Preparation Notes for Distributed Computing: Modules 3.2 and 4.1-4.2

PART 1: MODULE 3 - PROCESS, SYNCHRONIZATION, AND COORDINATION

This module examines the foundational elements of distributed processes: how they are structured, how they communicate, and how they coordinate their actions in a distributed environment lacking shared memory or a global clock.

Section 3.1.A: Process and Communication Models (Syllabus 3.1)

This section details the fundamental execution and architectural models, specifically threads, server designs, and the client-server paradigm.

Threads in Distributed Systems

A thread is the smallest unit of execution within a process. In the context of a distributed system, threads are the primary mechanism for enabling parallelism and concurrency. Their role is twofold:

1. **Local Concurrency:** Within a single node, threads improve performance and responsiveness. For example, a multi-threaded server can use one thread to handle a client request while other threads handle background tasks, such as network I/O or inter-node communication.
2. **Distributed Concurrency:** Threads are the "actors" that manage the system's distributed nature. They are often responsible for handling message passing, managing inter-node synchronization, and executing distributed tasks. A common use case is a web server that spawns a new thread to handle each incoming client connection concurrently.

Managing threads in a distributed system introduces challenges beyond those of a single operating system. Synchronization must occur between threads on different machines, load balancing must distribute work across all threads and nodes in the system, and resource allocation must be carefully managed (often using thread pools) to prevent contention and ensure stability.

Server Architectures

The design of a server in a distributed system involves a critical trade-off between performance, programming simplicity, and the I/O model. Three primary models exist for constructing a server :

1. **Multi-threaded Server:** This model achieves parallelism by using multiple threads of control. Its primary advantage is that it "makes programming easier" by allowing the use of simple, blocking system calls. For instance, when a thread handles a request and blocks on a disk read, other threads can continue to run and service other clients. This provides both programming ease and improved performance through parallelism.

2. **Single-threaded Server:** This model uses a single process to handle all requests sequentially. It "retains the ease and simplicity of blocking system calls" but, as a major drawback, "it gives up performance because there is no parallelism." A client blocking on a slow I/O operation will block the *entire server*, preventing any other clients from being served.
3. **Finite State Machine (FSM) Server:** This model, also known as an event-loop model, achieves high performance and parallelism (in the sense of concurrent I/O) by using non-blocking system calls. It is typically implemented with a single thread (or a small pool). However, this high performance comes at the cost of programming complexity; it is "hard to program" because it forces an asynchronous, event-driven paradigm where the programmer must manage state explicitly instead of relying on the blocking-call-and-return stack.

The choice of server model thus dictates the programming paradigm. The multi-threaded model allows for a traditional, procedural style (block, wait, continue), whereas the FSM model mandates a complex, event-driven style (asynchronous callbacks) but offers superior scalability for I/O-bound workloads.

The Client-Server Model

The client-server model is a foundational distributed architecture characterized by a clear separation of roles.

- **Client:** The entity that *initiates* communication by requesting a service or resource. The client typically handles user-interface logic and does not share its own resources.
- **Server:** The entity that *awaits* and *responds* to client requests. The server is a powerful, centralized provider of a service (e.g., email, database, web pages) and is responsible for managing that resource.

Communication follows a standard **request-response cycle**. This model offers significant advantages, including centralized data management, easier maintenance and backup, and simplified security enforcement.

However, this centralization is also the model's primary weakness. The server acts as a **centralized point of failure**; if the server crashes, the entire service becomes unavailable. It is also a natural **performance bottleneck** and a prime target for Denial of Service (DDoS) attacks. These inherent vulnerabilities of the simple client-server model are the primary motivation for **replication**, which is the core topic of Module 4. Replication attempts to solve the single-point-of-failure problem by creating multiple copies of the server, which in turn introduces the problem of **consistency**.

Section 3.1.B: Code Migration (Syllabus 3.1)

Code migration is the capacity for programs (or components of programs) to move between machines in a distributed system. This is a powerful mechanism for improving performance (e.g., by moving computation *to* the data) and increasing flexibility (e.g., by downloading code on demand).

Approaches to Code Migration (Strong vs. Weak Mobility)

The central distinction in code migration is based on what is being transferred, which dictates the state of the program upon arrival.

- **Weak Mobility (or Code Migration):**
  - **What Moves:** The code segment and potentially the data segment.
  - **Execution State:** The program *restarts from its initial state* on the target machine. No execution state (e.g., the program stack) is transferred.
  - **Examples:** This is a very common model. Java applets, JavaScript code downloaded by a browser, and mobile agents are all examples of weak mobility.
- **Strong Mobility (or Process Migration):**
  - **What Moves:** The code segment, the data segment, *and* the **execution segment**. The execution segment includes the program's run-time stack and the current process state.
  - **Execution State:** The program *resumes* execution on the target machine from the exact point at which it was suspended.
  - **Examples:** This is used for dynamic load balancing, such as moving a running process from a heavily-loaded machine to a lightly-loaded one in a high-performance computing cluster.

Migration can also be categorized by its initiation:

- **Sender-Initiated:** The machine *hosting* the code initiates the transfer (e.g., a client sending a query to a database server).
- **Receiver-Initiated:** The machine *receiving* the code initiates the transfer (e.g., a browser downloading a Java applet).

Migration and Local Resources

The most significant challenge in code migration, particularly for strong mobility, is the management of resources to which the migrating process is "attached". These attachments, or bindings, can be classified:

1. **Binding by Identifier:** The resource is fixed and location-independent (e.g., a specific URL or FTP server). This is easy to manage; the migrated process retains its reference.
2. **Binding by Value:** The resource is a standard library (e.g., Java's standard libraries). This is also easy; the migrated process finds and binds to the local copy of the same library on the new machine.
3. **Binding by Type:** The resource is generic and local to the machine (e.g., "a printer," "the local file system," an open file handle). This is the most difficult binding to manage.

This "local resource problem" is the primary reason why **strong mobility is rare** and **weak mobility is ubiquitous**. A weakly mobile program (like a JavaScript snippet) starts fresh and requests resources (like local storage) from its *new* environment (the browser). It has no memory of prior attachments.

Conversely, a strongly mobile process with an open file handle—a low-level kernel resource on the original machine—cannot simply resume on a new machine where that handle is meaningless. The system would require a complex management layer to either forward all I/O back to the origin machine (creating a new bottleneck) or attempt to re-bind the handle to an equivalent resource on the new machine, which is non-trivial. Thus, the practical utility of migration is often limited by the difficulty of migrating its resource bindings.

Section 3.2.A: Clock Synchronization (Syllabus 3.2)

In a distributed system, there is no single global clock. Each machine has its own physical clock, which is subject to "clock drift" and will inevitably fall out of sync with other machines. This

section covers algorithms to manage this problem, which are divided into two main categories: physical clock synchronization and logical clocks.

