# 5. Reliability

## Crash and crash recovery

- By *crash* all kinds of failures are denoted that bring down a server and cause all data in volatile memory to be lost (*soft crash*), but leave all data on stable secondary storage intact, i.e. not a (*hard crash*).
- A *crash recovery* algorithm restarts the server and brings its permanent data back to its most recent, consistent state, thereby ensuring atomicity and durability of transactions.
  - All updates of committed transactions are included: *redo recovery*,
  - No updates of uncommitted or aborted transactions are included: *undo recovery*.
- This functionality is called *failure resilience*, or *fault tolerance*, respectively *reliability*.

Today, a soft crash typically is produced by a so called *Heisenbug*, an error which cannot easily be eliminated by more extensive software testing because it appears in a „nondeterministic“ manner often related to concurrent threads or high system load.

# 5. Reliability

## Crash and crash recovery

- By *crash* all kinds of failures are denoted that bring down a server and cause all data in volatile memory to be lost (*soft crash*), but leave all data on stable secondary storage intact, i.e. not a (*hard crash*).
- A *crash recovery* algorithm restarts the server and brings its permanent data back to its most recent, consistent state, thereby ensuring atomicity and durability of transactions.
  - All updates of committed transactions are included: *redo recovery*,
  - No updates of uncommitted or aborted transactions are included: *undo recovery*.
- This functionality is called *failure resilience*, or *fault tolerance*, respectively *reliability*.

Today, a soft crash typically is produced by a so called *Heisenbug*, an error which cannot easily be eliminated by more extensive software testing because it appears in a „nondeterministic“ manner often related to concurrent threads or high system load.

# 5. Reliability

## Crash and crash recovery

- By *crash* all kinds of failures are denoted that bring down a server and cause all data in volatile memory to be lost (*soft crash*), but leave all data on stable secondary storage intact, i.e. not a (*hard crash*).
- A *crash recovery* algorithm restarts the server and brings its permanent data back to its most recent, consistent state, thereby ensuring atomicity and durability of transactions.
  - All updates of committed transactions are included: *redo recovery*,
  - No updates of uncommitted or aborted transactions are included: *undo recovery*.
- This functionality is called *failure resilience*, or *fault tolerance*, respectively *reliability*.

Today, a soft crash typically is produced by a so called *Heisenbug*, an error which cannot easily be eliminated by more extensive software testing because it appears in a „nondeterministic" manner often related to concurrent threads or high system load.

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time

### Recovery performance and system availability

MTBF: *mean time between failure*

MTTR: *mean time to repair*

*Availability*: probability for a server to be ready to serve:

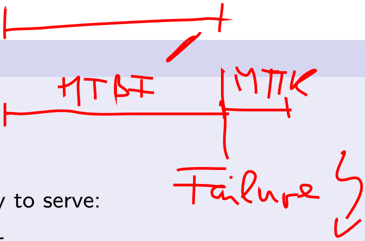
$$\frac{MTBF}{MTBF + MTTR}$$

### Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%

$\Longrightarrow$ Fast recovery is the key to high availability!

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time




## Recovery performance and system availability

MTBF: *mean time between failure*

MTTR: *mean time to repair*

*Availability*: probability for a server to be ready to serve:

$$\frac{MTBF}{MTBF + MTTR}$$

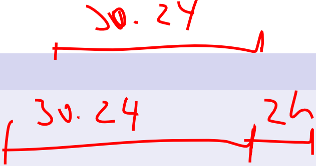
## Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%

$\Longrightarrow$ Fast recovery is the key to high availability!

During crash recovery after a system failure, a server and its data are unavailable to clients. Goal: minimize recovery time

### Recovery performance and system availability




MTBF: *mean time between failure*

MTTR: *mean time to repair*

*Availability*: probability for a server to be ready to serve:

$$\frac{MTBF}{MTBF + MTTR}$$

### Examples

- Server fails once a month and takes 2 hours to recover: availability of 99.7%
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%

$\Longrightarrow$ Fast recovery is the key to high availability!

