Midterm Paper 1 Answers (WhatsAppImage2025-03-11at10.35.07_753228

This document provides detailed answers to the questions from the first midterm paper image provided.

Question 1: Differentiate Synchronous and asynchronous executions with diagrams. (10 Marks)

Concept: Synchronous vs. Asynchronous Execution Models in Distributed Systems.

Syllabus Relevance: Module 1 (Synchronous versus asynchronous executions, Design issues and challenges).

Answer:

In distributed systems, the distinction between synchronous and asynchronous execution models is fundamental, primarily defined by assumptions about process execution speeds, message delivery times, and clock drift rates. These assumptions significantly impact the design and complexity of distributed algorithms.

Synchronous Distributed System: A synchronous distributed system is characterized by known, finite upper bounds on process execution speeds, message transmission delays, and the rate at which local clocks drift from real-time. Specifically: 1. **Bounded Message Delay:** There is a known maximum time `t_max` for any message sent between two connected processes to be received. 2. **Bounded Execution Steps:** The time taken for a process to execute a single step or instruction has a known upper bound. 3. **Bounded Clock Drift:** The rate at which any process's local clock deviates from a perfect reference clock is bounded by a known constant.

These bounds allow processes to operate in lock-step rounds. Processes can wait for a certain duration, knowing that any message sent in the previous round will have arrived. This simplifies algorithm design, particularly for tasks requiring coordination or agreement, as timeouts can reliably detect failures (a process not responding within the maximum time is considered failed). Consensus, for example, is solvable deterministically in synchronous systems even with certain types of failures.

Diagrammatic Representation (Conceptual): Imagine processes P1, P2, P3 executing in rounds. In each round `r`, a process performs computation, sends messages, and then waits for a duration `Δt` (calculated based on bounds) before starting round `r+1`. All messages sent in round `r` are guaranteed to arrive before the end of the waiting period `Δt`, allowing processes to start round `r+1` with complete information from round `r`.

```
 Time ->
P1: [Round 1 Comp/Send] --Wait Δt--> [Round 2 Comp/Send] --Wait Δt--> ...
P2: [Round 1 Comp/Send] --Wait Δt--> [Round 2 Comp/Send] --Wait Δt--> ...
P3: [Round 1 Comp/Send] --Wait Δt--> [Round 2 Comp/Send] --Wait Δt--> ...
   |<------ Round 1 ------->|<------ Round 2 ------->|
  (Messages sent in Round 1 arrive before Round 2 starts)
```

Asynchronous Distributed System: An asynchronous distributed system makes no assumptions about the relative timing of events. Specifically: 1. **Unbounded Message Delay:** Message transmission delays can be arbitrarily long, though finite (messages are eventually delivered if the recipient is correct and the network doesn't partition permanently). 2. **Unbounded Execution Steps:** The time taken for a process to execute a step is unpredictable. 3. **Unbounded Clock Drift:** Local clocks can drift at arbitrary rates relative to each other and real-time.

In this model, it is impossible to use timeouts to reliably detect process failures, as a delayed message is indistinguishable from a message sent by a crashed process. Algorithms must rely solely on message arrivals and cannot assume lock-step execution. This makes designing algorithms for problems like consensus significantly harder; for instance, the FLP impossibility result shows that deterministic consensus is impossible in a purely asynchronous system with even one potential crash failure. Algorithms often rely on message ordering guarantees (like FIFO or causal order) or failure detectors to achieve progress.