Physical Clock Synchronization

These algorithms attempt to synchronize the physical "wall-clocks" of machines to a common time, either an external standard (accuracy) or an internal average (precision).

- **Cristian's Algorithm:**
  - **Concept:** A client synchronizes its clock with an external, authoritative time server. This is a form of *external synchronization*.
  - **Algorithm Steps:**
    1. The client records its local time T_0 and sends a request to the time server.
    2. The server receives the request, records its current time T_s, and sends T_s back in a reply message.
    3. The client receives the reply at its local time T_1.
    4. The client calculates the Round-Trip Time (RTT): RTT = T_1 - T_0.
    5. The client assumes a symmetric network delay, estimating the one-way delay as \frac{RTT}{2}.
    6. The client sets its clock to T_s + \frac{RTT}{2}.
  - **Limitations:** The algorithm's accuracy is entirely dependent on the RTT being short and, more importantly, the network delay being *symmetric*. If the request takes 10 ms and the reply takes 90 ms, the algorithm's calculation will be highly inaccurate.
- **Berkeley Algorithm:**
  - **Concept:** An *internal synchronization* algorithm used when no external time source is available. A "master" node polls "slaves" and instructs them to adjust their clocks to a group average.
  - **Algorithm Steps:**
    1. A master is elected from the group (e.g., using an election algorithm).
    2. The master periodically *polls* all slaves for their current clock times.
    3. Slaves respond with their time. The master estimates their time, accounting for the RTT (similar to Cristian's).
    4. The master computes a fault-tolerant average, typically by ignoring outlier values.
    5. Critically, the master *does not* send the new time back. It sends each slave a relative *adjustment delta* (e.g., "+5 seconds," "-3 seconds").
  - **Key Features:**
    - **Robustness:** Because the master sends a *delta*, the transmission delay of this final message does not interfere with the synchronization, a key advantage over Cristian's algorithm.
    - **No Backward Clocks:** Setting a clock *backward* can break logical programs (e.g., file modification times). To avoid this, a node that receives a negative adjustment (e.g., "-5s") will often just slow its clock rate temporarily until it has "lost" the required 5 seconds.

Logical Clocks

When the exact wall-clock time is less important than the *order* of events, logical clocks are used.

- **Lamport's Timestamps:**
  - **Concept:** A simple algorithm that provides a *partial ordering* of events. It is based on the "happened-before" relation (\rightarrow), which states that A \rightarrow B if A and B are in the same process and A occurred first, or if A is the sending of a message and B is the receiving of that message.
  - **Algorithm Rules** :
    1. **IR1 (Local Event):** Before a local event, a process P_i increments its local clock: C_i = C_i + 1.
    2. **IR2 (Send Event):** When P_i sends a message m, it includes its current timestamp ts(m) = C_i.
    3. **IR3 (Receive Event):** When a process P_j receives a message m, it updates its clock C_j = \max(C_j, ts(m)) + 1.
  - **Property:** If A \rightarrow B, then C(A) < C(B).
  - **Limitation:** The converse is *not true*. If C(A) < C(B), it does *not* mean that A \rightarrow B. The events could be concurrent. Lamport's clocks can provide a total ordering (by using process IDs to break ties), but they *cannot distinguish causality from concurrency*.
- **Vector Clocks:**
  - **Concept:** An enhancement to Lamport's clocks that *exactly* captures the causal "happened-before" partial order. It can successfully identify concurrent events.
  - **Data Structure:** Each process P_i in a system of N processes maintains a vector (an array) VC_i of size N. VC_i[j] represents the number of events at process j that process i *knows about*.
  - **Algorithm Rules** :
    1. **Initialization:** All vectors start at [0, 0,..., 0].
    2. **IR1 (Local Event):** P_i increments *its own* entry in its vector: VC_i[i] = VC_i[i] + 1.
    3. **IR2 (Send Event):** P_i sends its *entire vector* VC_i with the message m.
    4. **IR3 (Receive Event):** When P_j receives m with vector VC_m: a. P_j increments *its own* entry: VC_j[j] = VC_j[j] + 1. b. P_j *merges* the received vector with its own: for all k from 1 to N, VC_j[k] = \max(VC_j[k], VC_m[k]).
  - **Property:** Two events A and B are *concurrent* if and only if VC(A) \not\leq VC(B) and VC(B) \not\leq VC(A).
  - **Trade-off:** This provides perfect causal tracking but at a high cost. A vector of size N must be sent with *every* message, which creates significant communication and processing overhead.

Table 1: Comparative Analysis of Clock Synchronization

Algorithm	Type	Synchronization	Goal	Message Overhead	Key Limitation
Cristian's	Physical	External	Accuracy (vs. UTC)	O(1)	Relies on symmetric network RTT.
Berkeley	Physical	Internal	Precision (Group Agreement)	O(N) (for polling)	No external accuracy; master is a

Algorithm	Type	Synchronization	Goal	Message Overhead	Key Limitation
					bottleneck.
Lamport's	Logical	Logical	Partial Ordering (Causality)	O(1) integer	Cannot distinguish concurrency from causality.
Vector	Logical	Logical	Exact Partial Ordering	O(N) vector	High communication/ processing overhead.

Section 3.2.B: Distributed Mutual Exclusion (Mutex) (Syllabus 3.2)

Ensuring mutual exclusion (i.e., that only one process can access a shared "critical section" at a time) is a fundamental problem. In a distributed system, this must be achieved without shared memory, relying only on message passing. Algorithms are divided into three main categories :

1. **Non-Token-Based (Permission-Based):** A process must acquire permission (votes) from other processes before entering the CS.
2. **Token-Based:** A single, unique "token" circulates. Only the process holding the token may enter the CS, guaranteeing mutual exclusion.
3. **Quorum-Based:** A hybrid approach where a process acquires permission from a *subset* (a quorum) of nodes, not all of them.

Non-Token-Based: Ricart-Agrawala Algorithm

- **Concept:** A permission-based algorithm that uses Lamport timestamps to create a total ordering of requests and resolve conflicts.
- **Algorithm Steps** :
    1. **Request:** Site S_i wants to enter the CS. It generates a timestamp (ts, i) and sends a REQUEST(ts, i) message to *all* other N-1 sites.
    2. **Receiving Request:** When site S_j receives REQUEST(ts, i):
        - If S_j is not in the CS and is not requesting the CS, it immediately sends a REPLY to S_i.
        - If S_j is already in the CS, it *defers* the request (does not reply).
        - If S_j is also requesting the CS, it compares its own timestamp (ts_j, j) with (ts, i).
            - If (ts_j, j) < (ts, i) (S_j has priority), S_j *defers* S_i's request.
            - If (ts, i) < (ts_j, j) (S_i has priority), S_j sends REPLY.
    3. **Enter CS:** S_i enters the CS *only* after it has received a REPLY from *all* other N-1 sites.
    4. **Release CS:** Upon exiting the CS, S_i sends a REPLY message to all sites whose requests it had deferred.
- **Message Complexity:** 2(N-1) messages per CS entry: (N-1) REQUEST messages and (N-1) REPLY messages.
- **Key Weakness:** The algorithm is *not fault-tolerant*. The requirement to receive N-1 replies means that the crash of *any single node* will cause the requesting process to wait forever (starve), halting progress.

