CMPU4021
Distributed Systems

Fault Tolerance

Fault Tolerance

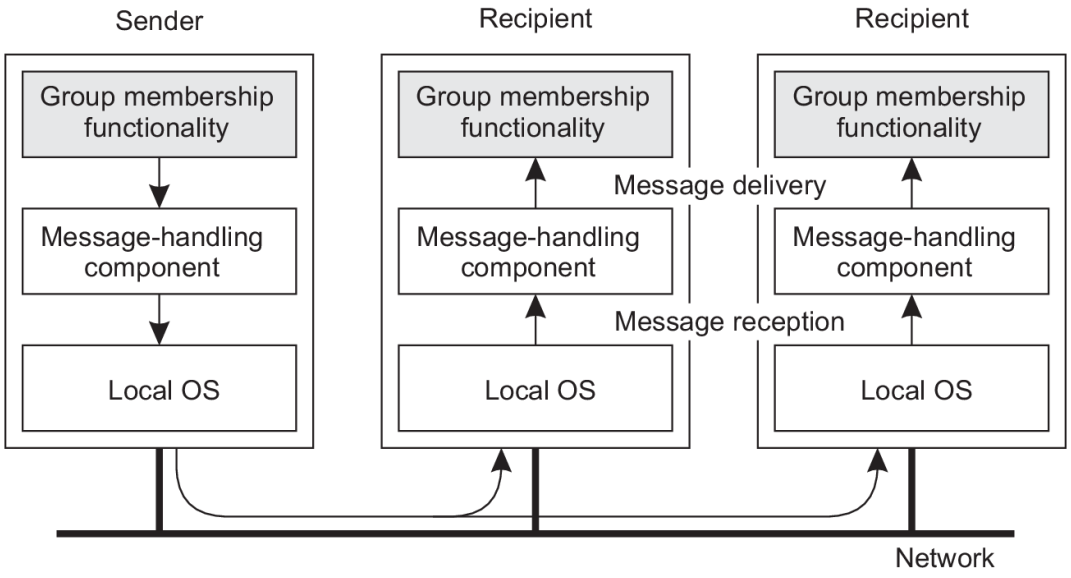
- Distributed systems - the notion of partial failure
  - part of the system is failing while the remaining part continues to operate
  - seemingly correctly

Types of failures

Type	Description of server's behaviour
Crash failure	Halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	Fails to respond to incoming requests Fails to receive incoming messages Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure *Value failure* *State-transition failure*	Response is incorrect The value of the response is wrong. Deviates from the correct flow of control

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.

