

Distributed Systems

1 Introduction to Distributed Systems	1
1.1 Defining Distributed Systems and Core Challenges	1
1.2 The Role of Middleware	2
1.3 The Four Key Design Goals	2
1.4 Classification of Distributed Systems	3
1.5 Common Pitfalls	4
2 Architectures	4
2.1 Why Architecture Matters	4
2.2 Architectural Styles	4
2.3 Middleware Organization	5
2.4 System Architectures	5
3 Communication	6
3.1 The Foundations of Distributed Communication	6
3.2 Remote Procedure Calls (RPC)	6
3.3 Message-Oriented Middleware (MOM)	7
3.4 Multicast and Epidemic Dissemination	8
4 Coordination and Synchronization	9
4.1 Clock Synchronization	9
4.2 Logical Clocks	9
4.3 Mutual Exclusion	10
4.4 Election Algorithms	10
4.5 Gossip-Based Coordination	11
4.6 Distributed Event Matching	11
4.7 Location Systems	11
5 Consistency and Replication	12
5.1 The Motivations and Challenges of Replication	12
5.2 Data-Centric Consistency Models	12
5.3 Client-Centric Consistency Models	13

5.4 Replica Management and Content Placement	13
5.5 Consistency Protocols	13
6 Fault Tolerance	14
6.1 The Defining Characteristic of Distributed Systems	14
6.2 Failure Vocabulary and Models	14
6.3 Masking Failures through Redundancy	15
6.4 Consensus with Crash Failures	15
6.5 Byzantine Fault Tolerance (BFT)	16
7 Security in Distributed Systems	16
7.1 Security Threats, Policies, and Mechanisms	16
7.2 Key Elements of Secure Designs	17
7.3 Cryptography Basics	17
7.4 Authentication	18
7.5 Authorization	18
7.6 Monitoring	19

1 Introduction to Distributed Systems

1.1 Defining Distributed Systems and Core Challenges

A distributed system is defined as a collection of autonomous computing elements that appear to its users as a single coherent system. In this context, “autonomous” means that each node possesses its own clock and state, and can fail independently of the others. “Coherent” means that the users or applications interacting with the system do not need to know which specific machine is handling their request.

While networked computer systems simply connect machines together, a distributed system builds upon this by ensuring that processes and resources are sufficiently spread across multiple computers to achieve specific goals, such as resource sharing, scalability, and reliability. However, building these systems introduces fundamental challenges: nodes can fail independently, communication networks are unreliable and can lose or delay messages,

there is no global clock to establish a universal time, and systems frequently suffer from partial failures where one part of the system crashes while the rest continues to operate.

1.2 The Role of Middleware

To manage the complexity of heterogeneous hardware, different operating systems, and network intricacies, distributed systems rely heavily on a software layer known as middleware. Middleware sits above the local operating systems and provides a uniform interface to applications. By offering standard services—such as naming, state management, and connectivity—middleware acts as the “glue” that allows applications to be written once and run anywhere within the distributed environment.

1.3 The Four Key Design Goals

1.3.1 Distribution Transparency

A primary goal of a distributed system is to hide the physical distribution of its processes and resources from the end user, achieving what is known as distribution transparency. The main types of transparency include:

- **Access Transparency:** Hides differences in data representation and how a resource is accessed, allowing a system to package operations (like submitting an entire form) without exposing the underlying transmission mechanics.
- **Location Transparency:** Conceals where a resource is physically located, often achieved through logical naming systems like URLs.
- **Relocation and Migration Transparency:** Hides that a resource may be moved by the system (relocation) or by the user (migration) without disrupting ongoing operations.
- **Replication Transparency:** Hides the fact that multiple copies of a resource exist to improve performance or availability.

- **Concurrency Transparency:** Ensures that multiple independent users can access shared resources concurrently without interfering with one another, often achieved through locking mechanisms or transactions.
- **Failure Transparency:** Masks the failure and automatic recovery of components from the user.

Achieving full transparency is practically impossible due to physical limitations, such as the speed of light introducing latency, and it is sometimes undesirable when users need context-aware or location-based services.

1.3.2 Scalability

Scalability ensures that a system can grow without suffering a severe drop in performance. It is measured across three dimensions:

- **Size Scalability:** The ability to add more users and resources to the system. Centralized services often become bottlenecks due to CPU, storage, or network limits. Queuing theory demonstrates that as a centralized server’s utilization approaches maximum capacity, response times increase exponentially.
- **Geographical Scalability:** The ability to operate effectively when users and resources are dispersed over large distances, a challenge often hindered by the latencies of synchronous communication and unreliable wide-area networks.
- **Administrative Scalability:** The ability to manage the system easily even when it spans multiple independent administrative organizations with conflicting policies.

To solve scalability bottlenecks, systems employ three main techniques: hiding communication latencies (e.g., using asynchronous communication or shifting computation to the client), partitioning and distribution (e.g., splitting the DNS namespace across many servers), and replication or caching (creating local copies to balance load and reduce latency, though this introduces data consistency challenges).

1.3.3 Openness

An open distributed system offers components that can be easily used by or integrated into other systems. This is achieved by defining services through standard Interface Definition Languages (IDLs). Openness guarantees **interoperability** (different systems can work together), **portability** (applications can run on different underlying platforms without modification), and **extensibility** (components can be swapped or added easily). To maximize flexibility, open systems strictly separate policies (how something is done or managed) from mechanisms (the underlying tools that do the work).

1.3.4 Dependability and Security

Dependability encompasses availability (readiness for immediate use), reliability (continuity of correct service), safety (avoiding catastrophic events upon failure), and maintainability (ease of repair). Masking partial failures to achieve fault tolerance is critical.

Security is equally foundational, ensuring confidentiality (authorized disclosure), integrity (protection against unauthorized alterations), and availability. Because communication happens over untrusted networks, systems rely heavily on cryptography, utilizing symmetric and asymmetric keys, digital signatures, and hash functions to mutually authenticate parties and secure data structures.

1.4 Classification of Distributed Systems

1.4.1 High-Performance Distributed Computing

- **Cluster Computing:** Designed for parallel programming, clusters consist of highly homogeneous, off-the-shelf compute nodes connected via high-speed local interconnects.
- **Grid Computing:** Focused on integrating heterogeneous resources spread across multiple administrative domains to form virtual organiza-

tions. Grid architectures rely on specific layers—Fabric, Connectivity, Resource, and Collective layers—to unify disparate systems.

- **Cloud Computing:** An evolution from grid computing that provides elastic, on-demand access to a shared pool of computing resources, shifting the financial model from buying hardware (CapEx) to renting services (OpEx) through Infrastructure (IaaS), Platform (PaaS), and Software (SaaS) layers.

