# Distributed Systems - Full Exam Notes

## Chapter 01: Introduction to Distributed Systems

Chapter 01: Introduction to Distributed Systems

1. What is a Distributed System?
- A collection of independent computers that appear to the user as a single coherent system.
- Designed to share resources, coordinate actions, and offer services despite physical distribution.

2. Distributed vs Decentralized Systems
- Centralized: One central point of control.
- Decentralized: No central control, but not necessarily interconnected.
- Distributed: Interconnected systems that coordinate actions and share responsibilities.

3. Design Goals of Distributed Systems
- Resource Sharing: E.g., cloud storage, email hosting.
- Openness: Components can interact with other systems via standard interfaces.
- Scalability: The system should perform well as the number of users/resources grows.
- Fault Tolerance: Continue operation despite partial failures.

4. Transparency in Distributed Systems
- Access Transparency: Uniform access to resources.
- Location Transparency: Resource location is hidden.
- Concurrency Transparency: Resources accessed concurrently without interference.
- Replication Transparency: Multiple instances appear as one.
- Failure Transparency: Recover from failures seamlessly.
- Migration Transparency: Resources can move without affecting the user.
- Performance Transparency: System adjusts to load.
- Scaling Transparency: Scales without affecting structure or performance.

5. Openness and Interoperability
- Open Systems: Use standard interfaces and protocols to interoperate (e.g., HTTP, SMTP).
- Middleware: Acts as a layer that supports interoperability, data marshaling, naming, and security.

6. Fault Tolerance and Dependability
- Dependability: Includes availability, reliability, safety, maintainability.
- Metrics:
  - MTTF: Mean Time To Failure.
  - MTTR: Mean Time To Repair.

- MTBF: Mean Time Between Failures = MTTF + MTTR.

7. Security Goals
- Confidentiality: Only authorized access to data.
- Integrity: Data cannot be altered improperly.
- Authentication: Verify identity.
- Authorization: Access control.
- Trust: Assumptions about entity behavior.
- Cryptographic Mechanisms: Symmetric/Asymmetric encryption, Hashing (e.g., H(data)).

## Chapter 02: Architectures of Distributed Systems

Chapter 02: Architectures of Distributed Systems

1. Architectural Styles
- Components: Replaceable modules with interfaces.
- Connectors: Mechanisms like RPC or message-passing that mediate communication.
- Configuration: The arrangement of components and connectors.

2. Layered Architectures
- Organizes systems in layers (presentation, logic, data).
- Example: Web systems (browser ↔ app logic ↔ database).

3. Object-based Style
- System built using objects connected through method calls.
- Emphasizes encapsulation, modularity.

4. Service-Oriented Architecture (SOA)
- System organized around services.
- Microservices: Small, independently deployed components.
- RESTful Services: Expose resources using URIs and HTTP methods (GET, POST, PUT, DELETE).

5. RESTful Architecture
- Stateless interactions, uniform interface.
- Resources are identifiable by URIs.
- Messages are self-contained.

6. Middleware in Distributed Systems
- Middleware provides standardized services: communication, naming, security, etc.
- Acts as a layer between OS and applications.

7. Coordination and Communication Styles

- Temporal Coupling: Sender and receiver must be active at the same time (e.g., RPC).
- Referential Coupling: Parties know each other's names/addresses.

8. Publish-Subscribe Architectures
- Decouples senders and receivers.
- Topic-based vs Content-based subscriptions.

9. Linda Tuple Space Example
- Shared data space with operations:
  - `in(t)`: Remove matching tuple.
  - `rd(t)`: Read matching tuple.
  - `out(t)`: Add tuple.
- Models asynchronous, decoupled interaction.

## Chapter 03: Processes

Chapter 03: Processes

1. Threads and Processes
- Thread: Minimal unit of execution (runs within a process).
- Process: Execution environment that may contain multiple threads.

2. Context Switching
- Thread Context: Includes registers and stack pointer.
- Process Context: Includes MMU settings, address space.
- Thread switching is cheaper than process switching.

3. Benefits of Threads
- Avoid Blocking: Non-blocking I/O through separate threads.
- Parallelism: Threads on multicore processors.
- Efficiency: Cheaper to create/destroy than processes.

4. Multithreaded Clients
- Useful in hiding network latency (e.g., web browsers downloading assets in parallel).
- Enables parallel RPCs to different servers.

5. Thread-Level Parallelism (TLP)
- TLP = $\sum (i * c_i) / (1 - c)$, where $c_i$ is time fraction for i threads running simultaneously.

6. User vs Kernel Threads
- User Threads: Managed in user space, fast but blocked together.
- Kernel Threads: Managed by OS, better concurrency but slower due to system calls.
- Hybrid Models: Mix both types for flexibility.

7. Thread Synchronization & Sharing
- Threads share address space → risk of race conditions.
- No protection from accessing shared memory improperly.

