

Distributed Systems

ECE428

Lecture 11

Adopted from Spring 2021

Today's agenda

- Wrap up Mutual Exclusion
 - Extending Maekawa's algorithm to break deadlocks.

Mutual exclusion in distributed systems

- Classical algorithms for mutual exclusion in distributed systems.
 - Central server algorithm
 - Ring-based algorithm
 - Ricart-Agrawala Algorithm
 - Maekawa Algorithm

Maekawa Algorithm: Actions

- state = Released, voted = false
- enter() at process P_i :
 - state = Wanted
 - Multicast **Request** message to all processes in V_i
 - Wait for **Reply (vote)** messages from all processes in V_i (including vote from self)
 - state = Held
- exit() at process P_i :
 - state = Released
 - Multicast **Release** to all processes in V_i

Maekawa Algorithm: Actions (contd.)

- When P_i receives a Request from P_j :
 - if (state == Held OR voted = true)
 - queue Request
 - else
 - send Reply to P_j and set voted = true
- When P_i receives a Release from P_j :
 - if (queue empty)
 - voted = false
 - else
 - dequeue head of queue, say P_k
 - Send Reply only to P_k
 - voted = true

Analysis: Maekawa Algorithm

- **Safety:**

- When a process P_i receives replies from all its voting set V_i members, no other process P_j could have received replies from all its voting set members V_j .

- **Liveness**

- Not satisfied. Can have deadlock!

- **Ordering:**

- Not satisfied.

Breaking deadlocks

- Maekawa algorithm can be extended to break deadlocks.
- Compare Lamport timestamps before replying (like Ricart-Agrawala).
- But is that enough?
 - *System of 6 processes $\{0, 1, 2, 3, 4, 5\}$. 0, 1, 2 want to enter critical section:*
 - $V_0 = \{0, 1, 2\}$: 0, 2 send **reply** to 0, but 1 sends **reply** to 1;
 - $V_1 = \{1, 3, 5\}$: 1, 3 send **reply** to 1, but 5 sends **reply** to 2;
 - $V_2 = \{2, 4, 5\}$: 4, 5 send **reply** to 2, but 2 sends **reply** to 0;
 - Suppose $(L1, P1) < (L0, P0) < (L2, P2)$.
 - *Deadlock can still happen based on when messages are received.*
 - P5 receives P2's request before P1's, and replies back to P2 first.
- *We need a way to take back the reply.*

Breaking deadlocks

- Say P_i 's request has a smaller timestamp than P_j .
- If P_k receives P_j 's request after replying to P_i , send **fail** to P_j .
- If P_x receives P_i 's request after replying to P_j , send **inquire** to P_j .
- If P_j receives an **inquire** and at least one **fail**, it sends a **relinquish** to release locks, and deadlock breaks.

Breaking deadlocks

- *System of 6 processes $\{0, 1, 2, 3, 4, 5\}$. 0, 1, 2 want to enter critical section:*
 - $V_0 = \{0, 1, 2\}$: 0, 2 send **reply** to 0, but 1 sends **reply** to 1;
 - $V_1 = \{1, 3, 5\}$: 1, 3 send **reply** to 1, but 5 sends **reply** to 2;
 - $V_2 = \{2, 4, 5\}$: 4, 5 send **reply** to 2, but 2 sends **reply** to 0;
- Suppose $(L1, P1) < (L0, P0) < (L2, P2)$.
- P2 will send **fail** to itself when it receives its own request after P0.
- P5 will send **inquire** to P2 when it receives P1's request.
- P2 will send **relinquish** to V_2 . P5 and P4 will set "voted = false". P5 will reply to P1.
- P1 can now enter CS, followed by P0, and then P2.