Token-Based: Suzuki-Kasami Algorithm

- **Concept:** A token-based algorithm that uses a broadcast REQUEST message to inform all sites of a need, and sequence numbers to manage out-of-order requests.
- **Data Structures** :
    - **On each site S_i:** An array RN[N] (Request Number). RN_i[j] stores the *largest* sequence number S_i has *received* from site S_j.
    - **In the Token:**
        1. An array LN[N] (Last Number). LN[j] stores the sequence number of the *last request executed* at site S_j.
        2. A Queue Q of site IDs waiting for the token.
- **Algorithm Steps** :
    1. **Request:** If S_i wants the CS and does not have the token: a. It increments its own sequence number: RN_i[i] = RN_i[i] + 1. b. It *broadcasts* a REQUEST(i, RN_i[i]) message to *all* other sites.
    2. **Receiving Request:** When S_j receives REQUEST(i, sn), it updates its local knowledge: RN_j[i] = \max(RN_j[i], sn).
    3. **Sending Token:** The site S_k currently holding the token will send it to S_i *if* S_k is not in the CS and S_i's request is outstanding (i.e., RN_k[i] == LN[i] + 1).
    4. **Release CS:** S_i (which now has the token) exits the CS. a. It updates the token's state: LN[i] = RN_i[i]. b. It checks all other sites k. If RN_i[k] == LN[k] + 1 (meaning k has an outstanding request) and k is not in Q, it adds k to Q. c. If Q is not empty, it pop(Q) and sends the token to that site. Otherwise, S_i keeps the token.
- **Message Complexity:** N messages (the broadcast) to request the token, or 0 if the site already holds it. This avoids the high-latency sequential passing of a simple token ring by *intelligently* passing the token only to sites that need it.

Quorum-Based: Maekawa's Algorithm

- **Concept:** This algorithm reduces the message overhead of Ricart-Agrawala by not requiring permission from *all* sites, but only from a *subset* (a "request set" or "quorum") R_i.
- **Key Property:** The quorums are constructed to satisfy the **intersection property**: \forall i \forall j : R_i \cap R_j \neq \emptyset. This common member ensures that two sites cannot gain permission simultaneously.
- **Algorithm Steps** :
    1. **Request:** S_i sends REQUEST to *only* the sites in its quorum R_i.
    2. **Receiving Request:** When S_j (in R_i) receives a REQUEST:
        - If S_j has not sent a REPLY (is "unlocked"), it sends REPLY to S_i and *locks* itself.
        - If S_j is already locked, it *queues* S_i's request.
    3. **Enter CS:** S_i enters the CS *only* after receiving a REPLY from *every* site in R_i.
    4. **Release CS:** S_i exits the CS and sends RELEASE to *all* sites in R_i.
    5. **Receiving Release:** S_j receives RELEASE. It "unlocks" itself. If its queue is not empty, it sends REPLY to the *next* queued request and remains locked.
- **Message Complexity:** In an optimal configuration, the quorum size |R_i| is \sqrt{N}. This results in a message complexity of **3\sqrt{N}**: \sqrt{N} REQUESTs + \sqrt{N} REPLYs +

\sqrt{N} RELEASEs.

- **Trade-off:** Maekawa's algorithm achieves a significant performance gain over Ricart-Agrawala (from O(N) to O(\sqrt{N})). However, it *loses a key safety property*. Ricart-Agrawala uses timestamps to prevent deadlocks. Maekawa's algorithm, in its basic form, does not use priorities and is **susceptible to deadlock** (a circular wait where sites are locked in a way that no site can achieve its full quorum).

Table 2: Comparative Analysis of Distributed Mutual Exclusion

Algorithm	Type	Message Complexity	Fault Tolerance	Key Failure Mode
Ricart-Agrawala	Non-Token (Permission)	2(N-1)	Poor	Crash of *any* node (starvation).
Suzuki-Kasami	Token-Based (Broadcast)	N (or 0)	Moderate	Token loss (requires regeneration).
Maekawa's	Quorum-Based	3\sqrt{N}	Moderate	**Deadlock** is possible.

Section 3.2.C: Election Algorithms (Syllabus 3.2)

Many distributed algorithms (like Berkeley's or centralized mutex) require a single "coordinator" process. Election algorithms are used to select this leader, especially when the current leader fails.

The Bully Algorithm

- **Concept:** The process with the *highest* unique process ID is always elected as the coordinator. It "bullies" lower-ID processes into submission.
- **Algorithm Steps** :
    1. Process P notices the coordinator is not responding.
    2. P initiates an election by sending an ELECTION message to *all* processes with a *higher ID* than P.
    3. P waits for a response (an OK message):
        - **Case A (No response):** If P receives no OK message before a timeout, it knows it is the highest-ID *alive* process. P *wins* the election.
        - P then sends a COORDINATOR (or "I WON") message to *all* processes with *lower IDs* to announce itself as the new leader.
        - **Case B (Response):** P receives an OK message from a higher-ID process Q. P *loses* the election. Its job is done, and it simply waits to receive a COORDINATOR message from the eventual winner.
    4. When Q (the higher-ID process) receives the ELECTION message from P, it first sends an OK back to P (to "bully" it and stop its election) and *then starts its own election* by going back to Step 2, sending ELECTION messages to processes with IDs *higher than Q*.

The Ring Algorithm

- **Concept:** Assumes all processes are arranged in a *logical* ring, where each process knows its immediate successor. The goal is typically to elect the highest-ID process.
- **Algorithm Steps** :
    1. Process P detects the coordinator has failed.
    2. P creates an ELECTION message containing its own ID in a list (e.g., [P]) and sends this message to its successor.
    3. When a process Q receives an ELECTION message, it *adds its own ID* to the list (e.g., [P] \rightarrow [P, Q]) and forwards this new message to *its* successor.
    4. The message circulates around the entire ring.
    5. The election "completes" when the message returns to the initiator, P, which recognizes its *own ID* in the incoming list.
    6. P now holds a list of all *alive* processes in the ring. P selects the new coordinator from this list (e.g., the one with the highest ID).
    7. P sends a new COORDINATOR message around the ring, containing the ID of the winner.
- **Fault Tolerance:** If Q's successor is down, Q must *skip* over it and send the message to the next-next successor, continuing until a live node is found.