## Today: global recovery

- Global recovery designed for distributed executions: commit coordination

## Not covered in this course: local recovery

- Local recovery designed for each site: advanced crash recovery algorithms[1]

2PC / 3PC

[1] additional literature: Concurrency Control and Recovery in Database Systems. Bernstein, Hadzilacos and Goodman, 1987 Addison Wesley. Download: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

# 5.1. Commit coordination

The coordination problem during the commit-phase.

Given a computation defined by a set of subtransactions each running at a seperate server. How can we ensure that either all subtransactions commit to the final result, or none of them do (atomicity)? To reach a unique decision among the subtransactions, a *coordinator* process is initiated running at one of the involved servers.

- A subtransaction may be aborted even after having reached the end because of some faulty other subtransaction.
- Therefore, during its commit-phase each subtransaction must figure out whether it and all the others will finish their commit-phase successfully.
- If this is not possible, all subtransactions have to be aborted.
- Reaching a global commit must be achieved by passing messages.

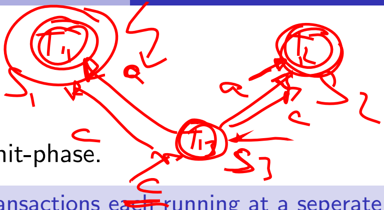
# 5.1. Commit coordination

The coordination problem during the commit-phase.

Given a computation defined by a set of subtransactions each running at a seperate server. How can we ensure that either all subtransactions commit to the final result, or none of them do (atomicity)? To reach a unique decision among the subtransactions, a *coordinator* process is initiated running at one of the involved servers.

- A subtransaction may be aborted even after having reached the end because of some faulty other subtransaction.
- Therefore, during its commit-phase each subtransaction must figure out whether it and all the others will finish their commit-phase successfully.
- If this is not possible, all subtransactions have to be aborted.
- Reaching a global commit must be achieved by passing messages.

# 5.1. Commit coordination

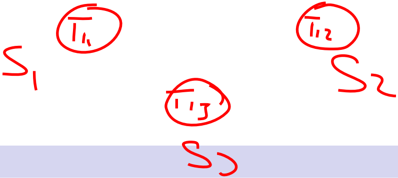


The coordination problem during the commit-phase.

Given a computation defined by a set of subtransactions each running at a seperate server. How can we ensure that either all subtransactions commit to the final result, or none of them do (atomicity)? To reach a unique decision among the subtransactions, a *coordinator* process is initiated running at one of the involved servers.

- A subtransaction may be aborted even after having reached the end because of some faulty other subtransaction.
- Therefore, during its commit-phase each subtransaction must figure out whether it and all the others will finish their commit-phase successfully.
- If this is not possible, all subtransactions have to be aborted.
- Reaching a global commit must be achieved by passing messages.

# 2-Phase-Commit Protocol




## how it works

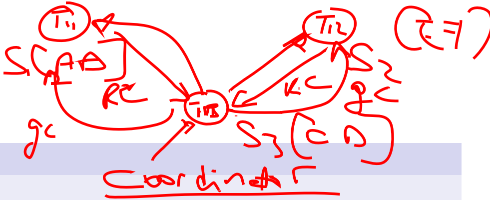
- The client who inititated the computation acts as coordinator; processes required to commit are the participants.
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly.

# 2-Phase-Commit Protocol

## how it works

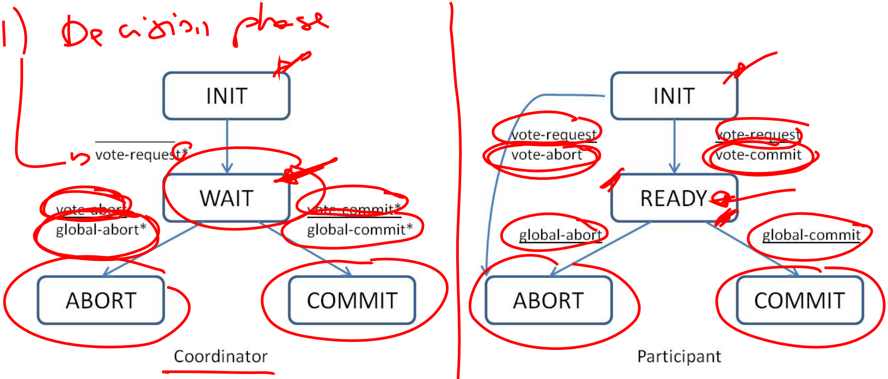
- The client who inititated the computation acts as coordinator; processes required to commit are the participants.
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, it sends *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly.