Diagrammatic Representation (Conceptual): Processes P1, P2, P3 execute independently. Messages (m1, m2) sent can experience vastly different delays. P1 might send m1 to P2, and P2 might send m2 to P3 much later, with m1 arriving after m2 is sent, or vice-versa. There are no synchronized rounds.

```
 Time ->
P1: Event -> Send(m1) -> Event -> ...
          |
P2: Event -> ... -> Receive(m1) -> Send(m2) -> ...
                          |
P3: Event -> ... -> Receive(m2) -> ...

(Message delays are arbitrary; no fixed rounds)
```

Key Differences Summarized:

Feature	Synchronous System	Asynchronous System
Message Delay	Bounded	Unbounded
Process Speed	Bounded step time	Unbounded step time
Clock Drift	Bounded	Unbounded
Failure Detection	Possible via timeouts	Impossible via timeouts
Execution Model	Lock-step rounds possible	Independent, event-driven
Algorithm Design	Simpler (e.g., consensus)	More complex (e.g., FLP)

Question 2: In the above space-time diagram of a distributed execution, global state G1 consisting of the given local states. Does the global state G1 is consistent? Justify your answer. G1 = { L1(e1^3), L2(e2^2), L3(e3^1) } (10 Marks)

Concept: Consistent Global State (Consistent Cut), Happened-Before Relation.

Syllabus Relevance: Module 1 (Global state, Cuts), Module 2 (Snapshot algorithms).

Answer:

A global state, represented as a cut across the space-time diagram of a distributed execution, is considered **consistent** if, for every message `m` delivered (received) by a process included in the cut, the corresponding send event of `m` by the source process is also included in the cut. In simpler terms, a cut is consistent if no message crosses the cut line from the future (events after the cut) to the past (events before the cut).

The given global state is G1 = { L1(e1^3), L2(e2^2), L3(e3^1) }. This corresponds to a cut C1 that passes through the state intervals immediately following events e1^3 on process P1, e2^2 on process P2, and e3^1 on process P3.

Let's analyze the messages exchanged relative to this cut C1:

1. **Message m12 (from P1 to P2):** Sent by event e1^1 (before e1^3 on P1, so included in the past of C1). Received by event e2^2 (the last event included on P2 by C1). Since both send and receive are before or at the cut, this message does not violate consistency.

2. **Message m13 (from P1 to P3):** Sent by event e1^2 (before e1^3 on P1, so included in the past of C1). Received by event e3^2 (after e3^1 on P3, so not included in the past of C1). This message crosses the cut from past to future, which is allowed and does not violate consistency.
3. **Message m21 (from P2 to P1):** Sent by event e2^1 (before e2^2 on P2, so included in the past of C1). Received by event e1^3 (the last event included on P1 by C1). Since both send and receive are before or at the cut, this message does not violate consistency.
4. **Message m31 (from P3 to P1):** Sent by event e3^1 (the last event included on P3 by C1). Received by event e1^4 (after e1^3 on P1, so not included in the past of C1). This message crosses the cut from past to future, which is allowed and does not violate consistency.
5. **Message m32 (from P3 to P2):** Sent by event e3^1 (the last event included on P3 by C1). Received by event e2^3 (after e2^2 on P2, so not included in the past of C1). This message crosses the cut from past to future, which is allowed and does not violate consistency.

Crucially, let's re-examine the diagram carefully for any message received before the cut whose send event happened after the cut. Looking at the cut defined by {e1^3, e2^2, e3^1}: * P1 receives m21 at e1^3. m21 was sent at e2^1 (before e2^2). OK. * P2 receives m12 at e2^2. m12 was sent at e1^1 (before e1^3). OK. * P3 is cut after e3^1. It has not received any messages up to this point.

All messages received up to the cut points {e1^3, e2^2, e3^1} were sent before the corresponding cut points on their respective sender processes. No message crosses the cut from the future to the past.

Justification: The global state G1 = { L1(e1^3), L2(e2^2), L3(e3^1) } represents a consistent cut of the distributed execution. This is because for every event `e` included in the state (i.e., `e` happened-before or is the cut event itself), if `e` is a receive event `Receive(m)`, then the corresponding send event `Send(m)` is also included in the state (i.e., `Send(m)` happened-before the cut event on the sender process). We have verified this for all messages received up to the cut points: m12 (sent e1^1, recv e2^2) and m21 (sent e2^1, recv e1^3). No message reception included in G1 has its corresponding send event excluded from G1.