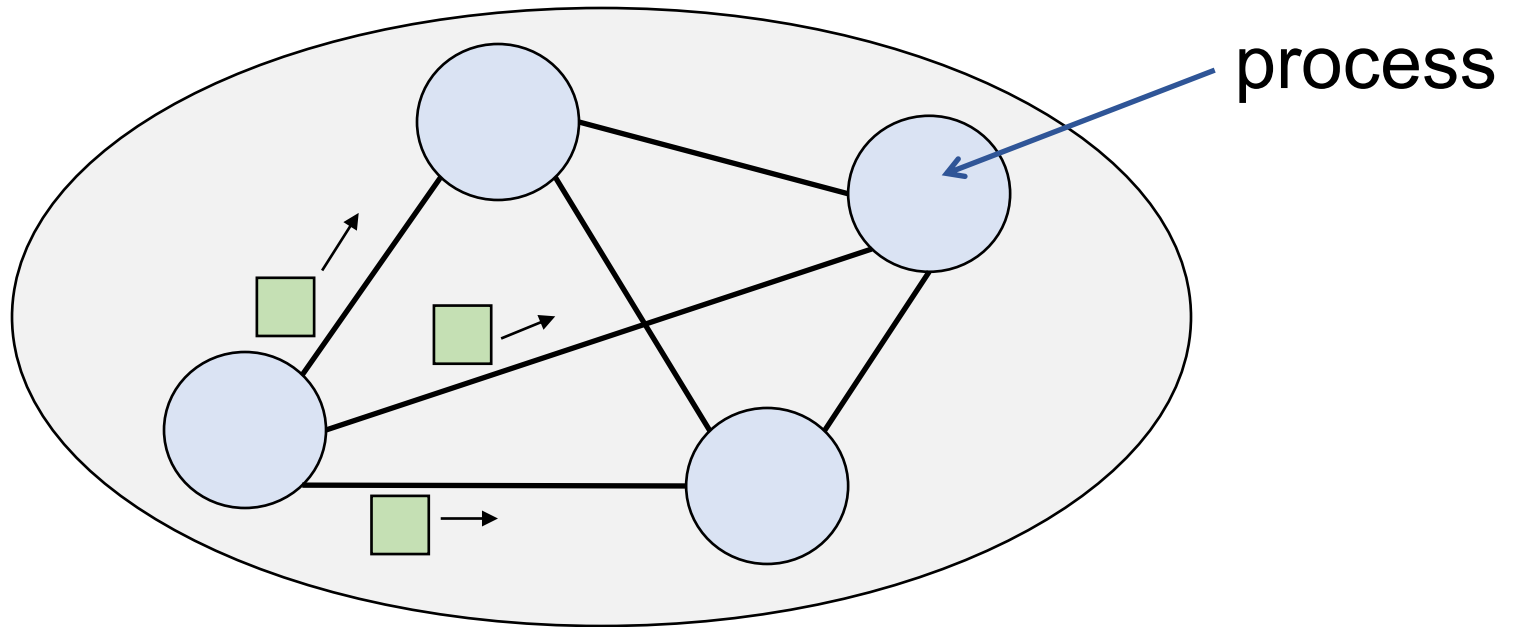
Mutual exclusion in distributed systems

- Classical mutual exclusion in distributed systems
 - Central server algorithm
 - Satisfies safety, liveness, but not ordering.
 - $O(1)$ bandwidth, and $O(1)$ client and synchronization delay.
 - Central server is scalability bottleneck
 - Ring-based algorithm
 - Satisfies safety, liveness, but not ordering.
 - Always uses bandwidth, $O(N)$ client & synchronization delay
 - Ricart-Agrawala algorithm
 - Satisfies safety, liveness, and ordering.
 - $O(N)$ bandwidth, $O(1)$ client and synchronization delay
 - Maekawa algorithm
 - Satisfies safety, but not liveness and ordering.
 - $O(\sqrt{N})$ bandwidth, $O(1)$ client and synchronization delay

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

What is a distributed system?



Independent components that are connected by a network and communicate by passing messages to achieve a common goal, appearing as a single coherent system.

Relationship between processes

- Two broad categories:
 - Client-server:
 - different roles/responsibilities.
 - Peer-to-peer:
 - similar role/responsibility.
 - run the same program/algorithm.

Key aspects of a distributed system

- Processes must communicate with one another to coordinate actions.
 - Communication channel between each pair of processes.
 - Time taken to transmit a message over a communication channel may vary.
- Different processes (on different computers) have different clocks.
 - These clocks *drift* from real time at different rates.
- Processes and communication channels may fail.

Two ways to model

- Synchronous distributed systems:
 - Known upper and lower bounds on time taken by each step in a process.
 - Known bounds on message passing delays.
 - Known bounds on clock drift rates.
- Asynchronous distributed systems:
 - No bounds on process execution speeds.
 - No bounds on message passing delays.
 - No bounds on clock drift rates.

