10. 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*[1], 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.

[1] A notion coined by Jim Gray.

10. 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*[1], 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.

[1] A notion coined by Jim Gray.

10. 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*[1], 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.

[1] A notion coined by Jim Gray.

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%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

$\Longrightarrow$ 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%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

$\Longrightarrow$ 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%, downtime of 26 h a year.
- Server fails once every 48 h and takes 30 sec to recover: availability of 99.98%, downtime of 105 min a year.

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

Outlook[2]

- Local recovery designed for each site: advanced crash recovery algorithms,
- Global recovery designed for distributed executions: commit coordination.

[2] 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

10.1. 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 subtransaction have to be aborted.
- Reaching a global commit must be achieved by passing messages.

10.1. 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 subtransaction have to be aborted.
- Reaching a global commit must be achieved by passing messages.

10.1. 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 subtransaction 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

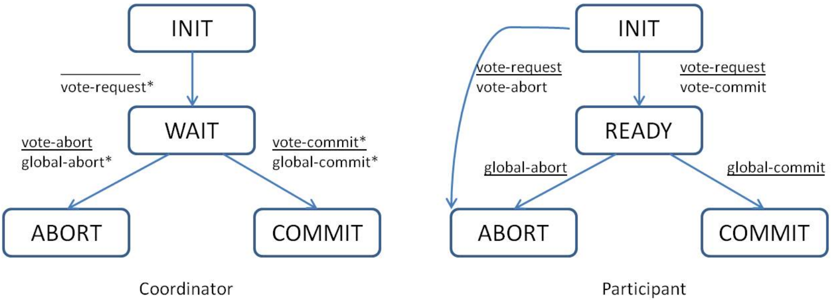
how it works

- 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 respont, 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 respont, 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 respont, 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 respont, 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

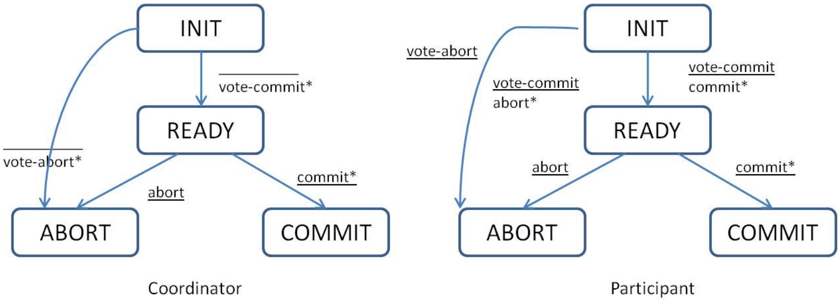
decentralized 2PC

- 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. $P_0, P_1, P_2, \ldots, P_n$, 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.

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.

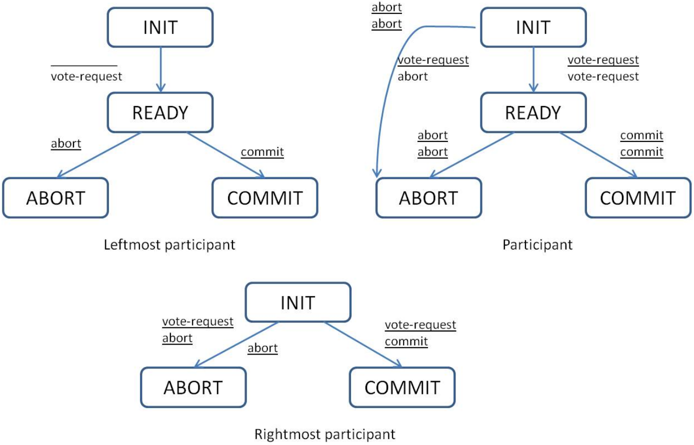
(S3) If process $P_i, 1 \leq i < n$, receives a message from its right neighbor:

- (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		
linear 2PC		

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		
linear 2PC		

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*).[3]

- 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] Also called *fail-stop*, because sites fail only by stopping, i.e. don't work incorrectly. Contrast this with Byzantine failures!