Therefore, the global state G1 is consistent.

Question 3: Consider the following three process A-execution diagram and determine whether it obeys causal order or not with justification. (10 Marks)

Concept: Causal Order (CO) in Message Delivery.

Syllabus Relevance: Module 2 (Message ordering paradigms, Causal order (CO)).

Answer:

Causal Order (CO) is a message ordering paradigm in distributed systems, particularly relevant for group communication or multicast messages. It states that if the sending of message `m1` happened-before the sending of message `m2`, then any process `p` that delivers both messages must deliver `m1` before it delivers `m2`. The happened-before relation (`->`) is defined by Lamport: a -> b if (i) a and b are events in the same process and a occurs before b, or (ii) a is the sending of a message m and b is the reception of m, or (iii) there exists an event c such that a -> c and c -> b (transitivity).

Let's analyze the provided A-execution diagram (which seems to depict asynchronous point-to-point communication rather than multicast, but the principle of causality still applies to the order of events observed by processes):

- **Process P1:** Sends m1 (event s1), then later sends m3 (event s3).
- **Process P2:** Sends m2 (event s2).
- **Process P3:** Receives m1 (event r1), then receives m2 (event r2), then receives m3 (event r3).

We need to check if any causal relationship between send events is violated by the receive order at P3.

1. **Causality between m1 and m3:** Event s1 (send m1) happens before s3 (send m3) on process P1. Thus, s1 -> s3.

2. **Delivery Order at P3:** P3 delivers m1 (at r1) and then delivers m3 (at r3). Since s1 -> s3, CO requires that m1 be delivered before m3 at any common destination. P3 delivers m1 before m3. This order respects causality.

3. **Other Potential Causal Chains:**

    - Is there a causal path from s2 (send m2) to s1 (send m1)? No direct path.
    - Is there a causal path from s2 (send m2) to s3 (send m3)? No direct path.
    - Is there a causal path from s1 (send m1) to s2 (send m2)? No direct path.
    - Is there a causal path from s3 (send m3) to s2 (send m2)? No direct path.

Let's consider the full happened-before relation based on the diagram: * s1 -> s3 (on P1) * s1 -> r1 (message m1) * s2 -> r2 (message m2) * s3 -> r3 (message m3) * r1 -> r2 -> r3 (on P3)

From these, we can infer causal chains: * s1 -> r1 -> r2 * s1 -> r1 -> r3 * s1 -> s3 -> r3 * s2 -> r2 -> r3

Now, let's re-evaluate the CO condition specifically for messages delivered at P3: * **Messages m1 and m3:** We established s1 -> s3. P3 delivers m1 (at r1) before m3 (at r3). This respects CO. * **Messages m1 and m2:** There is no causal relationship (s1 -> s2 or s2 -> s1) shown between the send events. Therefore, CO imposes no constraint on their delivery order relative to each other. P3 delivers m1 (r1) then m2 (r2). * **Messages m2 and m3:** There is no causal relationship (s2 -> s3 or s3 -> s2) shown between the send events. Therefore, CO imposes no constraint on their delivery order relative to each other. P3 delivers m2 (r2) then m3 (r3).

Justification: The execution obeys Causal Order. The only pair of messages with a causal dependency between their send events is (m1, m3), where send(m1) happened-before send(m3). Process P3 delivers m1 before delivering m3, satisfying the requirement of Causal Order. For all other pairs of messages delivered at P3 (m1, m2) and (m2, m3), there is no causal relationship between their respective send events, so any delivery order is permissible under CO.

Therefore, the execution shown obeys Causal Order.

Question 4: Consider the A-execution diagram in question 3. Do you think message ordering is satisfied. Justify your answer. (10 Marks)

Concept: Message Ordering Paradigms (FIFO, Causal, Total).

