

UNIT 8

Fault Tolerance

Distributed Systems (CACS352) — BCA III/VI

⚠ Please note

*These notes were prepared with AI assistance and **may contain errors or omissions**. Please cross-check definitions, algorithms and numerical examples against your prescribed textbook (e.g., Tanenbaum & van Steen, Distributed Systems), your class notes and your instructor before relying on them for exams.*

8.1 Introduction to Fault Tolerance

A system is **fault tolerant** if it can continue to provide its service in spite of faults. In a distributed system partial failure is normal — some components fail while others keep working — so fault tolerance is a central design goal.

Dependability

Requirement	Meaning
Availability	The system is ready to be used immediately; probability that it works at a given instant
Reliability	The system can run continuously without failure (measured over an interval)
Safety	When it fails, nothing catastrophic happens
Maintainability	How easily a failed system can be repaired

Example 8.1 — Availability

A server has a mean time to failure MTTF = 999 hours and a mean time to repair MTTR = 1 hour. Find its availability.

Solution

Availability = $MTTF / (MTTF + MTTR) = 999 / 1000 = 0.999 = 99.9\%$ (about 8.8 hours of downtime per year).

Faults, errors and failures

A **failure** occurs when a system cannot meet its promises. An **error** is the part of the system state that may lead to a failure; the cause of the error is a **fault**. Faults may be **transient** (occur once and vanish), **intermittent** (appear and disappear repeatedly, e.g., a loose contact) or **permanent** (remain until repaired).

Failure models

Type of failure	Description
Crash failure	Server halts but worked correctly until it halted
Omission failure	Server fails to respond to requests (receive or send omission)
Timing failure	Response lies outside the specified time interval
Response failure	Response is incorrect — value failure or state-transition failure
Arbitrary (Byzantine) failure	Server may produce arbitrary responses at arbitrary times, even maliciously

A **fail-stop** server halts and other servers can reliably detect it; **fail-silent** means others cannot tell a crash from a slow response; **fail-safe** means it fails only in a harmless way.

Failure masking by redundancy

- **Information redundancy** — extra bits (parity, Hamming codes, checksums) to detect/correct errors.
- **Time redundancy** — repeat an action if needed (retransmission, transactions); good for transient faults.
- **Physical redundancy** — extra hardware or processes. Example: **Triple Modular Redundancy (TMR)**: three copies of a unit feed a majority voter, so one faulty unit is masked.

8.2 Process Resilience

To protect against a faulty process, replicate it and organise the copies into a **process group**: a message sent to the group reaches all members, so if one fails another takes over. Groups may be **flat** (all peers, decisions collective, no single point of failure, but more overhead) or **hierarchical** (a coordinator decides; simple but a single point of failure). Group membership must be managed (join/leave/crash).

Failure masking and replication — k fault tolerance

A group is **k fault tolerant** if it survives faults in k members. With **crash / fail-silent** failures, $k + 1$ members suffice (one survivor can answer). With **Byzantine** failures where the group votes on the result, at least $2k + 1$ members are needed to have a correct majority — and for agreement in the presence of Byzantine processes $3k + 1$ members are needed.

Consensus in faulty systems

Processes must agree on a value (e.g., who is the leader, whether to commit). Requirements: **agreement** (all correct processes decide the same value), **validity** (the decided value was proposed by some process) and **termination** (every correct process eventually decides). The **FLP result** proves that no deterministic algorithm can guarantee consensus in an asynchronous system if even one process can crash; practical algorithms therefore use timeouts / randomness.

- **Paxos** and **Raft** — crash-tolerant consensus with a majority ($2k + 1$ nodes to tolerate k crashes); used for replicated state machines (Chubby, ZooKeeper's ZAB, etcd).
- **Byzantine agreement** (Lamport, Shostak, Pease) — with k arbitrary-faulty processes, agreement is possible only if $N \geq 3k + 1$ (with $k = 1$ you need at least 4 processes). PBFT is a practical protocol.
- **Flooding-based consensus** — in rounds, each process broadcasts its known proposals and picks by a deterministic rule.

Example 8.2 — How many replicas?

A service must tolerate $k = 2$ simultaneous faulty replicas. How many replicas are needed if the faults are (a) crash failures, (b) Byzantine failures?

Solution

(a) Crash: $k + 1 = 3$ replicas for masking (or $2k + 1 = 5$ when a majority is needed for consensus, e.g., Paxos).

(b) Byzantine: $3k + 1 = 3 \times 2 + 1 = 7$ replicas for agreement.

