Distributed Operating Systems - 25 Complete 10-Mark Answers

UNIT I: FOUNDATIONS OF DISTRIBUTED SYSTEMS

Q1: Architectures of Distributed Systems & System Architecture Types

Answer:

1. Introduction to Distributed System Architecture (2 marks) A distributed system architecture defines how independent computers are organized and communicate to work as a unified system. It represents the structural design pattern that determines component interaction, resource sharing, and activity coordination. Three main architectural paradigms exist based on organizational structure.

2. Client-Server Architecture (3 marks) - **Definition:** Centralized model where multiple clients request services from servers - **How it works:** Client initiates request → server processes → sends response - **Real example:** Gmail (browser requests emails from Google's servers) - **Advantages:** Easy to manage, centralized security, simple deployment - **Disadvantages:** Single point of failure, server bottleneck, expensive server resources

3. Peer-to-Peer (P2P) Architecture (2 marks) - **Definition:** All computers (peers) are equal, no central server - **Key features:** Decentralized control, direct peer communication, no hierarchy - **Real examples:** BitTorrent, Blockchain networks, Skype - **Advantages:** Highly scalable, fault-tolerant, load distributed - **Disadvantages:** Complex management, security harder, consistency issues

4. Multi-tier (N-tier) Architecture (2 marks) - **Definition:** System divided into logical layers (Presentation, Application, Data) - **Layers:** - Presentation: User interface - Application: Business logic - Data: Database storage - **Example:** E-commerce (product page → discount calculation → database storage) - **Advantages:** Clean separation, easier maintenance, independent scalability - **Disadvantages:** Complex communication, potential bottlenecks, increased latency

5. Comparison: Centralized vs Decentralized vs Hybrid (1 mark) | Aspect | Centralized | Decentralized | Hybrid | |———|————-|————–|———| | Control | One authority | All equal | Multiple centers | | Failure | Total risk | Graceful degradation | Partial impact | | Coordination | Easy | Complex | Moderate | | Example | Client-Server | P2P | DNS |

Q2: Challenges in Distributed Systems & Design Issues

Answer:

1. Introduction (1 mark) Distributed systems solve scalability problems but introduce new challenges: heterogeneity, transparency, concurrency, network failures, and security concerns.

2. Major Design Challenges (3 marks)

A. Heterogeneity Challenge - Different computers, OS, networks, programming languages - Solution: Middleware to abstract differences - Example: Java RMI enables different systems to call each other

B. Transparency (Hiding Distribution) - Location Transparency: Users don't know resource location ("/home/file.pdf" not "/Server5/…") - Access Transparency: Remote accessed like local - Failure Transparency: System continues despite failures - Concurrency Transparency: Users unaware of concurrent processes

C. Concurrency Challenge - Multiple processes accessing shared resources - Problems: Race conditions, deadlocks, data corruption - Solution: Proper synchronization mechanisms

3. Network & Reliability Issues (2 marks)

A. Unreliable Networks - Messages can be: lost, delayed, corrupted, duplicated, reordered - Example: "transfer $100" arrives twice = money transferred twice - Solution: Duplicate detection, acknowledgments

B. Partial Failures - Some components fail while others work - Hard to detect: Is server down or slow? Message lost or delayed? - Byzantine Failure: Components behave unpredictably

4. Consistency & State Management (2 marks)

A. Consistency Problem - Scenario: Database replicated on 3 servers, Client writes to Server A, Client 2 reads from Server B - Solutions: - Strong Consistency: All servers updated before returning (slow) - Eventual Consistency: Servers eventually match (fast) - Session Consistency: Consistent within session

B. Clock Synchronization - Each computer has own clock running at different speeds - Need to know which event happened first - Solution: Lamport timestamps

5. Security & Scalability (2 marks)

A. Security Issues - Authentication: Verify identity across network - Authorization: Who can access what? - Encryption: Keep data private in transit - Network attacks: Eavesdropping, man-in-the-middle, DoS

B. Scalability Challenge - Horizontal scaling: Add more computers (preferred) - Vertical scaling: Add more power to existing (limited) - Communication overhead increases with size - Coordination becomes harder

---

Q3: Communication Models in Distributed Systems

Answer:

1. Introduction (1 mark) Communication models define how processes exchange information in distributed systems. Fundamental choice between synchronous and asynchronous models impacts system design.

2. Synchronous Communication Model (2.5 marks) - **Definition:** Blocking communication where sender waits for receiver - **Characteristics:** - Sender blocks until message acknowledged - Receiver blocks until message arrives - Guaranteed delivery awareness - **Advantages:** Simple, no buffering needed, deadlock prevention easier - **Disadvantages:** Slow, processes can't do other work, potential for deadlocks - **Example:** Calling a friend (waiting for response)

3. Asynchronous Communication Model (2.5 marks) - **Definition:** Non-blocking where sender continues after sending - **Characteristics:** - Sender sends and continues immediately - Receiver checks for messages when ready - Message buffering required - **Advantages:** Fast, parallel execution, better performance - **Disadvantages:** Complex buffering, sender unaware of delivery, out-of-order issues - **Example:** Email (don't wait for read receipt)

4. Remote Procedure Call (RPC) (2 marks) - **Goal:** Make remote call look like local function call - **Steps:** 1. Client calls local stub function 2. Stub serializes parameters 3. Stub sends to server 4. Server deserializes and executes 5. Server sends result back 6. Client stub deserializes and returns - **Advantages:** Transparent, familiar to programmers - **Disadvantages:** Network failures cause errors, can't handle certain parameter types

5. Direct Message Passing (1.5 marks) - Explicit send/receive operations - Process A sends to specific process - Receiver retrieves from queue - More control than RPC - Better for distributed algorithms

6. Comparison (0.5 marks) RPC: Higher abstraction, easier programming Message Passing: Lower abstraction, more control, explicit synchronization

---

Q4: Lamport's Logical Clocks, Vector Clocks & Causal Ordering

Answer:

1. Problem Statement (1 mark) In distributed systems with unsynchronized physical clocks, determining event order is impossible. We need logical time based on events, not wall-clock.

2. Lamport's Logical Clocks (2.5 marks)

Core Idea: Count events instead of using wall-clock time.

Algorithm: - Each process maintains counter L - On local event: L = L + 1 - Before sending: Attach L to message - On receiving message with timestamp T: L = max(L, T) + 1

Example with 3 processes:

```
Process A: L=0 → event → L=1 → event → L=2 → send(ts=2)
Process B: L=0 → event → L=1 → receive(ts=2) → L=max(1,2)+1=3
Process C: L=0 → event → L=1 → event → L=2 → receive(ts=2) → L=3
```

Property: If Lamport(X) < Lamport(Y), then X happened-before Y

Limitation: Doesn't capture all causal relationships

3. Vector Clocks (2.5 marks)

Concept: Use array instead of single number

Algorithm: - For N processes, maintain vector of N numbers - On local event: V[i] = V[i] + 1 - Before sending: Attach entire vector - On receiving T: V[j] = max(V[j], T[j]) for all j, then V[i]++

Example:

```
Process A: V_A = [1,0] → local → [2,0] → send → receive [2,1] → [2,2]
Process B: V_B = [0,1] → receive [2,0] → [max(0,2), max(1,0)] → [2,1] → [2,2]
```

Ordering Rule: V_A < V_B if V_A[i] V_B[i] for all i AND at least one strictly < - [2,0] < [2,1] ? YES - [2,1] < [1,2] ? NO (not comparable, concurrent)

4. Causal Ordering (2.5 marks)

Definition: X causally precedes Y if: - X happens before Y at same process, OR - X sends message received by Y, OR - Chain: X → intermediate → Y

Example:

```
Alice: "Let's meet at coffee" → [message]
                                ↓
Bob: [receives] "Okay, 3 PM" → [message]
                                ↓
Charlie: [receives] "I'll come 3 PM"
```

Causal Ordering Guarantee: Messages delivered respecting causality - If Bob's message causally depends on Alice's, Charlie must receive Alice's first - Vector clocks ensure this

5. Applications (1.5 marks) - Distributed databases: Ensure consistent state - Message passing systems: Maintain message order - Event logging: Reconstruct system history - Distributed debugging: Understand event sequences

---

Q5: Interprocess Communication & Message Passing

Answer:

1. Introduction (1 mark) IPC is the mechanism allowing processes to exchange data and coordinate actions. Distributed systems need explicit message passing because processes have separate memory spaces.

