Distributed Systems

Fault Tolerance

WEEK 12: Fault Detection, Failure Models, Recovery Techniques, Redundancy, Process Resilience.

Online Lecture Series - 12

Felix Edesa

Addis Ababa Science and Technology University

Course Outline: Fault Tolerance (Week 12)

Topics We Will Cover

- **Introduction to Fault Tolerance** — Why systems must survive failures.
- **Failure Models** — Crash, omission, and Byzantine failures.
- **Redundancy Resilience** — Using replication to mask failures.
- **Consensus in Faulty Systems** — Ensuring agreement despite crashes.
- **Failure Detection** — Heartbeat and timeout techniques.
- **Reliable Communication** — Handling message loss in client–server systems.
- **Recovery Techniques** — Checkpointing, message logging, and rollback.
- **Industrial Insights** — Fault tolerance in cloud, IoT, and blockchain.

Fault Recovery — Introduction

Understanding Fault Tolerance

- In distributed systems, failures are inevitable.
- **Fault tolerance** ensures that a system continues to function correctly even when components fail.
- A dependable system is measured by:
  - **Availability:** readiness of a system for use at any moment.
  - **Reliability:** ability to operate without failure over time.
  - **Safety:** prevents catastrophic consequences during malfunction.
  - **Maintainability:** ease of identifying and restoring failed parts.

Example

- Banking systems maintain mirrored databases across regions.
- Even if one server fails, the backup continues processing requests.

Availability vs Reliability

Key Concepts

- **Availability** focuses on system uptime.
- **Reliability** focuses on uninterrupted operation — how long the system can run without failing.
- A system can be available but unreliable if it restarts frequently but recovers fast.
- Conversely, it can be reliable but unavailable during planned maintenance.
- Designing both high availability and high reliability requires redundancy, monitoring, and preventive maintenance.

Practical Insight

- A web app that restarts every hour for 1 second $\rightarrow$ 99.99% availability, but low reliability.
- Database clusters use replication to ensure both reliability and availability.

Safety and Maintainability

Core Principles

- **Safety:** Ensures that failures do not cause dangerous or irreversible results.
- Faults must degrade performance gracefully rather than catastrophically.
- **Maintainability:** Refers to how quickly a system can detect, isolate, and recover from faults.
- Maintainable systems include:
  - Automated diagnostics and alerts.
  - Fast rollback or patch mechanisms.
  - Modular components for easy replacement.

Example

- In an aircraft system, sensor faults trigger redundant sensors before data corruption.
- Cloud systems auto-restart failed containers and log recovery times.

Failure Detection Techniques

Detection Approaches

- Detecting failures early prevents cascading errors across distributed components.
- Common techniques include:
  - **Heartbeat Protocols:** Nodes send regular “alive” signals to coordinators.
  - **Timeout and Retries:** If no response within set time, node is suspected as failed.
  - **Distributed Monitoring:** Services like Prometheus or Datadog collect health metrics and alert on anomalies.
  - **Consensus Checks:** Nodes vote to confirm failure before triggering recovery (used in Paxos, Raft).

Example

- Kubernetes uses liveness probes to detect failed containers.
- Google Cloud auto-restarts unhealthy VMs after missed heartbeats.

Recovery and Re-Synchronization

Core Methods

- After detecting a fault, recovery aims to restore system state and synchronize data.
- **Checkpointing:** Periodically save process state for rollback after crash.
- **Replica Replay:** Retrieve data from replicas that remained active during failure.
- **Leader Election:** Elect new coordinators dynamically.
- **State Synchronization:** Compare timestamps or logs to reconcile differences.

Example

- Cassandra automatically synchronizes replicas after a node rejoins.
- Database logs replay recent transactions to rebuild consistency.

Practical Summary — Fault Recovery

Key Lessons for Engineers

- Always assume components will fail.
- Combine proactive monitoring with automated recovery tools.
- Redundancy, replication, and periodic checkpoints are core defense layers.
- Evaluate trade-offs: more redundancy increases cost but improves dependability.
- Document fault recovery strategies — logs, alert policies, and rollback plans.

Real-World Practices

- AWS auto-heals EC2 instances on hardware faults.
- Netflix uses “Chaos Monkey” to test fault tolerance.

Reliability vs Availability — Introduction

Understanding the Difference

- **Reliability** and **Availability** are two core dependability measures used in distributed and cloud systems.
- **Reliability ($R(t)$):** Probability that a component or system will function correctly for a given time period $[0, t)$ without failure.
- **Availability (A):** Probability that a system is operational and accessible at a given instant.
- A system may be highly available but not highly reliable — if it fails often but recovers quickly.