Failure detection

Failures are detected by **timeouts**: periodic *heartbeat* messages, or ping-ack. In an asynchronous network a timeout cannot be perfect — a slow process may be suspected wrongly (false positive) — so detectors are 'eventually accurate'. **Gossip-style** detectors let members exchange heartbeat tables to reduce false suspicions.

8.3 Reliable Client-Server Communication

Reliable **point-to-point** communication is usually provided by TCP (acknowledgements, retransmission, ordering). It masks omission failures, but not crashes — if a TCP connection breaks, the application must handle it. For RPC there are five classes of failure and standard solutions:

Failure	Possible solution
1. Client cannot locate the server	Raise an exception (loses transparency)
2. Client request is lost	Start a timer; retransmit on expiry (server discards duplicates using request ids)
3. Server crashes after receiving request	Client cannot tell whether the request was executed; choose at-least-once, at-most-once or exactly-once semantics
4. Server reply is lost	Use idempotent operations, or sequence numbers with a cached reply
5. Client crashes	Orphan computations must be eliminated

Server crash — when did it crash?

A server can crash *before* executing the request (no effect), or *after* executing but before replying (side effect done). Reissue strategies: never, always, only when acknowledged, only when not acknowledged. No combination works for all crash orderings (executing, printing, replying), which is why exactly-once is generally impossible.

Client crash — orphans

- **Extermination** — the client logs each RPC before sending; after reboot, orphans are killed.
- **Reincarnation** — time is divided into numbered *epochs*; after reboot the client broadcasts a new epoch and servers kill earlier computations.
- **Gentle reincarnation** — servers kill a computation only if its owner cannot be located.
- **Expiration** — each RPC gets a time quantum T ; it must finish or ask for more.

8.4 Reliable Group Communication

Reliable multicast ensures that a message sent to a group is delivered to *all* non-faulty members.

Basic reliable multicast

Sender attaches sequence numbers and keeps messages in a history buffer. **ACK-based:** receivers acknowledge every message; the sender retransmits on missing ACKs — suffers *feedback implosion* for large groups. **NACK-based:** receivers report only missing messages (detected by gaps in sequence numbers). With **feedback suppression** (as in SRM), a receiver waits a random time and multicasts its NACK; others that see it do not send their own.

Ordering guarantees

Ordering	Meaning
Unordered	No order guarantee among messages
FIFO	Messages from the same sender are delivered in the order sent
Causal	If $m_1 \rightarrow m_2$ then all deliver m_1 before m_2
Total (atomic)	All members deliver all messages in the same order (can be combined with FIFO or causal)

Atomic multicast and virtual synchrony

Atomic multicast guarantees that a message is delivered either to *all* members or to *none*, even in the presence of crashes. The group membership is described by a **view**. **Virtual synchrony** ensures that a multicast to view G is delivered to all members of G that do not crash *before* a view change; if the sender crashes, either all surviving members receive the message or none. A **view change** acts as a barrier: all pending messages are delivered (flushed) before the new view is installed. This makes the group appear as if it executes in lock-step, as in the ISIS toolkit.

8.5 Distributed Commit

The **atomic commit** problem: either *all* processes participating in a distributed transaction commit, or *all* abort. This is a special case of consensus that requires a coordinator.

Two-phase commit (2PC)

Phase 1 (voting): the coordinator sends VOTE-REQUEST to all participants; each replies VOTE-COMMIT (ready) or VOTE-ABORT. **Phase 2 (decision):** if all voted commit, the coordinator sends GLOBAL-COMMIT; otherwise GLOBAL-ABORT. Participants acknowledge.