# 2-Phase-Commit Protocol



## how it works

- The client who inititated the computation acts as coordinator; processes required to commit are the participants.
- 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 *global-commit* to all participants, otherwise it sends *global-abort*.
- Phase 2b: Each participant waits for *global-commit* or *global-abort* and reacts accordingly.




Notation: $\frac{\textit{message received}}{\textit{message sent}}$
*msg*$^*$: message sent-to/received-from all

State transitions during 2PC.

# Distributed Transaction Log: DT log at each site

## DT log maintenance

(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: DT log at each site

## DT log maintenance

(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: DT log at each site

## DT log maintenance

(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: DT log at each site

### DT log maintenance

(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: DT log at each site

## DT log maintenance

(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.

## Problems which may occur during 2PC: processes being blocked

- Participant is blocked in the *init-state*.

  A participant waits for a *vote-request*-message. As no decision for a global commit has been taken, the participant can abort without any harm.

- Cordinator is blocked in the *wait-state*.

  The coordinator waits for *vote-abort* and *vote-commit* messages. As no decision for a global commit has been taken so far, the coordinator can send *global-abort* to all participants having sent *vote-commit* so far.

- Participant is blocked in the *ready-state*.

  Participant, say $P$, has sent *vote-commit* and is waiting for the coordinators reply. $P$ does not know what to do, it cannot commit, because the coordinator did not respond, it cannot abort, because it voted for commit.

  Participant $P$ may contact another participant $Q$ to clarify the situation by executing the *cooperative termination protocol*:

| State of $Q$ | Action by $P$, $Q$ |
|---|---|
| COMMIT | $P$: Make transition to COMMIT |
| ABORT | $P$: Make transition to ABORT |
| INIT | $P$, $Q$: Make transition to ABORT |
| READY | $P$: Contact another participant |

## Problems which may occur during 2PC: processes being blocked

- Participant is blocked in the *init-state*.

  A participant waits for a *vote-request*-message. As no decision for a global commit has been taken, the participant can abort without any harm.

- Cordinator is blocked in the *wait-state*.

  The coordinator waits for *vote-abort* and *vote-commit* messages. As no decision for a global commit has been taken so far, the coordinator can send *global-abort* to all participants having sent *vote-commit* so far.

- Participant is blocked in the *ready-state*.

  Participant, say $P$, has sent *vote-commit* and is waiting for the coordinators reply. $P$ does not know what to do, it cannot commit, because the coordinator did not respond, it cannot abort, because it voted for commit.

  Participant $P$ may contact another participant $Q$ to clarify the situation by executing the *cooperative termination protocol*:

| State of $Q$ | Action by $P, Q$ |
|---|---|
| COMMIT | $P$: Make transition to COMMIT |
| ABORT | $P$: Make transition to ABORT |
| INIT | $P, Q$: Make transition to ABORT |
| READY | $P$: Contact another participant |

## Problems which may occur during 2PC: processes being blocked

- Participant is blocked in the *init-state*.

  A participant waits for a *vote-request*-message. As no decision for a global commit has been taken, the participant can abort without any harm.

- Cordinator is blocked in the *wait-state*.

  The coordinator waits for *vote-abort* and *vote-commit* messages. As no decision for a global commit has been taken so far, the coordinator can send *global-abort* to all participants having sent *vote-commit* so far.

- Participant is blocked in the *ready-state*.

  Participant, say $P$, has sent *vote-commit* and is waiting for the coordinators reply. $P$ does not know what to do, it cannot commit, because the coordinator did not respond, it cannot abort, because it voted for commit.