8. Example in Python
- `multiprocessing.Process` creates separate processes.
- `threading.Thread` allows concurrent execution within a process.

## Chapter 04: Communication in Distributed Systems

Chapter 04: Communication in Distributed Systems

1. Basic Networking Model
- Focuses on message-passing between nodes.
- Access transparency can be violated by low-level protocols.

2. OSI Layer Overview
- Physical Layer: Transmits raw bits.
- Data Link Layer: Frames, error correction.
- Network Layer: Routing of packets.
- Transport Layer: End-to-end communication (e.g., TCP, UDP).

3. Transport Protocols
- TCP: Reliable, connection-oriented (used by most systems).
- UDP: Best-effort, no guarantees, used for real-time or simple messages.

4. Middleware Layer
- Provides higher-level communication features:
  - Data marshaling, naming, security, replication, caching.
- Goal: standard services across applications.

5. Communication Types
- Transient vs Persistent:
  - Transient: Messages discarded if recipient unavailable.
  - Persistent: Messages stored until delivered.
- Synchronous vs Asynchronous:
  - Sync: Sender waits for reply.
  - Async: Sender continues execution.

6. Client-Server Model
- Transient, synchronous by default.
- Client waits for response, blocking.

7. Message-Oriented Middleware (MOM)
- Supports asynchronous, persistent communication.
- Examples: RabbitMQ, Kafka, ZeroMQ.

8. Remote Procedure Call (RPC)
- Client invokes server function as if local.
- Steps: Stub creation → Marshaling → Transmission → Execution → Response.
- Issues: Parameter passing, byte encoding, full access transparency is hard.

9. Asynchronous and Group RPC
- Async RPC: Call without waiting for response.
- Group RPC: One call sent to multiple servers.

10. Sockets & Messaging Patterns
- Sockets: Low-level TCP/IP interfaces.
- ZeroMQ: High-level patterns like Request-Reply, Publish-Subscribe, Pipeline.

## Chapter 05: Coordination in Distributed Systems

Chapter 05: Coordination in Distributed Systems

1. Clock Synchronization
- UTC: Based on atomic clocks; accurate and globally accepted.
- Precision: Deviation between any two clocks ($\pi$).
- Accuracy: Deviation from actual UTC ($\alpha$).

2. Clock Drift
- Clocks deviate at rate $\rho$.
- $F(t)/F$ must remain within $(1 - \rho)$ to $(1 + \rho)$.

3. Synchronization Types
- Internal: Clocks synchronized with each other.
- External: Clocks synchronized with an external standard (e.g., UTC).

4. Reference Broadcast Synchronization (RBS)
- Node broadcasts reference; others record receipt times.
- Offset calculated with linear regression to adjust for drift.

5. Logical Clocks & the Happened-Before Relation
- Lamport's Clocks:
  - P1: If $a \rightarrow b$, then $C(a) < C(b)$.
  - P2: If a is send, b is receive → $C(a) < C(b)$.

- Implementation: Each process has a counter; updated on events and messages.

6. Totally Ordered Multicast
- Use Lamport timestamps.
- Deliver messages in same order at all processes.
- Conditions: Message is at head and all others acknowledge higher timestamp.

7. Vector Clocks
- Captures causality more accurately.
- Each process maintains a vector VC[i].
- On send: increment own entry and attach.
- On receive: merge vector and increment own entry.

8. Causal Multicast
- Deliver message only after all causally preceding messages are delivered.
- Enforced using vector clock comparison.

9. Mutual Exclusion Algorithms
- Centralized: Coordinator grants access.
- Ricart & Agrawala: Send timestamped request; reply based on priority.
- Token Ring: Only token holder can enter critical section.

10. Key Insight
- Coordination relies on logical ordering (Lamport/Vector clocks), not real time.

## Chapter 06: Naming in Distributed Systems

Chapter 06: Naming in Distributed Systems

1. Purpose of Naming
- Names denote entities.
- Access points are referred to via addresses.
- Location-independent names do not reveal physical address.

2. Identifiers vs Names
- Pure Name: No semantic meaning, just for comparison.
- Identifier: Unique, refers to only one entity, stable over time.

3. Flat Naming Solutions
- Broadcasting: Send request to all nodes (not scalable).
- ARP: Resolve IP to MAC address via broadcast.

4. Forwarding Pointers

- Entity leaves pointer to new location when moving.
- Drawbacks: Long chains, not fault-tolerant, slow resolution.

5. Home-Based Approach
- A "home" node tracks the current location.
- First contact home, then connect to actual location.
- Issue: Poor scalability, single point of contact.

6. Distributed Hash Tables (DHTs) - Chord
- Nodes arranged in a ring; each has a unique m-bit ID.
- Each node maintains a finger table FT[i] = succ(p + 2^(i-1)).
- Efficient lookups using logarithmic hops.