1.4.2 Distributed Information Systems

These systems emerged to integrate enterprise applications and databases. At the lowest level, operations are encapsulated into distributed transactions that guarantee ACID properties (Atomicity, Consistency, Isolation, Durability). Nested transactions allow a top-level transaction to orchestrate multiple subtransactions across different databases, heavily relying on Transaction Processing (TP) monitors to coordinate atomic commits. At higher levels, Enterprise Application Integration (EAI) connects separate applications directly using Remote Procedure Calls (RPC) or Message-Oriented Middleware.

1.4.3 Pervasive and Mobile Systems

Pervasive systems blend computing naturally into the environment through tiny, battery-powered, wireless nodes.

- **Mobile and Edge Computing:** To overcome the high latencies of routing mobile data directly to remote clouds, edge computing pushes computation and storage out of the core internet to edge servers physically closer to the user.
- **Sensor Networks:** Consisting of thousands of tiny nodes, sensor networks avoid streaming massive amounts of raw data to central operators by performing in-network processing. Nodes aggregate, filter, and locally store data, returning only queried results to conserve limited bandwidth and battery life.

1.5 Common Pitfalls

When designing distributed systems, developers often fail because they rely on false assumptions about the underlying network. Peter Deutsch famously identified these pitfalls, noting that developers incorrectly assume the network is reliable, secure, homogeneous, and has zero latency with infinite bandwidth and zero transport costs. Recognizing that these assumptions are inherently false is the first step toward building robust, scalable, and secure distributed architectures.

2 Architectures

2.1 Why Architecture Matters

The architecture of a distributed system dictates how its software components are logically organized and physically placed across machines. Proper architectural design is crucial; a good architecture relies on clean interfaces and independent components, making the system scalable and easy to evolve. Conversely, a poor architecture results in tightly coupled “spaghetti” code that easily breaks upon modification. Mastering this complexity involves distinguishing between the logical software architecture (how components interact) and the physical system architecture (where components are instantiated).

2.2 Architectural Styles

An architectural style defines the components, their connections, the data exchanged, and how they are configured together. There are several foundational styles used to logically organize distributed systems:

2.2.1 Layered Architectures

In a layered style, components are organized in a stack where a higher layer typically makes a request (downcall) to the layer directly below it and waits for a response. Variations include one-way fire-and-forget calls or

upcalls, where a lower layer notifies an upper layer via a handle (such as an operating system event).

A key concept in layered communication is distinguishing between three elements:

- **Interface:** The API or contract a layer offers to the layer above it.
- **Service:** The actual functionality provided through that interface.
- **Protocol:** The rules that two peer layers on different machines follow to communicate.

This distinction allows developers to swap out the underlying protocol (e.g., from HTTP/1.1 to HTTP/3) without changing the interface exposed to the application.

2.2.2 Service-Oriented Architectures

Service-oriented architectures (SOAs) encapsulate functionality into separate, independent entities.

- **Object-Based Architecture:** The system consists of objects that encapsulate state (data) and methods (behavior). When a client invokes a remote object, it calls a local proxy with the exact same interface as the remote object. The proxy marshals the request across the network to a server-side skeleton, which demarshals the request and executes the actual method.
- **Microservices:** Inspired by the Unix philosophy, microservices decompose large applications into many small, independent networked processes. Each microservice typically owns its own data and can be deployed and scaled independently.
- **Resource-Based (RESTful) Architectures:** The architecture of the Web, known as Representational State Transfer (REST), treats the system as a vast collection of resources. Each resource is identified by a unique URL and manipulated using only four standard stateless operations: GET, PUT, POST, and DELETE.

2.2.3 Publish-Subscribe and Event-Based Architectures

To increase scalability, publish-subscribe styles decouple producers from consumers along two dimensions:

- **Temporal Decoupling:** Senders and receivers do not need to be online at the same time (e.g., email or message queues).
- **Referential Decoupling:** Senders do not need to know the identity of the receivers.

When a system is completely referentially and temporally decoupled, it forms a **Shared Data Space** (or Tuple Space). In systems like Linda, producers write typed tuples into a shared space using an **out** operation, while consumers use **in** or **rd** (read) operations to associative-match and retrieve tuples without ever directly communicating with the producers.

2.3 Middleware Organization

Middleware serves as the “glue” layer sitting above local operating systems, providing a uniform interface that hides hardware heterogeneity, differing OSes, and network complexities. To maintain openness and flexibility, middleware utilizes two primary design patterns:

- **Wrappers (Adapters) and Brokers:** Wrappers transform incompatible interfaces so legacy components can be integrated. To avoid an explosion of wrappers where every application needs a specific wrapper for every other application, a central broker can be used to mediate and route messages.
- **Interceptors:** These software constructs transparently break the standard control flow. A **request-level interceptor** can duplicate a call to invoke multiple replicas of an object without the client knowing. A **message-level interceptor** operates at the transport layer, perhaps fragmenting or encrypting a large payload before it hits the OS network stack.

2.4 System Architectures

System architecture refers to the physical placement of software components across machines.

2.4.1 Client-Server Architectures

The traditional model divides processes into clients (who request services) and servers (who process and reply). This is often divided into a three-tiered logical structure: the user-interface level, the processing level, and the data level. Physically, this translates to different tiers:

- **Two-Tiered:** Typically involves a “fat client” (where much of the processing runs locally on the user’s machine) or a “thin client” (where the client only handles the UI, and the server does the heavy lifting).
- **Three-Tiered:** An application server sits between the client and the database. A classic example is a Transaction Processing (TP) monitor that receives a single client request and coordinates nested subtransactions across multiple independent databases (like airlines and hotels) while guaranteeing ACID properties (Atomicity, Consistency, Isolation, Durability).

2.4.2 Symmetrically Distributed Systems (Peer-to-Peer)

In Peer-to-Peer (P2P) systems, there are no dedicated servers; every node acts symmetrically as both a client and a server. The main challenge is finding resources without a central directory.

- **Structured P2P:** Nodes are organized into a strict deterministic topology (like a ring or hypercube) using a semantic-free index called a Distributed Hash Table (DHT). In systems like Chord, nodes and data keys share the same ID space. A key is stored on the node with the next highest ID (its successor). Using shortcut connections (“finger tables”), lookup queries can be efficiently routed in $O(\log N)$ network hops.
- **Unstructured P2P:** Nodes form ad hoc random graphs. To locate data, nodes must search. **Flooding** sends a query to all neighbors up to a certain Time-To-Live (TTL), which is fast but network-heavy. Alternatively,

Random Walks forward the query to a single random neighbor. Because data is generally replicated, random walks often locate data far more efficiently than flooding without overwhelming the network.