Example

- A website that restarts instantly after each crash may appear "always online" — high availability, low reliability.

Reliability Metrics

Mathematical Foundations

- The **Reliability Function** $R(t)$ gives the probability that component C operates correctly in $[0, t)$ given it was working at $t = 0$.

$$R(t) = P(\text{Component works correctly during } [0, t))$$

- **Mean Time To Failure (MTTF):** The average time until a component fails.

$$\text{MTTF} = \int_0^\infty R(t)\, dt$$

Example

- Hard drives may have an MTTF of 1,000,000 hours.
- Cloud VMs are designed with reliability targets like "five nines" (99.999%) uptime.

Availability Metrics

How Availability is Measured

- **Mean Time To Repair (MTTR):** Average time needed to fix or restore a component after failure.
- **Mean Time Between Failures (MTBF):** Total expected operational cycle between two consecutive failures.

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

- **Availability Formula:**

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{\text{MTBF} - \text{MTTR}}{\text{MTBF}}$$

Example

- Web servers that automatically restart in 10 seconds achieve $A \approx 99.99\%$.
- High-availability clusters use load balancers to reroute traffic instantly.

Relationship Between Reliability and Availability

Key Differences and Connections

- Both metrics reflect dependability but from different perspectives:
  - Reliability — long-term operation without failure.
  - Availability — readiness at a given instant.
- A reliable system often implies high availability, but not always.
- Factors influencing both:
  - Hardware and network redundancy.
  - Fault detection and recovery speed.
  - Scheduled maintenance frequency.

Real-World Analogy

- A car that rarely breaks down (reliable) but is often at the mechanic for servicing (low availability).

Practical Engineering Considerations

Designing for Both

- To maximize reliability:
  - Use high-quality components and preventive monitoring.
  - Schedule predictive maintenance before failures occur.
- To maximize availability:
  - Deploy redundancy — multiple servers, replicas, or backup routes.
  - Implement rapid failover and self-healing mechanisms.
- Striking the right balance depends on the domain:
  - Financial and aerospace systems favor reliability.
  - Web services and IoT devices prioritize availability.

Example Systems

- AWS EC2 Auto-Scaling ensures 99.99% availability.
- SpaceX communication systems are built for extreme reliability.

Practical Problem & Solution — Reliability vs Availability

Problem Scenario

- **E-commerce platform:** Flash sale crashes servers; short downtimes frustrate users.
- **IoT smart home devices:** Temporary device failures during firmware updates.
- **Online exams:** Frequent short restarts interrupt students.
- **Cloud storage:** Backup servers sync slowly, causing temporary inconsistencies.

$\Rightarrow$ Systems may appear highly available but are not truly reliable if failures occur frequently or unpredictably.

Engineering Solutions

- **E-commerce:** Load balancers and auto-scaling prevent crashes.
- **IoT devices:** Gradual firmware updates with redundancy.

Practical Problem & Solution — Availability Calculation

Problem Scenario

- A streaming platform has:
  - Mean Time To Failure (MTTF) = 1200 hours
  - Mean Time To Repair (MTTR) = 10 hours
- Question: Compute system availability A.
- Observation: Users complain if downtime exceeds 0.5% annually.

Solution

- MTBF = MTTF + MTTR = 1200 + 10 = 1210 hours
- Availability:

$$A = \frac{1200}{1210} \approx 0.9917$$

Practical Problem & Solution — Reliability Calculation

Problem Scenario

- A mission-critical sensor network:
  - MTTF = 5000 hours
  - Exponential failure model assumed
- Task: Find reliability $R(t)$ at $t = 1000$ hours.
- Note: High reliability is essential to prevent false readings.

Solution

- Reliability function:

$$R(t) = e^{-t/\text{MTTF}}$$

$$= e^{-1000/5000} \approx 0.8187$$

Terminology — Failure, Error, Fault

Understanding Core Dependability Terms

- A **fault** is the underlying cause — such as hardware defects, software bugs, or network interruptions — that introduces incorrect system states.
- An **error** is the manifestation of a fault within the system's internal state, which can propagate through computations.
- A **failure** occurs when the system's external behavior deviates from its specification or expected service.

Real-World Examples

- **Failure:** A program crashes during execution.
- **Error:** A memory variable contains invalid data.
- **Fault:** Poor error-handling logic.