PART 2: MODULE 4 - CONSISTENCY AND REPLICATION

This module addresses the solutions to the fault-tolerance problem. **Replication** is the technique of keeping multiple copies of data or a process. **Consistency** is the challenge of ensuring that these copies remain correct and synchronized in the face of concurrent updates.

Section 4.1.A: Consistency Models (Syllabus 4.1)

A consistency model is a "contract" between the data store and the processes using it. It defines the rules for the order and visibility of read and write operations, determining how "correct" or "up-to-date" a read operation is guaranteed to be.

Data-Centric Consistency Models

These models define system-wide guarantees about the state of data, applying to *all* processes in the system. They exist on a spectrum, trading off strong correctness for high performance.

- **Strict Consistency:**
    - **Contract:** "Absolute time ordering". Any read on a data item x must return the value written by the *absolute most recent* write to x.
    - **Requirement:** This model assumes a perfect, instantaneous global clock.
    - **Practicality:** It is a theoretical ideal and **impossible to implement** in a real-world distributed system due to network latency.
- **Sequential Consistency:**
    - **Contract:** A slightly weaker model. "The result of any execution is the same as if all ops... were executed in *some* sequential order, and the operations of each individual process appear in this sequence in the order specified by its program".
    - **Key Idea:** There is no "real time" guarantee. However, *all processes must agree on the same* single interleaving (ordering) of all operations in the system. This global

synchronization requirement makes it correct but slow.

- **Causal Consistency:**
  - **Contract:** A further relaxation of sequential consistency. "All processes see *causally-related* shared accesses in the same order".
  - **Key Idea:** This model differentiates between events. Writes that are causally related (e.g., W_2 happens because it read the result of W_1) *must* be seen in the same order by all processes. However, *concurrent* writes (that are not causally related) may be seen in different orders by different processes. This relaxation allows for significantly better performance than sequential consistency.

This hierarchy is clear: **Strict** is "perfect" but impossible. **Sequential** is "very strong" (global total order) but slow. **Causal** is a "balance" (partial order). At the bottom of this hierarchy is **Eventual Consistency**, which offers no ordering guarantees, only that if no new writes occur, all replicas will *eventually* converge to the same value.

Table 3: Hierarchy of Data-Centric Consistency Models

Model	Guarantee (The "Contract")	Key Feature (The "How")	Practicality / Performance
Strict	All reads see the *absolute* most recent write.	Global clock, all ops instantaneous.	Impossible.
Sequential	All processes agree on *one* single ordering of all operations.	Total ordering, but not based on "real" time.	Very high cost, slow (global sync).
Causal	All processes agree on the order of *causally-related* operations.	Concurrent operations can be reordered.	Good performance/correctness balance.
Eventual	If no new writes occur, all replicas will *eventually* converge.	No ordering guarantees.	Highest performance, weakest guarantee.

Client-Centric (Session) Consistency Models

In many large-scale global systems, data-centric consistency is too expensive. Client-centric models offer a pragmatic solution, providing guarantees *only* for a single client's session, who may be connecting to different, out-of-sync replicas. These are often called "session guarantees."

- **Read-Your-Writes Consistency:**
  - **Contract:** "The effect of a write... by a process... will always be seen by a successive read... by the *same process*".
  - **Problem Solved:** A user updates their web page and hits "refresh." This model guarantees they will see the update they just made, not a stale, cached copy.
- **Monotonic Reads Consistency:**
  - **Contract:** "If a process reads... x, any successive read on x... will always return that *same value or a more recent value*".
  - **Problem Solved:** Prevents time from "going backward." A user reads a "final score" headline. They refresh and should not see an older "game starting" headline.

- **Monotonic Writes Consistency:**
  - **Contract:** "A write... on X is completed before any successive write... on X by the *same process*".
  - **Problem Solved:** Guarantees that a single client's updates are applied in the order they were issued. If a user posts "Part 1" and then "Part 2," other users should not see "Part 2" before "Part 1".
- **Writes-Follow-Reads Consistency:**
  - **Contract:** "A write... following a previous read... is guaranteed to take place on the same or a more recent value... that was read".
  - **Problem Solved:** Ensures that a write is causally "correct" based on what was read. For example, a user posts a *reply* to a news article. This guarantee ensures the reply is attached to the article version the user *read*, not an older, stale version.

Section 4.1.B: Replication and Consistency Management (Syllabus 4.1)

This section covers the "how" of implementing the models above: the strategies for managing multiple copies (replicas) of data.

Replication Strategies (Active vs. Passive)

- **Active-Passive (Primary-Backup):**
  - **Description:** One server is designated the "primary" (or "master") and is the single source of truth. It handles *all* write requests. The other "passive" (or "backup") servers are on standby and are updated by the primary.
  - **Pros:** Simple to implement and understand. Guarantees strong consistency because all writes are serialized through a single node.
  - **Cons:** Less resource-efficient (backups are idle). Failover is *not* instantaneous; there is a brief downtime while a backup is promoted to primary.
- **Active-Active (Multi-Master):**
  - **Description:** *All* nodes in the cluster are "active" and can accept write requests. The load is balanced across all nodes.
  - **Pros:** High availability, low latency, and efficient resource utilization.
  - **Cons:** *Extremely complex*. The system *must* have a mechanism to resolve concurrent write conflicts (e.g., if two users update the same data on two different replicas at the same time).

Quorum Consensus Protocols

- **Concept:** A protocol, often attributed to Gifford, that manages replicated data by defining a "quorum" (a minimum number of nodes) required for read and write operations. It provides a tunable trade-off between consistency, availability, and performance.
- **Formulas and Rules:** Let N be the total number of replicas.
  - N_W = Write Quorum (number of replicas that must acknowledge a write).
  - N_R = Read Quorum (number of replicas that must be read from).
  - **Rule 1: N_W + N_R > N**. This is the **Read-Write Intersection Rule**. It guarantees that *any* read quorum and *any* write quorum *must* overlap by at least one node.

This ensures that a read will always see at least *one* copy of the most recent write.
    - **Rule 2: N_W > \frac{N}{2}**. This is the **Write-Write Intersection Rule**. It ensures that two *write* operations cannot succeed concurrently at two disjoint sets of replicas. It prevents a "split-brain" scenario where the system has two conflicting "most recent" writes.
- **Example (N=5):**
    - **Balanced (Majority):** N_W=3, N_R=3. (Rule 1: 3+3 > 5. Rule 2: 3 > 2.5). This is a common, robust setup.
    - **Fast Read / Slow Write:** N_W=4, N_R=2. (Rule 1: 4+2 > 5. Rule 2: 4 > 2.5). This is valid and optimizes for read-heavy workloads.
    - **Fast Write / Slow Read:** N_W=2, N_R=4. (Rule 1: 2+4 > 5. Rule 2: 2 \ngtr 2.5). This is **INVALID** because it violates the Write-Write rule and can lead to data corruption.