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*).[3]

- 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] Also called *fail-stop*, because sites fail only by stopping, i.e. don't work incorrectly. Contrast this with Byzantine failures!

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*).[3]

- 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]Also called *fail-stop*, because sites fail only by stopping, i.e. don't work incorrectly. Contrast this with Byzantine failures!

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*).[3]

- 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] Also called *fail-stop*, because sites fail only by stopping, i.e. don't work incorrectly. Contrast this with Byzantine failures!

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

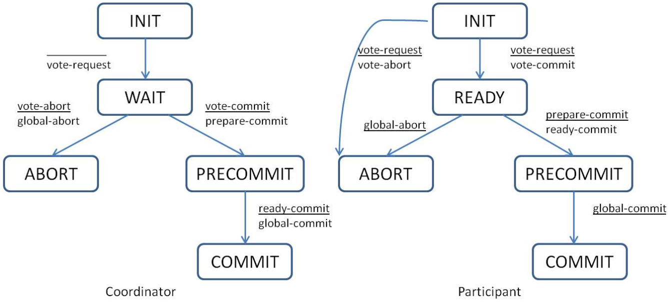
- Phase 1a: Coordinator sends *vote-request* to participants.
- Phase 1b: When participant receives *vote-request* it returns either *vote-commit* or *vote-abort* to coordinator. If it sends *vote-abort*, it aborts its local computation.
- Phase 2a: Coordinator collects all votes; if all are *vote-commit*, 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 determin 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[4], 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.

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

10.2. Crash recovery

System architecture

- *Stable database*

  Set of pages in stable storage, typically magnetic disks or SSDs (*solid state drives*).
- *Database cache*

  Dynamically evolving subset of the stable database copied into volatile memory.
- *Stable log*

  Set of *log entries* describing the history of updates on the cached database and possible additional bookkeeping records on the system history, prerequisites for redo and undo.
- *Log buffer*

  Data structure in volatile memory serving as a buffer in writing log entries to the stable log.

10.2. Crash recovery

System architecture

- *Stable database*

  Set of pages in stable storage, typically magnetic disks or SSDs (*solid state drives*).
- *Database cache*

  Dynamically evolving subset of the stable database copied into volatile memory.
- *Stable log*

  Set of *log entries* describing the history of updates on the cached database and possible additional bookkeeping records on the system history, prerequisites for redo and undo.
- *Log buffer*

  Data structure in volatile memory serving as a buffer in writing log entries to the stable log.

Physical vs. physiological log entries

- Physical: a full page.
  - *after image*: new content
  - *before image*: old content
- Physiological:
  - old and new values of the byte range actually modified in the page,
  - operation describing the update on the page.

moreover:

- Transactions follow the S2PL- or SS2PL-versions of the 2PL-protocol:
  - S2PL (*strict 2PL*): write locks are held until a transaction terminates,
  - SS2PL (*strong 2PL*): read and write locks are held until a transaction terminates.
- Granularity of locking are pages or smaller units, e.g. tuples. Smaller units than pages imply special recovery considerations.
- A page is written to the stable database only then, when it has been written to the stable log before (*write-ahead-log rule*).

Physical vs. physiological log entries

- Physical: a full page.
    - *after image*: new content
    - *before image*: old content
- Physiological:
    - old and new values of the byte range actually modified in the page,
    - operation describing the update on the page.

moreover:

- Transactions follow the S2PL- or SS2PL-versions of the 2PL-protocol:
    - S2PL (*strict 2PL*): write locks are held until a transaction terminates,
    - SS2PL (*strong 2PL*): read and write locks are held until a transaction terminates.
- Granularity of locking are pages or smaller units, e.g. tuples. Smaller units than pages imply special recovery considerations.
- A page is written to the stable database only then, when it has been written to the stable log before (*write-ahead-log rule*).