2. Message Passing Model (2 marks)

Basic Concept: Processes exchange data by sending/receiving messages through network

Two Fundamental Modes:

Synchronous Communication: - Sender blocks until message received - Receiver blocks until message arrives - Advantages: Simple, guaranteed delivery awareness - Disadvantages: Slow, processes can't do other work

Asynchronous Communication: - Sender sends and continues - Receiver checks for messages when ready - Advantages: Fast, parallel execution - Disadvantages: Complex buffering, out-of-order issues

3. Remote Procedure Call (RPC) (2 marks)

Goal: Make remote call look like local function call

Steps: 1. Client calls local stub function 2. Stub serializes parameters 3. Stub sends message to server 4. Server deserializes and calls actual function 5. Function executes and returns result 6. Server sends result back to client 7. Client stub deserializes and returns to caller

Advantages: Transparent, familiar programming model **Disadvantages:** Network failures cause errors, can't handle open files or functions

4. Message Buffering & Ordering (2 marks)

Buffering Strategies: - Unbuffered: Messages in OS buffer, limited size - Buffered (Queue): Messages stored in queue on receiver

Out-of-Order Problem:

```
A sends: MSG1 "Create account John"
         MSG2 "Deposit $1000"

Network delays MSG1, MSG2 arrives first

B receives:
1. "Deposit $1000" ERROR! Account doesn't exist
2. "Create account" Too late!
```

Solution: Ordered delivery or application-level handling

5. Reliability in Message Passing (2 marks)

Problems: - Message Loss: Network drops packets - Duplication: Message arrives twice - Corruption: Bits changed in transit - Delays: Unpredictable latency

Solutions:

Acknowledgments: - Receiver confirms message received - Sender retransmits if no ACK within timeout

Checksums: - Detect corrupted messages - Discard bad ones

Sequence Numbers: - Detect duplicates - Process each message only once

6. Message Passing vs Shared Memory (1 mark)

Aspect	Message Passing	Shared Memory
Distribution	Natural for networks	Hard to distribute
Synchronization	Explicit	Implicit
Performance	Network latency overhead	Fast (local memory)
Debugging	Complex	Easier

UNIT II: MUTUAL EXCLUSION

Q6: Lamport's Mutual Exclusion Algorithm

Answer:

1. Problem Statement (1 mark) Multiple processes need to access shared resources. Only ONE process should access critical section at a time. Need distributed algorithm without central coordinator.

2. Algorithm Overview (2 marks)

Data Structures: - Logical clock (Lamport timestamp) - Request queue (ordered by timestamp) - Message channels to all processes

Three Message Types: - REQUEST(ts, pid): "I want to enter critical section" - REPLY(ts, pid): "Request received, go ahead when ready" - RELEASE(ts, pid): "I'm done, leaving critical section"

3. Detailed Algorithm (3 marks)

When process P wants critical section: 1. Increment logical clock 2. Add request <ts, ProcessID> to own queue 3. Send REQUEST to ALL other processes 4. WAIT until: - Own request at HEAD of queue - Have REPLY from

ALL other processes 5. Enter Critical Section 6. Do critical task 7. Remove own request from queue 8. Send RELEASE to all

When P receives REQUEST from Q: 1. Update clock: LC = max(LC, request_ts) + 1 2. Add Q's request to queue (ordered by timestamp) 3. Immediately send REPLY to Q

When P receives RELEASE from Q: 1. Remove Q's request from queue 2. Check if own request now at head

4. Example with 3 Processes (2 marks)

```
Process A (ts=1, wants CS):
  Adds <1,A> to queue
  Sends REQUEST to B, C

Process B (receives REQUEST <1,A>):
  Updates clock, adds to queue: [<1,A>]
  Sends REPLY to A

Process C (receives REQUEST <1,A>):
  Updates clock, adds to queue: [<1,A>]
  Sends REPLY to A

Process A:
  Receives REPLY from B and C
  Queue: [<1,A>] (own request at head)
  Has all REPLYs → ENTERS CS

Process B (ts=2, wants CS):
  Sends REQUEST <2,B>
  Queue: [<1,A>, <2,B>]
  WAITS (A's request still at head)

Process A (done):
  Sends RELEASE
  Removes own request

Process B (receives RELEASE):
  Queue: [<2,B>]
  Own request at head
  ENTERS CS
```

5. Why This Works (1 mark)

Properties: - **Mutual Exclusion:** Only process with smallest timestamp can enter - **No Starvation:** FIFO order ensures fairness - **Deadlock-free:** No waiting for resources, only timestamps and replies

6. Message Complexity & Limitations (1 mark)

Message Count: 3(N-1) per critical section entry - REQUEST: N-1 messages - REPLY: N-1 messages - RELEASE: N-1 messages - For 100 processes: 29,700 messages/second!

Limitations: - Failure vulnerable: Process crash blocks others waiting for REPLY - Not suitable for dynamic systems: Processes joining/leaving complex - Asynchronous overhead: Must wait for synchronous replies

---

Q7: Ricart-Agrawala Algorithm vs Lamport's Algorithm

Answer:

1. Overview (1 mark) Both algorithms solve mutual exclusion in distributed systems without central coordinator. Ricart-Agrawala improves on Lamport by reducing message count.

2. Ricart-Agrawala Algorithm (2.5 marks)

Key Improvement: Combine REQUEST and REPLY logic

Concept: - When process receives REQUEST, immediately REPLY (not defer) - Wait until have REPLY from all before entering - On RELEASE, process waiting REQUEST with highest priority can enter

Algorithm: 1. Process wanting CS: LC++, send REQUEST to all 2. Receiver of REQUEST: - If not in CS and not wanting CS: send REPLY immediately - Else: defer reply (add to deferred queue) 3. Process enters CS when: - Has REPLY from all others - Own request at head of queue 4. On RELEASE: process all deferred REPLYs

Advantages: - Fewer messages: 2(N-1) vs 3(N-1) in Lamport - No explicit RELEASE message needed

3. Comparison: Ricart-Agrawala vs Lamport (2.5 marks)

Aspect	Lamport	Ricart-Agrawala
Message Count	3(N-1)	2(N-1)
Request Queue	Yes	Yes
Immediate REPLY	No (may defer)	Yes (if not in CS)
Deferred Replies	No	Yes
Complexity	Lower	Slightly higher
Performance	Moderate	Better
Failure Recovery	Complex	Complex