- **Hierarchical P2P:** Introduces “super peers” that act as local brokers or indexes for regular “weak peers,” drastically improving search efficiency.
- **BitTorrent:** A hybrid file-sharing architecture. Users find a `.torrent` file on a Web server, which points to a tracker. The tracker returns a list of active peers. Nodes then collaboratively trade missing file chunks. To prevent “free riding,” the protocol enforces a tit-for-tat system where a node restricts uploads to peers who do not reciprocate.

2.4.3 Hybrid System Architectures

Modern real-world systems blend these architectures into complex hybrid structures.

- **Cloud Computing:** Cloud providers offer elastic, on-demand virtualized resources, shifting enterprise costs from Capital Expenditure (CapEx) to Operational Expenditure (OpEx). It is divided into Infrastructure-as-a-Service (IaaS, renting VMs and storage), Platform-as-a-Service (PaaS, renting runtimes), and Software-as-a-Service (SaaS, full applications).
- **Edge and Fog Computing:** As the Internet of Things (IoT) grows, sending all data to a centralized cloud introduces unacceptable latency, wastes bandwidth, and raises privacy concerns. Edge computing pushes computation out of the cloud closer to the data sources (e.g., local sensor networks that pre-filter data or edge servers for autonomous driving). However, it introduces massive orchestration challenges regarding resource allocation and deciding exactly **where** to place services.
- **Blockchains:** Also known as distributed ledgers, blockchains are append-only chains of transactions replicated across P2P nodes. Validators group transactions into immutable blocks linked by cryptographic hashes. The architecture differs based on consensus mechanisms: **permissioned** blockchains rely on a small, known group of distributed validators, while

permissionless blockchains are fully decentralized and use resource-intensive leader election to agree on the state without a trusted third party.

3 Communication

3.1 The Foundations of Distributed Communication

At the heart of any distributed system is interprocess communication. While non-distributed platforms can rely on shared memory, distributed systems must rely on message passing over unreliable networks. This introduces fundamental challenges: communication latencies are unpredictable, partial failures can occur where one node crashes while another continues, and messages can be lost or reordered. To manage this complexity, communication is structured using protocols, and middleware is introduced to hide the intricacies of network-level message passing.

Communication in middleware can be classified along two main dimensions:

- **Persistence:** In transient communication, a message is only stored as long as the sending and receiving applications are executing; if it cannot be delivered, it is dropped. In persistent communication, the middleware stores the message (e.g., in a queue) as long as it takes to deliver it, meaning the sender and receiver do not need to be active simultaneously.
- **Synchronization:** In asynchronous communication, the sender continues executing immediately after submitting a message. In synchronous communication, the sender blocks until a specific condition is met—such as the message being handed to the network, delivered to the receiver, or fully processed and replied to by the receiver.

3.2 Remote Procedure Calls (RPC)

Introduced in the 1980s by Birrell and Nelson, the Remote Procedure Call (RPC) aims to make calling a function on a remote machine look and act as

much as possible like a local procedure call. This provides a high degree of access transparency by hiding the underlying message passing.

3.2.1 Stubs and Marshaling

To achieve this illusion, RPC relies on automatically generated pieces of code called stubs.

- The **client stub** mimics the target procedure locally. When the client application calls it, the stub packs the parameters into a message (a process called **marshaling**), asks the local OS to send the message, and then blocks, waiting for a reply.
- The **server stub** receives the incoming message, unpacks the parameters (**unmarshaling**), and makes a standard local call to the actual server procedure. Once the server finishes, the stub packs the return value and sends it back to the client.

Marshaling must account for heterogeneous systems, such as differences in byte ordering (big-endian vs. little-endian) and data representations. Furthermore, pointers cannot be directly passed across process boundaries because memory addresses are meaningless on a remote machine. Consequently, RPC typically uses **pass-by-value** or **copy/restore** semantics rather than pass-by-reference. To ensure both sides agree on the procedure signatures and data types, developers use an Interface Definition Language (IDL), which compilers use to automatically generate the client and server stubs.

3.2.2 RPC Failure Semantics

Unlike local calls, which either succeed or crash the program, remote calls face unique network failures (e.g., lost requests, lost replies, or server crashes). System designers must choose the appropriate failure semantics:

- **Exactly-once:** The ideal scenario where the operation is executed once and only once. It is incredibly difficult to guarantee due to the impossibility of distinguishing a crashed server from a lost reply.

- **At-least-once:** The client simply retries the request until it gets a response. This is only safe if the operation is **idempotent** (e.g., fetching a record or overwriting a specific value), meaning repeated executions do not change the system state beyond the initial application.
- **At-most-once:** The server keeps track of requests and actively detects and ignores duplicate messages, guaranteeing the operation is not accidentally run multiple times.

3.2.3 Variations on RPC

To overcome the strict, blocking nature of traditional synchronous RPC, several variations exist:

- **Asynchronous RPC:** The client sends the request and immediately continues execution once the server acknowledges receipt. No result is returned.
- **Deferred Synchronous RPC:** The client sends a request and continues working. It later receives the result either by polling the server or via a callback function triggered when the response arrives.
- **One-way RPC:** A “fire-and-forget” approach where the client does not even wait for an acknowledgment from the server.

3.2.4 Remote Method Invocation (RMI)

RMI is the object-oriented equivalent of RPC. When invoking methods on distributed objects, parameters can be passed in two ways. If a parameter is a reference to a remote object, it is passed by reference (the client receives a proxy stub). If the parameter is a local object, it is typically serialized and passed by value (copied entirely to the server).

3.3 Message-Oriented Middleware (MOM)

RPC assumes that the server is always up and that the client is willing to wait. When systems require temporal and referential decoupling—where producers generate data faster than consumers can process it, or where

parties may be temporarily offline—message-oriented communication is required.

3.3.1 Transient Messaging

Applications can use transient messaging primitives directly.

- **Sockets:** Provide a low-level, transport-layer interface (like TCP or UDP) for passing streams of bytes. They require developers to manually handle formatting, routing, and connection management.
- **ZeroMQ:** Enhances sockets by offering higher-level asynchronous communication patterns, such as request-reply, publish-subscribe, and pipeline (push-pull) configurations.
- **Message-Passing Interface (MPI):** Designed for high-performance computing clusters. It provides highly optimized, transient communication primitives for parallel applications where traditional networking stacks introduce too much overhead.