Algorithms

- A page in the database cache may be replaced and written back to the database before the commit of the updating transaction (*steal*) or not ($\neg$ *steal*).
- A page is forced to be written back to the database for all committed transactions (*force*) or not ($\neg$ *force*).

Classification

	force	$\neg$force
$\neg$steal	no redo no undo	redo no undo
steal	no redo undo	redo undo

in the following: steal/$\neg$force

Data actions

Given transaction T and page number *pageno*.

- read(*pageno*, T)

  Pinning the page to a fixed virtual-memory address in the database cache, reading the page contents of *pageno* and finally unpinning the page.

- write(*pageno*, T)

  Pinning the page to a fixed virtual-memory address in the database cache, reading the page contents and finally declaring the page to be dirty, unpinning the page and writing the page (*physiological action*).

- full-write(*pageno*, T)

  A new value is assigned to all bytes of a page and then it is written (*physical action*).

- fetch(*pageno*)

  Copies the previously uncached page *pageno* from the stable database into the database cache.

- flush(*pageno*)

  Copies the cached page *pageno* to the stable database.

- force()

  Forces all log entries in the log buffer to the stable log.

Data actions

Given transaction T and page number *pageno*.

- read(*pageno*, T)

  Pinning the page to a fixed virtual-memory address in the database cache, reading the page contents of *pageno* and finally unpinning the page.
- write(*pageno*, T)

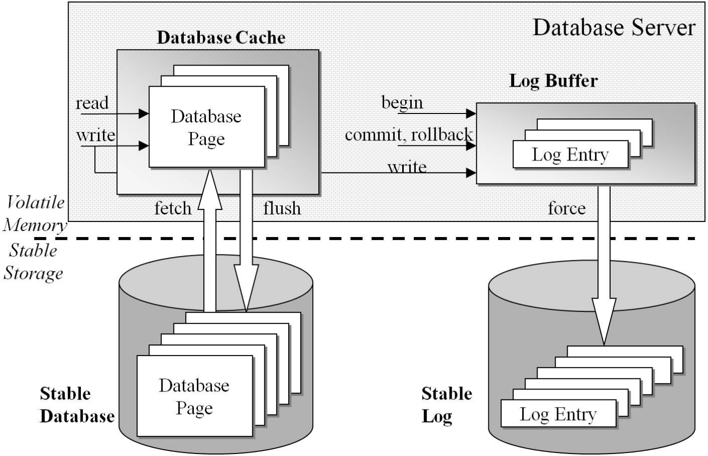
  Pinning the page to a fixed virtual-memory address in the database cache, reading the page contents and finally declaring the page to be dirty, unpinning the page and writing the page (*physiological action*).
- full-write(*pageno*, T)

  A new value is assigned to all bytes of a page and then it is written (*physical action*).
- fetch(*pageno*)

  Copies the previously uncached page *pageno* from the stable database into the database cache.
- flush(*pageno*)

  Copies the cached page *pageno* to the stable database.
- force()

  Forces all log entries in the log buffer to the stable log.



Overview of the system architecture components relevant to crash recovery.[5]

[5] Figure from Weikum and Vossen, *Transactional Information Systems*.

Numbering

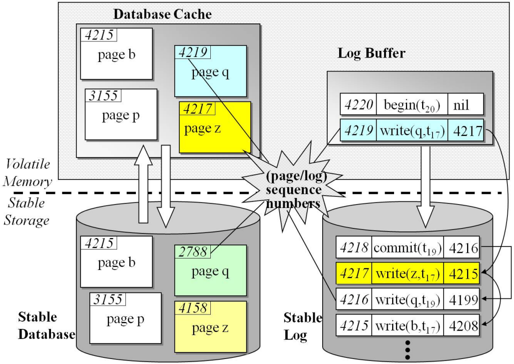
- Each action executed by the system is assigned a unique *sequence number* which is increasing among all actions that refer to the same page and among all actions that refer to the same transaction.
- Log entries are tagged with a chronologically increasing *log sequence number*.
- Each page in the stable database and the database cache carries a *page sequence number* that is coupled with log sequence numbers of the log entries for that page such that we can test the presence of a logged update in that page's state:

  The page sequence number of a pages is set to be the maximum log sequence number of the log entries that refer to this page.

