Fault Tolerance - Part II

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

Based on slides by Maarten Van Steen

What is the Problem?

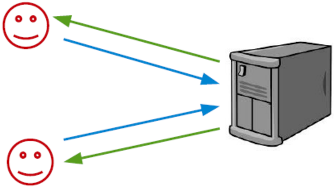
Two Generals' Problem

- Two generals need to be agree on time to attack to win.
- They communicate through messengers, who may be killed on their way.

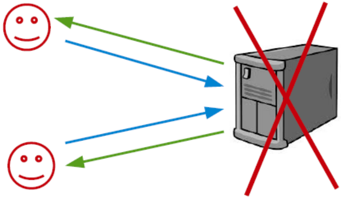
Two Generals' Problem

- Two generals need to be agree on time to attack to win.
- They communicate through messengers, who may be killed on their way.
- Agreement is the problem.

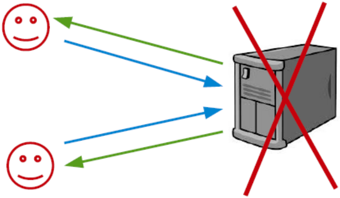
Replicated State Machine Problem (1/2)



Replicated State Machine Problem (1/2)



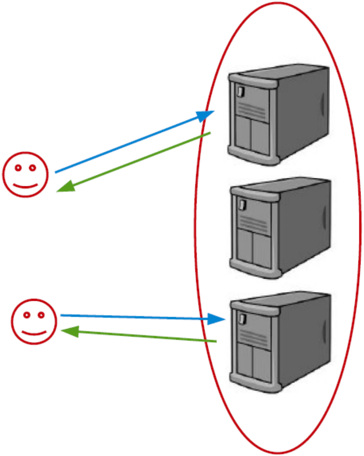
Replicated State Machine Problem (1/2)



- The solution: replicate the server.

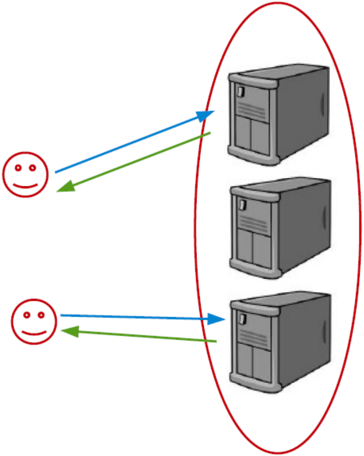
Replicated State Machine Problem (2/2)

- Make the server deterministic (state machine).
- Replicate the server.
- Ensure correct replicas step through the same sequence of state transitions (How?)



Replicated State Machine Problem (2/2)

- Make the server deterministic (state machine).
- Replicate the server.
- Ensure correct replicas step through the same sequence of state transitions (How?)
- Agreement is the problem.



Distributed Commit

The Agreement Problem

- Some nodes propose values (or actions) by sending them to the others.
- All nodes must decide whether to accept or reject those values.

The Agreement Problem

- Some nodes propose values (or actions) by sending them to the others.
- All nodes must decide whether to accept or reject those values.

- But, ...

The Agreement Problem

- Some nodes propose values (or actions) by sending them to the others.
- All nodes must decide whether to accept or reject those values.

- But, ...

- Concurrent processes and uncertainty of timing, order of events and inputs.
- Failure and recovery of machines/processors, of communication channels.

Distributed Commit

- Given a computation distributed across a process group, how can we ensure that either all processes commit to the final result, or none of them do (atomicity)?

- Possible solutions:
  - Two-Phase Commit (2PC)
  - Three-Phase Commit (3PC)

Two-Phase Commit (2PC)

Intuitive Example (1/3)

- You want to organize outing with 3 friends at 6pm Tuesday.
  - Go out only if all friends can make it.

Intuitive Example (2/3)

- What do you do?

Intuitive Example (2/3)

- What do you do?
  - Call each of them and ask if can do 6pm on Tuesday (voting phase)

Intuitive Example (2/3)

- What do you do?
  - Call each of them and ask if can do 6pm on Tuesday (voting phase)
  - If all can do Tuesday, call each friend back to ACK (commit)

Intuitive Example (2/3)

- What do you do?
  - Call each of them and ask if can do 6pm on Tuesday (voting phase)
  - If all can do Tuesday, call each friend back to ACK (commit)
  - If one cannot do Tuesday, call other three to cancel (abort)

Intuitive Example (3/3)

- Critical details

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.
  - You need to remember the decision and tell anyone whom you have not been able to reach during commit/abort phase.

