Fault Tolerance

Distributed Systems

Reliable remote procedure calls

What can go wrong?

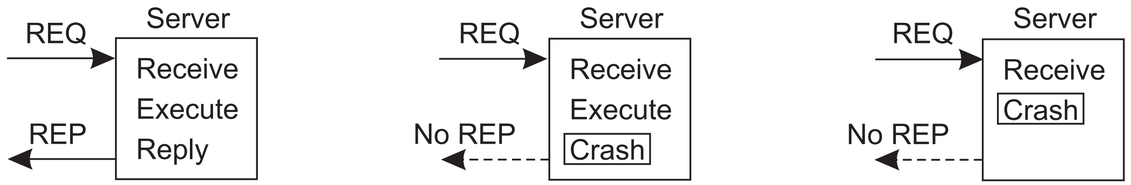
1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

Two “easy” solutions

1: (cannot locate server): just report back to client

2: (request was lost): just resend message

Reliable RPC: server crash



(a) (b) (c)

Problem

Where (a) is the normal case, situations (b) and (c) require different solutions. However, we don't know what happened. Two approaches:

- At-least-once-semantics: The server guarantees it will carry out an operation at least once, no matter what.
- At-most-once-semantics: The server guarantees it will carry out an operation at most once.

Reliable RPC: lost reply messages

The real issue

What the client notices, is that it is not getting an answer. However, it cannot decide whether this is caused by a lost request, a crashed server, or a lost response.

Partial solution

Design the server such that its operations are idempotent: repeating the same operation is the same as carrying it out exactly once:

- pure read operations
- strict overwrite operations

Many operations are inherently nonidempotent, such as many banking transactions.

Reliable RPC: client crash

Problem

The server is doing work and holding resources for nothing (called doing an orphan computation).

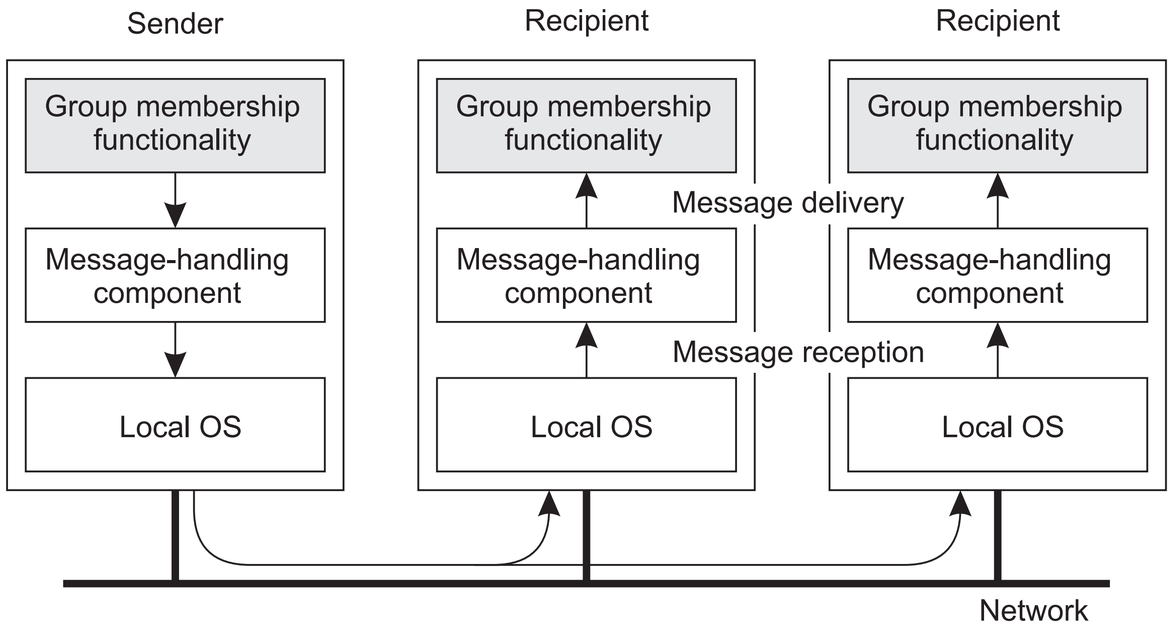
Solution

- Orphan is killed (or rolled back) by the client when it recovers
- Client broadcasts new epoch number when recovering $\Rightarrow$ server kills client's orphans
- Require computations to complete in a T time units. Old ones are simply removed.