  Participant $P$ may contact another participant $Q$ to clarify the situation by executing the *cooperative termination protocol*:

| State of $Q$ | Action by $P, Q$ |
|---|---|
| COMMIT | $P$: Make transition to COMMIT |
| ABORT | $P$: Make transition to ABORT |
| INIT | $P, Q$: Make transition to ABORT |
| READY | $P$: Contact another participant |

## Problems which may occur during 2PC: processes being blocked

- Participant is blocked in the *init-state*

  A participant waits for a *vote-request*-message. As no decision for a global commit has been taken, the participant can abort without any harm.

- Cordinator is blocked in the *wait-state*.

  The coordinator waits for *vote-abort* and *vote-commit* messages. As no decision for a global commit has been taken so far, the coordinator can send *global-abort* to all participants having sent *vote-commit* so far.

- Participant is blocked in the *ready-state*.

  Participant, say $P$, has sent *vote-commit* and is waiting for the coordinators reply. $P$ does not know what to do, it cannot commit, because the coordinator did not respond, it cannot abort, because it voted for commit.

  Participant $P$ may contact another participant $Q$ to clarify the situation by executing the *cooperative termination protocol*:

| State of $Q$ | Action by $P$, $Q$ |
|---|---|
| COMMIT | $P$: Make transition to COMMIT |
| ABORT | $P$: Make transition to ABORT |
| INIT | $P$, $Q$: Make transition to ABORT |
| READY | $P$: Contact another participant |

## Site S recovers from a failure

- 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.

## Site S recovers from a failure

- 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.

## Site S recovers from a failure

- 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.

## Site S recovers from a failure

- 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.

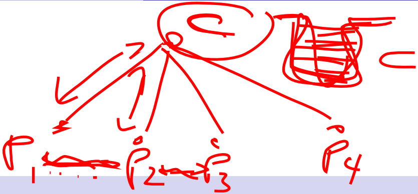
In the literature there are many optimizations described for 2PC - have a look into the Weikum-Vossen book, for example!

# 2-Phase-Commit Variants

## decentralized 2PC

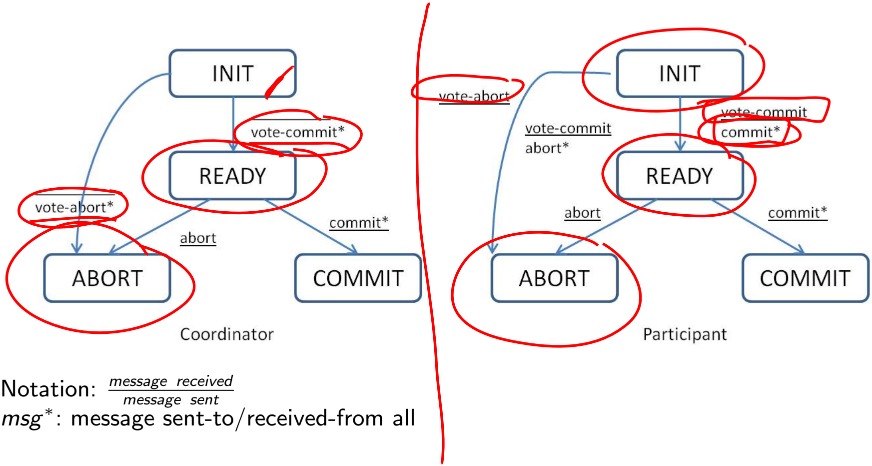
- Phase 1: Coordinator sends, depending on its vote, *vote-commit* or *vote-abort* to all participants.
- Phase 2a: When a participant receives *vote-abort* from the coordinator, it simply aborts. Otherwise it has received *vote-commit* and returns either *commit* or *abort* to coordinator and to all 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.




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

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

State transitions during decentralized 2PC.

## linear 2PC

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.

(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.

(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

All processes are linearly ordered, w.l.o.g. $P_0, P_1, P_2, \dots, P_n$, 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.

(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.

(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

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.

(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.

