Challenges in Distributed Systems

Parallel and distributed Computing
Prof. Maryam Saif

Introduction

A **distributed system** consists of multiple independent computers that communicate and coordinate their actions through a network to achieve a common goal.
While distributed systems provide advantages such as **scalability, resource sharing, and fault tolerance**, they also introduce significant **design and implementation challenges**.

- Two major challenges are:
  - **Heterogeneity**
  - **Fault Tolerance**

Heterogeneity

- **Heterogeneity** refers to the presence of **diverse and dissimilar components** within a distributed system.
  These differences may exist in hardware, software, networks, programming models, and administrative domains.
- **Why Heterogeneity is a Challenge?**
- Components may not be directly compatible
- Differences complicate communication and coordination
- System integration becomes complex
- Uniform performance and behavior are difficult to guarantee

Network Heterogeneity

Hybrid Networks

- Combination of multiple network types
- Example: LAN + WAN + wireless networks
- Challenges:
- Variable latency
- Different bandwidths
- Complex routing and communication protocols

Cloud Networks

- Infrastructure provided by cloud service providers
- Highly scalable and virtualized
- Challenges:
- Network latency unpredictability
- Dependence on third-party providers
- Multi-tenancy issues

Network Hetrogeneity

Private Networks

- Owned and managed by a single organization
- More control over security and performance
- Challenges:
- Limited scalability
- Cost of maintenance

Public Networks

- Example:
- Internet
- Open and widely accessible
- Challenges:
- Security threats
- Variable performance
- Packet loss and congestion

Computer Hardware Heterogeneity

- Distributed systems often include machines with:
- Different CPU architectures (x86, ARM)
- Different processing speeds
- Different memory sizes
- Different storage types
- **Challenges**
  - Performance imbalance
  - Difficult load balancing
  - Architecture-dependent code optimization

Operating System Heterogeneity

- **Different nodes may run:**
  - Windows
  - Linux
  - macOS
  - UNIX variants
- **Challenges**
  - Differences in system calls
  - Process management variations
  - File system differences
  - Resource management incompatibility

Programming Language Heterogeneity

- **Distributed components may be written in:**
  - C/C++, Java, Python, Go, JavaScript
- **Challenges**
  - Data representation differences
  - Interoperability issues
  - Different runtime environments
- **Solution Approaches**
  - Language-neutral interfaces (IDLs)
  - APIs
  - Serialization formats (JSON, XML, Protocol Buffers)

Heterogeneity Due to Different Developers

- **Distributed systems are often built by:**
  - Multiple teams
  - Different organizations
  - Different time periods
- **Challenges**
  - Inconsistent design choices
  - Different coding standards
  - Varying documentation quality
  - Integration difficulties

Mobile Code Heterogeneity

- **Mobile code** refers to code that can move across systems during execution.
- **Examples:**
  - Java applets
  - Mobile agents
  - Containerized applications
- **Challenges**
  - Security risks
  - Execution environment compatibility
  - Resource availability differences

Middleware Heterogeneity

- **Middleware** acts as a layer between applications and the underlying system.
- Examples:
  - CORBA, RPC, RMI
- Message-oriented middleware
- **Challenges**
  - Different middleware standards
  - Compatibility issues
  - Performance overhead
  - Version mismatches

Role of Middleware in Handling Heterogeneity

- Middleware helps by:
  - Providing uniform interfaces
  - Masking low-level differences
  - Enabling interoperability
  - Supporting communication across heterogeneous components

Types of faults in Distributed System

- **Hardware faults**
  - Disk crashes
  - CPU failure
  - Power outages
- **Software faults**
  - Bugs
  - Memory leaks
  - Deadlocks
- **Network faults**
  - Message loss
  - Network partition
  - Latency spikes
- **Operational faults**
  - Configuration errors
  - Human mistakes

Fault Tolerance in Distributed Systems

- **Fault tolerance is the ability of a distributed system to:**
  - Continue functioning correctly
  - Even in the presence of faults or failures
- **Failures are inevitable due to:**
  - Hardware faults
  - Software bugs
  - Network failures
  - Human errors

Attributes of Fault Tolerance

1. **Availability**

- Availability refers to the probability that a system is operational and accessible when required.
  - Measured as uptime percentage
  - High availability systems aim for minimal downtime

Example:

99.9% availability → system down for ~8.7 hours/year

- **Challenges**
  - Network partitions
  - Hardware failures
  - Maintenance downtime

Attributes of Fault Tolerance

2. Reliability:

Reliability is the ability of a system to perform correctly over a specified period without failure.

"Will the system continue to work correctly over time?"

- **Challenges:**
  - Data inconsistency
  - Message duplication or loss
  - Clock synchronization issues
- **Techniques to Improve Reliability:**
  - Fault detection and recovery
  - Transaction management
  - Checkpointing and rollback

Attributes of Fault Tolerance

3. Safety:

- **Safety** ensures that the system never reaches an incorrect or dangerous state, even in the presence of faults.

"Nothing bad ever happens."

- **Challenges:**
  - Inconsistent states
  - Incorrect coordination among nodes
- **Techniques to Ensure Safety:**
  - Atomic transactions
  - Distributed locking
  - Strict validation rules

Attributes of Fault Tolerance

4. Maintainability:

- **Maintainability** is the ease with which a system can be repaired, modified, or updated.

  "How quickly and easily can the system be fixed or upgraded?"

- **Challenges in Distributed Systems:**
  - Large number of components
  - Complex interdependencies
  - Difficult debugging
  - Heterogeneous environments
- **Techniques to Improve Maintainability:**
  - Modular architecture
  - Monitoring and logging
  - Self-healing mechanisms