Unit 6: Software Fault Tolerance (5 hours)

Introduction to Software Fault Tolerance

What is Fault Tolerance?

Fault tolerance is the ability of a system (such as a computer, network or software) to continue operating properly even when one or more of its components fail. The goal is to ensure **high availability, reliability, and uninterrupted service** despite hardware failures, software bugs, or external disruptions.

Key Aspects of Fault Tolerance:

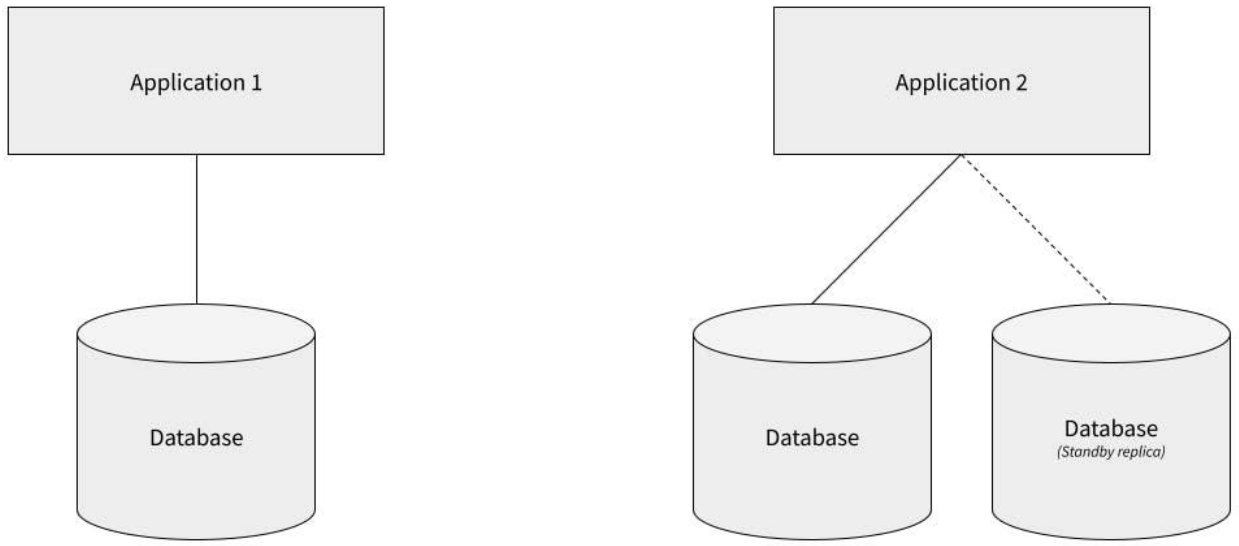
1. **Failure Detection** – The system must identify faults quickly (e.g., through health checks, timeouts, or redundancy checks).
2. **Recovery Mechanisms** – Automatically switch to backup systems or repair issues without human intervention.
3. **Redundancy** – Duplicate critical components (e.g., servers, databases, power supplies) so that if one fails, another takes over.
4. **Graceful Degradation** – If a failure occurs, the system reduces functionality rather than crashing entirely.

Examples of Fault Tolerance:

- **Hardware:** RAID (Redundant Array of Independent Disks) keeps data accessible even if one disk fails.
- **Cloud Computing:** AWS, Google Cloud, and Azure distribute workloads across multiple servers to prevent single points of failure.
- **Networking:** Routers and switches use backup paths if the primary connection fails.
- **Software:** Databases with replication (e.g., PostgreSQL, MongoDB) ensure data remains available if a node goes down.

Fault Tolerance vs. High Availability (HA):

- **Fault Tolerance** aims for **zero downtime** (seamless operation during failures).
- **High Availability** minimizes downtime but may have brief interruptions during failover.
-
- *Let's try to understand more about fault tolerance with example.*
- For example, here's a simple demonstration of comparative fault tolerance in the database layer. In the diagram below, **Application 1** is connected to a single database instance. **Application 2** is connected to two database instances — the primary database and a standby replica.



-
  - In this scenario, Application 2 is more fault tolerant. If its primary database goes offline, it can switch over to the standby replica and continue operating as usual.

- Application 1 is not fault tolerant. If its database goes offline, all application features that require access to the database will cease to function.