(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

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.

(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.

(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

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.

(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.

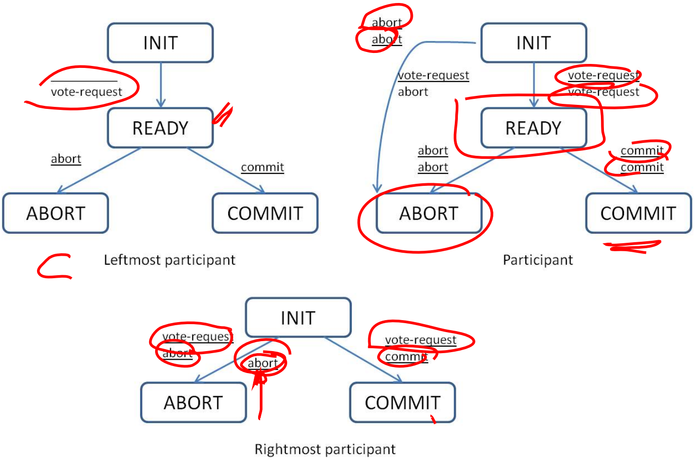
(S3) If process $P_i, 1 \leq i < n$, receives a message from its right neighbor:

- (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.

(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.




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

State transitions during linear 2PC.

## 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 | $n^2 + n$ | 2 |
| linear 2PC | $2n$ | $2n$ |

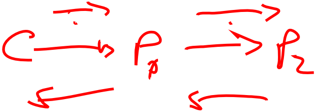
$n$ participants.

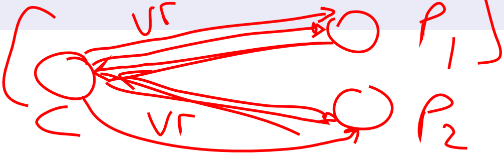
## Comparison

*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 | $n^2 + n$ | 2 |
| linear 2PC | $2n$ | $2n$ |

$n$ participants.





## Under which assumptions does 2PC work correctly, i.e. will not block?

### Possible failures

Assumption: A site is either working correctly (is *operational*) or not working at all (is *down*).

- partial site failure:

  Some sites are operational, some sites are down.

- total site failure:

  All sites are down.

- communication failure:

  Some site $A$ is not able to communicate with some site $B$, even though none of them is down. This may be due to broken communication links or site failures.

2PC may be blocking even in case of only partial failures. $\Longrightarrow$ 3PC

## Under which assumptions does 2PC work correctly, i.e. will not block?

### Possible failures

Assumption: A site is either working correctly (is *operational*) or not working at all (is *down*).

- partial site failure:

  Some sites are operational, some sites are down.

- total site failure:

  All sites are down.

- communication failure:

  Some site $A$ is not able to communicate with some site $B$, even though none of them is down. This may be due to broken communication links or site failures.

2PC may be blocking even in case of only partial failures. $\Longrightarrow$ 3PC

## Under which assumptions does 2PC work correctly, i.e. will not block?

### Possible failures

Assumption: A site is either working correctly (is *operational*) or not working at all (is *down*).

- partial site failure:

  Some sites are operational, some sites are down.

- total site failure:

  All sites are down.

- communication failure:

  Some site $A$ is not able to communicate with some site $B$, even though none of them is down. This may be due to broken communication links or site failures.

2PC may be blocking even in case of only partial failures. $\Longrightarrow$ 3PC

## Under which assumptions does 2PC work correctly, i.e. will not block?

### Possible failures

Assumption: A site is either working correctly (is *operational*) or not working at all (is *down*).

- partial site failure:

  Some sites are operational, some sites are down.

- total site failure:

  All sites are down.

- communication failure:

  Some site $A$ is not able to communicate with some site $B$, even though none of them is down. This may be due to broken communication links or site failures.

2PC may be blocking even in case of only partial failures. $\Longrightarrow$ 3PC

# 3-Phase-Commit Protocol

**In contrast to 2PC, 3PC tolerates partial failures by guaranteeing the property NB**

- 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.