3.3.2 Persistent Messaging and Queues

Persistent asynchronous communication is provided by Message-Queuing Systems. Applications communicate by inserting messages into specific queues managed by queue managers. Basic queue operations include **PUT** (append a message), **GET** (block and wait for a message), **POLL** (check for a message without blocking), and **NOTIFY** (trigger a callback when a message arrives). Because messages are stored durably, a sender can submit a message even if the receiver is offline, and the receiver can retrieve it later.

3.3.3 Message Brokers and Publish-Subscribe

In large, heterogeneous systems (like Enterprise Application Integration), it is impractical for every application to understand the proprietary messaging formats of every other application. **Message Brokers** act as application-level gateways that sit between producers and consumers. They route messages and transform their formats (e.g., converting delimiters or restructuring fields) so that legacy applications can communicate seamlessly.

In **Publish-Subscribe** systems, communication is completely decoupled. Publishers send messages to a broker without knowing who will receive them. Subscribers express interest in specific data—either through **topic-based** subscriptions (e.g., subscribing to a named channel) or **content-based** subscriptions (e.g., subscribing to messages where a specific field meets a condition). The broker matches messages to subscriptions and routes them accordingly. In large-scale systems, brokers are often arranged hierarchically to prevent a single central broker from becoming a bottleneck.

3.4 Multicast and Epidemic Dissemination

Beyond one-to-one communication, distributed systems frequently need to disseminate information from one sender to a large group of receivers.

3.4.1 Application-Level Multicasting

Because network-level IP multicasting is rarely supported across wide-area networks, multicasting is usually implemented at the application layer using overlay networks. Nodes organize themselves into either a **tree** or a **mesh**.

- **Trees** are efficient because they require exactly $O(N)$ messages, but they are fragile; if an interior node crashes, the entire subtree below it is disconnected.
- **Meshes** are highly resilient because they provide multiple paths between nodes, trading network bandwidth for fault tolerance.

The performance of application-level multicast trees is evaluated using three metrics: **link stress** (how many times a single packet crosses the same physical network link), **stretch** or relative delay penalty (the ratio of the delay in the overlay network compared to the optimal physical network route), and **tree cost** (the total aggregated cost of the links used).

3.4.2 Flooding

If the overlay network forms a mesh, a naive way to broadcast a message is flooding, where every node forwards an incoming message to all its neighbors (except the sender). Because this is highly inefficient and generates

massive duplicate traffic, systems employ **probabilistic flooding**, where nodes only forward messages with a certain probability. In structured peer-to-peer networks (like a Hypercube or Chord DHT), flooding can be carefully controlled along specific logical dimensions to guarantee that a broadcast reaches all nodes using exactly $N - 1$ messages without duplicates.

3.4.3 Gossip-Based Data Dissemination

Inspired by biology, epidemic (or gossip) algorithms provide an incredibly robust, decentralized way to spread information. There is no central coordinator; instead, nodes randomly select peers to exchange updates.

- **Anti-entropy:** Nodes continuously synchronize state. A node can push its updates, pull updates from a peer, or perform a push-pull exchange. The push-pull strategy is the most efficient, ensuring that a single update exponentially propagates to all nodes in the system in $O(\log N)$ rounds.
- **Rumor Spreading:** A node actively gossips a newly received update to random peers. However, if it contacts a peer that already knows the update, it loses interest and stops gossiping with a certain probability. While extremely fast and network-efficient, it does not mathematically guarantee 100% coverage.

Because epidemic algorithms rely purely on spreading new data, propagating the deletion of an item is inherently difficult (if an item is simply erased, an older copy from another node might re-infect the system). To solve this, deletions are handled by spreading **death certificates**—special update messages that act as tombstones, recording the permanent deletion of the data so that obsolete copies are ignored.

4 Coordination and Synchronization

4.1 Clock Synchronization

Because distributed systems lack a global clock, synchronizing time across multiple machines is a fundamental challenge. Hardware clocks use quartz

crystal oscillators, which are subject to clock drift (running at slightly different frequencies) and clock skew (showing different times at a given moment) due to physical limitations and temperature variations.

To achieve global agreement, systems often synchronize with Coordinated Universal Time (UTC), which periodically incorporates leap seconds to stay aligned with the Earth's solar day.

- **Network Time Protocol (NTP):** Clients contact a time server to determine current time. To account for network latency, a client calculates its offset and the transmission delay by tracking the timestamps of the request and response. If a clock is fast, it cannot simply be set backward, as this breaks software logic; instead, the clock is slowed down gradually in a process called slewing. NTP organizes servers in hierarchical strata, where stratum-1 servers are connected directly to atomic clocks or GPS.
- **Reference Broadcast Synchronization (RBS):** Used in wireless sensor networks, a sender broadcasts a reference message, and the receivers synchronize their clocks with each other. This eliminates the variable delay of network access time on the sender's side.
- **TrueTime:** Used in Google's Spanner database, this service utilizes GPS and atomic clocks to bound time uncertainty to a highly precise interval (e.g., 6 milliseconds), ensuring absolute transaction ordering across global datacenters.

4.2 Logical Clocks

In many distributed applications, absolute time is not required; it is only necessary that processes agree on the correct order of events.

- **Lamport's Logical Clocks:** Based on the "happened-before" relation, where event A happens before event B if they occur sequentially in the same process or represent the sending and receiving of a specific message. Each process maintains a local counter incremented before any event. When sending a message, the counter value is attached; upon receipt, the receiver updates its clock to be greater than both its own current time and

the received timestamp. This provides a total ordering of events, useful for tasks like totally ordered multicasting to replicas.

- **Vector Clocks:** Lamport clocks cannot determine if two events are causally related or concurrent just by looking at their timestamps. Vector clocks solve this by maintaining an array of counters (a vector) tracking the causal history of the entire system. A receiver updates its vector by taking the maximum of each corresponding counter in the sender's vector. This allows systems to detect potential causality conflicts and enforce causally ordered message delivery.

4.3 Mutual Exclusion

To prevent data corruption, distributed algorithms enforce mutual exclusion so that only one process accesses a shared resource at a time.

- **Centralized Algorithm:** A single coordinator queues requests and grants permission one by one. It is highly efficient, requiring only three messages (request, grant, release) per access, but the coordinator is a single point of failure and a potential performance bottleneck.
- **Distributed Algorithm (Ricart-Agrawala):** A process multicasts a request with a Lamport timestamp to all nodes. A node replies immediately with an "OK" if it is not interested, or queues the request if it is already in the critical section. If two nodes request simultaneously, the lower timestamp wins. This avoids a central bottleneck but requires $2(N-1)$ messages and fails if any single node crashes.
- **Token-Ring Algorithm:** Processes form a logical ring and pass a token. Access is granted only when holding the token. This guarantees fairness and prevents starvation, but detecting and recovering a lost token is complex.
- **Decentralized Algorithm:** Resources are replicated, and a process must obtain a majority vote from multiple coordinators to proceed. It provides strong fault tolerance but suffers from low resource utilization when multiple nodes compete heavily.