4. Example Comparison (2 marks)

Lamport (3 messages): - A sends REQUEST - B sends REPLY - C sends REPLY - A RELEASES - Total: 3 messages for A to enter CS

Ricart-Agrawala (2 messages): - A sends REQUEST - B sends REPLY (immediate, not waiting) - C sends REPLY (immediate, not waiting) - A automatically handles deferred cases - Total: 2 messages for A to enter CS

5. Practical Considerations (1.5 marks)

When to use Lamport: - Simpler implementation - Explicit control over deferral

When to use Ricart-Agrawala: - Lower message overhead critical - Dynamic systems where traffic matters - Network-constrained environments

Both algorithms: - Don't handle process failures well - Not practical for large-scale systems - Theoretical importance for understanding distributed coordination

6. Modern Alternatives (0.5 marks) - Token-based algorithms: Lower latency - Voting-based: Better fault tolerance - Quorum-based: Scalable approach

---

Q8: Distributed Mutual Exclusion Using Token Ring

Answer:

1. Introduction (1 mark) Token ring algorithm uses a logical token passed among processes in a ring. Only token holder can enter critical section, reducing message overhead compared to voting algorithms.

2. System Model (1.5 marks)

Structure: - Processes arranged logically in a ring - Token circulates among processes - Only process holding token can enter CS - Token moves in specific order (e.g., P0 $\rightarrow$ P1 $\rightarrow$ P2 $\rightarrow$ ... $\rightarrow$ PN $\rightarrow$ P0)

Advantages: - Fair: Every process eventually gets token - Deterministic: No competition or starvation - Low message overhead: Only token passing messages

3. Algorithm Details (2 marks)

When process wants CS: 1. Wait for token to arrive 2. Check if process has token 3. If yes, enter CS 4. Do critical task 5. Pass token to next process 6. If no token arrives, wait

Token Movement:

```
Initial: Token at P0

P0 (has token):
  Enters CS
```

```
  Executes critical task
  Passes token to P1

P1 (receives token):
  Enters CS
  Executes critical task
  Passes token to P2

P2 (receives token):
  Enters CS
  Executes critical task
  Passes token to P3
```

4. Performance Analysis (2 marks)

Advantages: - Message Complexity: O(N) (only N messages per round) - Fair scheduling: Round-robin fairness - Starvation-free: Every process eventually gets token - Simple implementation: No complex distributed algorithms

Disadvantages: - If token lost: System deadlock - Time to CS unpredictable: May wait for N-1 processes - Overhead when system idle: Token still circulates - Process failure: Token handling complex

Example Metrics: - N=10 processes - Time to CS: 0 to 9 token passes - Average wait: 5 token passes

5. Recovery Mechanism (1.5 marks)

Token Loss Detection: - Monitor token timeout - If no token received within time T: token lost - Regenerate token through election algorithm

Process Failure Handling: - Neighbor detection: Check if neighbor alive - Token regeneration: Elect new token holder - State recovery: Difficult in asynchronous systems

6. Comparison with Other Algorithms (1 mark)

Algorithm	Messages	Fairness	Recovery
Lamport	3(N-1)	Yes	Hard
Ricart-Agrawala	2(N-1)	Yes	Hard
Token Ring	N	Fair RR	Hard
Centralized	3	Possible unfairness	Single point failure

Q9: Centralized Mutual Exclusion Algorithm

Answer:

1. Overview (1 mark) Simplest approach: Designate one process as coordinator who grants permission to enter critical section.

2. System Model (1.5 marks)

Coordinator Role: - Maintains queue of requests - Grants permission to one process at a time - Tracks who's in CS

Process Roles: - Sends REQUEST to coordinator - Waits for REPLY (permission) - Enters CS - Sends RELEASE to coordinator - Coordinator grants next

3. Algorithm Details (2 marks)

When process P wants CS: 1. Send REQUEST to coordinator 2. Wait for REPLY from coordinator 3. Enter critical section 4. Execute critical task 5. Send RELEASE to coordinator

Coordinator Logic: 1. Receive REQUEST from P 2. If CS free: Send REPLY immediately 3. Else: Add P to queue, defer REPLY 4. Receive RELEASE from P holding CS 5. Remove P from CS 6. If queue not empty: Send REPLY to first in queue

Queue Management:

```
Queue: [P2, P3, P1]

P2 in CS currently
P3 next
P1 third

When P2 releases:
  Remove P2 from CS
  Send REPLY to P3
  P3 can now enter CS
  Queue becomes [P1]
```

4. Message Complexity (1.5 marks)

Per CS Entry: 3 messages 1. REQUEST: Process to coordinator 2. REPLY: Coordinator to process 3. RELEASE: Process to coordinator

Total: 3 messages (very efficient!)

Comparison: - Lamport: 3(N-1) = 9 messages (N=4) - Centralized: 3 messages (N=4) - Improvement: 67% reduction!

5. Advantages & Disadvantages (2 marks)

Advantages: - Very simple to implement - Minimal message overhead - Fair queuing: FIFO order - No starvation: Requests served in order - Easy to understand and verify

Disadvantages: - Single point of failure: Coordinator crashes = system down - Not suitable for large systems: Bottleneck at coordinator - Not distributed: Defeats purpose of distributed system - Coordinator failure hard to detect: Sender doesn't know if REPLY lost or coordinator dead - Loss of transparency: Coordinator location must be known

6. Failure Scenarios (1 mark)

Scenario 1: Coordinator Crashes - Processes waiting for REPLY stuck - Impossible to recover without new coordinator - Need election algorithm

Scenario 2: RELEASE message lost - Coordinator thinks CS still in use - Next requestor never gets permission - Deadlock

Scenario 3: Coordinator becomes bottleneck - 1000 processes requesting CS - Coordinator receives 1000 REQUESTs/sec - Can't handle throughput - System slows down

7. Comparison with Distributed Algorithms (0.5 marks) Centralized: Simplicity vs reliability trade-off. Good for small systems, bad for large distributed systems.

Q10: Dining Philosophers Problem in Distributed Context

Answer:

1. Problem Statement (1.5 marks)

Scenario: - N philosophers sit around circular table - Between each pair: one chopstick (N total chopsticks) - Philosopher needs 2 chopsticks to eat - Eat → Think → repeat

Challenge: Design algorithm so all can eat infinitely often without deadlock

Original Problem vs Distributed Version: - Original: Shared memory on single machine - Distributed: Each philosopher on different node, chopsticks distributed

2. Deadlock Scenario (1.5 marks)

How Deadlock Occurs:

```
P0 picks left chopstick (0)
P1 picks left chopstick (1)
P2 picks left chopstick (2)
P3 picks left chopstick (3)
...
All wait for right chopstick (circular dependency)

DEADLOCK! No one can eat!
```

Conditions Met: 1. Mutual Exclusion: Each chopstick used by one 2. Hold and Wait: Holding left, waiting for right 3. No Preemption: Can't take chopstick 4. Circular Wait: P0→P1→P2→…→P0

3. Solution 1: Asymmetric Approach (1.5 marks)

Rule: - Odd philosopher: Pick left first, then right - Even philosopher: Pick right first, then left

Algorithm:

```
if (philosopher_id is odd) {
    pick_left_chopstick();
    pick_right_chopstick();
} else {
    pick_right_chopstick();
    pick_left_chopstick();
}
eat();
put_left_chopstick();
put_right_chopstick();
```