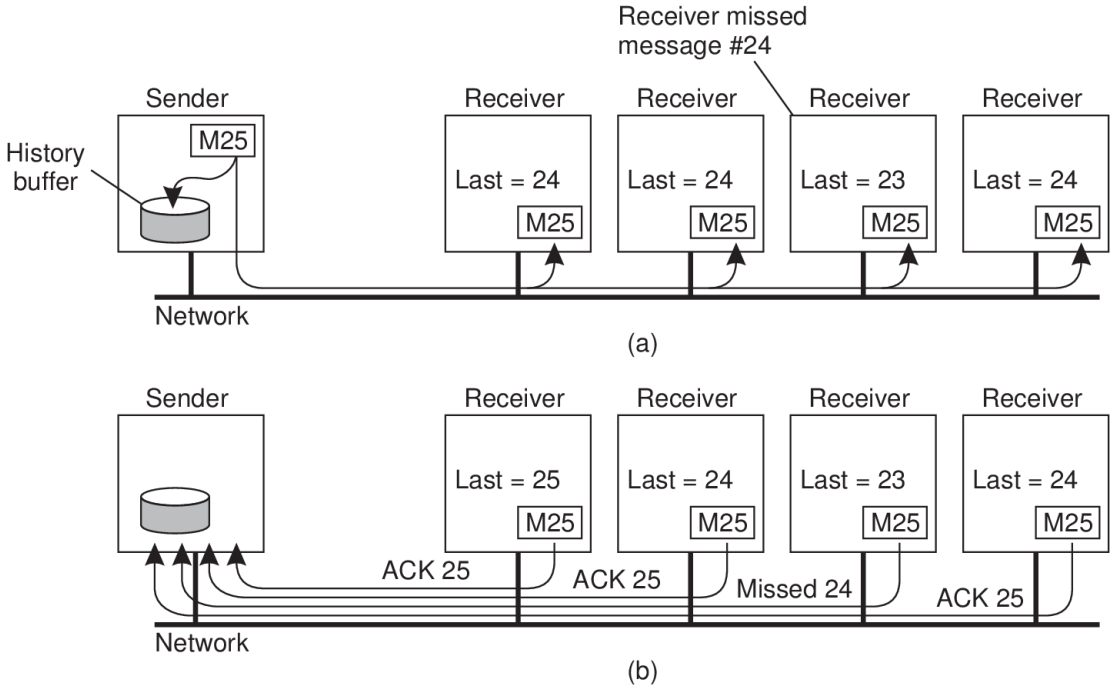


Less simple reliable group communication

- 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** non-faulty 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
  - assume nonfaulty processes
- Reliable group communication
  - boils down to **reliable multicasting**:
    - is a message received and delivered to **each** recipient, as intended by the sender.



Atomic multicast

- Problem
  - How to achieve reliable multicasting in the presence of process failures.
- In particular – often needed in a distributed system
  - The guarantee that a message is delivered to either all group members or to none at all.
  - This is also known as the **atomic multicast problem**.
- The atomic multicasting problem is an example of a more general problem, known as **distributed commit**.

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.

Distributed Commit

- The distributed commit problem involves having an operation being performed by each member of a process group, or none at all.
- In the case of reliable multicasting, the operation is the delivery of a message.
- With distributed transactions, the operation may be the commit of a transaction at a single site that takes part in the transaction.

Failure model for the commit protocols

- Commit protocols are designed to work in an asynchronous system in which
  - servers may crash
  - messages may be lost
- It is assumed that an underlying request-reply protocol removes corrupt and duplicated messages.
- There are no Byzantine faults
  - servers either crash or obey the messages they are sent

Atomic commit protocols

- One coordinator and multiple participants
- Protocols for atomic distributed commit
  - One-phase
    - the coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out.
  - Two-phase
    - designed to allow any participant to abort its part of a transaction
    - can result in extensive delays for participants in the uncertain state.
  - Three-phase
    - designed to alleviate delays due to participants in the uncertain state.
    - more expensive in terms of the number of messages and the number of rounds
    - required for the normal (failure-free) case.

TWO-PHASE COMMIT PROTOCOL

The two-phase commit protocol

- During the progress of a transaction, the only communication between coordinator and participant is the *join* request
  - The client request to commit or abort goes to the coordinator
    - if client or participant request abort, the coordinator informs the participants immediately
    - if the client asks to commit, the 2PC comes into use
- 2PC
  - *voting phase*: coordinator asks all participants if they can commit
    - if yes, participant records updates in permanent storage and then votes
  - *completion phase*: coordinator tells all participants to commit or abort
  - the next slide shows the operations used in carrying out the protocol

Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

This is a request with a reply

These are asynchronous requests to avoid delays

Asynchronous request

- participant interface- *canCommit?, doCommit, doAbort*
- coordinator interface- *haveCommitted, getDecision*

The two-phase commit protocol

Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.
4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