The above scenario is just a simple example though. In reality, fault tolerance must be considered in every layer of a system (not just the database), and there are degrees of fault tolerance. While Application 2 is more fault tolerant than Application 1, it's still less fault tolerant than many modern applications.

Fault tolerance can also be achieved in a variety of ways. These are some of the most common approaches to achieving fault tolerance:

Multiple hardware systems capable of doing the same work. For example, Application 2 in our diagram above could have its two databases located on two different physical servers, potentially in different locations. That way, if the primary database server experiences an error, a hardware failure, or a power outage, the other server might not be affected.

Multiple instances of software capable of doing the same work. For example, many modern applications make use of **containerization (Containerization** is a method of packaging software so that it can run reliably and consistently across different computing environments. It involves bundling an application along with all its dependencies (like libraries, configuration files, and binaries) into a single, lightweight unit called a **container**.

) platforms such as Kubernetes so that they can run multiple instances of software services. One reason for this is so that if one instance encounters an error or goes offline, traffic can be routed to other instances to maintain application functionality.

Backup sources of power, such as generators, are often used in on-premises systems to protect the application from being knocked offline if power to the servers is impacted by, for example, the weather. That type of outage is more common than you might expect

Fault tolerance goals

Building fault-tolerant systems is more complex and generally also more expensive. If we think back to our simple example from earlier, Application 2 is more fault tolerant, but it also has to pay for and maintain an additional database server. Thus, it's important to assess the level of fault tolerance your application requires and build your system accordingly.

Normal functioning vs. graceful degradation

When designing fault-tolerant systems, you may want the application to remain online and fully functional at all times. In this case, your goal is **normal functioning** — you want your application, and by extension the user's experience, to remain unchanged even if an element of your system fails or is knocked offline.

Another approach is aiming for what's called **graceful degradation,** where outages and errors are allowed to impact functionality and degrade the user experience, but not knock the application out entirely. For example, if a software instance encounters an

error during a period of heavy traffic, the application experience may slow for other users, and certain features might become unavailable.

Building for normal functioning obviously provides for a superior user experience, but it’s also generally more expensive. The goals for a specific application, then, might depend on what it’s used for. Mission-critical applications and systems will likely need to maintain normal functioning in all but the most dire of disasters, whereas it might make economic sense to allow less essential systems to degrade gracefully.

System Survivability and Measuring System Survivability

What is system Survivability?

System survivability is a subset of **dependability**, focusing on a system's ability to **continue functioning (possibly in a degraded mode) during and after failures, attacks, or adverse conditions**. Unlike basic fault tolerance (which handles known failures), survivability also deals with **unexpected disruptions**, such as cyberattacks, natural disasters, or unforeseen software errors.

Key Concepts:

1. Relationship with Dependability

Dependability encompasses multiple attributes:

- **Reliability** (continuous correct service)
- **Availability** (readiness for use)
- **Safety** (no catastrophic failures)
- **Security** (protection against attacks)
- **Maintainability** (ease of repair)
- **Integrity** (no unauthorized modifications)

Survivability extends these by ensuring the system can **detect, resist, recover, and adapt** to disruptions while maintaining critical functions.

2. Core Principles of Survivability

A survivable system must:

1. **Resist Attacks/Failures** – Prevent or mitigate damage (e.g., firewalls, redundancy).
2. **Recognize Threats** – Detect intrusions or failures (e.g., anomaly detection, logging).
3. **Recover Quickly** – Restore essential services (e.g., failover mechanisms, backups).
4. **Adapt & Evolve** – Modify behavior to maintain functionality (e.g., load balancing, throttling).

3. Techniques for Enhancing Survivability

a) Redundancy & Diversity

- **Hardware Redundancy** (backup servers, RAID storage)
- **Software Diversity** (multiple implementations of critical functions to avoid common failures)
- **Geographical Distribution** (cloud-based multi-region deployments)

b) Fault Containment & Isolation

- **Microservices Architecture** – Limits blast radius of failures.
- **Sandboxing** – Isolates compromised components.
- **Circuit Breakers** – Prevents cascading failures.(failures in one part leads to another part failures.)

c) Self-Healing & Autonomous Recovery

- **Rollback Mechanisms** – Reverts to a stable state after failure.
- **Dynamic Reconfiguration** – Adjusts resources based on demand/threats.

d) Intrusion Tolerance & Cyber-Resilience

- **Byzantine Fault Tolerance (BFT)** – Handles malicious nodes in distributed systems.
- **Honeypots & Deception** – Diverts attackers from real systems.