Why it works: - Breaks circular dependency - At least one philosopher can always get both chopsticks - No deadlock possible

4. Solution 2: Centralized Monitor (1.5 marks)

Approach: - Centralized coordinator maintains state - Philosopher requests both chopsticks together - Coordinator grants only if both available

Algorithm:

```
REQUEST both chopsticks from coordinator
Wait for permission
If granted:
  eat()
  Release both chopsticks to coordinator
Else:
  Wait in queue
  Retry when notified
```

Advantages: Simple, no deadlock **Disadvantages:** Centralized (coordinator bottleneck, single point failure)

5. Solution 3: Distributed Token-Based (1.5 marks)

Approach: - Each chopstick is a token - Philosopher requests tokens from neighbors - Token passed only if neighbor not eating

Algorithm:

```
For each needed chopstick:
  Send REQUEST to neighbor holding token
```

```
  Wait for token reply
Once have both tokens:
  Enter CS (eat)
  Release both tokens
  Send to neighbors
```

Message Complexity: O(N) requests, responses in distributed manner

6. Solution 4: Resource Hierarchy (1 mark)

Rule: Total order on chopsticks

```
Chopsticks numbered 0 to N-1
Philosopher always picks lower-numbered first
Then higher-numbered

P0: Pick 0, then 1
P1: Pick 1, then 2
P2: Pick 2, then 3
...
PN: Pick N, then 0 (wrap around with ordering)

Actually: Pick min(i, (i+1)%N), then max(i, (i+1)%N)
```

Why works: Prevents circular dependency

7. Comparison of Solutions (1 mark)

Solution	Deadlock-free	Starvation	Efficiency	Complexity
Asymmetric	Yes	Possible	Fair	Low
Centralized	Yes	No	Poor	Very Low
Token-based	Yes	No	Fair	Medium
Hierarchy	Yes	Possible	Good	Low

Best for distributed: Token-based (fair, no single point failure)

UNIT III: DEADLOCK DETECTION

Q11: Deadlock Detection Algorithms

Answer:

1. Definition & Conditions (1.5 marks)

Deadlock: State where processes can't proceed because each waits for resources held by others

Necessary Conditions (Coffman): 1. Mutual Exclusion: One process at a time 2. Hold and Wait: Holding while waiting 3. No Preemption: Can't forcibly take 4. Circular Wait: Cyclic resource request

Break any one = no deadlock

2. Wait-For Graph Construction (2 marks)

Building the Graph: - Nodes: Processes - Edges: P1 → P2 means P1 waits for resource held by P2

Example:

```
P1 waits for resource held by P2
P2 waits for resource held by P3
P3 waits for resource held by P1

Graph:
P1 → P2
P2 → P3
P3 → P1

CYCLE DETECTED = DEADLOCK!
```

Cycle Detection Algorithm: 1. Build wait-for graph 2. Detect cycle using DFS/BFS 3. If cycle exists: Deadlock detected

3. Detection Algorithm Details (2 marks)

Algorithm:

```
1. Build wait-for graph from resource allocation state
2. Detect all cycles in graph
3. All processes in any cycle are deadlocked
4. On detection: Declare deadlock, take recovery action
```

Complexity: - Time: O(N + E) where N=processes, E=edges - Space: $O(N^2)$ for graph representation

Implementation:

```
Create adjacency matrix for wait-for graph
For each process waiting for another:
  Add edge to matrix
Perform DFS from each node:
  If return to same node: CYCLE DETECTED
```

4. Distributed Cycle Detection (2 marks)

Challenge: No global view of wait-for graph in distributed systems

Approach 1: Centralized Detection - Collect wait-for info at central node - Build global graph - Detect cycles - Limitation: Central node bottleneck, single

point failure

Approach 2: Distributed Detection - Use token circulation to detect cycles - Each process passes token through system - Token creates virtual edges - If token returns: Cycle exists

Phantom Deadlock Problem: - Network delays create false deadlocks - Message delayed makes system appear deadlocked - Message arrives later, deadlock resolves - Hard to distinguish real vs phantom

5. Recovery Mechanisms (1.5 marks)

Once Deadlock Detected:

Strategy 1: Victim Selection - Choose process to kill (victim) - Selection criteria: - Lowest priority process - Process used least resources - Process with least execution time - Kill victim process

Strategy 2: Rollback & Restart - Victim rolled back to checkpoint - Resources released - Other processes can now proceed - Victim restarted later

Strategy 3: Resource Preemption - Forcibly take resources from some process - Give to deadlocked process - Complex in distributed systems

6. Disadvantages & Alternatives (1.5 marks)

Disadvantages: - Building global graph hard in distributed systems - Phantom deadlocks: False positives - Recovery costly: Loss of work by victim - No prevention: Allows bad states - Detection delay: Deadlock persists until detected

Better Alternatives: - Prevention: Ensure deadlock can't occur - Avoidance: Don't enter unsafe states (Banker's algorithm) - Ignore deadlocks: Accept and handle at application level

---

Q12: Deadlock Prevention & Avoidance

Answer:

1. Overview (1 mark) Prevention makes deadlock impossible by breaking one Coffman condition. Avoidance predicts unsafe states before they happen.

2. Deadlock Prevention (3 marks)

Strategy 1: Break Mutual Exclusion - Make resources sharable (if possible) - Problem: Many resources can't be shared (locks, devices) - Rarely applicable

Strategy 2: Break Hold and Wait - Require all resources before starting - Request all upfront or request one at a time, release before requesting next - Problem: Inefficient resource usage, processes hold resources unnecessarily

Strategy 3: Break No Preemption - Allow forcibly taking resources - If process needs resource held by another, preempt and take it - Problem: Difficult to implement, context loss, rollback needed

Strategy 4: Break Circular Wait (BEST!) - Impose ordering on resources - All processes must request in same order - Example: Always request A before B

Implementation:

```
No Ordering (Deadlock possible):
P1: Lock(A) → Lock(B)
P2: Lock(B) → Lock(A)
Deadlock!

With Ordering (No Deadlock):
P1: Lock(A) → Lock(B)
P2: Lock(A) → Lock(B)
No way to form cycle!
```

Problem: Application might naturally need different order

3. Deadlock Avoidance (2 marks)

Concept: Detect unsafe states before they happen, avoid transitions to unsafe states

Banker's Algorithm Example:

```
Resources Available: 10 units
Process 1: Max needed = 5, Has = 3, Still needs = 2
Process 2: Max needed = 8, Has = 6, Still needs = 2

Question: Can we safely give 2 to Process 1?

Check:
If Process 1 gets 2: Uses 5 total, 5 free left
Process 1 finishes: Releases 5, now 10 total free
Process 2 can now get its 2 more

Result: SAFE to grant
```

Algorithm: 1. Simulate request granting 2. Check if system remains in safe state 3. Safe = All processes can eventually complete 4. If safe: Grant request 5. If unsafe: Deny request, wait

Advantages: - Allows more concurrency than prevention - Prevents deadlock like prevention

Disadvantages: - Need to know max demands upfront (hard in distributed) - Overhead: Must simulate for each request - Overly conservative: Denies safe

requests sometimes

4. Comparison: Prevention vs Avoidance (2 marks)

Aspect	Prevention	Avoidance
Approach	Break condition	Detect unsafe state
Max Info Needed	No	Yes (max demands)
Overhead	Low (ordering only)	High (simulate each)
Concurrency	Low (restrictions)	Medium (better than prevention)
Implementation	Simple	Complex
Practical Use	Limited	Limited in distributed