Intuitive Example (3/3)

- Critical details
  - While you were calling everyone to ask, people who have promised they can do 6pm Tuesday must reserve that slot.
  - You need to remember the decision and tell anyone whom you have not been able to reach during commit/abort phase.

- That is exactly how 2PC works.

2PC Players

- Coordinator: the client who initiated the computation.
- Participants: the processes required to commit.

2PC (1/2)

- Phase 1a: the coordinator sends vote-request to participants.

2PC (1/2)

- Phase 1a: the coordinator sends vote-request to participants.
- Phase 1b: when a participant receives vote-request, it returns either vote-commit or vote-abort to coordinator.
    - If it sends vote-abort, it aborts its local computation.

2PC (2/2)

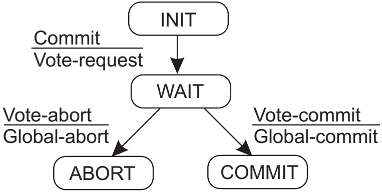
- Phase 2a: the coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort.

2PC (2/2)

- Phase 2a: the coordinator collects all votes; if all are vote-commit, it sends global-commit to all participants, otherwise it sends global-abort.

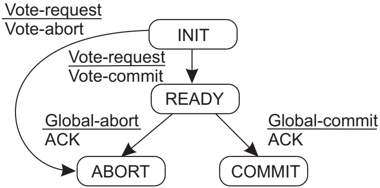
- Phase 2b: each participant waits for global-commit or global-abort and handles accordingly.

2PC States



(a)

Coordinator



(b)

Participant

2PC - Failing Participant (1/2)

- Initial state: no problem, participant was unaware of protocol.

2PC - Failing Participant (1/2)

- Initial state: no problem, participant was unaware of protocol.
- Ready state: the participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision.

2PC - Failing Participant (1/2)

- Initial state: no problem, participant was unaware of protocol.
- Ready state: the participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision.
- Abort state: remove the workspace of results.

2PC - Failing Participant (1/2)

- Initial state: no problem, participant was unaware of protocol.
- Ready state: the participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make ⇒ log the coordinator's decision.
- Abort state: remove the workspace of results.
- Commit state: copying workspace to storage.

2PC - Failing Participant (2/2)

- Alternative: when a recovery is needed to READY state, check state of other participants $\Rightarrow$ no need to log coordinator's decision.

2PC - Failing Participant (2/2)

- Alternative: when a recovery is needed to READY state, check state of other participants ⇒ no need to log coordinator's decision.
- Recovering participant P contacts another participant Q:

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

2PC - Failing Participant (2/2)

- Alternative: when a recovery is needed to READY state, check state of other participants ⇒ no need to log coordinator's decision.
- Recovering participant *P* contacts another participant *Q*:

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- If all participants are in the READY state, the protocol blocks. Apparently, the coordinator is failing. Note: The protocol prescribes that we need the decision from the coordinator.

2PC - Failing Coordinator

- The real problem lies in the fact that the coordinator's final decision may not be available for some time or lost.

2PC - Failing Coordinator

- The real problem lies in the fact that the coordinator's final decision may not be available for some time or lost.

- Alternative: let a participant P in the READY state timeout when it hasn't received the coordinator's decision; P tries to find out what other participants know.

2PC - Failing Coordinator

- The real problem lies in the fact that the coordinator's final decision may not be available for some time or lost.

- Alternative: let a participant P in the READY state timeout when it hasn't received the coordinator's decision; P tries to find out what other participants know.

- Essence of the problem is that a recovering participant cannot make a local decision: it depends on other (possibly failed) processes.

Three-Phase Commit (3PC)

3PC (1/3)

- Phase 1a: the coordinator sends vote-request to participants.
- Phase 1b: when a participant receives vote-request, it returns either vote-commit or vote-abort to coordinator.
  - If it sends vote-abort, it aborts its local computation.