- **ZooKeeper:** Uses a hierarchical namespace where a client creates an ephemeral file node (e.g., `/lock`) to acquire a lock. If the node exists, the client asynchronously subscribes to a notification triggered when the node is deleted, rather than polling.

4.4 Election Algorithms

Many protocols require a coordinator. If the coordinator crashes, an election algorithm must automatically agree on a new one.

- **The Bully Algorithm:** A process initiates an election by sending messages to all nodes with a higher ID. If none respond, it declares itself the leader. If a higher node answers, that node takes over the election, ensuring the "biggest" node always wins.
- **Ring Algorithm:** An election message circulates a logical ring. Each active process appends its ID to the message. Once the message completes the circle, the highest ID in the list is appointed coordinator.
- **ZooKeeper Leader Election:** Within a server ensemble, a server proposes itself as leader by broadcasting its process ID and the latest transaction ID it has processed. Servers update their candidate choice when they receive a broadcast with a more up-to-date transaction, promoting a leader only when a majority agrees.
- **Raft Leader Election:** Operates in numbered terms. A candidate requests votes, and followers cast their vote for the first candidate they hear from. Randomized timeouts ensure that split votes are resolved quickly.
- **Large-Scale Elections (Blockchains):** In permissionless networks, elections happen via "proof of work," where validators race to solve a computationally heavy cryptographic puzzle. Alternatively, "proof of stake" randomly selects a leader proportionally based on the amount of digital tokens they own, heavily reducing computational waste.
- **Wireless Environments:** Traditional algorithms fail in mobile networks due to dynamic topologies. Instead, a source node broadcasts an election tree. Leaf nodes report their resources (like battery capacity) upward,

allowing the source to select the objectively best leader rather than simply the highest ID,.

4.5 Gossip-Based Coordination

Epidemic (gossip) protocols mimic biological infections to propagate information reliably without central coordinators.

- **Aggregation:** Nodes continually pick a random peer and average their respective values. This exponential spread quickly converges the entire network to a global average, allowing decentralized estimation of network size or capacities.
- **Peer-Sampling Service:** To allow a node to randomly pick peers without a global directory, nodes maintain a small “partial view” of neighbors. By periodically exchanging and swapping these partial views, the local lists continuously shift, creating a statistically random sample of the network,.
- **Overlay Construction:** A two-layer architecture where the bottom layer performs random peer-sampling, and the top layer applies a ranking function (like physical distance) to keep the best neighbors. This allows unstructured networks to automatically organize into structured topologies like grids or tori,.
- **Secure Gossiping:** Malicious “hub attacks” can pollute partial views by continually passing references to colluding attackers. Benign nodes combat this by analyzing inbound references (indegree distributions) and silently forcing suspicious nodes to play by the rules to avoid being blacklisted,.

4.6 Distributed Event Matching

Publish-subscribe systems rely on event matching to route notifications from publishers to interested subscribers.

- **Architectures:** Simple deployments use a centralized broker. To scale, brokers form an overlay network. They can broadcast all subscriptions to build selective routing filters, ensuring that notifications are forwarded only along paths where interested clients reside,.

- **Content-Based Gossiping:** When subscriptions are highly expressive (e.g., numeric ranges), nodes use gossiping to find others with intersecting interests. They cluster into minimal disjoint hyperspaces and form bidirectional rings to disseminate matching notifications rapidly without flooding the network,.
- **Secure Publish-Subscribe:** To preserve privacy against untrusted cloud brokers, techniques like Public Key Encryption with Keyword Search (PEKS) are utilized. Publishers encrypt tags, and subscribers provide trapdoors, enabling the broker to match events without ever decrypting the underlying message,.

4.7 Location Systems

Distributed systems frequently need to optimize the physical placement of processes or data based on network proximity.

- **Global Positioning System (GPS):** A dedicated satellite system where receivers solve quadratic equations using the timestamps of broadcasts from at least four satellites, determining their exact 3D coordinates and calculating receiver clock offset simultaneously.
- **WiFi Localization:** Indoors, GPS fails. Devices instead use databases of known WiFi access points, estimating their location by computing the centroid of detected access points, often weighted by signal strength,.
- **Logical Positioning (Network Coordinates):** To optimize latency (e.g., for Content Delivery Networks), nodes are assigned coordinates in an m-dimensional geometric space where distance represents network latency. This can be computed centrally using fixed landmark nodes, or in a decentralized manner using a simulated spring model like Vivaldi, where nodes iteratively push and pull their coordinates to minimize the error between actual latency and geometric distance.

5 Consistency and Replication

5.1 The Motivations and Challenges of Replication

Replication is a fundamental technique in distributed systems driven by several core motivations. First, data is replicated to increase **reliability**; if one replica crashes or becomes corrupted, the system can continue operating by switching to another available copy. Second, replication significantly improves **performance** and **scalability**. By distributing the workload across multiple servers, a system can handle a much larger number of read requests. Furthermore, placing replicas geographically close to end users reduces network latency, resulting in faster access times. Examples of massively replicated systems include the Domain Name System (DNS) and content delivery networks (CDNs) like Akamai.

However, replication introduces a fundamental catch: the consistency problem. Whenever one replica is updated, it temporarily diverges from the others. To maintain a coherent system, modifications must be propagated to all copies. Handling conflicting operations—such as read-write conflicts (where one process reads while another writes) or write-write conflicts (where multiple processes update the same data simultaneously)—requires the system to enforce a global execution order. This global synchronization over a wide-area network is inherently expensive in terms of performance. Therefore, distributed system designers often weaken consistency requirements to reduce coordination overhead.

5.2 Data-Centric Consistency Models

A data-centric consistency model acts as a contract between concurrent processes and the data store, specifying what results a read operation is allowed to return when data is being updated.

Sequential Consistency This is a foundational model which guarantees that the result of any execution is the same as if all operations by all processes were executed in some sequential order, and the operations of

each individual process appear in this sequence in the order specified by its program. Importantly, sequential consistency does not rely on a global clock or real-time execution; it merely requires that all processes see the same valid interleaving of write operations.

Linearizability Linearizability is a stricter model than sequential consistency. It requires that each operation appears to take effect instantaneously at some exact moment between its start and completion. This introduces a real-time ordering constraint: if an operation fully completes before another begins, the system must respect this real-time order. Because it relies on absolute time, linearizability is highly expensive to implement but is used in advanced systems like Google's Spanner, which relies on TrueTime to bound clock uncertainty.