5. Practical Considerations (1.5 marks)

Prevention used when: - Resource ordering naturally exists - Low concurrency acceptable - Simple implementation important

Avoidance used when: - Need higher concurrency - Max demands predictable - Overhead acceptable

In Distributed Systems: - Prevention preferred: Simple - Avoidance hard: Don't know global max demands - Most systems allow deadlocks, handle at app level

---

UNIT IV: FILE SYSTEMS

Q13: Distributed File Systems Architecture

Answer:

1. Introduction & Motivation (1.5 marks)

Problem: Users want files accessible from anywhere, any device

Solution: Distributed File System (DFS)

Goals: - Files appear local despite being remote - Transparency: Users unaware of distribution - Reliability: Survive node failures - Performance: Efficient access despite network

2. DFS Architecture Components (2 marks)

Clients: - Run on user machines - Issue file requests - Cache frequently accessed files - Have local filesystem interface

Servers: - Store actual file data - Handle concurrent requests - Manage metadata (permissions, ownership) - Provide consistency guarantees

Network: - Interconnects clients and servers - RPC-based communication - Reliability through retries

Naming Service: - Maps file names to server locations - Maintains directory hierarchy - Provides transparency

Storage: - Disks on servers - Replicated for fault tolerance

3. Naming & Transparency (2 marks)

Location Transparency: - Users see: `/home/alice/file.pdf` - Actually stored on: Server 5, Block 2340 - Users don't need to know location

Benefits: - Files can move between servers - User code unchanged - Easy load balancing - Hide server details

Implementation: - Naming service maintains mapping - On file open: Query naming service - Get server location - Contact appropriate server

4. Caching Strategies (2 marks)

Write-Through: - Client writes → immediately updates server - Advantages: Consistent - Disadvantages: Every write network latency

Write-Back: - Client writes → cache locally - Later: Flush to server - Advantages: Fast - Disadvantages: Data loss risk on crash

Delayed-Write (Popular): - Write at file close - Advantages: Balanced (good performance + reasonable consistency) - Used in: NFS, many systems

Cache Invalidation: - Server notifies on changes - AFS: Callback mechanism - NFS: Timeout-based

5. Consistency Models (1.5 marks)

Strict Consistency: - All see latest write immediately - Expensive: Requires sync updates

Session Consistency: - Within session consistent - Between sessions may be stale - Popular: Practical compromise

Eventual Consistency: - Eventually everyone sees same data - Temporarily inconsistent OK - Used in: Cloud systems (Google Drive, OneDrive) - Cheap, scalable

6. Real Examples (0.5 marks)

NFS (Network File System): - Simple client-server - Stateless servers - Client caching - Session consistency

AFS (Andrew File System): - Whole-file caching - Callbacks for consistency - Strong consistency

---

Q14: Caching in Distributed File Systems

Answer:

1. Motivation for Caching (1 mark) Direct server access too slow. Caching at client provides local access speed, reduces network traffic.

2. Caching Strategies (2.5 marks)

Block-Level Caching: - Cache individual blocks (4KB pages) - Advantages: Fine-grained, less redundant - Disadvantages: Complex, more network trips for small changes

File-Level Caching: - Cache entire files - AFS approach - Advantages: Simple, atomic operations - Disadvantages: Wastes space for large files

Directory Caching: - Cache directory entries - Reduces metadata lookups - Critical for performance

3. Consistency Issues (2.5 marks)

Problem: Multiple clients caching same file, one modifies

Close-to-Open Consistency: - Write becomes visible after file close - Read sees last visible version - Practical and efficient

Callback Mechanism: - Server notifies on file change - Client invalidates cache - AFS uses this

Timestamp-Based: - Check timestamp on server - Invalidate if changed - NFS approach

4. Update Propagation (2 marks)

Write-Through: Update visible immediately (expensive)

Write-Behind (Delayed): - Update local cache - Flush on close or timeout - Risk: Crash loses updates

Batch Updates: - Collect multiple writes - Flush together - Reduces network traffic

5. Advantages & Disadvantages (1.5 marks)

Advantages: - Dramatic speed improvement - Reduced network traffic - Better user experience

Disadvantages: - Cache coherence complexity - Stale data risk - Crash safety issues - Invalidation overhead

6. Implementation Considerations (0.5 marks) - Cache size management - Eviction policies (LRU) - Consistency level trade-offs

---

Q15: Replication in Distributed File Systems

Answer:

1. Why Replication? (1 mark) Single copy = data loss if server fails. Multiple copies = redundancy and availability.

2. Replication Strategies (2.5 marks)

Primary Copy (Master-Slave): - One primary server handles all writes - Backup replicas read-only - On primary failure: Promote backup

Quorum-Based: - Write to majority (quorum) of replicas - Read from any replica - Ensures consistency

Full Replication: - Write to all replicas - Most consistent but slowest

Consistency Guarantees: - Replica must become consistent - Time varies by strategy

3. Data Consistency (2 marks)

Strong Consistency: - All replicas updated before returning - Guarantees reads see latest - Slow: Multiple network round-trips

Eventual Consistency: - Replicas eventually match - Temporarily inconsistent - Fast: Can return immediately

Causal Consistency: - Related updates respect causal order - Middle ground

4. Failure Scenarios (1.5 marks)

Server Failure: - Other replicas continue serving - Reads: Go to available replica - Writes: Go to quorum of available

Network Partition: - Replicas split into groups - CAP theorem: Choose 2 of 3 (Consistency, Availability, Partition tolerance)

Data Divergence: - Different replicas have different versions - Need resolution mechanism - Often: Last-write-wins or application-specific

5. Replication Overhead (1.5 marks)

Advantages: - High availability - Better performance (read from nearest) - Fault tolerance

Disadvantages: - Storage overhead - Consistency complexity - Network traffic for updates - Slower writes (update all/quorum)

6. Real Examples (0.5 marks) - Google Bigtable: Eventual consistency with quorum writes - Amazon S3: Replication across regions

---

UNIT V: LOAD DISTRIBUTION & DSM

Q16: Distributed Shared Memory (DSM)

Answer:

1. Introduction (1 mark) DSM allows programmers to write parallel programs as if on single multiprocessor, but actually runs on networked computers. Hides distribution.