Terminology — Handling Faults

Introduction to Fault Management

- Fault management is the set of strategies used to **handle faults** in distributed and critical systems.
- Four main approaches:
  1. Fault Prevention
  2. Fault Tolerance
  3. Fault Removal
  4. Fault Forecasting
- Each approach addresses faults **at different stages** of the system lifecycle: design, operation, and maintenance.

Example

- Fault Prevention: Avoid sloppy coding practices.
- Fault Tolerance: Use redundant developers for critical components.

Fault Prevention

Preventing Faults at Source

- Goal: Avoid the introduction of faults into the system.
- Strategies:
  - Careful recruitment and training of programmers and engineers.
  - Strict coding standards, code reviews, and testing protocols.
  - Design practices that reduce complexity and potential error sources.
- Emphasizes **proactive measures** rather than reactive fixes.

Example

- Do not hire sloppy programmers.
- Implement thorough code review and automated static analysis tools.

Fault Tolerance

Masking Faults in Operation

- Goal: Ensure the system **continues operating correctly** even if faults occur.
- Techniques:
  - Redundancy: Duplicate critical components or processes.
  - Voting mechanisms to resolve discrepancies between replicas.
  - Graceful degradation to maintain partial functionality.
- Focuses on **resilience during runtime**, critical for distributed systems, cloud services, and IoT networks.

Example

- Build each component by two independent programmers.
- Use RAID storage to prevent data loss if a disk fails.

Fault Removal

Reducing Faults Post-Detection

- Goal: Decrease the presence, severity, or number of faults in a system.
- Strategies:
  - Debugging and fixing detected errors.
  - Removing low-quality code and replacing it with well-tested modules.
  - Conduct regression testing to ensure fixes do not introduce new faults.
- This approach is **reactive** and focuses on improving system dependability.

Example

- Get rid of sloppy programmers or unmaintainable code.
- Update firmware to fix recurring errors in devices.

Fault Forecasting

Predicting Fault Presence and Impact

- Goal: Estimate current and future incidence of faults and their consequences.
- Strategies:
  - Statistical analysis of historical failure data.
  - Predictive modeling for system vulnerabilities.
  - Risk assessment to prioritize preventive or corrective measures.
- Helps plan maintenance, redundancy, and resource allocation efficiently.

Example

- Estimate how a recruiter is performing when hiring sloppy programmers.
- Predict server failure likelihood.

Failure Models — Types of Failures

Understanding Failure Models

- Failure models describe how a component or system **may fail** in distributed environments.
- They help **design fault-tolerant systems** by anticipating possible abnormal behaviors.
- Common failure types in servers and distributed systems:
  1. Crash failures
  2. Omission failures (receive/send)
  3. Timing failures
  4. Response, value, and state-transition failures
  5. Arbitrary (Byzantine) failures

Examples

- Crash failure: Server halts unexpectedly.
- Receive omission: Server ignores incoming requests.
- Send omission: Messages not delivered to clients.

Crash and Omission Failures

Crash and Omission Failures Explained

- **Crash failure:** Component stops functioning but works correctly until the crash.
- **Omission failures:** Component fails to send or receive messages.
  - *Receive omission:* Does not respond to incoming requests.
  - *Send omission:* Fails to deliver messages to other components.
- Implications:
  - Systems must detect and recover from silent failures.
  - Redundancy and heartbeat mechanisms help identify crashed nodes.

Practical Examples

- Crash: Web server unexpectedly stops serving pages.
- Receive omission: Microservice ignores API requests.

Timing and Response Failures

Delayed or Incorrect Responses

- **Timing failure:** Component responds outside the expected time interval.
- **Response failure:** Component produces an incorrect response.
- **Value failure:** Response is technically valid but contains wrong values.
- **State-transition failure:** System deviates from its correct flow of control.

Examples

- Timing: Slow API response beyond SLA.
- Response: Wrong invoice sent to a client.

Arbitrary (Byzantine) Failures

Unpredictable Failures

- **Arbitrary failure:** Component can produce any behavior, including malicious or random actions.
- Hardest to detect and mitigate because failures are not systematic.
- Often referred to as **Byzantine failures**.
- Requires consensus protocols and redundancy to tolerate.
- Example in distributed systems: Faulty or compromised node sending inconsistent messages to replicas.

Practical Example

- Blockchain node sending invalid transactions.
- Microservices producing conflicting results intermittently.

Dependability vs Security — Overview

Key Concepts

- Distributed systems aim to be dependable and secure.
- **Dependability:** System continues to operate correctly despite faults.
- **Security:** System protected against deliberate malicious actions.
- Both require anticipating failures and handling them efficiently.