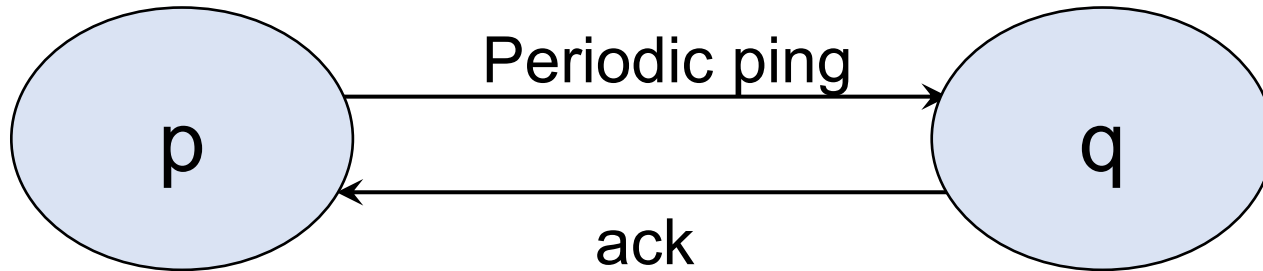
Types of failure

- Omission: when a process or a channel fails to perform actions that it is supposed to do.
 - Process may crash.
 - Fail-stop: if other processes can detect that the process has crashed.
 - Communication omission: a message sent by process was not received by another.
- Arbitrary (Byzantine) Failures: any type of error, e.g. process executing incorrectly, sending a wrong message, etc.
- Timing Failures: Timing guarantees are not met.
 - Applicable only in synchronous systems.

Topics for first midterm

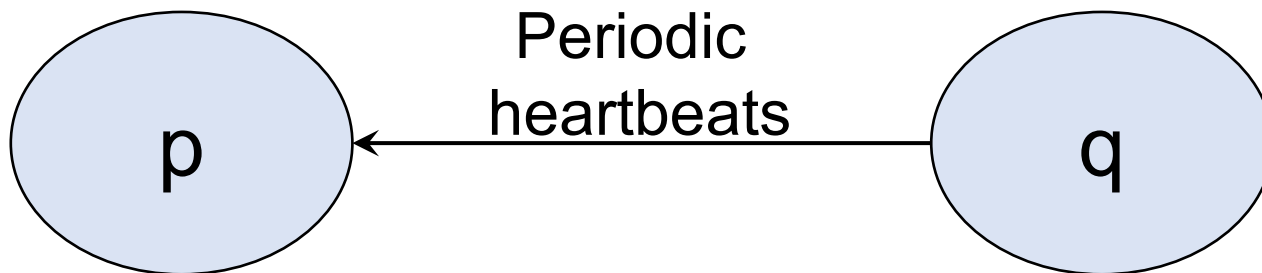
- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

How to detect a crashed process?



p sends pings to q every T seconds ($T = \text{period}$).

If p doesn't receive an ack after sending a ping within a specified **timeout**, declare q has failed.



q sends heartbeats to p every T seconds ($T = \text{period}$).

If p doesn't receive a heartbeat from q for a specified **timeout**, declare q has failed.

Computing timeout values

- Can precisely compute timeout value in synchronous systems.
 - In the worst case, how long would take to receive an ack after sending a ping?
 - In the worst case, what is the maximum time gap between two consecutive heartbeats?
- Can estimate timeout value based on observed round-trip times in asynchronous systems.

Metrics for evaluating failure detector

- Correctness:
 - Completeness: Every failed process is *eventually* detected.
 - Accuracy: Every detected failure corresponds to a crashed process (no mistakes).
- Performance:
 - Worst-case failure detection time: maximum time gap between when a failure occurs to when it is detected.
 - Bandwidth usage: No. of messages exchanged for failure detection per unit time.

Extending to N processes

- Centralized heartbeat
 - All processes send heartbeats to a central server.
- Ring-based failure detector
 - A process sends heartbeats to its ring successor.
- All-to-all failure detector
 - All processes send heartbeats to each-other.

Trade-off in completeness and bandwidth usage.