2. System Models (2 marks)

Page-Based DSM: - Shared memory divided into pages (4KB) - Each page on different node - Page fault → fetch from remote - Most common approach

Object-Based DSM: - Share objects instead of raw pages - More semantic information - Better for OOP (Java, C++)

Variable-Based DSM: - Fine-grained sharing at variable level - Overhead usually prohibitive

3. Coherence Protocols (2 marks)

Write-Invalidate: - Writer invalidates copies at others - Others refetch on next access - Good when: Few writers, read-heavy - Bad when: Multiple writers to same data

Write-Update: - Writer broadcasts new value to all - Copies updated automatically - Good when: Frequent shared writes - Bad when: Network overhead

Implementation:

```
Process 1: [Page cached]
           Process 2: Write to page
           ↓
           [Invalidate message to Process 1]
Process 1: [Cache invalid]
           [Next access: Page fault]
           [Fetch latest from Process 2]
```

4. Design Issues (2 marks)

Granularity Problem: - Large pages (1MB): Few faults but false sharing - Small pages (4KB): More faults but less false sharing - Trade-off: Typical = 4KB

False Sharing: - Two different variables on same page - One written → whole page invalidated - Other variable needs unnecessary refetch

Thrashing: - Page constantly moving between nodes - Repeated invalidate/fetch cycles - Network congested

5. Data Placement & Performance (1.5 marks)

Initial Data Placement: - Place near first accessor - Reduces initial latency

Dynamic Migration: - Move pages to frequent accessors - Detect hot pages - Migrate during idle time

Synchronization Overhead: - Locks require distributed coordination - Network round-trips for every lock - Lock caching: Local acquisition if possible

6. Advantages & Disadvantages (1 mark)

Aspect	Advantage	Disadvantage
Programming	Familiar shared-memory	Non-deterministic bugs
Performance	Can be good	Network latency overhead
Scalability	Theoretically good	Practically limited
Debugging	Harder than message passing	Complex to trace

Best for: Shared-memory algorithm applications on clusters

Q17: Load Balancing & Task Distribution

Answer:

1. Problem Statement (1 mark) In distributed systems, work unevenly distributed. Some nodes heavily loaded, others idle. Goal: Balance load for better overall performance.

2. Load Balancing Goals (1.5 marks)

Objectives: - Minimize mean response time - Maximize throughput - Reduce idle time on any node - Prevent any node becoming bottleneck - Balance communication vs computation trade-off

Metrics: - CPU utilization - Memory usage - I/O queue length - Network bandwidth

3. Policies for Load Balancing (2.5 marks)

Centralized Approach: - Central load balancer - Monitors all nodes - Assigns new tasks - Advantages: Simple, global view - Disadvantages: Bottleneck, single point failure

Distributed Approach: - Peers make local decisions - Query neighbors for status - Advantages: No bottleneck, scalable - Disadvantages: Inconsistent view, delayed info

Task Transfer Policies:

```
Threshold Policy:
If node load > threshold: Send to neighbor
```

```
Probe Policy:
Before accepting job: Query other nodes
Choose least loaded

Demand-Driven:**
Only transfer when specifically requested
Reactive vs proactive
```

4. Algorithms (2 marks)

Round-Robin: - Assign to nodes in sequence - Simple, assumes equal capacity - No load info needed - Problem: Uneven if tasks have varying load

Least-Loaded: - Assign to node with lowest current load - Better than round-robin - Problem: Overhead of querying load, network changes

Shortest-Queue: - Assign to node with smallest job queue - Considers queue length - Problem: New task might be long-running

Token-Based: - Fixed tokens (one per process slot) - Send token to request work - Load implicitly balanced

5. Migration vs Remote Execution (1.5 marks)

Remote Execution: - Job sent to remote node - Executes there - Results returned - Advantages: Simpler, less overhead - Disadvantages: Depends on network for I/O

Process Migration: - Entire process moved to another node - Can transfer state, open files - Advantages: Local I/O access - Disadvantages: Complex, state transfer overhead

6. Practical Considerations (0.5 marks) - Trade-off between balancing overhead and benefit - Network bandwidth often bottleneck - Communication locality more important than perfect balance - Dynamic vs static balancing

---

Q18: Job Scheduling in Distributed Systems

Answer:

1. Overview (1 mark) Job scheduling: Assigning jobs to nodes at right time. Goal: Minimize turnaround time, maximize utilization, meet deadline constraints.

2. Scheduling Strategies (2.5 marks)

FCFS (First Come First Served): - Jobs processed in arrival order - Fair but not optimal - Convoy effect: Long job blocks short jobs

SJF (Shortest Job First): - Shortest job first - Minimizes average wait time - Problem: Need to know job length upfront

Priority Scheduling: - Jobs with higher priority processed first - Real-time systems - Problem: Lower priority starvation

RR (Round Robin): - Each job gets time slice - Preemption allows fairness - Overhead from context switching

3. Distributed Scheduling Challenges (2 marks)

Heterogeneous Systems: - Nodes have different capacities - Job execution time varies by node - Need to estimate and adapt

Dynamic Job Arrivals: - Jobs arrive unpredictably - Can't plan in advance - Adaptive scheduling needed

Communication Locality: - Minimize data movement - Assign jobs close to data - Network bottleneck consideration

4. Scheduling Algorithms (1.5 marks)

Centralized Scheduler: - Central node assigns jobs - Has global view - Bottleneck, single point failure

Distributed Scheduler: - Local decisions at each node - Cooperate with neighbors - Scalable, resilient

Bidding Protocol:

```
Node A has job J
Broadcasts request for bidding
Nodes B, C, D reply with bids (estimated completion time)
Node A chooses lowest bid (node C)
Send job to C
```

5. Real-Time Scheduling (1 mark) - Deadlines must be met - Reject if can't guarantee deadline - Priority based on deadline

6. Energy Efficiency (0.5 marks) - Modern concern: Power consumption - Schedule to minimize active nodes - Consolidate onto fewer nodes

---

Q19: Batch Processing & Parallel Job Execution

Answer:

1. Batch Processing Concept (1 mark) Collect multiple jobs, process them together. Efficient for systems without interactive load, common in supercomputing, data processing.

2. Batch Scheduling (2 marks)

Queue Discipline:

```
Incoming jobs → Queue → Available slot → Execute
```

Backfill Scheduling: - Main schedule: Assign time slots - Backfill: Use gaps with smaller jobs - Maximizes utilization

Gang Scheduling: - Schedule all processes of job simultaneously - Prevents deadlock from partial scheduling - Improves locality and performance

3. Parallel Job Execution (2 marks)

Problem: Job might need multiple nodes

Allocation Strategies: - Contiguous: All nodes together - Non-contiguous: Scattered allocation - Contiguous better for communication locality

Processor Allocation: - 2D mesh allocation - Tree allocation - Minimize communication distance

Example:

```
Job needs 4 processors for matrix multiplication

Option 1 (Contiguous):
[A][B][C][D] → All together
Good communication locality

Option 2 (Scattered):
[A][...][B][...][C][...][D]
Bad communication locality
```

4. Load Balancing in Parallel Jobs (1.5 marks)

Challenges: - Uneven processor speeds - Network congestion - Synchronization points

Dynamic Load Balancing: - Migrate work between processors - Detect imbalance - Move tasks to underutilized processors

Example:

```
Matrix multiplication:
Processor 1: Done early
Processor 2: Still computing
Processor 1 helps Processor 2
Migrate some tasks from 2 to 1
```

5. Fault Tolerance (1.5 marks)

Challenges: - One processor fails = entire job fails - Checkpointing overhead - Recovery time

Strategies: - Checkpoint periodic state - On failure: Restart from checkpoint - Replication: Run on 2 nodes - Redundancy: Extra processor for recovery

6. Performance Metrics (0.5 marks) - Job completion time - System utilization - Turnaround time (arrival to completion) - Waiting time

---

Q20: Process Migration in Distributed Systems

Answer:

1. Definition & Motivation (1.5 marks)

Process Migration: Moving running process from one node to another

Why Migrate? - Load balancing: Move from overloaded to underutilized - Fault recovery: Before node crashes - Maintenance: Shut down node for updates - Resource consolidation: Save energy - Communication locality: Move process closer to data

2. Migration Mechanism (2 marks)

What to Migrate: - Process state: Registers, stack, heap - Open files: File descriptors, positions - Network connections: TCP connections - Cached data: Recent values