Numbering

- Each action executed by the system is assigned a unique *sequence number* which is increasing among all actions that refer to the same page and among all actions that refer to the same transaction.
- Log entries are tagged with a chronologically increasing *log sequence number*.
- Each page in the stable database and the database cache carries a *page sequence number* that is coupled with log sequence numbers of the log entries for that page such that we can test the presence of a logged update in that page's state:

  The page sequence number of a pages is set to be the maximum log sequence number of the log entries that refer to this page.



- In the cache, pages *q* and *z* are dirty.
- The last update of page *q* is not yet recorded in the stable log and the stable database either; the respective transaction has not yet committed.
- Log entries are backwards chained on a per-transaction basis.

Sequence numbers in log entries and page headers.[6]

[6] Figure from Weikum and Vossen, *Transactional Information Systems*.

Basic data structures

```
type Page: record of
        PageNo: identifier;
        PageSeqNo: identifier;
        Status: (clean, dirty) /* only for cached pages*/;
        Contents: array [PageSize] of char;
     end;
persistent var StableDatabase: set of Page indexed by PageNo;
var DatabaseCache:
        set of Page indexed by PageNo;
type LogEntry: record of
        LogSeqNo: identifier;
        TransId: identifier;
        PageNo: identifier;
        ActionType:(write, full-write, begin, commit, rollback);
        UndoInfo: array of char;
        RedoInfo: array of char;
        PreviousSeqNo: identifier;
     end;
persistent var StableLog: ordered set of LogEntry indexed by LogSeqNo;
var LogBuffer:
        ordered set of LogEntry indexed by LogSeqNo;
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
     end;
var ActiveTrans: set of TransInfo indexed by TransId;
```

Actions During Normal Operation 1

```
write or full-write (pageno, transid, s):
   DatabaseCache[pageno].Contents := modified contents;
   DatabaseCache[pageno].PageSeqNo := s;
   DatabaseCache[pageno].Status := dirty;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := write or full-write;
   newlogentry.TransId := transid;
   newlogentry.PageNo := pageno;
   newlogentry.UndoInfo := information to undo update
        (before-image for full-write);
   newlogentry.RedoInfo := information to redo update
        (after-image for full-write);
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   ActiveTrans[transid].LastSeqNo := s;
   LogBuffer += newlogentry;

fetch (pageno):
   DatabaseCache += pageno;
   DatabaseCache[pageno].Contents := StableDatabase[pageno].Contents;
   DatabaseCache[pageno].PageSeqNo := StableDatabase[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;
```

Actions During Normal Operation 2

```
flush (pageno):
   if there is logentry in LogBuffer with logentry.PageNo = pageno
      then force ( );
   StableDatabase[pageno].Contents := DatabaseCache[pageno].Contents;
   StableDatabase[pageno].PageSeqNo := DatabaseCache[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

force ( ):
     StableLog += LogBuffer;
     LogBuffer := empty;

begin (transid, s):
   ActiveTrans += transid;
   ActiveTrans[transid].LastSeqNo := s;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := begin;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := nil;
   LogBuffer += newlogentry;

commit (transid, s):
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := commit;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   LogBuffer += newlogentry;
   ActiveTrans -= transid;
   force ( );
```

simple Three-Pass Algorithm

(1) Analysis pass:
Determine start of stable log from master record; perform forward scan to determine *winner*, i.e. commit log entry is encountered, and *loser* transactions, i.e. no commit log entry exists.