Consistency Protocols

These are the underlying mechanisms used to implement the replication strategies.

- **Primary-Based Protocols:** These protocols implement the **Active-Passive** strategy.
    - **Remote-Write Protocol:** All clients write to a *fixed, remote* primary server, which then propagates the update to backups. This is the classic primary-backup model.
    - **Local-Write Protocol:** The "primary" status *migrates* to the client that wants to write. That client writes locally (for low latency), and then propagates updates to the other replicas.
- **Replicated-Write Protocols:** These protocols implement the **Active-Active** strategy.
    - **Active Replication:** The write *operation* (not just the data) is forwarded to all replicas, which execute it. This requires a total ordering of operations to ensure all replicas end up in the same state.
    - **Quorum-Based Protocols:** The *client* is responsible for writing the new data to a N_W quorum of replicas, as described in the Quorum Consensus section.

Section 4.2: Fault Tolerance (Syllabus 4.2)

Fault tolerance is the *ability* of a system to continue operating correctly even in the presence of failures. This is the primary *reason* for using replication.

Taxonomy of Failures

The type of failure dictates the difficulty of building a fault-tolerant system.

- **Crash Failure (Fail-Stop):** A process or server halts and permanently stops all execution. This is the simplest and "cleanest" failure to handle, as the process stops sending messages entirely.
- **Omission Failure:** A process *fails to send* a message (send-omission) or *fails to receive* a message (receive-omission). From an observer's perspective, this can be indistinguishable from a network failure (lost message).
- **Byzantine (Arbitrary) Failure:** This is the most severe and difficult failure. The process behaves *arbitrarily*. It can lie, send conflicting messages to different nodes, or act maliciously to disrupt the system. Tolerating Byzantine failures requires complex consensus algorithms.

Core Fault Tolerance Techniques

Achieving fault tolerance relies on a hierarchy of concepts:

1. **Goal: Fault Tolerance:** The *ability* to continue functioning despite faults.
2. **Method: Redundancy:** The *strategy* of having duplicate components (hardware, software, data) so that a backup can take over.
3. **Technique: Replication:** The *act* of *creating* the redundant duplicates.

The key techniques used to manage this are:

- **Replication:** (Covered in 4.1.B). This is the primary technique for fault tolerance. By having multiple copies of data or a service (e.g., in an Active-Passive setup), the failure of the primary component is not catastrophic, as a backup can take over.
- **Checkpointing and Rollback Recovery:** This technique involves periodically *saving the state* of a process (a "checkpoint") to stable storage. If the process fails, it can be *restarted* from the last known good checkpoint, rather than from the very beginning. This "rollback recovery" minimizes lost work.
- **Failure Detection:** A system cannot recover from a failure it doesn't know about. The most common detection technique is the use of **Heartbeat** messages. Processes periodically send "I'm alive" messages to a monitor. If the monitor does not receive a heartbeat within a certain **timeout** period, it declares the process "dead" and initiates a recovery (e.g., promoting a passive replica to primary).

PART 3: GATE-PATTERN SELF-ASSESSMENT QUIZ

This section contains Multiple Choice Questions (MCQs) designed to test analytical understanding of the above topics, as per a "GATE-pattern" exam.

Module 3.1 & 3.2 Questions (Synchronization & Coordination)

Q1. A server architecture uses a single thread and non-blocking I/O to handle many concurrent client connections. Which model is this, and what is its primary trade-off? (a) Multi-threaded, and it has high programming complexity. (b) Single-threaded, and it has poor performance. (c) Finite State Machine (FSM), and it has high programming complexity. (d) Multi-threaded, and it has high context-switching overhead.

Q2. Which of the following statements about code mobility is TRUE? (a) Strong mobility is common in web browsers (e.g., JavaScript). (b) Weak mobility involves transferring the execution stack. (c) Process migration is an example of weak mobility. (d) Weak mobility requires the program to restart from an initial state on the new machine.

Q3. A process P with a Java applet has an open file handle on Machine A. The process migrates to Machine B using strong mobility. What is the primary challenge? (a) The Java Virtual Machine on Machine B may be a different version. (b) The process's resource binding to the file handle is "by type" and is invalid on Machine B. (c) The network connection will be lost. (d) The code segment must be recompiled for Machine B.

Q4. In Cristian's Algorithm, a client sends a request at its local time 10:00:00.000. It receives a reply at 10:00:00.080. The reply contains the server's timestamp T_s = 10:00:00.050. What time should the client set its clock to? (a) 10:00:00.050 (b) 10:00:00.090 (c) 10:00:00.130 (d) 10:00:00.040

Q5. A set of nodes in a private data center, with no access to an external UTC source, needs to synchronize their clocks *among themselves* to ensure precise internal event ordering. Which algorithm is most appropriate? (a) Cristian's Algorithm (b) The Bully Algorithm (c) Berkeley Algorithm (d) Network Time Protocol (NTP)
Q6. Which of the following statements about Lamport's Timestamps is FALSE? (a) If event A \rightarrow B (A happened-before B), then C(A) < C(B). (b) It is a logical clock algorithm. (c) It can be used to create a total ordering of events by using process IDs to break ties. (d) If C(A) < C(B), then A \rightarrow B.
**Q7. In a system with 3 processes (P1, P2, P3), the vector clocks are V1, V2, V3.

- Event A at P1: $V1 = $
- Event B at P2: $V2 = $ (P2 received a message from P1 with V=)
- Event C at P3: $V3 = $ What is the causal relationship between events B and C?** (a) B \rightarrow C (B happened-before C) (b) C \rightarrow B (C happened-before B) (c) B and C are concurrent (B \parallel C) (d) Not enough information.

Q8. A system has 49 nodes (N=49). What is the theoretical message complexity per CS entry for Ricart-Agrawala's and Maekawa's algorithms, respectively? (a) 48 and 21 (b) 96 and 21 (c) 96 and 49 (d) 48 and 48
Q9. A key disadvantage of Maekawa's algorithm for mutual exclusion, which is not present in Ricart-Agrawala's algorithm, is: (a) High message complexity of O(N). (b) The possibility of deadlock. (c) The need for a central coordinator. (d) The reliance on Lamport timestamps.
Q10. In the Ricart-Agrawala algorithm, site S5 (timestamp 10,5) and site S10 (timestamp 12,10) both request the CS simultaneously. What happens when S5's request reaches S10? (a) S10 sends REPLY to S5 immediately. (b) S10 defers S5's request. (c) S10 enters the CS. (d) S10 sends a RELEASE message.
Q11. In the Bully Algorithm, Process P7 (ID=7) detects that the coordinator (P10) has failed. P7 sends ELECTION messages to P8, P9, and P10. P8 and P9 are alive; P10 is dead. What happens next? (a) P7 wins the election because P10 is dead. (b) P8 receives the message, sends OK to P7, and starts its own election. (c) P7, P8, and P9 all hold elections and the winner is decided by a vote. (d) P9 receives the message, sends OK to P7, and becomes the new coordinator.
Q12. In the Ring Algorithm, an ELECTION message initiated by P3 arrives at P5. The message contains the list . What action does P5 take? (a) P5 discards the message because it has a higher ID. (b) P5 sends the message to its successor. (c) P5 adds its own ID, creating , and sends it to its successor. (d) P5 sends a COORDINATOR message, declaring itself the winner.