4. Real-World Examples

- **NASA Space Probes** – Must operate despite radiation, hardware faults, and communication delays.
- **Financial Systems (e.g., Stock Exchanges)** – Use redundant data centers to prevent trading halts.
- **Military Command Systems** – Designed to function even if partially compromised.
- **Cloud Services (AWS, Azure)** – Auto-scaling and multi-zone deployments ensure uptime.

5. Survivability vs. Fault Tolerance vs. High Availability

Aspect	Fault Tolerance	High Availability (HA)	Survivability
Goal	Zero downtime	Minimal downtime	Continued operation (possibly degraded)
Scope	Known failures	Planned failovers	Unknown threats (attacks, disasters)
Response	Immediate failover	Fast recovery	Adaptive recovery & evolution
Example	RAID storage	Load-balanced web servers	Cyber-resilient military systems

6. Challenges in Achieving Survivability

- **Complexity** – Managing redundancy and failover logic.
- **Performance Overhead** – Extra checks and replication slow down systems.
- **Cost** – Requires additional infrastructure.
- **Security Trade-offs** – More components mean more attack surfaces.

Measuring System Survivability

Survivability is a critical aspect of **system dependability**, but unlike traditional reliability or availability metrics, it focuses on **maintaining essential functionality during and after failures, attacks, or unexpected disruptions**. Measuring survivability requires a combination

of **quantitative metrics, qualitative assessments, and resilience evaluation frameworks**.

1. Key Metrics for Survivability

To assess survivability, we use a mix of **dependability metrics** (e.g., reliability, availability) and **resilience-specific measures** (e.g., recovery time, degradation tolerance). Below are the most important ones:

A. Core Dependability Metrics (Baseline)

These metrics help establish the system's baseline dependability:

1. **Mean Time Between Failures (MTBF)**
   - Measures how often failures occur.
   - Higher MTBF → More reliable system.
2. **Mean Time To Repair (MTTR)**
   - Time taken to recover from a failure.
   - Lower MTTR → Faster recovery.
3. **Availability (%)**
   - Availability=Uptime/Uptime +Downtime * 100
   - Survivability extends this by ensuring **critical functions remain available** even if some parts fail.
4. **Failure Rate (λ)**
   - Number of failures per unit time.

B. Survivability-Specific Metrics

These focus on **how well the system maintains operations under stress**:

1. **Degradation Ratio (DR)**
   - Measures how much performance drops during failure.
   - DR=Reduced Capacity/Full Capacity

- Example: A cloud service running at 60% capacity during an attack has DR=0.6

2. **Recovery Time Objective (RTO)**
   - Maximum acceptable time to restore critical functions.
3. **Recovery Point Objective (RPO)**
   - Maximum tolerable data loss (e.g., last backup time).
4. **Survivability Index (SI)**
   - Combines multiple factors (e.g., detection time, recovery success rate).
   - Example formula:
     SI=Successful Recoveries/Total Disruptions×1/MTTR

5. **Attack/Disruption Resistance Rate**
   - Percentage of attacks/failures the system resists without total failure.
6. **Adaptability Score**
   - Measures how well the system adjusts to new threats (e.g., via machine learning, dynamic reconfiguration).

2. **Evaluation Frameworks for Survivability**

A. Quantitative Models

1. **Markov Models**
   - Predicts survivability by modeling system states (normal, degraded, failed).
   - Used in safety-critical systems (e.g., aerospace, nuclear plants).
2.

3. **Fault Tree Analysis (FTA)**
   - Identifies potential failure paths and their impact on survivability.
4. **Reliability Block Diagrams (RBDs)**
   - Evaluates how redundancy and failover mechanisms improve survivability.