Syllabus Relevance: Module 2 (Message ordering paradigms).

Answer:

This question asks if

"message ordering" is satisfied in the execution shown in Question 3. Message ordering isn't a single property but rather a set of different guarantees, such as FIFO (First-In, First-Out), Causal Order (CO), and Total Order (TO).

Let's evaluate the common ordering guarantees based on the diagram:

1. **FIFO Order:** This guarantees that messages sent between the same pair of processes (sender and receiver) are delivered in the order they were sent.

   - Consider the channel P1 -> P3: P1 sends m1 (event s1) and later sends m3 (event s3). Process P3 receives m1 (event r1) and later receives m3 (event r3). The delivery order (m1 then m3) matches the sending order (m1 then m3). So, FIFO order is satisfied for the P1 -> P3 channel.
   - Consider the channel P2 -> P3: P2 sends only one message, m2. FIFO is trivially satisfied.
   - No other channels have multiple messages in this diagram.
   - Therefore, FIFO order is satisfied for all relevant communication channels shown.

2. **Causal Order (CO):** As established in the answer to Question 3, if send(m1) -> send(m2), then any process delivering both must deliver m1 before m2.

   - We found that s1 -> s3. P3 delivers m1 (at r1) before m3 (at r3). This respects CO.
   - There were no other causal dependencies between send events for messages delivered at P3.
   - Therefore, Causal Order is satisfied.

3. **Total Order (TO):** This guarantees that all processes that deliver a set of messages deliver them in the same relative order. This is typically applied to multicast messages. The diagram shows point-to-point messages, and only P3 receives messages m1, m2, and m3. P3 delivers them in the sequence m1, m2, m3. Without other recipients, we cannot fully verify or falsify Total Order across multiple processes. However, there is no violation shown at the single recipient P3.

Justification: Message ordering, interpreted as either FIFO or Causal Order, is satisfied in the given A-execution diagram. * **FIFO:** Messages sent from P1 to P3 (m1, then m3) are received in the same order (r1, then r3). * **Causal:** The only causal dependency between send events (send(m1) -> send(m3)) is respected by the delivery order at P3 (deliver(m1) before deliver(m3)).

Since the fundamental ordering properties (FIFO and Causal) are met based on the diagram, we can conclude that message ordering is satisfied.

Question 5: Explain the three basic approaches for implementing distributed mutual exclusion with system model. (10 Marks)

Concept: Distributed Mutual Exclusion (DME) Approaches, System Model.

Syllabus Relevance: Module 3 (Distributed mutual exclusion algorithms, System model).

Answer:

Distributed Mutual Exclusion (DME) ensures that at most one process can access a shared resource (Critical Section - CS) at any given time in a distributed system where processes communicate only via message passing.

System Model: We typically assume a distributed system consisting of `N` autonomous processes (P1, P2, ..., PN) connected by a communication network. Key assumptions often include: * **Processes:** Processes operate asynchronously at potentially different speeds. They can fail (e.g., crash failures). * **Communication:** Processes communicate solely by exchanging messages. The network might be assumed to be reliable (messages are eventually delivered without corruption) or unreliable. Message delivery might be FIFO ordered between pairs of processes or unordered. The system might be synchronous (bounded message delays) or asynchronous (unbounded delays). * **No Shared Memory:** Processes do not share physical memory.

Basic Approaches for DME:

1. **Non-Token-Based (Permission-Based) Approach:**
    - **Idea:** A process wishing to enter the CS must request permission from some set of other processes and wait until sufficient permissions are granted. Timestamps (like Lamport's logical clocks or vector clocks) or sequence numbers are typically used to order requests and resolve conflicts fairly.
    - **Mechanism:** When Pi wants to enter the CS, it sends a REQUEST message (often timestamped) to a set of other processes (e.g., all other processes in Lamport's or Ricart-Agrawala's algorithms). A receiving process Pj grants permission (sends a REPLY) based on its own state (whether it's in the CS, requesting the CS, or idle) and the priority of the requests (e.g., lower timestamp has higher priority). Pi enters the CS only after receiving REPLY messages from the required set of processes. Upon exiting the CS, Pi sends RELEASE messages or informs deferred requesters.
    - **Examples:** Lamport's Algorithm, Ricart-Agrawala Algorithm.