Process State:

```
CPU state: Program counter, registers
Memory: Stack and heap data
Resources: Open files, network sockets
```

3. Pre-Migration & Post-Migration (2 marks)

Pre-Migration: 1. Suspend process 2. Freeze all updates 3. Capture state: Registers, memory 4. Mark pages as transferable

Migration: 1. Send process state over network 2. Update page table at destination 3. Establish resource references 4. Update process location registry

Post-Migration: 1. Resume process at new node 2. Redirect incoming messages 3. Update file descriptor references 4. Resume normal execution

4. Challenges (1.5 marks)

Open Files: - Seek position must be preserved - File locks affected - Solution: Maintain file handle reference to original node

Network Connections: - TCP socket tied to machine - Can't move active connection - Solution: - Let connection die (restart) - Redirect through proxy - Tunneling through old node

Processor Heterogeneity: - Different instruction sets - Binary incompatible - Solution: Virtual machine (Java bytecode)

5. Weak vs Strong Migration (1 mark)

Weak Migration: - Migrate code only, not full state - Process restarts at destination - Simpler but less transparent - Example: Mobile agent moving

Strong Migration: - Migrate entire execution state - Resume immediately where left off - Complex but transparent - Example: Commercial systems

6. Practical Examples (0.5 marks) - MOSIX: Preemptive process migration - Condor: Virtual machine migration - Container migration: Docker swarm

---

UNIT VI: ADDITIONAL TOPICS

Q21: Byzantine Agreement & Fault Tolerance

Answer:

1. Problem Statement (1.5 marks)

Byzantine Agreement: Distributed system must reach agreement despite some processes behaving maliciously

Motivating Scenario:

```
Generals Problem:
Multiple generals coordinate attack on city
Some may be traitors (send conflicting messages)
Non-traitors must reach agreement despite traitors

Military analogy: Trust but verify
```

2. Byzantine Fault Model (1.5 marks)

Byzantine Behavior: - Send conflicting messages to different processes - Delay messages selectively - Forge messages - Arbitrary behavior (not just crash)

Assumptions: - Processes know list of all participants - Can send authenticated messages - Synchronous or asynchronous communication - Up to F nodes can be Byzantine ($F < N/3$)

Why N/3 threshold: - With F Byzantine out of N total - Need 2F+1 honest nodes to detect/correct - Minimum: N 3F+1

3. Byzantine General's Algorithm (2 marks)

Simplified Version (N=4, F=1):

```
3 generals + 1 commander (possibly traitor)
```

```
Round 1: Commander sends order to each general
Round 2: Each general broadcasts what they received
Round 3: Vote on majority received

Honest generals agree even if commander is traitor
Because honest majority will receive same order
```

Detailed Algorithm:

```
All processes propose value
Round 1: Each broadcasts proposal
Round 2: Each broadcasts received values
Round 3: Each broadcasts received in round 2
Continue F+1 rounds
Final: Take majority value

With N  3F+1:
- Honest processes can detect majority opinion
- Byzantine nodes can't force incorrect agreement
```

4. Practical Agreement Protocols (1.5 marks)

Two-Phase Commit: - Coordinator: "Should we commit?" - Participants: "Vote yes/no" - Coordinator: "Commit/Abort" - Problem: Coordinator failure unresolved (blocking)

Three-Phase Commit: - Phase 1: Cancommit (vote) - Phase 2: Precommit (prepare) - Phase 3: Commit (final) - Non-blocking but higher overhead

Paxos Algorithm: - Tolerates arbitrary message delays - Guarantees agreement - Complex but practical - Used in: Google Chubby, Zookeeper

5. Consensus vs Byzantine Agreement (1 mark)

Aspect	Consensus	Byzantine
Failure Type	Crash	Arbitrary
Threshold	N 2F+1	N 3F+1
Complexity	Simple	Complex
Rounds	Few	Many (F+1)

6. Practical Limitations (1 mark) - Byzantine agreement expensive (F+1 rounds) - High message complexity - Only for small F - Most systems assume crash failures only

---

Q22: Election Algorithms in Distributed Systems

Answer:

1. Why Elections? (1 mark) Many distributed algorithms need one process as coordinator: mutual exclusion, deadlock detection, leader for replication. Need algorithm to choose one.

2. Bully Algorithm (2 marks)

Idea: Process with highest ID wins

Algorithm:

```
When P detects coordinator dead:
1. P sends ELECTION message to all processes with higher ID
2. If any respond: P is not coordinator
3. If none respond: P becomes coordinator
4. New coordinator sends ELECTED to all

When P receives ELECTION from lower-ID process:
1. P responds "I'm alive"
2. P initiates its own ELECTION
```

Example (4 processes, IDs 1,2,3,4):

```
Process 2 detects coordinator (say process 4) dead

2 sends ELECTION to 3,4
3 responds "I'm alive"
3 sends ELECTION to 4
4 doesn't respond

3 becomes coordinator
3 sends ELECTED to all
```

Advantages: Simple, clear **Disadvantages:** Many messages ($O(N^2)$ worst case)

3. Ring Algorithm (2 marks)

Idea: Processes arranged in logical ring, election message passes around

Algorithm:

```
When P wants election:
1. P creates message with [P_ID]
2. Sends to next process in ring
3. Each process adds own ID to message
4. If message returns to initiator:
   - List of all IDs collected
   - Maximum ID is new coordinator
   - Announce to all

Process receiving election message:
```

```
1. Add own ID to list
2. Pass to next process
```

Example:

```
Process 2 starts election
Message: [2] → 3 → 4 → 1 → 2

Back at 2: [2,3,4,1]
Max = 4, so 4 becomes coordinator
Announce: "4 is coordinator"
```

Advantages: Only O(N) messages per election **Disadvantages:** One process down breaks ring, need repair

4. Candidate Algorithm (1.5 marks)

For Partial Synchrony:

```
Processes vote for leader
Multiple rounds if no consensus
Message passing with timeouts
Leader with highest votes wins

Handles process crashes gracefully
```

5. Election with Crash Tolerance (1 mark)

Challenge: Process crashes during election

Solutions: - Timeout mechanism: Retry if no response - Multiple rounds: Ensure all hear result - Persistent storage: Remember elections

6. Applications (0.5 marks) - Mutual exclusion: Need one coordinator (Lamport's algorithm) - Distributed file system: Primary copy selection - Database replication: Master election - Token ring: Choose token initiator

---

Q23: Distributed Transactions & Two-Phase Commit

Answer:

1. Transaction Concept (1.5 marks)

Definition: Sequence of operations that must atomically succeed or fail

Properties (ACID): - **Atomicity:** All or nothing - **Consistency:** Valid state to valid state - **Isolation:** No interference with other transactions - **Durability:** Survives failures

Distributed Challenge: - Operations span multiple nodes - Network failures mid-transaction - Partial completion possible

2. Two-Phase Commit (2PC) (2 marks)

Phase 1 (Voting/Prepare):

```
Coordinator: "Can you commit?"
  Participant 1: "Yes, I can (locked resources)"
  Participant 2: "Yes, I can"
  Participant 3: "Yes, I can"
```

Phase 2 (Commit/Abort):

```
Coordinator: "Commit!"
  Participant 1: "Committed"
  Participant 2: "Committed"
  Participant 3: "Committed"
```

