commits without violating NB.



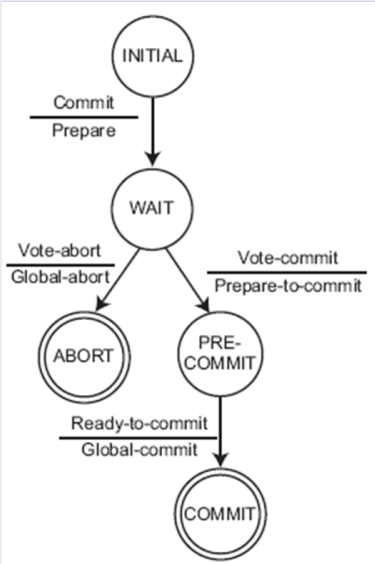
Notation: $\frac{\textit{message received}}{\textit{message sent}}$

State transitions during 3PC.

Termination Protocol: Coordinator Timeouts

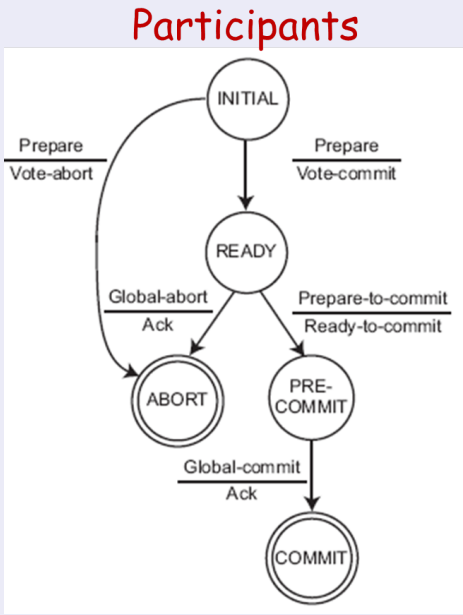
- Timeout @ PRECOMMIT
  - Participants must be at least in READY.
  - Move all the participants to PRECOMMIT.
  - Globally commit
- Timeout @ ABORT / COMMIT
  - Ignore and treat as completed
  - Participants are either at PRECOMMIT or READY and they can continue to termination.

Coordinator



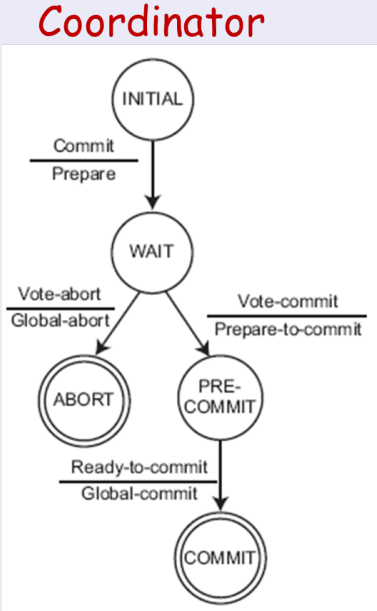
Termination Protocol: Participant Timeouts

- Timeout @ INITIAL
  - Coordinator must have failed at INITIAL.
  - Can abort.
  - If Prepare arrives later, can either Vote-abort or ignore it (i.e., let the coordinator timeout @WAIT).
- Timeout @ READY
  - Voted to commit, but does not know the coordinator's global decision.
  - Elect a new coordinator and terminate using a special protocol.
- Timeout @ PRECOMMIT
  - Same as Timeout @ READY



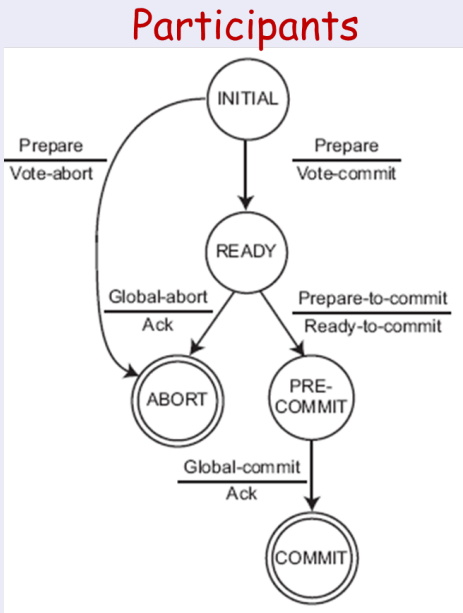
Recovery Protocol: Coordinator Failures

- Failure @ INITIAL
  - Start the commit process upon recovery.
- Failure @ WAIT
  - The participants may have elected a new coordinator and terminated.
  - Ask around for the fate of the transaction
- Failure @ PRECOMMIT
  - Ask around for the fate of the transaction
- Failure @ ABORT / COMMIT
  - If all ACKs have been received, nothing to do.
  - Else, invoke the termination protocol.



Recovery Protocol: Participant Failures

- Failure @ INITIAL
  - Abort upon recovery.
- Timeout @ READY
  - The coordinator has already been informed about the local decision.
  - Upon Recovery, ask around
- Timeout @ PRECOMMIT
  - Ask around how the others have terminated the transaction
- Timeout @ ABORT/COMMIT
  - Nothing to do



Analysis of 3PC

Correctness of 3 PC

Given the incresased complexity, not a full proof

- **Validity**: A value is chosen that is proposed by at least one participant
- **Termination**:
  - If nodes never fail, the protocol with eventually terminate (even under asynchronous semantics).
  - If nodes fail before reaching a commit consensus, the protocol will terminate with abort
  - If nodes fail after a commit consensus, a new coordinator will recover the commit decision

Are we done? Did we overcome FLP?

3 PC and network splits

Consider the case in which

- the network is split in two during the second phase (prepare to commit)
- and the coordinator fails

Further assume that

- on one partition (A), all participants received the "prepare to commit"
- on the other (B), none

Each side will pick a coordinator, which in turn contacts the available participants

- Partition (A) $\rightarrow$ commit
- Partition (B) $\rightarrow$ abort

3PC can become unsafe.