B. Qualitative Assessments

1. **Survivability Levels (SL)**
   - **SL0**: No survivability (total failure on disruption).
   - **SL1**: Minimal functionality retained.
   - **SL2**: Partial recovery with manual intervention.
   - **SL3**: Automatic recovery with minor degradation.
   - **SL4**: Full resilience (self-healing, no downtime).
2. **Cyber-Resilience Frameworks (e.g., MITRE, NIST SP 800-160)**
   - Assess survivability against cyber threats.
   - Includes metrics like:

- **Time to Detect (TTD): Time interval between the occurrence of a fault** and the **moment the system or monitoring mechanism detects it**. The shorter the TTD, the quicker a system can react or recover. Longer TTD means the attacker can attack for longer period.
- **Time to Respond (TTR):** The amount of **time taken to initiate a corrective or protective action** after a fault, failure, or attack has been detected. The shorter TTR means faster containment of issues and better survivability.
- **Time to Recover (TTR):** The **amount of time it takes to restore a system or service** back to its **normal operating condition** after a fault, failure, or disruption has occurred and a response has been initiated. The shorter TTR improves resilience and survivability and helps meet Service Level Agreements.

- **Design Patterns for Fault Tolerance**

1. Redundancy Pattern

Concept: Duplicate critical components so if one fails, another can take over.

Types:

- **Hardware Redundancy** (e.g., backup servers)
- **Software Redundancy** (e.g., N-version programming)
- **Data Redundancy** (e.g., RAID, replicated databases)

Example:
Cloud platforms using **multi-zone replication** for database availability.

2. Retry Pattern

Concept: If a transient failure occurs (like a network timeout), retry the operation after a short delay.

Features:

- Often used with exponential backoff
- Needs max retry count and timeout

Used In:
Microservices communication, HTTP API calls

3. Circuit Breaker Pattern

Concept: Prevents a system from repeatedly calling a failing service.

How it works:

- If failures exceed a threshold, the circuit "opens"
- Calls are blocked until the system stabilizes

Used In:
Distributed systems, to isolate failing services and avoid cascading failures

4. Failover Pattern

Concept: Automatically switch to a backup system when the primary fails.

Variants:

- **Active-Passive**: Backup is idle until needed
- **Active-Active**: All instances serve traffic, share load

Used In:
Databases, load balancers, clustered apps

5. Watchdog Timer Pattern

Concept: A separate component monitors the main system, and **reboots or restarts it** if it becomes unresponsive.

Used In:
Embedded systems, real-time systems, robotics

6. Health Check Pattern

Concept: Continuously monitor components to detect failure early.

Examples:

- Kubernetes **liveness** and **readiness** probes
- Load balancer health checks

7. Graceful Degradation Pattern

Concept: When full functionality can't be provided, offer a reduced (but still useful) level of service.

Example:

- A video app downgrading quality instead of stopping playback.
- E-commerce site disables recommendations during high load.

8. Bulkhead Pattern

Concept: Isolate parts of the system so a failure in one area doesn't bring down the entire system.

Named After: Ship compartments designed to limit flooding.

Used In:
Containerized apps, micro services, concurrent threads

9. Self-Healing Pattern

Concept: The system detects anomalies and autonomously attempts to fix them.

Examples:

- Auto-restart crashed containers

- Auto-scale services under load

10. Audit and Monitoring Pattern

Concept: Continuously log and analyze data to detect and respond to faults before they escalate.

Tools:
ELK Stack, Prometheus + Grafana, AWS Cloud Watch

- **Fault Minimization and Tolerance Techniques**

Fault minimization techniques are methods used in software engineering to reduce the number of errors or faults in a software system. Some common techniques include:

1. **Code Reviews**: Code review is a process where code is evaluated by peers to identify potential problems and suggest improvements.
2. **Unit Testing**: Unit testing involves writing automated tests for individual units of code to ensure that each component works as expected.
3. **Test-Driven Development**: Test-driven development (TDD) is a software development process where unit tests are written before writing the code, ensuring that the code meets the requirements.
4. **Continuous Integration and Continuous Deployment (CI/CD**): CI/CD is a software engineering practice where code changes are automatically built, tested, and deployed to production.
5. **Static Code Analysis**: Static code analysis is the process of automatically analyzing code to find potential problems without actually executing it.
6. **Design and Architecture Reviews**: Reviews of design and architecture help ensure that the overall design of the software is correct and scalable.

7. **Error Handling**: Proper error handling helps prevent the spread of faults and ensures that the software can handle unexpected situations gracefully.
8. **Documentation**: Clear and up-to-date documentation can help reduce the likelihood of errors by providing a clear understanding of the software's functionality.