The Voting Rules

1. Each participant has one vote which can be either 'commit‘ or 'abort’;

2. Having voted, a participant cannot change its vote;

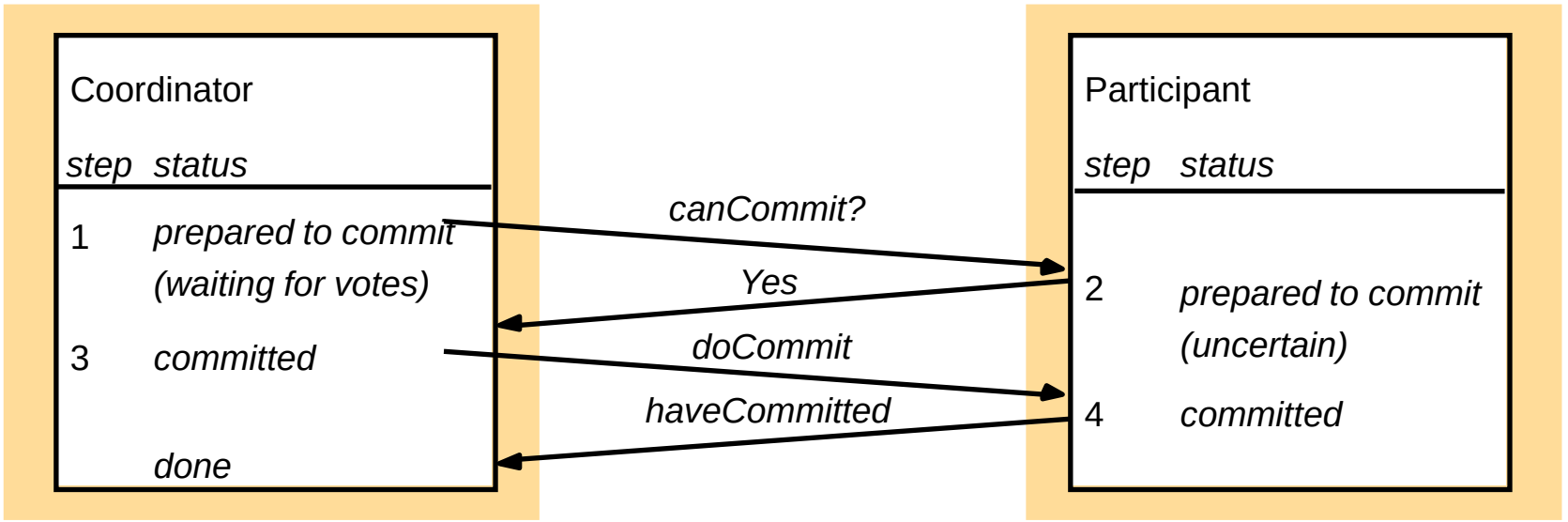
3. If a participant votes 'abort' then it is free to abort the transaction immediately; any site is in fact free to abort a transaction at any time up until it records a 'commit' vote. Such a transaction abort is known as a unilateral abort.

4. If a participant votes ‘commit’, then it must wait for the co-ordinator to broadcast either the 'global-commit' or 'global-abort' message;

5. If all participants vote 'commit' then the global decision by the co-ordinator must be 'commit';

6. The global decision must be adopted by all participants.

Communication in two-phase commit protocol



- Time-out actions in the 2PC
  - to avoid blocking forever when a process crashes or a message is lost
  - *uncertain* participant (step 2) has voted yes. it can't decide on its own
    - it uses `getDecision` method to ask coordinator about outcome
  - participant has carried out client requests, but has not had a `Commit?` from the coordinator. It can abort unilaterally
  - coordinator delayed in waiting for votes (step 1). It can abort and send `doAbort` to participants.

Communication in two-phase commit protocol

- A participant may be delayed
  - Carried out all its client requests, but has not yet received a *`canCommit`*? Call from the coordinator.
  - As the client sends the *`closeTransaction`* to the `Coordinator`, a participant can only detect such a situation if it notices that it has not had a request in a particular transaction for a long time, e.g.
    - By a timeout period on a lock.
  - As no decision has been made at this stage, the participant can decide to *abort* unilaterally after some period of time.

Communication in two-phase commit protocol

- The coordinator may be delayed:
  - When it is waiting for votes from the participants.
  - As it has not yet decided the fate of the transaction, it may decide to abort the transaction after some period of time.
  - It must then announce *doAbort* to the participants who have already sent their votes.
  - Some participant may try to vote *Yes* after this, but their votes will be ignored and they will enter the *uncertain* state.

Performance of the two-phase commit protocol

- If there are no failures, the 2PC involving *N* participants requires
  - *N canCommit?* messages and replies, followed by *N doCommit* messages.
    - the cost in messages is proportional to 3*N*, and the cost in time is three rounds of messages.
    - The *haveCommitted* messages are not counted
- There may be arbitrarily many server and communication failures
- 2PC is guaranteed to complete eventually, but it is not possible to specify a time limit within which it will be complete
  - delays to participants in uncertain state
  - some 3PCs designed to alleviate such delays
    - they require more messages and more rounds for the normal case

THREE-PHASE COMMIT (3PC) PROTOCOL

Three-phase commit (3PC) protocol: Phase 1

Phase 1: *Voting phase*

- The coordinator sends a `canCommit?` request to each of the participants in the transaction.
  - Purpose: Find out if everyone agrees to commit
- It the coordinator gets a `timeout` from any participant, or any NO replies are received
  - Send an `abort` to all participants
- If a participant times out waiting for a request from the coordinator
  - It `aborts` itself (assume coordinator crashed)
- Else continue to phase 2

3PC protocol: Phase 2

Phase 2: *Prepare to commit phase*

- The coordinator collects the votes and makes a decision.
  - If it is `No`, it `aborts` and informs participants that voted `Yes`
  - if the decision is `Yes`, it sends a `preCommit` request to all the participants.
  - Participants that voted `Yes` wait for a `preCommit` or `doAbort` request.
  - They acknowledge `preCommit` requests and carry out `doAbort` requests.

