আন্তর্জাতিক ইসলামী বিশ্ববিদ্যালয় চট্টগ্রাম

International Islamic University Chittagong

Challenges and Solutions in Distributed Database Systems

Presented By

Mujibul Haque Tanim

ID: MC-243102

Course Teacher

Dr. Mohammad Aman Ullah

Professor,

Dept of Computer Science & Engineering, IIUC.

Table of Contents

Authentication in Distributed System

Definition: The process of verifying the identity of a principal (user, device, service) attempting to access resources or perform actions.

Authentication is typically based on:

- **Something you know**: Passwords, PINs.
- **Something you have**: Tokens, smart cards, devices.
- **Something you are**: Biometrics like fingerprints, retinal patterns.

Classification of Authentication Protocol

- **Password-Based:** Authenticates using user-provided passwords, often simple but vulnerable to attacks.
- **Token-Based:** Uses physical or digital tokens (e.g., smart cards) for secure, hardware-backed authentication.
- **Biometric-Based:** Relies on unique biological traits (e.g., fingerprints) for high-security, user-specific authentication.
- **Certificate-Based:** Employs digital certificates (e.g., X.509) to verify identity via public key infrastructure.
- **Multi-Factor Authentication (MFA):** Combines multiple methods (e.g., password + token) for enhanced security.

Challenges & Solutions

Challenges in Authentication Protocols

- **Security Vulnerabilities**: Protocols like Kerberos face risks from replay attacks or stolen tickets.
- **User Experience**: OAuth’s token management can confuse users, leading to misuse or errors.
- **Performance Overhead**: TLS/SSL handshakes introduce latency due to cryptographic computations.

Solutions to Challenges

- **Security Vulnerabilities**: Implement time-stamping and nonces in Kerberos to prevent replay attacks.
- **User Experience**: Simplify OAuth flows with clear user consent interfaces and documentation.
- **Performance Overhead**: Use TLS session resumption to reduce handshake latency in repeated connections.

Distributed Shared Memory

- **Introduction**: Shared memory enables processes in distributed systems to access common data, mimicking a centralized memory model.
- **Advantages**:
  - **Simplified programming:** Processes communicate via memory rather than explicit messages.
  - **High performance:** Fast data access compared to message-passing.
  - Scalability: Supports concurrent operations across distributed nodes.

Memory Consistency Models

- **Definition**: Rules governing the order and visibility of memory operations across processes.
- **Types**:
  - **Sequential Consistency**: All operations appear in a global sequential order.
  - **Causal Consistency**: Causally related operations are seen in order.
  - **Eventual Consistency**: Updates propagate eventually, prioritizing availability.
- **Trade-offs**: Stronger consistency increases latency; weaker models enhance performance.

Synchronization and Distributed Mutual Exclusion

- **Synchronization-based Consistency**:
  - Uses locks, semaphores, or barriers to enforce consistent memory access.
  - Ensures data integrity in concurrent environments.
- **Shared Memory Mutual Exclusion**:
  - Prevents simultaneous access to shared resources using locks or atomic operations.
- **Distributed Mutual Exclusion Algorithms**:
  - **Centralized**: Single coordinator grants access.
  - **Token-based**: Token possession allows resource access.
  - **Ricart-Agrawala**: Timestamp-based request ordering.
- **Wait-Freedom and Synchronization Techniques**:
  - **Wait-Freedom**: Guarantees process completion without waiting, using atomic operations like compare-and-swap.
  - **Techniques**: Lock-free data structures, transactional memory for robust synchronization.

Introduction to Dead-lock Detection

- **Definition**: Deadlock occurs when multiple transactions wait indefinitely for resources held by each other in a distributed database.

- **Impact**: Causes system delays, reduced throughput, and potential transaction failures.

- **Key Challenge**: Detecting and resolving deadlocks across distributed nodes efficiently.