A **fault** is a defect in the program that, when executed under particular conditions causes a different result of the program operation from its requirements. It is the condition that causes the software to fail to perform its required functionality. The following are the techniques used to reduce faults in software:

1. **Fault Prevention -** Fault Prevention/Avoidance strategies identify all potential areas where a fault can occur and close the gaps. These prevention strategies address system requirements and specifications, software design methods, re-usability, or formal methods. They are employed during the development phase of the software to avoid or prevent fault occurrence. They contribute to the system dependability through the rigorous specification of the system requirements, programming methods, and software re-usability. But it is difficult to quantify the impact of fault avoidance strategies on system dependability. So, despite fault prevention efforts, faults are created, so fault removal is needed.
2. **Fault Removal -** Fault removal strategies are dependability-enhancing techniques employed during verification and validation. They improve by detecting existing faults and eliminating the defected faults. They are employed after the development phase of the software to contribute to the validation of the software. Common fault removal techniques involve testing. It follows that minimizing component size and interrelationship maximizes accurate testing. The difficulties encountered in testing programs are often related to the prohibitive costs and exhaustive testing. Therefore, fault removal is imperfect, hence fault tolerance is needed.

3. **Fault Tolerance -** Fault tolerance includes dependability-enhancing techniques that are used during the validation of software to estimate the presence of faults. It is used to reduce system design faults and enhance the reliability of the software. Fault tolerance techniques are employed during the development phase of the software which enables the system to tolerate faults remaining in the system after its development and provide operation complying with the requirements specification in spite of faults. Therefore, when a fault occurs it prevents the system failure.

Fault prevention, fault removal, and fault tolerance represent the successive lines of defense against the contingency of faults of software systems and their impact on system. Despite the fact, that the benefits of each of these techniques are remarkable, the law of diminishing returns advocates that they should be used in unison where each one is applied wherever it is most effective.

Advantages and Disadvantages:

Advantages of fault minimization techniques in software engineering:

1. **Improved software quality**: By reducing the number of faults in a software system, the quality of the software is improved, resulting in a better user experience.
2. **Increased reliability**: With fewer faults, the software is more reliable and less likely to fail, leading to fewer downtime incidents and a better reputation for the company.
3. **Reduced maintenance costs**: By reducing the number of faults in a software system, the cost of maintenance is reduced as there are fewer issues to fix.
4. **Faster problem resolution**: By using techniques like unit testing and continuous integration, problems are caught early, making it easier and faster to resolve them.

Fault Tolerance Techniques

Fault-tolerance is the process of working of a system in a proper way in spite of the occurrence of the failures in the system. Even after performing the so many testing processes there is possibility of failure in system. Practically a system can't be made entirely error free. hence, systems are designed in such a way that in case of error availability and failure, system does the work properly and given correct result. Any system has two major components - Hardware and Software. Fault may occur in either of it. So there are separate techniques for fault-tolerance in both hardware and software. **Hardware Fault-tolerance Techniques:** Making a hardware fault-tolerance is simple as compared to software. Fault-tolerance techniques make the hardware work proper and give correct result even some fault occurs in the hardware part of the system. There are basically two techniques used for hardware fault-tolerance:

1. **BIST -** BIST stands for Build in Self-Test. System carries out the test of itself after a certain period of time again and again, that is BIST technique for hardware fault-tolerance. When system detects a fault, it switches out the faulty component and switches in the redundant of it. System basically reconfigure itself in case of fault occurrence.
2. **TMR -** TMR is Triple Modular Redundancy. Three redundant copies of critical components are generated and all these three copies are run concurrently. Voting of result of all redundant copies are done and majority result is selected. It can tolerate the occurrence of a single fault at a time.
3. **Check-pointing and Rollback Recovery -** This technique is different from above two techniques of software fault-tolerance. In this technique, system is tested each time when we perform some computation. This techniques is basically useful when there is processor failure or data corruption.

Fault Tolerance Architecture

N-version Programming: **N-Version Programming** is a **software fault tolerance technique** that involves developing **multiple functionally equivalent versions** of a program independently and running them in parallel to ensure correct execution, even if some versions contain faults. N-Version Programming involves implementing **N independently developed versions** of a software module from the same specification. The outputs of these versions are then compared using a **voting mechanism** to determine the correct result.