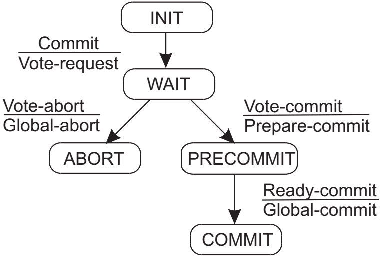
3PC (2/3)

- Phase 2a: the 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 waits for prepare-commit, or waits for global-abort after which it halts.

3PC (3/3)

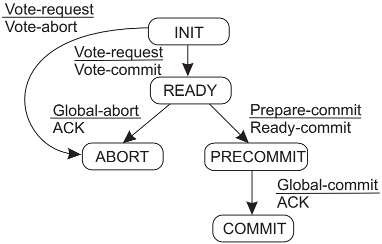
- Phase 3a: the coordinator waits until all participants have sent ready-commit, and then sends global-commit to all.
- Phase 3b: each participant waits for global-commit.

3PC States



(a)

Coordinator



(b)

Participant

3PC - Failing Participant

- Can P find out what it should do after crashing in the READY or PRE-COMMIT state, even if other participants or the coordinator failed?

3PC - Failing Participant

- Can *P* find out what it should do after crashing in the READY or PRE-COMMIT state, even if other participants or the coordinator failed?
- If a participant timeouts in READY state, it can find out at the coordinator or other participants whether it should abort, or enter PRE-COMMIT state.

3PC - Failing Participant

- Can *P* find out what it should do after crashing in the READY or PRE-COMMIT state, even if other participants or the coordinator failed?

- If a participant timeouts in READY state, it can find out at the coordinator or other participants whether it should abort, or enter PRE-COMMIT state.

- If a participant already made it to the PRE-COMMIT state, it can always safely commit (but is not allowed to do so for the sake of failing other processes).

Recovery

Recovery

- When a failure occurs, we need to bring the system into an error-free state:

Recovery

- When a failure occurs, we need to bring the system into an error-free state:
  - Forward error recovery: find a new state from which the system can continue operation.

Recovery

- When a failure occurs, we need to bring the system into an error-free state:
    - Forward error recovery: find a new state from which the system can continue operation.
    - Backward error recovery: bring the system back into a previous error-free state.

Recovery

- When a failure occurs, we need to bring the system into an error-free state:
    - Forward error recovery: find a new state from which the system can continue operation.
    - Backward error recovery: bring the system back into a previous error-free state.

- Use backward error recovery, requiring that we establish recovery points.

Recovery

- When a failure occurs, we need to bring the system into an error-free state:
  - Forward error recovery: find a new state from which the system can continue operation.
  - Backward error recovery: bring the system back into a previous error-free state.

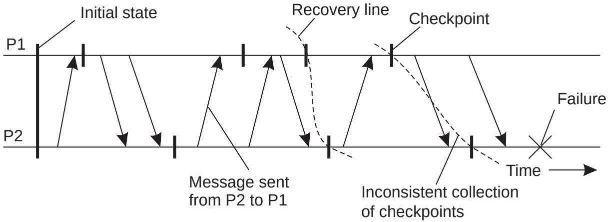
- Use backward error recovery, requiring that we establish recovery points.

- Recovery in distributed systems is complicated by the fact that processes need to cooperate in identifying a consistent state from where to recover.

Checkpointing

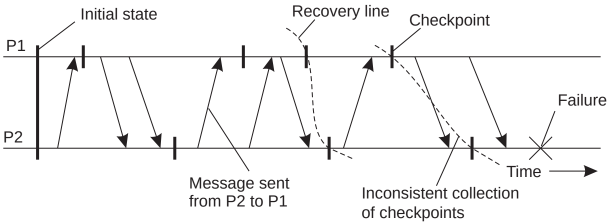
Consistent Recovery State

- Requirement: every message that has been received is also shown to have been sent in the state of the sender.