Causal Consistency Causal consistency weakens the strict global ordering requirements by only guaranteeing that operations which are causally related are seen by all processes in the same order. If one write is influenced by an earlier write (e.g., a reply to a social media post), everyone will observe the original post before the reply. However, concurrent writes that have no causal link can be observed in different orders by different processes.

Eventual Consistency and the CAP Theorem In scenarios where concurrent write-write conflicts are rare (like DNS or web caches), systems often adopt eventual consistency. This model guarantees that if no new updates are made, all replicas will eventually converge to the exact same state, though no guarantees are made about what intermediate values clients might observe.

This leads directly into the **CAP Theorem**, which states that a distributed system cannot simultaneously guarantee **Consistency** (all nodes see the same data at the same time), **Availability** (every request receives a response), and **Partition tolerance** (the system continues to operate despite network failures). Because network partitions are inevitable in distributed environments, systems are generally forced to choose between being CP

(Consistent and Partition tolerant, like ZooKeeper) or AP (Available and Partition tolerant, like Cassandra or DNS).

5.3 Client-Centric Consistency Models

While data-centric models provide guarantees for all concurrent clients, client-centric models focus on the experience of a single (often mobile) client accessing different replicas over time. If a user modifies data at one replica and then transparently connects to a different, potentially stale replica, they might experience jarring inconsistencies. To solve this, systems provide session guarantees:

- **Monotonic Reads:** Once a process reads a specific value, any successive read will return that same value or a newer one; the client will never travel “back in time” to an older version.
- **Monotonic Writes:** A write operation by a process is completely propagated and executed before any successive write operation by the same process can take place.
- **Read Your Writes:** The effect of a write operation will always be seen by a successive read operation by the same process, preventing situations where a user updates a web page but sees the old cached version upon refreshing.
- **Writes Follow Reads:** A write operation that follows a read is guaranteed to take place on the same or a more recent version of the data that was read.

ZooKeeper: Mixing Consistency Models Apache ZooKeeper is a practical example of a coordination service that mixes these models. It provides sequential consistency for write operations (all writes are globally ordered by a primary leader), but allows reads to be served locally by any replica, meaning reads can be stale. However, on a per-session basis, ZooKeeper guarantees monotonic reads and read-your-writes consistency. This deliberate trade-off allows ZooKeeper to scale effectively while avoiding the massive coordination overhead of strict global consistency.

5.4 Replica Management and Content Placement

Deciding where, when, and how to replicate content is typically organized into three logical concentric rings:

- **Permanent Replicas:** The initial, statically configured core servers or mirror sites of the data store.
- **Server-Initiated Replicas:** Temporary copies created dynamically to enhance performance. Servers track file access counts grouped by client locations. If the demand exceeds a specific replication threshold, a replica is pushed closer to those clients. If demand drops below a deletion threshold, the replica is removed.
- **Client-Initiated Replicas:** Local client caches (e.g., browser caches). These are lightweight, ephemeral, and managed entirely by the client to bypass network latency altogether.

When a replica needs to be updated, the system can use one of three propagation strategies:

- **Invalidation:** The server simply notifies replicas that their data is stale. This uses minimal bandwidth but forces a cache miss on the next access.
- **Transfer of Data:** The server pushes the actual updated data to the replicas immediately, keeping them fresh at the cost of higher bandwidth.
- **Transfer of Operation (Active Replication):** The server sends the write operation itself, and each replica independently re-executes the operation to reach the new state.

5.5 Consistency Protocols

Consistency protocols provide the actual implementation mechanisms to achieve these theoretical models.

Primary-Backup Protocols In primary-backup architectures, one designated server (the primary) is uniquely responsible for accepting and coordinating all write operations. When a write arrives, the primary executes it, propagates the update to all backup servers, waits for their acknowledgments, and only then acknowledges the client. Reads can be directed to any

backup (if stale reads are acceptable) or strictly to the primary (to guarantee sequential consistency). If the primary crashes, the backups detect the failure and elect a new leader.

Quorum-Based Protocols Instead of relying on a single primary bottleneck, quorum protocols use a voting mechanism. To perform an operation, a client must request permission from a subset of the N replicas. A read requires a read quorum (N_R), and a write requires a write quorum (N_W). To ensure consistency, the system enforces two mathematical constraints:

1. $N_R + N_W > N$: This ensures the read and write quorums overlap, guaranteeing that a reader will always query at least one server containing the most recent write.
2. $N_W > \frac{N}{2}$: This prevents write-write conflicts by ensuring that two concurrent writes cannot both achieve a majority.

A popular configuration of this protocol is **ROWA** (Read One, Write All), where $N_R = 1$ and $N_W = N$. This provides extremely fast, single-server reads, but forces write operations to synchronously update every single replica in the system.

6 Fault Tolerance

6.1 The Defining Characteristic of Distributed Systems

Unlike a single machine where a crash typically results in a total system halt, distributed systems are defined by the notion of partial failure. In a distributed environment, one component may fail while the rest of the system continues to operate, often seemingly correctly. Because these systems are heavily reliant on asynchronous networks, it is fundamentally impossible to distinguish a component that has completely crashed from one that is merely running very slowly or experiencing severe network delays. The ultimate goal of distributed systems design is to build architectures that can automatically and gracefully tolerate these partial failures without severely impacting overall performance.

6.2 Failure Vocabulary and Models

To build dependable systems, it is crucial to understand the exact mechanisms of failure:

- **Fault:** The underlying root cause, such as a programming bug, a loose network connector, or a cosmic ray bit-flip. Faults can be transient (occurring once and vanishing), intermittent (appearing and disappearing sporadically), or permanent (remaining until physically repaired).
- **Error:** An invalid or incorrect state in the system caused by a fault, such as a race condition or a corrupted memory value.
- **Failure:** The ultimate result when the system or component no longer lives up to its designed specifications.

Failures are classified into several models based on their severity:

- **Crash failures:** The server halts prematurely but was operating correctly right up until the moment it stopped. This is the easiest type of failure to mask.
- **Omission failures:** The server fails to respond to incoming requests, either because it failed to receive the message (receive omission) or failed to send the reply (send omission).
- **Timing failures:** The server's response lies outside a specified real-time window, such as delivering streaming video frames too late.
- **Response failures:** The server responds, but the value is completely wrong, or the server deviates from the correct flow of control (state-transition failure).
- **Arbitrary (Byzantine) failures:** The most severe failure model. A process may produce arbitrary responses at arbitrary times, send contradictory messages to different peers, or act maliciously.