Observation

- Dependability focuses on accidental faults.
- Security addresses deliberate, malicious failures.

Omission Failures — Impact on Dependability and Security

Concept

- **Omission failures:** A component fails to perform an action it should have.
- Examples:
  - Payment system fails to record a transaction.
  - IoT sensor does not send its measurement.
- Observations:
  - Unintentional omission → dependability problem.
  - Deliberate omission → security issue (e.g., tampering with logs).

Practical Notes

- Short downtime can appear like omission failure.
- Redundancy and monitoring can detect and mitigate omissions.

Commission Failures — Impact on Dependability and Security

Concept

- **Commission failures:** A component performs an action it should not have.
- Examples:
  - Unauthorized access or data modification.
  - Sending erroneous messages in a distributed system.
- Observations:
  - Accidental commission → rare dependability issue.
  - Deliberate commission → security attack.
  - Hard to distinguish deliberate vs accidental actions.

Practical Notes

- Logging and auditing can help identify commission failures.
- Security protocols are essential to prevent malicious commission.

Halting Failures — Scenario

Understanding Halting Failures

- A halting failure occurs when a component stops operating and no longer responds.
- Example: Component C no longer perceives any activity from C^* — is it a crash or a temporary omission/timing failure?
- Detecting halting failures requires careful system modeling and monitoring.

Observation

- Halting failures appear as unresponsive nodes.
- Redundant monitoring and heartbeat messages help detect potential halts.

Halting Failures — Asynchronous vs Synchronous Systems

System Timing Assumptions

- **Asynchronous systems:** No assumptions about process speeds or message delays.
  - Cannot reliably detect crash or halting failures.
- **Synchronous systems:** Execution speeds and message delivery times are bounded.
  - Can reliably detect omission and timing failures.
- **Partially synchronous systems:** Usually behave synchronously, but timing bounds may be violated occasionally.
  - Most of the time, crash failures can be reliably detected.

Practical Note

- Heartbeats and timeout mechanisms are used in partially synchronous systems.
- Many distributed algorithms assume partial synchrony.

Halting Failures — Assumptions We Can Make

Understanding Halting Assumptions

- Halting failures can be categorized based on what assumptions we can safely make about failure detection.
- These assumptions help in designing fault-tolerant systems and distributed algorithms.
- Key halting assumptions include:
  - Fail-stop
  - Fail-noisy
  - Fail-silent
  - Fail-safe
  - Fail-arbitrary

Practical Notes

- Fail-stop: Node crashes but is detectable via heartbeat.
- Fail-noisy: Node crashes, but monitoring eventually notices it.

Halting Failures — Examples in Practice

Real-World Scenarios

- **Fail-stop:** Cloud VM crashes are immediately detected by monitoring services; failover occurs.
- **Fail-noisy:** IoT devices lose connectivity; detection happens after retries or timeouts.
- **Fail-silent:** Microservices drop messages silently.
- **Fail-safe:** Database replicas ignore corrupt writes but remain operational.
- **Fail-arbitrary:** Malicious nodes in blockchain networks attempting invalid transactions.

Key Takeaways

- Different failure assumptions require different detection and recovery strategies.

Redundancy for Failure Masking — Types

Understanding Redundancy

- Redundancy is a key technique to mask failures and ensure system dependability.
- By adding extra resources (information, time, or physical components), failures can be tolerated without affecting overall system operation.
- **Types of Redundancy:**
  - **Information redundancy:** Extra bits added to data units; errors can be detected and corrected.
  - **Time redundancy:** Operations can be repeated to handle transient faults.

Examples in Practice

- Information redundancy: Error-correcting codes (ECC) in RAM or disk storage.
- Time redundancy: Retransmission in network protocols.

Redundancy for Failure Masking — Engineering Notes

Design Considerations

- Choose redundancy type based on fault characteristics:
  - Transient faults → time redundancy is effective.
  - Permanent hardware faults → physical redundancy is essential.
  - Data corruption → information redundancy (checksums, ECC).
- Redundancy introduces cost and complexity; balance reliability gains against these factors.
- Combine multiple types for critical systems (e.g., RAID arrays + ECC + retry mechanisms).

Key Points

- Redundancy masks failures without fixing the root cause immediately.
- It is widely used in distributed, cloud, and high-availability systems.