7. Chord Lookup Example
- Uses finger table to locate the responsible node for a key.

8. Network Proximity in DHTs
- Logical closeness may not match physical closeness.
- Solutions:
  - Proximity routing: Forward to nearest physical node.
  - Proximity neighbor selection.

9. Hierarchical Naming – HLS (Hierarchical Location Service)
- Domain-based hierarchy with directories per domain.
- Each level knows part of the tree; root knows all.
- Lookup: Start at leaf, go up if not found, then back down.

10. Scaling HLS
- Distribute load by assigning physical servers to logical domains.
- Avoid overloading root by spreading records per entity.

Summary:
- Naming enables entity identification and location.
- Solutions vary in scalability, fault tolerance, and efficiency.

## Chapter 07: Consistency and Replication

Chapter 07: Consistency and Replication

1. Why Replication?
- Reliability: Switch to a backup if one replica fails.
- Performance: Distribute load across replicas.
- Availability: Access data from geographically closer replicas.

2. Problem: Keeping Replicas Consistent
- Updates must be propagated to all replicas.
- Conflicts may arise from concurrent operations.

3. Data-Centric Consistency Models
- Define contract between data store and processes.

4. Sequential Consistency
- Result is as if operations executed in some sequential order.
- Operations from the same process appear in program order.

5. Linearizability (Strong Consistency)
- Operations appear to take effect instantly at some point between start and end.
- Stronger than sequential consistency.

6. Causal Consistency
- Writes that are causally related must be seen in the same order by all processes.
- Concurrent writes can be seen in different orders.

7. Serializability (from transactions)
- The final result matches a serial execution of transactions.
- Used to check correctness of concurrent operations.

8. Entry Consistency
- Locks ensure consistency of accessed data.
- All operations must complete before data access.

9. Eventual Consistency
- All replicas will eventually converge to the same value in absence of new updates.
- Common in large-scale systems like DNS or shopping carts.

10. Strong Eventual Consistency
- Conflicts resolved using mechanisms like "last-writer-wins" based on time.

11. Continuous Consistency (Conit)
- Allows fine-grained consistency using bounds on:
  - Value deviation (g)
  - Order deviation (p)
  - Time deviation (d)

12. Client-Centric Consistency Models
- Ensure consistency from an individual user's point of view.

a. Monotonic Reads:
- Subsequent reads return same or newer data.

b. Monotonic Writes:
- Writes are propagated in order.

c. Read Your Writes:
- You see your own updates.

d. Writes Follow Reads:
- Updates follow previous reads.

Summary:
- Strong models ensure correctness but reduce performance.
- Weaker models (e.g., eventual consistency) trade consistency for scalability.

## Chapter 08: Fault Tolerance in Distributed Systems

Chapter 08: Fault Tolerance in Distributed Systems

1. Dependability
- A system is dependable if it is reliable, available, safe, and maintainable.

2. Faults, Errors, and Failures
- Fault: The cause of an error.
- Error: Incorrect state.
- Failure: When the system deviates from its intended behavior.

3. Reliability vs Availability
- Reliability R(t): Probability system works correctly during [0, t).
- Availability A(t): Fraction of time the system is operational.
- Metrics:
  - MTTF: Mean Time To Failure.
  - MTTR: Mean Time To Repair.
  - MTBF: MTTF + MTTR.

4. Failure Models
- Crash (halting) failures: Process stops functioning.
- Omission: Fails to respond.
- Timing: Responds too late.
- Arbitrary (Byzantine): Unpredictable behavior, possibly malicious.

5. Fault Handling Techniques
- Fault Prevention: Avoid introduction of faults.
- Fault Detection: Identify presence of faults.
- Fault Tolerance: Continue operation in spite of faults.
- Fault Recovery: Return to normal operation.

6. Redundancy for Fault Masking
- Information Redundancy: Extra bits for error detection/correction.
- Time Redundancy: Retry operations.
- Physical Redundancy: Extra hardware/software (replication).

7. Process Resilience
- Process Groups: Group of replicas act as a single process.
- Flat vs Hierarchical groups.

8. k-Fault Tolerant Groups
- With crash failures: Need k+1 replicas.
- With arbitrary failures: Need 2k+1 for majority agreement.

9. Consensus in Faulty Systems
- Goal: Non-faulty processes agree on a command.

10. Flooding-based Consensus
- Multicast proposed commands to all.
- Merge received commands and deterministically select next.

11. Raft Protocol
- Leader-based log replication.
- Majority acknowledgment commits the command.
- If leader crashes, new leader elected based on up-to-date log.

12. Paxos Protocol
- Handles crash failures in partially synchronous networks.
- LEARN messages ensure operation acknowledged by majority before execution.

13. Failure Detection
- Reliable failure detection is impossible.
- Timeout-based mechanisms used, but may cause false positives.

14. Paxos Rule
- A server can execute an operation only after receiving LEARN from a majority.

Summary:

- Fault tolerance is critical in distributed systems.
- Replication, consensus algorithms, and detection mechanisms form the backbone of resilient design.