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

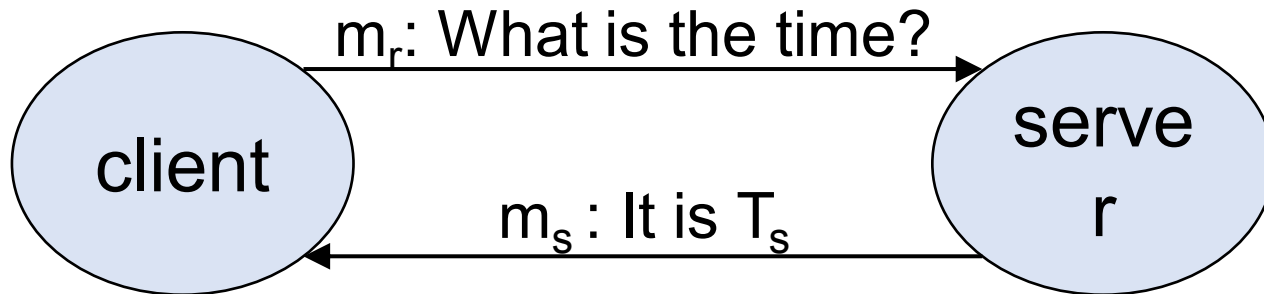
Clock Skew and Drift Rates

- Each process has an internal **clock**.
- Clocks between processes on different computers differ:
 - Clock **skew**:
 - relative difference between two clock values.
 - Clock **drift rate**:
 - change in skew from a perfect reference clock per unit time (measured by the reference clock).

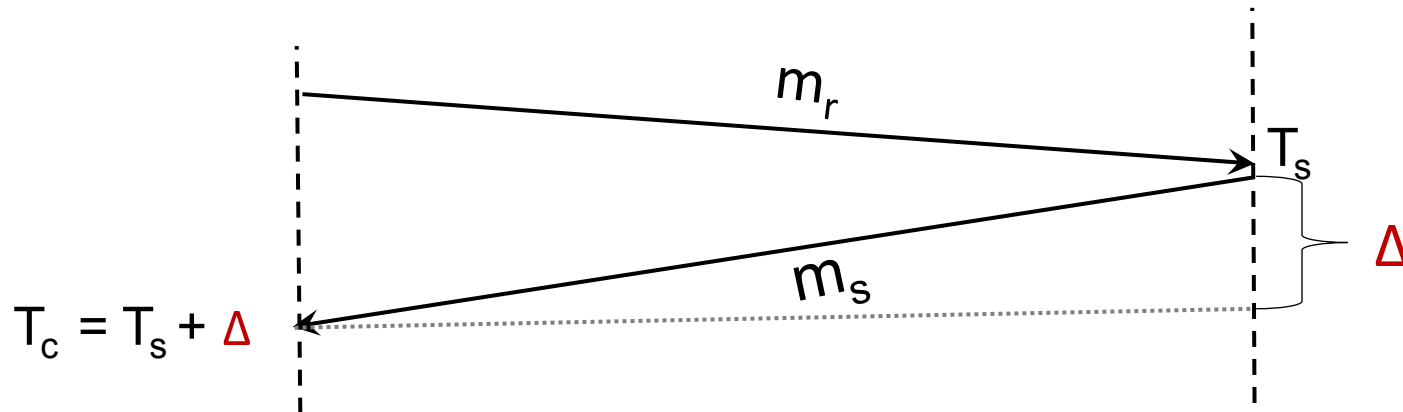
Clock synchronization

- External synchronization
 - Synchronize time with an authoritative clock.
- Internal synchronization
 - Synchronize time internally between all processes in a distributed system.
- Synchronization bound (D) between two clocks A and B over a real time interval I .
 - $|A(t) - B(t)| < D$, for all t in the real time interval I .
 - $\text{Skew}(A, B) < D$ during the time interval I .
- *Important metric: worst-case skew right after synchronization.*

Clock Synchronization



What time T_c should client adjust its local clock to after receiving m_s ?



But the value of Δ is unknown.

Clock synchronization

- In a synchronous system:
 - use known maximum and minimum network delays to find the Δ value that results in smallest worst-case skew.
- In asynchronous system:
 - Use observed round-trip time (RTT).
 - Cristian algorithm: Estimates Δ as $RTT/2$.
 - What is the worst-case skew?

Other clock synchronization protocols

- Berkeley algorithm for internal synchronization.
 - Central server collects and estimates local timestamps, computes updated time as average of estimated local times, and disseminates offsets from updated time.
- Network Time Protocol:
 - External time synchronization service over the Internet.
 - Symmetric mode synchronization:
 - Two servers exchange a pair of messages (A to B and B to A)
 - Estimate offset and accuracy bound using the send and receive timestamps at A and B for both messages.