   - **Diagram (Conceptual - Ricart-Agrawala):** ``` P1 (Wants CS) --REQUEST(ts1)--> P2 P1 (Wants CS) --REQUEST(ts1)--> P3

     P2 (Idle) --REPLY--> P1 P3 (Wants CS, ts3 > ts1) --REPLY--> P1

     P1 (Receives all Replies) --> Enter CS --> Exit CS --> Send deferred Replies ```

2. **Token-Based Approach:**

   - **Idea:** A unique token exists in the system. Only the process currently holding the token is allowed to enter the CS. Mutual exclusion is guaranteed because only one process can possess the token at a time.
   - **Mechanism:** Processes are typically organized in a logical structure (e.g., a ring or a tree). A process wanting to enter the CS requests the token. If it doesn't have the token, the request is forwarded along the logical structure towards the current token holder. When the token holder exits the CS (or if it wasn't using it), it passes the token to the next requesting process according to the algorithm's rules (e.g., along the ring, down the tree, or directly in broadcast-based algorithms).
   - **Examples:** Suzuki-Kasami's Broadcast Algorithm, Raymond's Tree-Based Algorithm.

   - **Diagram (Conceptual - Ring):** ``` P1 --> P2 --> P3 --> P4 --> P1 (Logical Ring) (Token initially at P1)

     P3 (Wants CS) --> Sends Request towards P1 P1 (Exits CS/Idle) --> Sends Token --> P2 --> P3 P3 (Receives Token) --> Enter CS --> Exit CS --> Sends Token to next requester/neighbor ```

3. **Quorum-Based Approach:**

   - **Idea:** A process wishing to enter the CS requests permission from only a subset of processes, called a quorum (or request set). These quorums are carefully constructed such that any two quorums in the system have at least one process in common (non-empty intersection property).
   - **Mechanism:** When Pi wants to enter the CS, it sends REQUEST messages only to the processes in its quorum (Ri). A process Pj in Ri grants permission (sends REPLY) only if it hasn't granted permission to another process already (it locks itself for Pi). Pi enters the CS only after receiving permission from all processes in its quorum Ri. The intersection property ensures that no two processes can get permission from their respective quorums simultaneously, because the common member(s) will grant permission to only one of them at

a time. Upon exiting, Pi sends RELEASE messages to its quorum members, allowing them to grant permission to others.

- **Example:** Maekawa's Algorithm (uses finite projective planes to construct quorums of size approx. sqrt(N)).

- **Diagram (Conceptual - Maekawa):** ``` N=7 processes. Quorums Ri, Rj (size ~sqrt(7) ~ 3). Intersection: Ri ∩ Rj ≠ ∅ (e.g., Pk is in both)

  P1 (Wants CS) --REQUEST--> Quorum R1 = {P1, P2, P4} P5 (Wants CS) --REQUEST--> Quorum R5 = {P2, P5, P6}

  P1 gets REPLY from P1, P4. P5 gets REPLY from P5, P6.

  P2 (Common member) receives requests from P1 and P5. P2 grants permission (REPLY) to only one (e.g., P1 based on timestamp). P2 defers REPLY to P5.

  P1 receives all replies --> Enter CS. P5 waits for reply from P2.

  P1 (Exits CS) --RELEASE--> R1 = {P1, P2, P4} P2 (Receives RELEASE from P1) --> Sends REPLY to P5. P5 (Receives reply from P2) --> Enter CS. ```

Each approach has different trade-offs regarding message complexity, synchronization delay, fault tolerance, and ease of implementation.