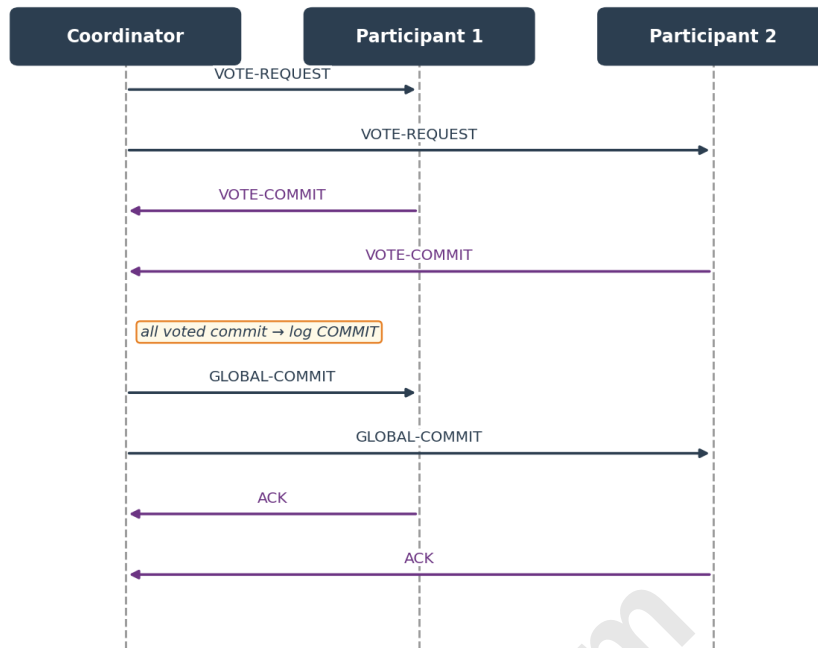


Figure 8.1 — Two-phase commit (all participants vote commit).

Actor	States
Coordinator	INIT → WAIT → COMMIT or ABORT
Participant	INIT → READY → COMMIT or ABORT

Failures in 2PC

- **Participant waits in INIT** for VOTE-REQUEST — timeout: it may abort unilaterally.
- **Coordinator waits in WAIT** for votes — timeout: it decides GLOBAL-ABORT.
- **Participant in READY** waits for the decision — it cannot decide alone. It *blocks*, or asks other participants: if any is in COMMIT or ABORT, follow it; if any is in INIT, abort; if all are in READY, it must keep waiting (blocking) until the coordinator recovers.
- Coordinator and participants write their state to a **log on stable storage** before sending messages so that they can recover.

*Note: 2PC is **blocking**: if the coordinator crashes while participants are READY, they cannot safely decide.*

Three-phase commit (3PC)

3PC adds a **PRECOMMIT** state between READY and COMMIT so that no state is adjacent to both COMMIT and ABORT, and any crash can be resolved by the surviving processes; it is **non-blocking** as long as there is no network partition and fail-stop crashes are assumed. It needs an extra round of messages and is rarely used in practice; most systems use 2PC with replicated coordinators (or Paxos Commit).

8.6 Recovery

Recovery brings a failed system back to a correct state. **Backward recovery** rolls back to a previously saved consistent state (checkpoint) and continues; **forward recovery** moves to a new correct state (e.g., using error-correcting codes or repair). Backward recovery is general but costs time and may lose work.

Checkpointing

Each process periodically saves its state on **stable storage** (survives crashes). The set of the most recent checkpoints, one per process, forms the **recovery line** and must be a **consistent global state** (a *distributed snapshot*): no message is recorded as received but not sent.

- **Independent checkpointing** — each process checkpoints alone; simple, but recovery may cascade backwards through many checkpoints (the **domino effect**) to find a consistent line.
- **Coordinated checkpointing** — processes synchronise so all checkpoints are consistent, e.g., using the **Chandy–Lamport snapshot algorithm**: an initiator saves its state and sends a *marker* on all outgoing channels; on first receiving a marker, a process saves its state and sends markers; messages arriving on a channel after the local state was saved but before the marker on that channel are recorded as channel state.

Message logging

Checkpointing is expensive, so processes also **log messages** and replay them after restoring the last checkpoint (relying on *piecewise deterministic* execution). A process becomes an **orphan** if it depends on a message whose log was lost.

Approach	Idea	Trade-off
Pessimistic logging	Log a message before delivering it	No orphans; slow (extra writes)
Optimistic logging	Log asynchronously; roll back orphans after a crash	Faster; complex recovery
Causal logging	Keep dependency info in messages	Balance of both

Key Takeaways

- ✓ Dependability = availability, reliability, safety, maintainability; availability = $MTTF / (MTTF + MTTR)$.
- ✓ Failure types: crash, omission, timing, response, Byzantine; mask them with information, time or physical redundancy.
- ✓ Process groups: $k + 1$ members for crash faults, $3k + 1$ for Byzantine agreement; consensus via Paxos/Raft; FLP shows limits.
- ✓ RPC failures: five cases; orphan handling by extermination, reincarnation, gentle reincarnation or expiration.
- ✓ Reliable multicast uses sequence numbers and NACKs; virtual synchrony gives atomic delivery across view changes.
- ✓ 2PC (vote, decide) may block; 3PC avoids blocking with PRECOMMIT; recovery uses checkpoints, message logs and consistent snapshots.

— *End of Unit 8* —