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by \rightarrow .
 - $e \rightarrow e'$ means e *happened before* e' .
 - $e \rightarrow_i e'$ means e *happened before* e' , as observed by p_i .
- HB rules:
 - If $\exists p_i$, $e \rightarrow_i e'$ then $e \rightarrow e'$.
 - For any message m , $\text{send}(m) \rightarrow \text{receive}(m)$
 - If $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$
- Also called “*causal*” relationship.

Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.
- Each process maintains a single integer clock to logically timestamp each event.
- Checkout algorithm to assign Lamport timestamps.
- If $e \rightarrow e'$ then $L(e) < L(e')$.
- What can we conclude if $L(e) < L(e')$?

Vector Clocks

- Each process maintains vector of clocks V_i
 - $V_i[j]$ is the clock for process p_j
- Checkout algorithm to assign vector timestamps.
- Let $V(e) = V$ and $V(e') = V'$
 - $V = V'$, iff $V[i] = V'[i]$, for all $i = 1, \dots, n$
 - $V \leq V'$, iff $V[i] \leq V'[i]$, for all $i = 1, \dots, n$
 - $V < V'$, iff $V \leq V' \ \& \ V \neq V'$
iff $V \leq V' \ \& \ \exists j \text{ such that } (V[j] < V'[j])$
- $e \rightarrow e'$ iff $V < V'$
 - $(e \rightarrow e' \text{ implies } V < V')$ and $(V < V' \text{ implies } e \rightarrow e')$
- $e \parallel e'$ iff $(V \not\leq V' \text{ and } V' \not\leq V)$

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

Global snapshot

- State of each process (and each channel) in the system at a given instant of time.
- Difficult to capture a global snapshot of the system.
 - Requires precise clock synchronization across processes.
- *How do we capture global snapshots without precise time synchronization across processes?*
 - Relax the requirement for capturing the state of different processes and channels at the same real time instant.
 - As long as the global state is *consistent*, it is still useful in reasoning about properties of the system.

Notations and Definitions

- For a process p_i , where events e_i^0, e_i^1, \dots occur:

history(p_i) = $h_i = \langle e_i^0, e_i^1, \dots \rangle$

prefix history(p_i^k) = $h_i^k = \langle e_i^0, e_i^1, \dots, e_i^k \rangle$

s_i^k : p_i 's state immediately after k^{th} event.

- For a set of processes $\langle p_1, p_2, p_3, \dots, p_n \rangle$:

global history: $H = \cup_i (h_i)$

a **cut** $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_n^{c_n}$

the **frontier** of $C = \{e_i^{c_i}, i = 1, 2, \dots, n\}$

global state S that corresponds to cut $C = \cup_i (s_i^{c_i})$

Notations and definitions

- A cut C is **consistent** if and only if
$$\forall e \in C \text{ (if } f \rightarrow e \text{ then } f \in C)$$
- A global state S is consistent if and only if it corresponds to a consistent cut.

Notations and definitions

- A **run** is a total ordering of events in H that is consistent with each h_i 's ordering.
- A **linearization** is a run consistent with happens-before (\rightarrow) relation in H .
- Linearizations pass through consistent global states.
- **Execution lattice**: a way to reason about linearizations and the set of all consistent global states.

Chandy-Lamport Algorithm

- Records a consisted global snapshot
 - identifies a consistent cut.
- Key system assumptions:
 - Two uni-directional communication channels between each ordered process pair : p_j to p_i and p_i to p_j .
 - *Communication channels are FIFO-ordered (first in first out).*
 - No failures (messages are not dropped, process doesn't crash).
- Checkout the algorithm!

Chandy-Lamport Algorithm

- Records a consisted global snapshot
 - identifies a consistent cut.
- Key system assumptions:
 - Two uni-directional communication channels between each ordered process pair : p_j to p_i and p_i to p_j .
 - *Communication channels are FIFO-ordered (first in first out).*
 - No failures (messages are not dropped, process doesn't crash).
- Useful for reasoning about system *properties*.

Liveness

- **Liveness** = guarantee that something **good** will happen, **eventually**
- Examples:
 - A distributed computation will terminate.
 - “Completeness” in failure detectors: the failure will be detected.
 - All processes will eventually decide on a value.
- A global state S_0 satisfies a liveness property P iff:
 - $\text{liveness}(P(S_0)) \equiv \forall L \in \text{linearizations from } S_0, L \text{ passes through a } S_L \text{ \& } P(S_L) = \text{true}$
 - For any linearization starting from S_0 , P is true for **some** state S_L reachable from S_0 .