Module 4.1 & 4.2 Questions (Consistency & Replication)

Q13. Which of the following is the "contract" for Sequential Consistency? (a) All processes see causally-related writes in the same order, but not concurrent writes. (b) All processes see all operations in *some* single sequential order, and that order matches their program order. (c) All reads return the value of the *absolute* most recent write. (d) All processes will eventually read the same value if no new writes occur.
Q14. A client-centric consistency model is needed to prevent a user from seeing an older version of an article *after* they have already seen a newer version. Which model provides this guarantee? (a) Read-Your-Writes (b) Monotonic Reads (c) Monotonic Writes (d) Writes-Follow-Reads

Q15. A user posts a reply to a forum thread. The system must ensure that this reply is associated with the version of the thread the user was reading. Which consistency model is designed to guarantee this? (a) Causal Consistency (b) Read-Your-Writes (c) Monotonic Reads (d) Writes-Follow-Reads

Q16. What is the fundamental trade-off between an Active-Passive and an Active-Active replication strategy? (a) Active-Passive has lower latency but Active-Active has stronger consistency. (b) Active-Passive is simpler and more consistent, but Active-Active has higher availability and no failover downtime. (c) Active-Passive uses resources more efficiently, but Active-Active is simpler to program. (d) Active-Passive is only for databases, while Active-Active is for web servers.

Q17. A replicated data store has 10 nodes (N=10). The system is configured with a Write Quorum N_W = 7 and a Read Quorum N_R = 3. Which statement is FALSE? (a) The configuration is invalid because N_W + N_R = N. (b) The configuration optimizes for fast writes. (c) The configuration satisfies the Read-Write intersection rule (N_W + N_R > N). (d) The configuration satisfies the Write-Write intersection rule (N_W > N/2).

Q18. Using the same system (N=10, N_W=7, N_R=3), what is the *actual* flaw in this configuration? (a) N_W + N_R > N is not satisfied. (b) N_W > N/2 is not satisfied. (c) Both N_W + N_R > N and N_W > N/2 are violated. (d) Both rules are satisfied, but N_W + N_R = 10, which is not *greater than* 10. The Read-Write rule is violated.

Q19. A database uses a "primary-backup" model where all write operations are sent to a single master node, which then propagates them to passive slaves. This is an implementation of: (a) A Replicated-Write, Active Replication protocol. (b) A Replicated-Write, Quorum-Based protocol. (c) A Primary-Based, Remote-Write protocol. (d) A Primary-Based, Local-Write protocol.

Q20. What is the correct relationship between Fault Tolerance, Redundancy, and Replication? (a) Replication is the *goal*, Redundancy is the *method*, and Fault Tolerance is the *technique*. (b) Fault Tolerance is the *goal*, Redundancy is the *strategy*, and Replication is the *technique* used to implement it. (c) Redundancy is the *goal*, Replication is the *problem*, and Fault Tolerance is the *strategy*. (d) All three terms mean the same thing.

Q21. A process stops all execution and sends no further messages. This is a: (a) Byzantine Failure (b) Omission Failure (c) Crash (Fail-Stop) Failure (d) Checkpoint Failure

Q22. A node in a distributed system appears to be working, but it sends a message "Value=A" to Node 1 and a message "Value=B" to Node 2 for the same data. This is a: (a) Byzantine Failure (b) Omission Failure (c) Crash Failure (d) Network Partition

Q23. A fault-tolerance technique involves periodically saving a process's state to stable storage, allowing the process to restart from its last saved state after a crash. This is called: (Example) (a) Replication (b) Heartbeating (c) Checkpointing and Rollback (d) Quorum Consensus

Q24. In the Bully Algorithm, what is the purpose of the OK message? (a) To acknowledge that a lower-ID process has won the election. (b) To tell a lower-ID process to stop its election, because a higher-ID process is taking over. (c) To tell all processes that a new coordinator has been found. (d) To request the token.

Q25. Which of the following scenarios violates Monotonic Reads consistency? (a) A user writes "A", refreshes, and sees "A". (b) A user writes "A", then writes "B", and another user sees "A" then "B". (c) A user reads "B", refreshes, and sees "A" (an older value). (d) A user reads "A", refreshes, and sees "A".

Answer Key

1. **(c) Finite State Machine (FSM), and it has high programming complexity.**
2. **(d) Weak mobility requires the program to restart from an initial state on the new machine.**
3. **(b) The process's resource binding to the file handle is "by type" and is invalid on Machine B.**
4. **(b) 10:00:00.090**. (RTT = 80ms. One-way = 40ms. New Time = T_s + One-way = 10:00:00.050 + 40ms = 10:00:00.090)
5. **(c) Berkeley Algorithm**. (It is an *internal* synchronization algorithm for a group of nodes *without* an external source)
6. **(d) If C(A) < C(B), then A \rightarrow B.** (This is the converse, which is not true. C(A) < C(B) can also occur if A and B are concurrent)
7. **(c) B and C are concurrent (B \parallel C)**. ($V(B) = $ is not $\leq V(C) = $, and V(C) is not \leq V(B). Therefore, they are concurrent)
8. **(b) 96 and 21**. (Ricart-Agrawala: 2(N-1) = 2(48) = 96. Maekawa's: 3\sqrt{N} = 3\[span_35](start_span)[span_35](end_span)sqrt{49} = 3 * 7 = 21)
9. **(b) The possibility of deadlock.** (Maekawa's does not use timestamps and can have a circular wait)
10. **(a) S10 sends REPLY to S5 immediately.** (S10 is requesting, but S5 has a lower timestamp (10 < 12), so S5 has priority)
11. **(b) P8 receives the message, sends OK to P7, and starts its own election.** (P8 is alive and has a higher ID than P7, so it "bullies" P7 and takes over the election)
12. **(c) P5 adds its own ID, creating , and sends it to its successor.**
13. **(b) All processes see all operations in *some* single sequential order, and that order matches their program order.**
14. **(b) Monotonic Reads**. (Prevents time from "going backward" for a client)
15. **(d) Writes-Follow-Reads**. (Ensures the write [the reply] is based on the data that was read [the thread])
16. **(b) Active-Passive is simpler and more consistent, but Active-Active has higher availability and no failover downtime.**
17. **(a) The configuration is invalid because N_W + N_R = N.**
18. **(d) Both rules are satisfied, but N_W + N_R = 10, which is not *greater than* 10. The Read-Write rule (N_W + N_R > N) is violated.**
19. **(c) A Primary-Based, Remote-Write protocol.**
20. **(b) Fault Tolerance is the *goal*, Redundancy is the *strategy*, and Replication is the *technique* used to implement it.**
21. **(c) Crash (Fail-Stop) Failure**.
22. **(a) Byzantine Failure**. (The node is "lying" and sending conflicting information)
23. **(c) Checkpointing and Rollback**.
24. **(b) To tell a lower-ID process to stop its election, because a higher-ID process is taking over.**
25. **(c) A user reads "B", refreshes, and sees "A" (an older value).** (This violates the guarantee that subsequent reads will see the *same or newer* value)