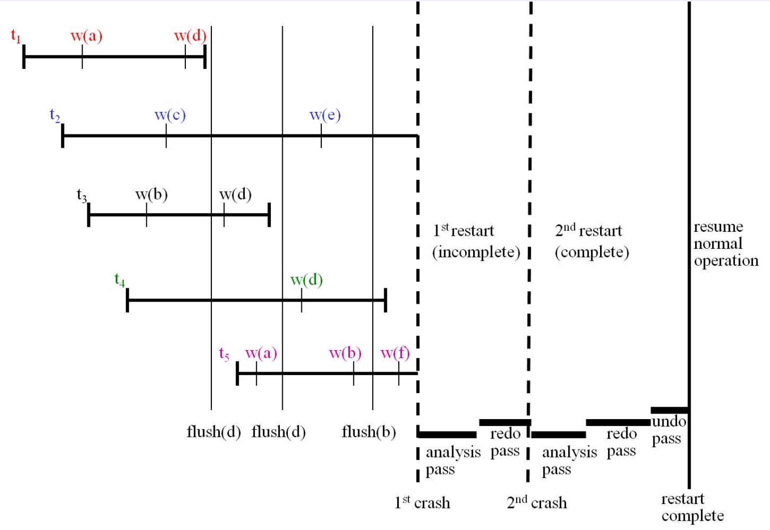
(2) Redo pass:
Perform forward scan of stable log to redo all winner actions in chronological (LSN) order until end of log is reached.

(3) Undo pass:
Perform backward scan of stable log to traverse all loser log entries in reverse chronological order and undo the corresponding actions.

Benefits of forward processing

- After the analysis pass the pages to be processed are known. Therefore, acessing the pages can be optimized.
- Physiological logging can be applied.

Normal processing, crash and repeated crash. Recovery must be idempotent. [7]



[7] Figure from Weikum and Vossen, *Transactional Information Systems*.

Example continued

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1: begin (t_1)			1: begin(t_1)	
2: begin (t_2)			2: begin (t_2)	
3: write (a, t_1)	a: 3		3: write (a, t_1)	
4: begin (t_3)			4: begin (t_3)	
5: begin (t_4)			5: begin (t_4)	
6: write (b, t_3)	b: 6		6: write (b, t_3)	
7: write (c, t_2)	c: 7		7: write (c, t_2)	
8: write (d, t_1)	d: 8		8: write (d, t_1)	
9: commit (t_1)			9: commit (t_1)	1, 2, 3, 4, 5, 6, 7, 8, 9
10: flush (d)		d: 8		
11: write (d, t_3)	d: 11		11: write (d, t_3)	
12: begin (t_5)			12: begin (t_5)	
13: write (a, t_5)	a: 13		13: write (a, t_5)	
14: commit (t_3)			14: commit (t_3)	11, 12, 13, 14
15: flush (d)		d: 11		
16: write (d, t_4)	d: 16		16: write (d, t_4)	
17: write (e, t_2)	e: 17		17: write (e, t_2)	
18: write (b, t_5)	b: 18		18: write (b, t_5)	
19: flush (b)		b: 18		16, 17, 18
20: commit (t_4)			20: commit (t_4)	20
21: write (f, t_5)	f: 21		21: write (f, t_5)	
ϟSYSTEM CRASH ϟ				

Example continued

RESTART
analysis pass: losers = $\{t_2, t_5\}$

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
redo (6)	b: 6			
flush (a)		a: 3		
redo (8)	d: 8			
flush (d)		d: 8		
redo (11)	d:11			
ϟSECOND SYSTEM CRASH ϟ				

Example continued

SECOND RESTART				
analysis pass: losers = $\{t_2, t_5\}$				
Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo(3)	a: 3			
redo(6)	b: 6			
redo(8)	d: 8			
redo(11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 6			
undo(17)	e: 0			
undo(13)	a: 3			
undo(7)	c: 0			
SECOND RESTART COMPLETE: RESUME NORMAL OPERATION				