Safety

- **Safety** = guarantee that something **bad** will **never** happen.
- Examples:
 - There is no deadlock in a distributed transaction system.
 - “Accuracy” in failure detectors: an alive process is not detected as failed.
 - No two processes decide on different values.
- A global state S_0 satisfies a safety property P iff:
 - $\text{safety}(P(S_0)) \equiv \forall S \text{ reachable from } S_0, P(S) = \text{true}.$
 - For **all** states S reachable from S_0 , $P(S)$ is true.

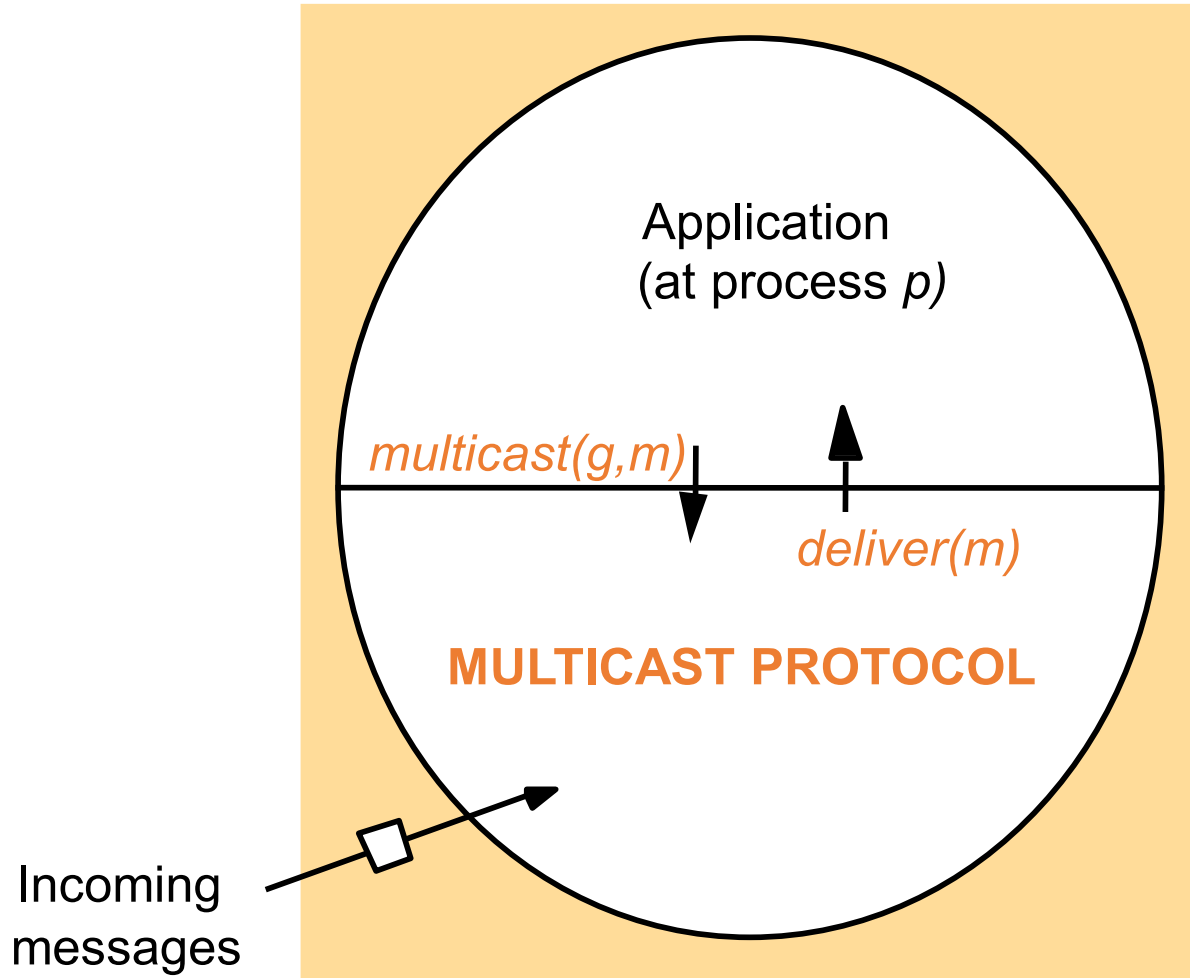
Stable Global Predicates

- once true, stays true forever afterwards (for stable liveness)
 - True for a state S , true for all states reachable from S .
- once false, stays false forever afterwards (for stable non-safety)
 - False for a state S , false for all states reachable from S .
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- **Multicast**

Multicast Protocol



Distinction between
when a message
arrives at process
 p 's node
vs

when the message
is **delivered** to the
application at p .