Simple reliable group communication

Intuition

A message sent to a process group **G** should be delivered to each member of **G**. Important: make distinction between receiving and delivering messages.



Less simple reliable group communication

Reliable communication in the presence of faulty processes

Group communication is reliable when it can be guaranteed that a message is received and subsequently delivered by all nonfaulty group members.

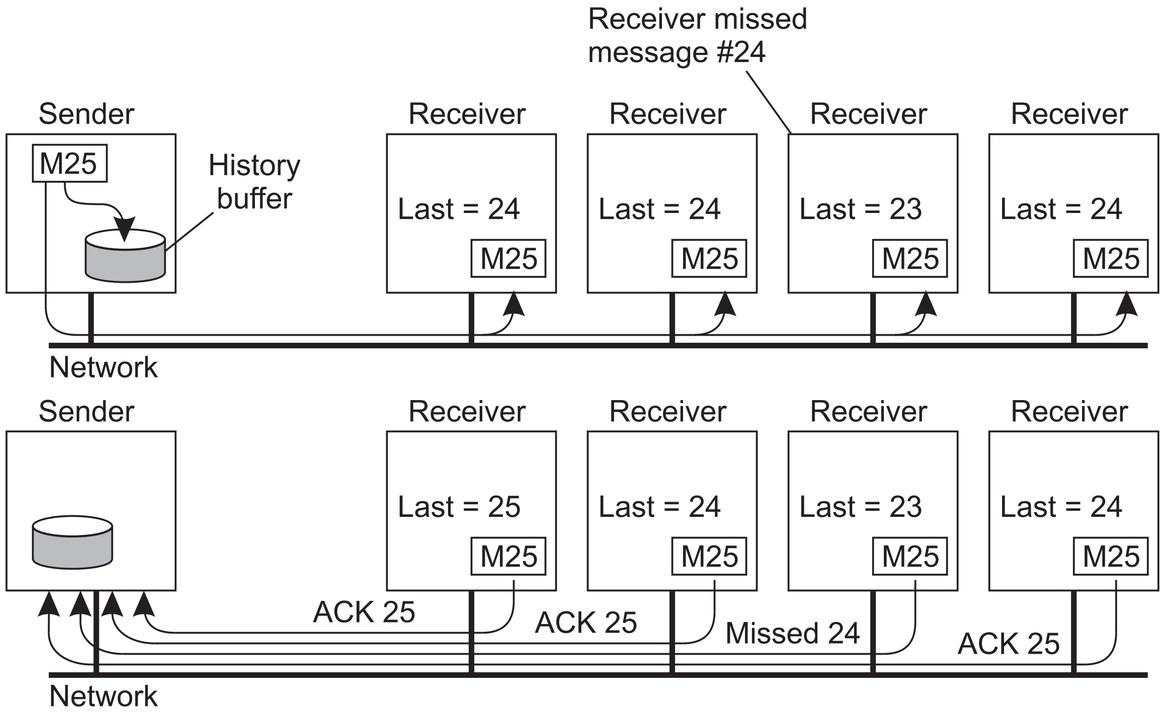
Tricky part

Agreement is needed on what the group actually looks like before a received message can be delivered.

Simple reliable group communication

Reliable communication, but assume nonfaulty processes

Reliable group communication now boils down to reliable multicasting: is a message received and delivered to each recipient, as intended by the sender.



Distributed commit protocols

Problem

Have an operation being performed by each member of a process group, or none at all.

- Reliable multicasting: a message is to be delivered to all recipients.
- Distributed transaction: each local transaction must succeed.