Models of Dead-lock

- **Wait-for Graph (WFG)**: Directed graph showing transactions waiting for resources; a cycle indicates a deadlock.
- **Resource Allocation Graph**: Models resource requests and assignments to identify conflicts.
- **Distributed Nature**: Global deadlocks span multiple nodes, complicating detection compared to local deadlocks.

Deadlock Handling Strategies and Detection Issues

- **Strategies**:
  - **Prevention**: Enforce protocols to avoid deadlock conditions (e.g., strict resource ordering).
  - **Avoidance**: Dynamically allocate resources to prevent deadlock formation.
  - **Detection and Resolution**: Identify deadlocks via WFG and abort/rollback transactions.
- **Issues in Detection**:
  - **Scalability**: High overhead in monitoring large distributed systems.
  - **False Positives**: Inaccurate detection due to incomplete global state information.
  - **Communication Delays**: Node coordination increases latency in detecting global deadlocks.

Check-pointing and Rollback

- **Checkpointing:** Periodically save the state of each database node.Essential for minimizing data loss and recovery time.
- **Types:**
  - **Coordinated:** All nodes checkpoint together (ensures global consistency).
  - **Uncoordinated:** Nodes checkpoint independently (more complex recovery).

- **Rollback:** Upon failure, restore the database to a previous consistent state (a checkpoint).
- **Goal:** To undo the effects of failed transactions.

Recovery and Considerations

- **Recovery:** Process of restoring the DDB to a functional state after a failure. Involves using checkpoints and logs to ensure data consistency.

- **Key Considerations:**
  - Consistency: Maintaining global database consistency across all nodes.
  - Atomicity: Ensuring transactions are either fully completed or have no effect.
  - Durability: Committed transactions are permanent.

Failure Detectors

A component that monitors processes in a distributed system and provides information about whether those processes have failed.

Purpose:

- Detect process failures
- Crucial for initiating recovery procedures
- Enable fault-tolerant distributed systems

Types of Failure Failure Detectors

- **Perfect (P):** Accurately detects all crashes immediately, never suspecting correct processes.
- **Strong (S):** Ensures at least one correct process is never suspected, may falsely suspect others.
- **Eventually Perfect (♦P):** Eventually detects all crashes accurately, may make initial mistakes.
- **Eventually Strong (♦S):** Eventually ensures one correct process is never suspected, may err initially.

Failure Detectors Properties

- **Completeness**: Every faulty process is eventually detected.
  - Strong Completeness: Every faulty process is eventually *permanently* suspected by every correct process.
  - Weak Completeness: Every faulty process is eventually permanently suspected by *some* correct process.

- **Accuracy**: No correct process is ever wrongly suspected of having failed.
  - Strong Accuracy: No correct process is ever suspected.
  - Weak Accuracy: Some correct process is never suspected.

- **Eventually Strong Completeness**: There is a time after which every faulty process is permanently suspected by every correct process.

- **Eventually Strong Accuracy**: There is a time after which some correct process is never suspected.

Challenges in Failure Detectors

- **Completeness**: Every faulty process is eventually detected.
  - Strong Completeness: Every faulty process is eventually *permanently* suspected by every correct process.
  - Weak Completeness: Every faulty process is eventually permanently suspected by *some* correct process.
- **Accuracy**: No correct process is ever wrongly suspected of having failed.
  - Strong Accuracy: No correct process is ever suspected.
  - Weak Accuracy: Some correct process is never suspected.
- **Eventually Strong Completeness**: There is a time after which every faulty process is permanently suspected by every correct process.
- **Eventually Strong Accuracy**: There is a time after which some correct process is never suspected.

Key Takeaways

- **Authentication:** Crucial for secure access and data protection in distributed systems.
- **Distributed Shared Memory:** Simplifies application development but introduces consistency challenges.
- **Deadlock Detection:** Essential to prevent system stalls; various algorithms exist, each with trade-offs.
- **Checkpointing, Rollback, and Recovery:** Vital techniques for fault tolerance and system resilience.
- **Failure Detectors:** Fundamental components for detecting node failures, enabling fault-tolerant mechanisms.

Thank You