Because failure detection relies on timeouts, its accuracy depends on network synchrony. In a purely synchronous system, failures are **fail-stop** and can be detected with absolute certainty. In practical, partially synchronous systems, failures are **fail-noisy** (eventually detected, but with possible false

positives), or **fail-silent** (clients cannot distinguish a crash from a simple omission).

6.3 Masking Failures through Redundancy

The key to achieving fault tolerance is redundancy.

Physical Redundancy Hardware-level protection often uses Triple Modular Redundancy (TMR). In TMR, three identical components process the same inputs in parallel. Their outputs are fed into a voter component that passes the majority result. If one component fails or produces a faulty result, the other two outvote it, completely masking the failure. While highly effective for mission-critical systems, it requires 3x the hardware and introduces the voter itself as a potential single point of failure.

Process Redundancy and Groups Software-level protection is achieved by organizing identical processes into groups. A client sends a message to the group as a single logical abstraction. If one group member crashes, the others take over.

- **Flat groups** are perfectly symmetrical and lack a single point of failure, but reaching decisions requires complex voting mechanisms.
- **Hierarchical groups** use a coordinator to make rapid decisions, but if the coordinator crashes, the system halts until a leader election algorithm promotes a new coordinator.

To survive k concurrent crash failures, a system typically requires $k + 1$ identical members. However, to survive k Byzantine (arbitrary) failures, the system requires $2k + 1$ members if only a simple majority vote on a static result is needed. If the group must actively reach a distributed consensus, the threshold rises to $3k + 1$ members.

6.4 Consensus with Crash Failures

To act as a single fault-tolerant entity, all non-faulty processes in a group must execute the exact same commands in the exact same order. Reaching this agreement is known as consensus. The FLP Impossibility Theorem

mathematically proves that in a fully asynchronous system, consensus is impossible to guarantee if even one process crashes. Therefore, practical consensus algorithms rely on partial synchrony and timeouts.

Flooding-Based Consensus In a simple round-based approach, processes broadcast their known commands to everyone else. If a process crashes mid-round, surviving processes might have differing information. A process cannot safely decide on an action until it is certain that other survivors have the same data, meaning failures force the system into extra communication rounds to ensure everyone is synchronized.

Raft Raft is a consensus algorithm explicitly designed for understandability. It uses a strong primary-backup architecture.

- **Leader Election:** All servers start as followers. If a follower hears nothing from a leader, it becomes a candidate and requests votes. To prevent infinite split-vote ties, Raft uses randomized timeouts (e.g., 150–300 ms). The first candidate to secure a majority of votes becomes the leader.
- **Log Replication:** The leader receives client requests, appends them to its local log as uncommitted entries, and sends `AppendEntries` RPCs to the followers. Once a majority of followers acknowledge the entry, the leader commits the operation, executes it, and informs the followers to do the same. This strict control flow guarantees the Log Matching Property (if two logs have the same index and term, they are identical up to that point) and Leader Completeness.

Paxos Paxos is a highly robust but notoriously complex consensus protocol that operates without a single permanent leader. It separates processes into Proposers, Acceptors, and Learners. To prevent split-brain scenarios where multiple isolated proposers believe they are in charge, Paxos uses two phases:

1. **Prepare/Promise:** A proposer generates a unique timestamp and asks a majority of acceptors to promise they will ignore any older proposals.

If an acceptor has already accepted a previous operation, it returns that operation to the proposer.

2. **Accept/Learn:** If the proposer receives a previously accepted operation from the acceptors, it is forced to adopt it. It then broadcasts an **Accept** message for that operation. Once a majority accepts it, the learners are notified to execute the operation.

6.5 Byzantine Fault Tolerance (BFT)

Byzantine failures involve processes that lie, collude, or send conflicting data—a scenario famously known as the Byzantine Generals Problem.

Why $3k + 1$ Processes are Needed If a system must tolerate $k = 1$ Byzantine failure, having 3 processes is insufficient. If a faulty primary tells Backup 1 to execute ‘True’ and tells Backup 2 to execute ‘False’, the backups will exchange their values and both end up with the set {True, False}. Neither backup can determine if the primary lied or if the other backup is lying. By increasing the group size to $3k + 1$ (e.g., 4 processes for 1 failure), the system guarantees that the $2k + 1$ non-faulty processes will form a strict mathematical majority capable of outvoting the k traitors.

Practical Byzantine Fault Tolerance (PBFT) PBFT achieves consensus in environments with arbitrary failures using signed messages and a three-phase commit process:

1. **Pre-Prepare:** The primary assigns a timestamp and view number to a client’s operation and broadcasts it.
2. **Prepare:** Replicas verify the message and broadcast a **Prepare** message to all other replicas. A non-faulty replica achieves a “prepare certificate” only when it collects $2k$ matching prepare messages, mathematically proving that a majority of the network agrees on the sequence order.
3. **Commit:** Replicas broadcast a **Commit** message. Once a replica collects $2k$ matching commit messages, it permanently executes the operation and replies to the client.

View Changes and Scaling If the PBFT primary is detected as faulty, backups initiate a View Change. The new primary collects all prepared certificates from a quorum to ensure that no previously committed operations are abandoned, thereby preserving safety and liveness. Because PBFT requires $O(N^2)$ message exchanges, it is generally limited to small groups. Modern distributed ledgers and blockchains solve BFT at a massive scale using protocols like HotStuff, which reduces message overhead to $O(N)$ using leader-relays, and by integrating economic deterrents (Proof of Work/ Proof of Stake) alongside cryptographic threshold signatures.

7 Security in Distributed Systems

7.1 Security Threats, Policies, and Mechanisms

Securing a distributed system requires addressing a negative goal: protecting the system against all unauthorized actions. To understand how systems are compromised, security professionals analyze the attack vector (the specific path an attacker uses to gain access), the attack surface (the complete set of all possible attack vectors), and the kill chain or attack path (the sequence of steps an attacker takes to exploit vulnerabilities and create an impact).

Threats generally fall into four main categories:

- **Disclosure:** Unauthorized reading of data, breaching confidentiality.
- **Modification:** Unauthorized writing or silent tampering of data, breaching integrity.
- **Denial of use:** Blocking legitimate access, such as through Denial of Service (DoS) attacks, breaching availability.
- **Resource theft:** Hijacking infrastructure, such as running crypto-miners or ransomware on someone else’s CPU.

To defend against these, organizations must define a strict security policy that dictates precisely what each entity may and may not do. Without a policy, security mechanisms have nothing to enforce, rendering security mere

theater. Mechanisms used to enforce these policies include cryptography, authentication (verifying identity), authorization (verifying permissions), continuous monitoring, attack path analysis, and automatic remediation.