Works cited

1. What are threads in a distributed system? - Design Gurus, https://www.designgurus.io/answers/detail/what-are-threads-in-a-distributed-system 2. Threads in Distributed Systems - GeeksforGeeks, https://www.geeksforgeeks.org/system-design/threads-in-distributed-systems/ 3. Threads in distributed system | Distributed System | Lec-37 | Bhanu Priya - YouTube, https://www.youtube.com/watch?v=klorWr04oQg 4. The Client-Server Model Explained: From Basics To Implementation - Unstop, https://unstop.com/blog/client-server-model 5. Client–server model - Wikipedia, https://en.wikipedia.org/wiki/Client%E2%80%93server_model 6. Client-Server Model - GeeksforGeeks, https://www.geeksforgeeks.org/system-design/client-server-model/ 7. Understanding the Client-Server Model Basics - SynchroNet, https://synchronet.net/client-server-model/ 8. Code mobility - Wikipedia, https://en.wikipedia.org/wiki/Code_mobility 9. Code, Process, and VM Migration Part 1: Migration Introduction, https://none.cs.umass.edu/~shenoy/courses/677content/slides/spring24/Lec10.pdf 10. Code Migration in Distributed System - GeeksforGeeks, https://www.geeksforgeeks.org/system-design/code-migration-in-distributed-system/ 11. Code and Process Migration! Motivation! - LASS, https://lass.cs.umass.edu/~shenoy/courses/fall09/lectures/Lec06.pdf 12. Code migration and distributed scheduling, http://www.cs.iit.edu/~iraicu/teaching/CS550-S11/lecture12.pdf 13. Clock Synchronization in Distributed Systems - GeeksforGeeks, https://www.geeksforgeeks.org/distributed-systems/clock-synchronization-in-distributed-system/ 14. Time Synchronization in Distributed Systems | by Dung Le - Medium, https://medium.com/distributed-knowledge/time-synchronization-in-distributed-systems-a21808928bc8 15. Cristian's Algorithm For Clock Synchronization - Unstop, https://unstop.com/blog/cristians-algorithm 16. Simulation Engine for Analysis and Comparison between Cristian's ..., http://antares.cs.kent.edu/~mikhail/classes/aos.f07/ProjectReports/sehgal.report.pdf 17. 5.1.4 Clock Synchronization - IOE, CSIT, BIM, BCA - Easy Explanation, https://ezexplanation.com/course/distributed-system/514-clock-synchronization 18. Cristian's Algorithm for Clock Synchronization - YouTube, https://www.youtube.com/watch?v=OZE6lVhIVv0 19. Berkeley algorithm - Wikipedia, https://en.wikipedia.org/wiki/Berkeley_algorithm 20. Lecture 12: March 18 12.1 Overview 12.2 Clock Synchronization - LASS, https://lass.cs.umass.edu/~shenoy/courses/spring19/lectures/Lec12_notes.pdf 21. Berkeley's Algorithm - Tutorials Point, https://www.tutorialspoint.com/berkeley-s-algorithm 22. Berkeley's Algorithm for Clock Synchronization - YouTube, https://www.youtube.com/watch?v=jKyvgxx0-Jk 23. Lamport Clock | Baeldung on Computer Science, https://www.baeldung.com/cs/lamport-clock 24. Time, Clocks, and the Ordering of Events in a Distributed System - Leslie Lamport, https://lamport.azurewebsites.net/pubs/time-clocks.pdf 25. Lamport's logical clock - GeeksforGeeks, https://www.geeksforgeeks.org/dsa/lamports-logical-clock/ 26. Lamport Clocks: Determining the Order of Events in Distributed Systems - Medium, https://medium.com/outreach-prague/lamport-clocks-determining-the-order-of-events-in-distributed-systems-41a9a8489177 27. Logical Clocks - Documentation - Aeron.io, https://aeron.io/docs/distributed-systems-basics/logical-clocks/ 28. Difference between Lamport timestamps and Vector clocks, https://cs.stackexchange.com/questions/101496/difference-between-lamport-timestamps-and-vector-clocks 29. Vector Clocks | Kevin Sookocheff,