Process Resilience

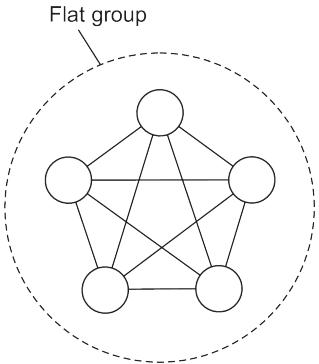
Understanding Process Resilience

- Process resilience ensures system stability by **protecting against malfunctioning processes**.
- Achieved through **process replication**: multiple instances of a process operate together to mask failures.
- Organize processes into **process groups** for coordinated fault-tolerance:
  - **Flat groups:** All processes are peers, equal responsibilities.
  - **Hierarchical groups:** Processes organized in layers; higher layers coordinate or monitor lower layers.

Examples in Practice

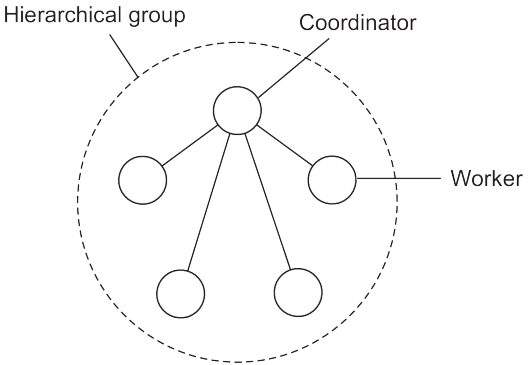
- Database clusters: Multiple nodes replicate transactions to tolerate crashes.
- Microservices: Redundant service instances.

Detecting and adjusting incorrect times



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Detecting and adjusting incorrect times



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

k-Fault Tolerant Groups — Concept

Definition

- A **k-fault tolerant group** can mask up to **k concurrent member failures**.
- **Halting failures (crash/omission/timing):** Need $k + 1$ members; one correct member ensures correctness.

Example

- A database cluster with 3 nodes ($k = 2$) can tolerate 2 crashes and still provide correct results.

k-Fault Tolerant Groups — Byzantine Failures

Arbitrary Failures

- **Arbitrary (Byzantine) failures:** Need $2k + 1$ members; majority vote ensures correct results.
- **Assumptions:**
  - All members are identical.
  - All members process commands in the same order.

Example

- Blockchain networks: Require 3f+1 nodes to tolerate f malicious nodes.
- Microservices redundancy: Majority voting ensures correct service responses.

Consensus — Prerequisite

Prerequisite for Consensus

- In a fault-tolerant process group, each **nonfaulty process** executes the same commands.
- Commands are executed in the **same order** by all nonfaulty members.
- Ensures that all nonfaulty processes have a consistent internal state.
- Forms the basis for agreement on future actions in distributed systems.

Example

- Distributed databases: Each node must process transactions in the same order to maintain consistency.

Consensus — Reformulation

Need for Agreement

- Nonfaulty group members must **reach consensus** on which command to execute next.
- Prevents inconsistencies due to delayed or missing messages.
- Consensus protocols (e.g., Paxos, Raft) are designed to achieve this under different failure models.
- Crucial for reliable and fault-tolerant distributed systems.

Example

- Leader election in Raft ensures all nodes agree on the next command.
- Paxos guarantees all nonfaulty nodes decide the same value.

Recovery Techniques — Checkpointing

Checkpointing

- **Definition:** Periodically saving the state of a process or system to stable storage.
- **Purpose:** Allows recovery from failures by rolling back to the last saved state.
- **Types:**
  - *Coordinated checkpoints:* All processes synchronize to save states together.
  - *Uncoordinated checkpoints:* Processes save states independently; may require recovery protocols to avoid inconsistency.

Example

- Long-running scientific simulations: Save checkpoints every hour.
- Databases: Periodically save transaction logs and snapshots.

Recovery Techniques — Message Logging & Rollback

Message Logging and Rollback

- **Message Logging:** Recording messages received by processes to reconstruct state after failure.
- **Rollback:** Reverting process to a previous consistent state after detecting a failure.
- **Advantages:**
  - Supports fine-grained recovery without restarting the whole system.
  - Works in conjunction with checkpointing for complete fault recovery.

Example

- Distributed online games: Rollback to previous game state after a crash.
- Banking systems: Message logs enable recovery of pending transactions.

Questions?

Thank you for your attention!

References

Suggested References

1. M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., Version 4.02, February 2024.
2. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2012.
3. N. Lynch, *Distributed Algorithms*, 2nd ed., Morgan Kaufmann, 1996.
4. R. E. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 4th ed., Pearson, 2024.