7.2 Key Elements of Secure Designs

The design of secure systems relies heavily on several long-standing principles:

- **Fail-safe defaults:** Access should be denied by default; the burden of proof is on granting access, not denying it.
- **Open design:** Security should not rely on obscurity. Algorithms should be public, and only the cryptographic keys should be kept secret.
- **Separation of privilege:** Critical operations should require multiple conditions or parties to agree, avoiding an all-powerful superuser.
- **Least privilege:** Processes and users should operate with just enough access to perform their job, and nothing more.
- **Least common mechanism:** Components should be isolated where possible, because shared code introduces a shared attack surface.

7.2.1 The Trusted Computing Base (TCB)

Dependencies between services lead to the concept of the Trusted Computing Base, which represents the root of trust—all firmware, hardware, and software necessary to enforce a security policy. To ensure a pristine environment, modern hardware employs mechanisms like ARM TrustZone, which physically and logically divides the processor into a Normal state and a Secure state. Address bus qualifiers select tenancy, ensuring secure memory controllers grant access only to secure OS kernels and trusted applications.

This hardware foundation allows for Secure Boot. When a device powers on, it verifies a digital certificate burned into the hardware. This certificate checks the cryptographic hash of the Boot-ROM, which in turn verifies the OS kernel. If the hash is valid, the system boots, ensuring the entire software stack remains untampered.

7.3 Cryptography Basics

Cryptography is the foundational mechanism for ensuring confidentiality and integrity over untrusted networks.

- **Symmetric Cryptosystems:** The sender and receiver share the same secret key for both encryption and decryption.
- **Asymmetric Cryptosystems:** Keys come in pairs. A public key encrypts data (ensuring privacy since only the private key can decrypt it), while a private key encrypts data or hashes (ensuring authentication and non-repudiation, as anyone with the public key can verify the signature). The RSA algorithm is a prime example, relying on the mathematical difficulty of finding the private key when given the public key and the product of two large prime numbers.

Hash functions are one-way mathematical operations that take an arbitrary-length message and produce a fixed-length output. They must be collision-resistant. When combined with asymmetric cryptography, hashes enable digital signatures: a sender computes the hash of a message and encrypts the hash with their private key. The receiver decrypts the signature with the sender's public key and computes their own hash of the message to verify integrity.

7.3.1 Diffie-Hellman Key Exchange

To securely establish a shared symmetric session key over an insecure channel, systems use the Diffie-Hellman key exchange. Both parties agree on a public base and modulus. Each party picks a private, secret number, computes a public derivative, and sends it to the other. By applying their private number to the received public derivative, both parties arrive at the exact same shared secret mathematically, without ever transmitting their private keys over the network.

7.4 Authentication

Authentication ensures entities are who they claim to be. Modern systems rely on multi-factor authentication, utilizing combinations of:

- **What a user knows:** Passwords, PINs, or passphrases.
- **What a user has:** Physical ID cards, smartphones, or hardware tokens.
- **What a user is (Static Biometrics):** Fingerprints, iris scans, or facial geometry.
- **What a user does (Dynamic Biometrics):** Voice patterns or keystroke typing rhythms.

7.4.1 Authentication Protocols

To prevent replay attacks, authentication protocols rely on challenge-response mechanisms using nonces (numbers used once). If Alice and Bob share a secret key, Bob can challenge Alice to encrypt a nonce. If Alice returns the properly encrypted nonce, Bob knows he is talking to Alice. However, protocol design is notoriously fragile. Naive optimizations, such as combining multiple challenges into fewer messages, can open the door to reflection attacks, where a malicious actor tricks a system into encrypting its own challenge to bypass the authentication check.

7.4.2 Key Distribution Center (KDC) and Kerberos

In large systems, having every pair of users share a secret key is mathematically unscalable. Instead, systems use a centralized Key Distribution Center. In the Needham-Schroeder protocol, Alice asks the KDC for a session key to talk to Bob. The KDC generates a key and gives Alice a ticket (encrypted with Bob's secret KDC key) containing the session key. Alice passes this ticket to Bob.

Kerberos is a widely deployed framework based on this concept. It utilizes two components: an Authentication Server (AS) and a Ticket-Granting Service (TGS). A user logs in, and the AS grants them a Ticket-Granting Ticket. The user then presents this ticket to the TGS to securely request session

keys for specific applications, enabling seamless single sign-on across the distributed system.

7.4.3 TLS 1.3

For web-based communication, the Transport Layer Security (TLS) protocol establishes secure channels. In TLS 1.3, the client and server immediately execute an ephemeral Diffie-Hellman key exchange during their initial “hello” messages. They negotiate cryptographic parameters and compute a shared secret. The server then authenticates itself by sending its public key certificate (signed by a trusted Certificate Authority), encrypted using the newly derived session key, ensuring both privacy and authenticated identity.

7.5 Authorization

Once an identity is authenticated, a reference monitor enforces authorization, dictating what the identity is allowed to do. There are four primary access control models:

- **Mandatory Access Control (MAC):** The system strictly enforces security labels (e.g., top secret). Owners cannot override these rules; they are used in high-security environments.
- **Discretionary Access Control (DAC):** The resource owner grants or revokes read, write, or execute rights at their own discretion.
- **Role-Based Access Control (RBAC):** Users are authorized based on their organizational role (e.g., teacher, student) rather than personal identity, with roles carrying fixed permission sets.
- **Attribute-Based Access Control (ABAC):** The most fine-grained model, where access rules evaluate attributes of the user, the target object, the environment, and the network connection dynamically.

These models are implemented using either Access Control Lists (ACLs) or Capabilities. With an ACL, the server checks a local list to see if the requesting subject is permitted to access the object. With a capability (such

as a JSON Web Token or JWT), the client holds an unforgeable digital ticket that inherently carries the access rights.

7.5.1 Delegation

Often, a user needs an application to act on their behalf (e.g., an email client accessing a mail server). Instead of handing over their master password, the user delegates rights through protocols like OAuth. The user authorizes the application, which receives a secure access token representing a restricted subset of the user's rights.

7.6 Monitoring

Security is never perfect, making monitoring essential. Firewalls establish a perimeter between networks. A packet-filtering router inspects headers to block unauthorized IP addresses, while an application gateway inspects the actual payload of the traffic. However, firewalls cannot stop threats that originate from within or that exploit legitimate application traffic.

To catch these, systems employ Intrusion Detection Systems (IDS). Signature-based IDSs look for known patterns of malicious network packets, but struggle against novel attacks. Anomaly-based IDSs use machine learning to establish a baseline of normal behavior and flag deviations. Because a single viewpoint is insufficient in large networks, Collaborative Intrusion Detection Systems (CIDS) pool data from multiple sensors to optimize accuracy and precision while minimizing false positives.