If Any Can't Commit:

```
Participant 2: "No, I can't"
Coordinator: "Abort!"
  Participant 1: "Aborted (rolled back)"
  Participant 2: "Aborted"
  Participant 3: "Aborted (rolled back)"
```

3. Protocol Details (2 marks)

Participant State Machine:

```
Initial → Working (locks held)
          Vote Yes → Prepared (ready to commit)
                     Commit → Committed
                     Abort → Aborted (rollback)
          Vote No → Aborted
```

Recovery Mechanism: - Log all decisions to persistent storage - On crash: Read log, recover state - Ensures durability

4. Problems with 2PC (1.5 marks)

Blocking: - Participant waits for phase 2 decision - Resources locked during wait - If coordinator crashes: Locked indefinitely

Example:

```
Participant locked resource X
Waiting for coordinator commit decision
Coordinator crashes
Resource X stays locked → Deadlock possible
```

Scalability: - Synchronous: All must vote synchronously - Large N: Latency increases - Overhead significant

Network Partitions: - Coordinator in partition 1, participant in partition 2 - Can't communicate - Participant doesn't know to abort

5. Three-Phase Commit (1.5 marks)

Improvement: Add precommit phase

```
Phase 1: Cancommit? (vote)
Phase 2: Precommit (prepare without final decision)
Phase 3: Commit (final)

Advantage: Non-blocking
  If coordinator fails after phase 2:
   Participants can unilaterally commit
  Resources not locked indefinitely

Disadvantage: More messages, more complex
```

6. Modern Alternatives (0.5 marks) - Saga pattern: Compensating transactions - Event sourcing: Track all events - Eventual consistency: Accept temporary inconsistency

---

Q24: Distributed Data Consistency Models

Answer:

1. Consistency Overview (1 mark) Challenge: Multiple clients accessing replicated data see inconsistent state. Need model defining what "consistent" means.

2. Strong Consistency Models (2 marks)

Linearizability (Atomic Consistency): - All writes visible in single order - Every read sees latest write - Total order on all operations - Very expensive: Requires coordination

Sequential Consistency: - Writes appear in same order to all - But not absolute order relative to physical time - Less strict than linearizability

Serializability: - Concurrent operations appear atomic - Equivalent to some serial order - Common in databases

Example:

```
Process A writes X=1
Process B writes X=2

Linearizable: All see [1,2] order or [2,1] order, not mixed

Sequential: All see same order, but not tied to real time
            (might see 2 happened before 1)
```

3. Weak Consistency Models (2 marks)

Causal Consistency: - Respects causal ordering - A's write → B sees it → B's write → All see B's causally after A's

FIFO Consistency: - Within process: Writes ordered - Between processes: No guarantee

Session Consistency (Popular): - Within one user's session: Consistent - Different user might see stale - Good balance: Simple + practical

Eventual Consistency: - Eventually consistent (no time bound) - Temporarily inconsistent OK - Used in: Cloud systems, social media

4. Implementation Trade-offs (2 marks)

Strong Consistency: - Advantages: Programmer friendly, no anomalies - Disadvantages: High latency, limited scalability

Weak Consistency: - Advantages: High availability, fast responses - Disadvantages: Complex programming, anomalies possible

Example Anomaly (Eventual Consistency):

```
Alice writes email to Bob
Alice checks "sent"
Bob checks email
Bob might not see yet (eventual)

Bob waits, email appears
Acceptable for some systems (not banks)
```

5. Verification & Specification (1.5 marks)

How to verify consistency: - Formal models (TLA+) - Testing: Run scenarios - Simulation: Model checker

Specification: Define precisely - What is consistent state? - What guarantees provided? - When consistency guaranteed?

6. CAP Theorem (0.5 marks) Choose 2 of 3: - Consistency: All see same data - Availability: System responds - Partition tolerance: Survive network partition

Most distributed systems choose AP (availability + partition), accepting eventual consistency.

Q25: Distributed Monitoring & Debugging

Answer:

1. Challenges (1.5 marks)

Distributed Debugging Hard Because: - No global clock (events interleaved unpredictably) - No global state (each process has own view) - Non-determinism: Same input same execution - Network failures: Can't observe all messages - Scale: Too many events

2. Event Logging & Replay (2 marks)

Logging Approach: - Record all events at each process - Events: Send, receive, local execution - Timestamps: Lamport or Vector clocks

Replay: - Recreate execution from logs - Deterministic replay: Same execution as original - Helps reproduce bugs

Example:

```
Process A: [Event1] [sends to B] [Event2]
Process B: [receives from A] [Event3]

Logs:
A: E1, send(msg), E2
B: receive(msg), E3

Replay: Execute in same order, same interleaving
Debug: Where did execution diverge?
```

Challenges: - Logs huge (millions of events) - Replay slow - Non-determinism hard to capture

3. Tracing & Instrumentation (2 marks)

Instrumentation: - Add logging statements to code - "Entering function X" - "Sending message to Y" - "Received response"

Distributed Tracing: - Trace request through entire system - Unique ID for request - Each component logs processing - Reconstruct full path

Tools: - Zipkin: Open-source tracing - Jaeger: Distributed tracing - Application Insights: Cloud tracing

5. Monitoring & Alerting (1.5 marks)

Metrics to Monitor: - Response time: Latency percentiles - Error rate: % of failures - CPU/Memory: Resource usage - Network: Bandwidth, packet loss - Throughput: Operations/second

Alerting: - Threshold-based: Alert if metric $>$ limit - Anomaly detection: Alert on unusual patterns - Composite: Alert on combination

Example: - Alert if response time 99th percentile $>$ 100ms - Alert if error rate $>$ 1% - Alert if CPU usage $>$ 80%

4. Fault Localization (1 mark)

Finding Root Cause: - Error message: Symptom, not cause - Trace dependencies: Which system caused failure? - Correlation analysis: What events preceded error?

Techniques: - Compare logs of good vs bad runs - Statistical analysis: Unusual patterns - Causal analysis: Event dependencies

5. Tools & Frameworks (1 mark)

Logging: - ELK Stack: Elasticsearch, Logstash, Kibana - Splunk: Commercial log analysis - Fluentd: Log aggregation

Metrics: - Prometheus: Metrics collection - Graphite: Time-series storage - InfluxDB: Time-series database

6. Best Practices (0.5 marks) - Log at appropriate level (DEBUG, INFO, ERROR) - Include context (request ID, component name) - Structured logging: Parse-able format (JSON) - Retention: Balance storage vs history - Privacy: Don't log sensitive data

---

EXAM TIPS FOR 10-MARK ANSWERS

Key Points:

1. **Define first:** Start with clear definition
2. **Structure with headings:** Shows organization (+1 mark)
3. **Real examples:** Gmail, Facebook, Bitcoin (+1-2 marks)
4. **Algorithms/steps:** Clear numbered flow (+2 marks)
5. **Advantages & Disadvantages:** Critical thinking (+1 mark)
6. **Diagrams:** Simple boxes/arrows (+1-2 marks bonus)
7. **Trade-offs:** "This is fast but inconsistent…" (+1 mark)
8. **Time:** 15-20 minutes per 10-mark question

Answer Format:

- Line 1: Definition (1 mark)
- Lines 2-8: Core explanation (4-5 marks)
- Lines 9-10: Examples (2 marks)
- Lines 11-12: Comparison (1-2 marks)
- **Total: ~2 pages, 1000 words per question**

TOTAL: 25 $\times$ 10 marks = 250 marks (Full course coverage)