3PC protocol: Phase 3

Phase 3: *Finalize phase*

- The coordinator collects the acknowledgements.
- When all are received, it commits and sends `doCommit` requests to the participants.
- Participants wait for a `doCommit` request.
- When it arrives, they `commit`.

Commit protocols: delays handling

Assumptions: communication does not fail:

- Two-phase commit protocol
  - the ‘uncertain/delay’ period occurs because a participant has voted yes but has not yet been told the outcome.
    - It can no longer abort unilaterally
- Three-phase commit protocol
  - The participants ‘uncertain’ period lasts from when the participant votes yes until it receives the *preCommit* request.
  - At this stage, no other participant can have committed. Therefore if a group of participants discover that they are all ‘uncertain’ and the coordinator cannot be contacted, they can decide unilaterally to abort.

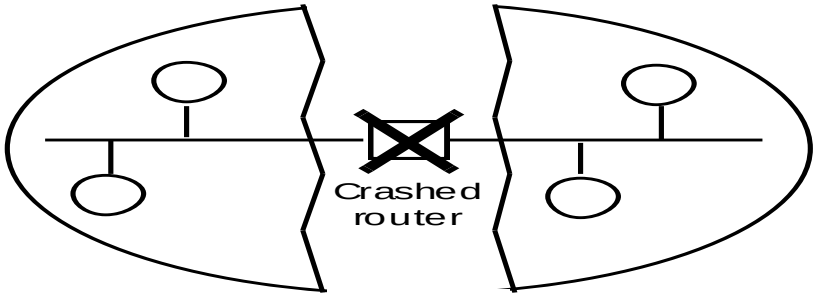
3PC Weaknesses

- It may result in inconsistent state when a crashed coordinator recovers

- It is not resilient against network partitions
  - Consensus based protocols are designed to be resilient against network partitions
    - Raft, Paxos

A network partition

- Network partition example:
  - The failure of a router between two networks
  - May mean that a collection of four processes is split into two pairs
  - Intra-pair communication is possible over their respective networks
  - Inter-pair communication is not possible while the router has failed.



Paxos: High Overview

- Paxos is a family of protocols providing distributed consensus
- Goal
  - Agree on a single value even if multiple systems propose different values concurrently
- Common use: provide a consistent ordering of events from multiple clients
  - All machines running the algorithm agree on a proposed value from a client
  - The value will be associated with an event or action
  - Paxos ensures that no other machine associates the value with another event

Consistency, availability, and partitioning (CAP) theorem

In 2000, Eric Brewer posed an important theorem which was later proven to be correct

CAP theorem

Any networked system providing shared data can provide only two of the following three properties:

C: *consistency*, by which a shared and replicated data item appears as a single, up-to-date copy

A: *availability*, by which updates will always be eventually executed

P: Tolerant to the *partitioning* of process group.

Conclusion

In a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request.

CAP theorem intuition

- Simple situation: two interacting processes
  - P and Q can no longer communicate:
    - Allow P and Q to go ahead ⇒ no consistency
    - Allow only one of P, Q to go ahead ⇒ no availability
  - P and Q have to be assumed to continue communication ⇒ no partitioning allowed.

CAP Theorem Practical Ramification

- The CAP theorem is all about reaching a trade-off between safety and liveness, based on the observation that obtaining both in an inherently unreliable system cannot be achieved.
  - Practical distributed systems are inherently unreliable.

- One can argue that the CAP theorem moves designers of distributed systems from theoretical solutions to *engineering* solutions.

- In practical distributed systems, one simply has to make a choice to proceed despite the fact that another process cannot be reached.
  - In other words, we need to do something when a partition manifests itself through high latency
  - Exactly deciding on how to proceed is application dependent

Summary

- Fault tolerance is defined as the characteristic by which a system can mask the occurrence and recovery from failures.

- In other words, a system is fault tolerant if it can continue to operate in the presence of failures.

References

- Chapter 17: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5thEd.

- Chapter 8: Maarten van Steen, Andrew S. Tanenbaum, Distributed Systems, 3rd edition (2017)