Goals of NVP

- Increase **reliability** and **fault tolerance**.
- Detect and tolerate **design and implementation faults** (as opposed to just transient faults).
- Avoid **common-mode failures** by introducing **design diversity**.

Key Components of NVP

Component	Description
N Versions	Multiple independently implemented versions (usually 3 or more) of the same function/module
Voter	A decision component that collects outputs from each version and selects the majority or correct one
Consensus Algorithm	Usually majority voting, but can use weighted or median voting
Specification	All versions are developed from the **same specification**, which must be unambiguous and complete

How It Works (Execution Flow)

1. **Input is received** by all N versions simultaneously.
2. Each version **processes the input independently**.
3. Outputs are sent to a **voter**.
4. Voter uses a **voting algorithm** to decide the final result.
    - Majority voting: If at least ⌈N/2⌉ versions agree, their result is accepted.
    - Median voting or consensus methods in case of numeric or approximate outputs.
5. If a discrepancy is detected, the system may:
    - **Log an error**,
    - **Isolate the faulty version**,
    - **Trigger alerts**, or
    - **Replace faulty module** (if dynamically updatable).

Example Scenario: NVP in Aerospace System

Imagine a flight control software that calculates the position of control surfaces on an aircraft:

- You develop 3 versions of the control logic:
    - **Version A**: Implemented in C by Team 1
    - **Version B**: Implemented in Ada by Team 2
    - **Version C**: Implemented in Rust by Team 3
- All versions receive the same sensor data.
- A **voter module** compares their outputs:
    - If A and C agree but B differs, then A and C's output is used.

This ensures that a bug in one version (e.g., due to a coding mistake or compiler issue) does not lead to failure.

Advantages

- High **resilience** to software faults.

- Can **detect and tolerate design bugs** (not just hardware faults).
- Especially useful in **safety-critical systems** (aerospace, nuclear, medical devices).

Challenges and Limitations

Challenge	Description
Cost	Very high cost – developing multiple versions triples effort.
Specification Consistency	Ambiguities in spec can cause all versions to be faulty in the same way (common-mode failure).
Voter Complexity	Designing a fair, deterministic, and reliable voter is complex.
Performance	Increased runtime overhead due to multiple versions running concurrently.
Development Independence	True independence is hard to achieve – teams might interpret specs similarly or share mental models/tools.

Recovery Blocks

Recovery blocks are a fault tolerance technique in computer programming that uses multiple alternative code blocks (called "alternates") to perform the same function. If the primary alternate fails an acceptance test, the block switches to an alternate and re-runs it until one passes. This mechanism helps ensure that the program can continue to function even if a fault occurs.

Core Concept:

- A recovery block consists of a primary alternate (the main code) and one or more secondary alternates.

- Each alternate is designed to perform the same task, but with potentially different algorithms or approaches.
- A recovery block also includes an "acceptance test" that verifies the results of each alternate.
-

2. **How it Works**:

- When a recovery block is executed, the primary alternate is first executed.
- After the primary alternate completes, the acceptance test is performed to verify that the results are correct and valid.
- If the acceptance test passes, the recovery block is considered successful, and the results are returned to the calling code.
- If the acceptance test fails, the recovery block switches to the next alternate and attempts to execute it.
- This process repeats until one of the alternates passes the acceptance test or all alternates have been tried.

Benefits of Recovery Blocks:

- **Fault Tolerance:**

  If one alternate fails, the program can switch to another alternate, potentially continuing to operate correctly.
- **Error Detection and Recovery:**

  The acceptance test provides a mechanism to detect errors and allows for recovery by switching to an alternate.
- **Increased Reliability:**

  By incorporating redundancy and error recovery, recovery blocks can increase the reliability of software systems.
- **Cost-Effective:**
  Recovery blocks can be a cost-effective way to implement fault tolerance compared to using multiple hardware channels.

4. **Example**:

- Imagine a function that calculates a value. The primary alternate might use a specific algorithm, while an alternate might use a different, more robust algorithm.
- The acceptance test might verify that the calculated value falls within a certain range or meets specific criteria.
- If the primary algorithm fails (e.g., due to a bug), the alternate algorithm can be used, and the acceptance test will ensure that the result is still valid.