https://sookocheff.com/post/time/vector-clocks/ 30. Vector Clocks. Like Lamport's Clock, Vector Clock is… | by Sruthi Sree Kumar | Big Data Processing | Medium, https://medium.com/big-data-processing/vector-clocks-182007060193 31. Vector Clocks for Ordering of Events in Distributed Systems - YouTube, https://www.youtube.com/watch?v=b2Tud5Kkue8 32. Logical Time, Lamport Timestamps and Vector Clocks in distributed systems - Codemia, https://codemia.io/knowledge-hub/path/logical_time_lamport_timestamps_and_vector_clocks_in_distributed_systems 33. Chapter 9: Distributed Mutual Exclusion Algorithms, https://www.cs.uic.edu/~ajayk/Chapter9.pdf 34. Survey on Token-Based Distributed Mutual Exclusion Algorithms - arXiv, https://arxiv.org/html/2502.04708v1 35. Difference between Token-based and Non-Token-based Algorithms in Distributed Systems, https://www.tutorialspoint.com/difference-between-token-based-and-non-token-based-algorithms-in-distributed-systems 36. A Quorum-Based Group Mutual Exclusion Algorithm for a Distributed System with Dynamic Group Set - The University of Texas at Dallas, https://personal.utdallas.edu/~neerajm/publications/journals/quorum.pdf 37. Ricart–Agrawala algorithm - Wikipedia, https://en.wikipedia.org/wiki/Ricart%E2%80%93Agrawala_algorithm 38. Ricart–Agrawala Algorithm in Mutual Exclusion in Distributed ..., https://www.geeksforgeeks.org/operating-systems/ricart-agrawala-algorithm-in-mutual-exclusion-in-distributed-system/ 39. Distributed Algorithms (CAS 769) - Week 5: Mutual Exclusion, https://www.cse.msu.edu/~borzoo/teaching/15/CAS769/lectures/week5.pdf 40. Ricart–Agrawala algorithm is an algorithm to for mutual exclusion in a distributed system prop - Rohini College, https://www.rcet.org.in/uploads/academics/rohini_98619579057.pdf 41. Ricart Agrawala Mutual Exclusion algorithm in Distributed Systems Synchronization, https://www.youtube.com/watch?v=EcoTlY0GXrA 42. Suzuki–Kasami algorithm - Wikipedia, https://en.wikipedia.org/wiki/Suzuki%E2%80%93Kasami_algorithm 43. Suzuki–Kasami Algorithm for Mutual Exclusion in Distributed System - GeeksforGeeks, https://www.geeksforgeeks.org/operating-systems/suzuki-kasami-algorithm-for-mutual-exclusion-in-distributed-system/ 44. Maekawa's Algorithm for Mutual Exclusion in Distributed System ..., https://www.geeksforgeeks.org/operating-systems/maekawas-algorithm-for-mutual-exclusion-in-distributed-system/ 45. Maekawa's Mutual Exclusion algorithm - Quorum based approach - YouTube, https://www.youtube.com/watch?v=lsotFDYrTRI 46. Maekawa mutual exclusion algorithm- Distributed systems- Video 17 - YouTube, https://www.youtube.com/watch?v=HpQyZDK0R90 47. Lecture 14: March 21 14.1 Overview 14.2 Leader Election - LASS, https://lass.cs.umass.edu/~shenoy/courses/spring22/lectures/Lec14_notes.pdf 48. Lecture 6: Coordinator Election - UCSD CSE, https://cseweb.ucsd.edu/classes/sp16/cse291-e/applications/ln/lecture6.html 49. Election Algorithms:, https://www.kdkce.edu.in/pdf/HVG-8IT-DS-Election%20Algorithm.pdf 50. Election Algorithms - A ring algorithm - Codemia, https://codemia.io/knowledge-hub/path/election_algorithms_-_a_ring_algorithm 51. Understanding Data-Centric Consistency Models - Medium, https://medium.com/@paulsapna153/understanding-data-centric-consistency-models-c19ed0e5bd66 52. Consistency model - Wikipedia, https://en.wikipedia.org/wiki/Consistency_model 53. Distributed Systems Unit-III Replication and Fault Tolerance - Akshay Jain, https://jainakshay781.wordpress.com/wp-content/uploads/2022/02/dcs-final-unit-iii.pdf 54. Fault-Tolerance II: Replication, Time and Consistency - cs.Princeton, https://www.cs.princeton.edu/courses/archive/fall15/cos518/lectures/L4-replication-consistency.pdf 55. Consistency Model in Distributed System - GeeksforGeeks,

https://www.geeksforgeeks.org/operating-systems/consistency-model-in-distributed-system/ 56. What are Consistency Models? Definition & FAQs | ScyllaDB, https://www.scylladb.com/glossary/consistency-models/ 57. Client-Centric Consistency Models - Gao's blog, https://vitaminac.github.io/posts/Client-Centric-Consistency-Models/ 58. Client-Centric Consistency Models in Distributed Systems - YouTube, https://www.youtube.com/watch?v=9XSWe-sp9M8 59. Chapter 7, https://csis.pace.edu/~marchese/CS865/Lectures/Chap7/Chapter7fin.htm 60. Chapter 7: CONSISTENCY AND REPLICATION - UTSA, https://www.cs.utsa.edu/~korkmaz/teaching/resources-os-grad/tk-slides/ts/ch07-ts-tk-consistency-replication.pdf 61. Active-Active vs. Active-Passive: High-Availability Guide | Aerospike, https://aerospike.com/blog/active-active-vs-active-passive/ 62. Active and Passive Replication in Distributed Systems | Jaksa's Blog - WordPress.com, https://jaksa.wordpress.com/2009/05/01/active-and-passive-replication-in-distributed-systems/ 63. Understanding the Differences and Advantages of Active-Active vs. Active-Passive Replication - pgEdge, https://www.pgedge.com/blog/understanding-the-differences-and-advantages-of-active-active-vs-active-passive-replication 64. Active-Active Vs. Active-Passive High-Availability Clustering | JSCAPE, https://www.jscape.com/blog/active-active-vs-active-passive-high-availability-cluster 65. Quorum-Based Replication Strategies - GeeksforGeeks, https://www.geeksforgeeks.org/system-design/quorum-based-replication-strategies/ 66. Lecture 17: March 30 17.1 Overview 17.2 Implementing Consistency Models - LASS, https://lass.cs.umass.edu/~shenoy/courses/spring22/lectures/Lec17_notes.pdf 67. Remote Write Protocol in Distributed Systems - GeeksforGeeks, https://www.geeksforgeeks.org/operating-systems/remote-write-protocol-in-distributed-systems/ 68. Lecture 6 Consistency and Replication - Electrical and Computer Engineering Department, http://ece.uprm.edu/~wrivera/ICOM6006/Lecture6.pdf 69. Fault Tolerance in Distributed System - GeeksforGeeks, https://www.geeksforgeeks.org/computer-networks/fault-tolerance-in-distributed-system/ 70. Erasure Coding vs. Replication for Fault Tolerant Systems - GeeksforGeeks, https://www.geeksforgeeks.org/system-design/erasure-coding-vs-replication-for-fault-tolerant-systems/ 71. Failure and Failure Detection - Department of Computer Science ..., https://cse.buffalo.edu/~eblanton/course/cse486-2024-0s/materials/06-failures.pdf 72. The Spectrum of Failure Models in Distributed Systems | by ALameer Ashraf | Medium, https://medium.com/@alameerashraf/the-spectrum-of-failure-models-in-distributed-systems-1951bdb3ce72 73. Faults and fault-tolerance, https://homepage.divms.uiowa.edu/~ghosh/16612.week10.pdf 74. Byzantine fault - Wikipedia, https://en.wikipedia.org/wiki/Byzantine_fault 75. Failure modes in distributed systems - Alvaro Videla, https://alvaro-videla.com/2013/12/failure-modes-in-distributed-systems.html 76. Fault Tolerance in Distributed Systems: Strategies and Case Studies - DEV Community, https://dev.to/nekto0n/fault-tolerance-in-distributed-systems-strategies-and-case-studies-29d2 77. Design Patterns: 5 Expert Techniques for Boosting Fault Tolerance ..., https://www.designgurus.io/kb/design-patterns-5-expert-techniques-for-boosting-fault-tolerance-in-distributed-systems 78. Fault Tolerance Techniques to Know for Parallel and Distributed Computing - Fiveable, https://fiveable.me/lists/fault-tolerance-techniques 79. What Is Fault Tolerance? | Creating a Fault-tolerant System - Fortinet, https://www.fortinet.com/resources/cyberglossary/fault-tolerance 80. Fault Tolerance Techniques in Distributed System - International Journal of Engineering Innovation and Research, http://www.ijeir.org/administrator/components/com_jresearch/files/publications/IJEIR_50_Final.p

df