# 3-Phase-Commit Protocol

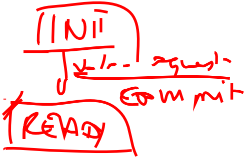
In contrast to 2PC, 3PC tolerates partial failures by guaranteeing the property NB

- 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.

# 3-Phase-Commit Protocol




In contrast to 2PC, 3PC tolerates partial failures by guaranteeing the property NB

- 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.

## 3-phase commit (3PC) protocol

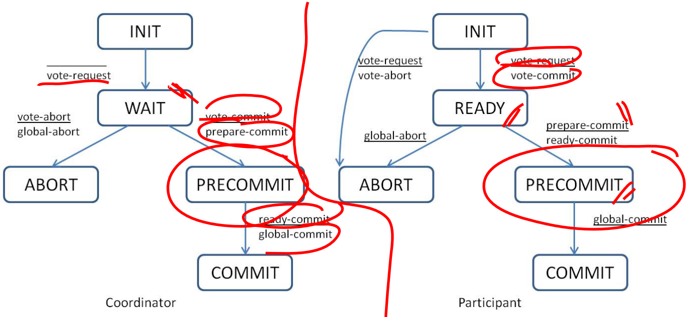
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, 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.

To proof correctness and termination of 3PC is difficult. Let's look at one case to demonstrate what could happen.

**If a participant *P* times out in state PRECOMMIT, why can't it ignore the timeout and simply decide for commit?**

- The coordinator may have failed after having sent a *prepare-commit*-messsage to *P* but before sending it to some other *Q*.
- Thus *P* times out outside its uncertainty period while *Q* will time out inside its uncertainty period.
- Thus, committing of *P* would violate NB.
- Therefore, before committing, *P* must assure, that all operational participants have received a *prepare-commit*-messsage and therefore moved outside their uncertainty period.
- To this end a dedicated termination protocol has to be applied.

To proof correctness and termination of 3PC is difficult. Let's look at one case to demonstrate what could happen.

**If a participant *P* times out in state PRECOMMIT, why can't it ignore the timeout and simply decide for commit?**

- The coordinator may have failed after having sent a *prepare-commit*-messsage to *P* but before sending it to some other *Q*.
- Thus *P* times out outside its uncertainty period while *Q* will time out inside its uncertainty period.
- Thus, committing of *P* would violate NB.
- Therefore, before committing, *P* must assure, that all operational participants have received a *prepare-commit*-messsage and therefore moved outside their uncertainty period.
- To this end a dedicated termination protocol has to be applied.

To proof correctness and termination of 3PC is difficult. Let's look at one case to demonstrate what could happen.

If a participant $P$ times out in state PRECOMMIT, why can't it ignore the timeout and simply decide for commit?

- The coordinator may have failed after having sent a *prepare-commit*-messsage to $P$ but before sending it to some other $Q$.
- Thus $P$ times out outside its uncertainty period while $Q$ will time out inside its uncertainty period.
- Thus, committing of $P$ would violate NB.
- Therefore, before committing, $P$ must assure, that all operational participants have received a *prepare-commit*-messsage and therefore moved outside their uncertainty period.
- To this end a dedicated termination protocol has to be applied.

### Termination rules

By applying an election protocol among all operational processes determine a new coordinator.

(1) If some process is Aborted, the coordinator decides Abort, sends ABORT messages to all participants, and stops.

(2) If some process is Committed[2], the coordinator decides Commit, sends COMMIT messages to all participants, and stops.

(3) If all processes that reported their state are Uncertain, the coordinator decides Abort, sends ABORT messages to all participants, and stops.

(4) If some process is Committable but none is Committed, the coordinator first sends PRE-COMMIT messages to all processes that reported Uncertain, and waits for acknowledgments from these processes. After having received these acknowledgments the coordinator decides Commit, sends COMMIT messages to all processes, and stops.

Processes may fail during the termination protocol! The protocol then has to be repeated - either it will be finished by some coordinator or all processes will fail.

[2] This may have happened in a previous round of the termination protocol.