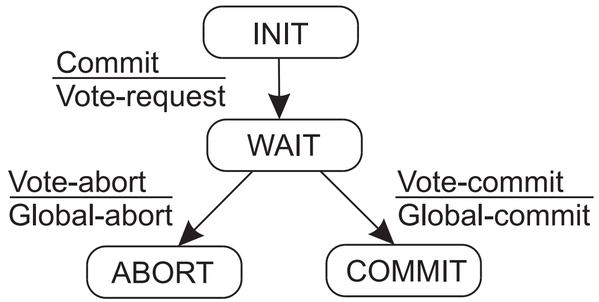
Two-phase commit protocol (2PC)

Essence

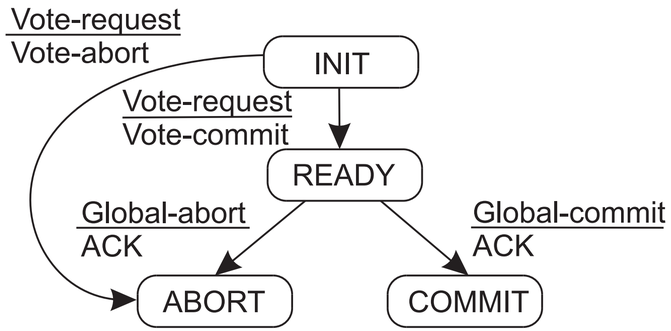
The client who initiated the computation acts as coordinator; processes required to commit are the participants.

- Phase 1a: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- 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 handles accordingly.

2PC - Finite state machines



Coordinator



Participant

2PC – Failing participant

Analysis: participant crashes in state S, and recovers to S

- *INIT*: No problem: participant was unaware of protocol
- *READY*: Participant is waiting to either commit or abort. After recovery, participant needs to know which state transition it should make $\Rightarrow$ log the coordinator's decision
- *ABORT*: Merely make entry into abort state idempotent, e.g., removing the workspace of results
- *COMMIT*: Also make entry into commit state idempotent, e.g., copying workspace to storage.

Observation

When distributed commit is required, having participants use temporary workspaces to keep their results allows for simple recovery in the presence of failures.

2PC – Failing participant

Alternative

When a recovery is needed to *READY* state, check state of other participants $\Rightarrow$ 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

Result

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

Observation

The real problem lies in the fact that the coordinator's final decision may not be available for some time (or actually 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 (as discussed).