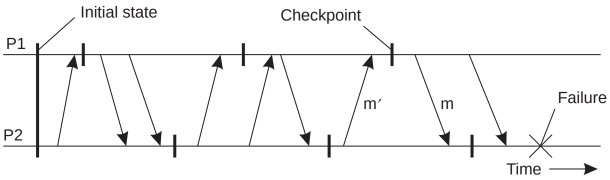
Consistent Recovery State

- Requirement: every message that has been received is also shown to have been sent in the state of the sender.
- Recovery line: assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Cascaded rollback

- If checkpointing is done at the wrong instants, the recovery line may lie at system startup time ⇒ cascaded rollback



Independent Checkpointing (1/2)

- Each process independently takes checkpoints.

Independent Checkpointing (1/2)

- Each process independently takes checkpoints.
- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$.

Independent Checkpointing (1/2)

- Each process independently takes checkpoints.
- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$.
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i, m).

Independent Checkpointing (1/2)

- Each process independently takes checkpoints.
- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$.
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i, m).
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$.

Independent Checkpointing (1/2)

- Each process independently takes checkpoints.
- Let $CP[i](m)$ denote m^{th} checkpoint of process P_i and $INT[i](m)$ the interval between $CP[i](m-1)$ and $CP[i](m)$.
- When process P_i sends a message in interval $INT[i](m)$, it piggybacks (i, m).
- When process P_j receives a message in interval $INT[j](n)$, it records the dependency $INT[i](m) \rightarrow INT[j](n)$.
- The dependency $INT[i](m) \rightarrow INT[j](n)$ is saved in a stable storage when taking checkpoint $CP[j](n)$.

Independent Checkpointing (2/2)

- If process P_i rolls back to $CP[i](m-1)$, P_j must roll back to $CP[j](n-1)$.

Independent Checkpointing (2/2)

- If process P_i rolls back to $CP[i](m-1)$, P_j must roll back to $CP[j](n-1)$.
- How can P_j find out where to roll back to? we can build a dependency graph between checkpoints to discover the recovery line.

Coordinated Checkpointing

- Each process takes a checkpoint after a globally coordinated action.

Coordinated Checkpointing

- Each process takes a checkpoint after a globally coordinated action.
- Simple solution: use a two-phase blocking protocol:

Coordinated Checkpointing

- Each process takes a checkpoint after a globally coordinated action.
- Simple solution: use a two-phase blocking protocol:
    - A coordinator multicasts a checkpoint request message.

Coordinated Checkpointing

- Each process takes a checkpoint after a globally coordinated action.
- Simple solution: use a two-phase blocking protocol:
  - A coordinator multicasts a checkpoint request message.
  - When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint.

Coordinated Checkpointing

- Each process takes a checkpoint after a globally coordinated action.
- Simple solution: use a two-phase blocking protocol:
  - A coordinator multicasts a checkpoint request message.
  - When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint.
  - When all checkpoints have been confirmed at the coordinator, it latter broadcasts a checkpoint done message to allow all processes to continue.

Message Logging

Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

- We assume a piecewise deterministic execution model:

Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

- We assume a piecewise deterministic execution model:
    - The execution of each process can be considered as a sequence of state intervals.

Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.
- We assume a piecewise deterministic execution model:
  - The execution of each process can be considered as a sequence of state intervals.
  - Each state interval starts with a nondeterministic event (e.g., message receipt).

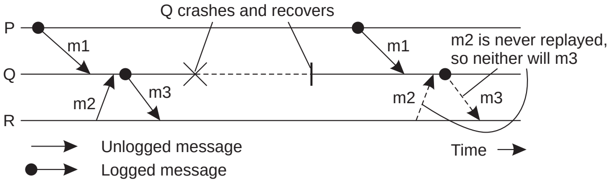
Message Logging

- Instead of taking an (expensive) checkpoint, try to replay your (communication) behavior from the most recent checkpoint ⇒ store messages in a log.

- We assume a piecewise deterministic execution model:
  - The execution of each process can be considered as a sequence of state intervals.
  - Each state interval starts with a nondeterministic event (e.g., message receipt).
  - Execution in a state interval is deterministic.

Message Logging and Consistency

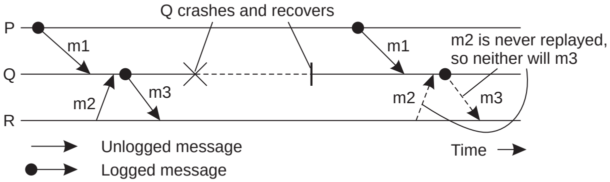
- Example:
    - Process Q has just received and subsequently delivered messages m_1 and m_2.
    - Assume that m_2 is never logged.
    - After delivering m_1 and m_2, Q sends message m_3 to process R.
    - Process R receives and subsequently delivers m_3.