Analysis pass

```
analysis pass ( ) returns losers:
var losers: set of record
              TransId: identifier;
              LastSeqNo: identifier;
           end indexed by TransId;
   losers := empty;
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max do
       case StableLog[i].ActionType:
          begin: losers += StableLog[i].TransId;
                 losers[StableLog[i].TransId].LastSeqNo := nil;
          commit: losers -= StableLog[i].TransId;
          full-write: losers[StableLog[i].TransId].LastSeqNo := i;
      end /*case*/;
   end /*for*/;
```

Redo pass (full writes)

```
redo pass ( ):
   min := LogSeqNo of oldest log entry in StableLog;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := min to max
   do
       if StableLog[i].ActionType = full-write and
          StableLog[i].TransId not in losers
       then
          pageno = StableLog[i].PageNo;
          fetch (pageno);
          full-write (pageno)
             with contents from StableLog[i].RedoInfo;
       end /*if*/;
   end /*for*/;
```

Undo pass (full writes)

```
undo pass ( ):
   while there exists t in losers
         such that losers[t].LastSeqNo <> nil
   do
       nexttrans = TransNo in losers
          such that losers[nexttrans].LastSeqNo =
          max {losers[x].LastSeqNo | x in losers};
       nextentry = losers[nexttrans].LastSeqNo;
       if StableLog[nextentry].ActionType = full-write
       then
          pageno = StableLog[nextentry].PageNo;
          fetch (pageno);
          full-write (pageno)
             with contents from StableLog[nextentry].UndoInfo;
          losers[nexttrans].LastSeqNo :=
             StableLog[nextentry].PreviousSeqNo;
       end /*if*/;
   end /*while*/;
```

Idempotence of a recovery algorithm means, that when the recovery algorithm crashes, it will, when restarted again, perform the same steps as it did during the previous restart.

The problem of idempotence

The restart operations are performed in the cache and may be arbitrarily flushed - so it is not clear in general, whether a certain redo has to be repeated during a repeated restart.

- full-writes:

  Full-writes assign values to all bytes of a page. Before- and after-image are full page contents. Therefore, idempotence is guaranteed.

- general writes:

  If redo- or undo-information is given by operations - e.g. an *insert*-operation or a *shift* of certain bytes - then idempotence is not guaranteed and extra mechanisms have to be added.

Incorporating general writes as physiological log entries

- State testing during the redo pass:

  for log entry for page p with log sequence number i, redo write only if $i > p.PageSeqNo$ *and subsequently set* $p.PageSeqNo := i$

- State testing during the undo pass:

  for log entry for page p with log sequence number i, undo write only if $i \leq p.PageSeqNo$ *and subsequently set* $p.PageSeqNo := i\text{-}1$

Example continued

RESTART				
analysis pass: losers = $\{t_2, t_5\}$				

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
redo (3)	a: 3			
consider-redo (6)	b: 18			
flush (a)		a: 3		
consider-redo (8)	d: 11			
consider-redo (11)	d: 11			
ϟSECOND SYSTEM CRASH ϟ				

Example continued

SECOND RESTART
analysis pass: losers = $\{t_2, t_5\}$

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable Database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
consider-redo(3)	a: 3			
consider-redo(6)	b: 18			
consider-redo(8)	d: 11			
consider-redo(11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 17			
consider-undo(17)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			
SECOND RESTART COMPLETE: RESUME NORMAL OPERATION				

Simple Three-Pass Algorithm with General Writes

```
redo pass ( ):
   ...
         fetch (pageno);
         if DatabaseCache[pageno].PageSeqNo < i
         then
            read and write (pageno)
                 according to StableLog[i].RedoInfo;
            DatabaseCache[pageno].PageSeqNo := i;
         end /*if*/;
   ...
undo pass ( ):
   ...
   fetch (pageno);
         if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo
         then
            read and write (pageno)
                according to StableLog[nextentry].UndoInfo;
            DatabaseCache[pageno].PageSeqNo := nextentry.LogSeqNo - 1;
         end /*if*/;
   ...
```