Observation

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

Recovery: Background

Essence

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

Practice

Use backward error recovery, requiring that we establish recovery points

Observation

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

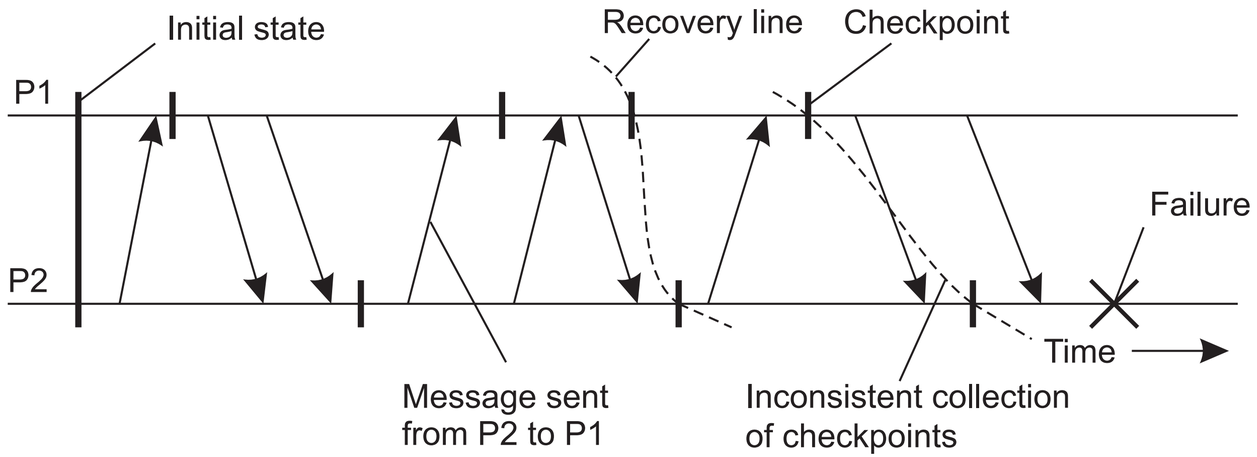
Consistent recovery state

Requirement

Every message that has been received is also shown to have been sent in the state of the sender.

Recovery line

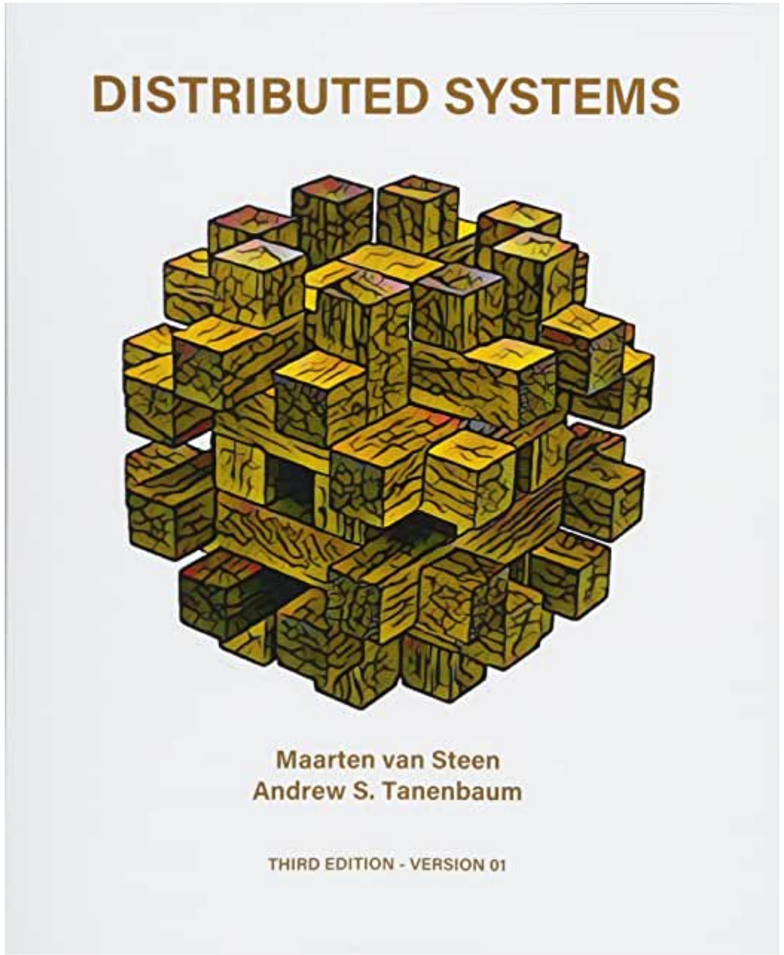
Assuming processes regularly checkpoint their state, the most recent consistent global checkpoint.



Coordinated checkpointing

In **coordinated checkpointing** all processes synchronize to jointly write their state to local storage. The main advantage of coordinated checkpointing is that the saved state is automatically globally consistent. A simple solution is to use a two-phase blocking protocol. A coordinator first multicasts a CHECKPOINT-REQUEST message to all processes. When a process receives such a message, it takes a local checkpoint, queues any subsequent message handed to it by the application it is executing, and acknowledges to the coordinator that it has taken a checkpoint. When the coordinator has received an acknowledgment from all processes, it multicasts a CHECKPOINT-DONE message to allow the (blocked) processes to continue.

Text Book


DISTRIBUTED SYSTEMS
Maarten van Steen
Andrew S. Tanenbaum
THIRD EDITION - VERSION 01