Message Logging and Consistency

- Example:
  - Process Q has just received and subsequently delivered messages m_1 and m_2.
  - Assume that m_2 is never logged.
  - After delivering m_1 and m_2, Q sends message m_3 to process R.
  - Process R receives and subsequently delivers m_3.

- Orphan process: a process that survives the crash of another process, but whose state is inconsistent with the crashed process after its recovery.



Message-Logging Schemes (1/4)

- *HDR*[*m*]: the header of message *m* containing its source, destination, sequence number, and delivery number.

Message-Logging Schemes (1/4)

- *HDR*[*m*]: the header of message *m* containing its source, destination, sequence number, and delivery number.
  - The header contains all information for resending a message and delivering it in the correct order.

Message-Logging Schemes (1/4)

- *HDR*[*m*]: the header of message *m* containing its source, destination, sequence number, and delivery number.
  - The header contains all information for resending a message and delivering it in the correct order.
  - A message *m* is stable if *HDR*[*m*] cannot be lost (e.g., because it has been written to stable storage).

Message-Logging Schemes (1/4)

- $HDR[m]$: the header of message m containing its source, destination, sequence number, and delivery number.
  - The header contains all information for resending a message and delivering it in the correct order.
  - A message m is stable if $HDR[m]$ cannot be lost (e.g., because it has been written to stable storage).

- $DEP[m]$: the set of processes to which message m has been delivered, as well as any message that causally depends on delivery of m.

Message-Logging Schemes (1/4)

- $HDR[m]$: the header of message m containing its source, destination, sequence number, and delivery number.
  - The header contains all information for resending a message and delivering it in the correct order.
  - A message m is stable if $HDR[m]$ cannot be lost (e.g., because it has been written to stable storage).
- $DEP[m]$: the set of processes to which message m has been delivered, as well as any message that causally depends on delivery of m.
- $COPY[m]$: the set of processes that have a copy of $HDR[m]$ in their volatile memory.

Message-Logging Schemes (2/4)

- If C is a collection of crashed processes, then $Q \notin C$ is an orphan if there is a message m such that $Q \in DEP[m]$ and $COPY[m] \subseteq C$.

Message-Logging Schemes (2/4)

- If C is a collection of crashed processes, then $Q \notin C$ is an orphan if there is a message m such that $Q \in DEP[m]$ and $COPY[m] \subseteq C$.

- We want $\forall m \forall C :: COPY[m] \subseteq C \Rightarrow DEP[m] \subseteq C$.
    - This is the same as saying that $\forall m :: DEP[m] \subseteq COPY[m]$.

Message-Logging Schemes (2/4)

- If C is a collection of crashed processes, then $Q \notin C$ is an orphan if there is a message m such that $Q \in DEP[m]$ and $COPY[m] \subseteq C$.

- We want $\forall m \forall C :: COPY[m] \subseteq C \Rightarrow DEP[m] \subseteq C$.
  - This is the same as saying that $\forall m :: DEP[m] \subseteq COPY[m]$.

- Goal: no orphans, means that for each message m, $DEP[m] \subseteq COPY[m]$.

Message-Logging Schemes (3/4)

- Pessimistic protocol: for each unstable message m, there is at most one process dependent on m, that is $|DEP[m]| \leq 1$.

- Consequence: an unstable message in a pessimistic protocol must be made stable before sending a next message.

Message-Logging Schemes (4/4)

- Optimistic protocol: for each unstable message m, we ensure that if $COPY[m] \subseteq C$, then eventually also $DEP[m] \subseteq C$, where C denotes a set of processes that have been marked as faulty.

- Consequence: to guarantee that $DEP[m] \subseteq C$, we generally rollback each orphan process Q until $Q \notin DEP[m]$.

Summary

Summary

- Distributed commit: 2PC and 3PC
- Recovery: checkpointing and message logging

Reading

- Chapter 8 of the Distributed Systems: Principles and Paradigms.

Questions?