It is the message
delivery that
matters!

Basic Multicast (B-Multicast)

- Straightforward way to implement B-multicast:
 - use a reliable one-to-one send (unicast) operation:
B-multicast(group g, message m):
 for each process p in g, send (p,m).
receive(m): B-deliver(m) at p.
- Guarantees: message is eventually delivered to the group if:
 - Processes are non-faulty.
 - The unicast “send” is reliable.
 - *Sender does not crash.*
- *Can we provide reliable delivery even after sender crashes?*

Reliable Multicast (R-Multicast)

- Integrity: A *correct* (i.e., non-faulty) process *p* delivers a message *m* at most once.
 - *Assumption: no process sends the same message twice*
- Validity: If a *correct* process multicasts (sends) message *m*, then it will *eventually* deliver *m* itself.
 - *Liveness for the sender.*
- Agreement: If a *correct* process delivers message *m*, then all other *correct* processes in group(*m*) will *eventually* deliver *m*.
 - *All or nothing.*
- Validity and agreement together ensure overall liveness: if some correct process multicasts a message *m*, then, all correct processes deliver *m* too.

Implementing R-Multicast

On initialization

`Received := {};`

For process p to R-multicast message m to group g

`B-multicast(g,m);` ($p \in g$ is included as destination)

On B-deliver(m) at process q with $g = \text{group}(m)$

`if ($m \notin \text{Received}$):`

`Received := Received \cup { m };`

`if ($q \neq p$): B-multicast(g,m);`

`R-deliver(m)`

Ordered Multicast

- FIFO ordering: If a correct process issues $\text{multicast}(g, m)$ and then $\text{multicast}(g, m')$, then every correct process that delivers m' will have already delivered m
- Causal ordering: If $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$ then any correct process that delivers m' will have already delivered m .
 - Note that \rightarrow counts messages multicast delivered to the application, rather than all network messages.
- Total ordering: If a correct process delivers message m before m' , then any other correct process that delivers m' will have already delivered m .

HB Relationship for Causal Ordering

- HB rules in causal ordered multicast:
 - If $\exists p_i, e \rightarrow_i e'$ then $e \rightarrow e'$.
 - If $\exists p_i, \text{multicast}(g,m) \rightarrow_i \text{multicast}(g,m')$, then $\text{multicast}(g,m) \rightarrow \text{multicast}(g,m')$
 - If $\exists p_i, \text{delivery}(m) \rightarrow_i \text{multicast}(g,m')$, then $\text{delivery}(m) \rightarrow \text{multicast}(g,m')$
 - For any message m , $\text{send}(m) \rightarrow \text{receive}(m)$
 - For any *multicast* message m , $\text{multicast}(g,m) \rightarrow \text{delivery}(m)$
 - If $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$
 - $\text{multicast}(g,m) \rightarrow \text{delivery}(m)$
 - $\text{delivery}(m) \rightarrow_i \text{multicast}(g,m')$
 - $\text{multicast}(g,m) \rightarrow \text{multicast}(g,m')$
- *Application can only see when messages are sent (multicast) and delivered, not when they are received at the protocol.*

Implementing Ordered Multicast

- Basic idea:
 - Sequence number (or vector, in case of causal-ordered multicast) associated with each multicast message.
 - Multicast protocol buffers the message until the conditions for the next expected sequence number/vector are satisfied.
- Two ways to implement total-ordered multicast:
 - Central server based algorithm
 - Decentralized ISIS algorithm
- Checkout algorithms to implement FIFO, Causal, and Total ordered multicasts.

Underlying multicast mechanisms

- Unicast to each process in the group.
- Tree-based multicast.
 - Construct a minimum spanning tree of processes and unicast along the tree.
- Gossip
 - Each process sends a message to 'b' random processes.

Topics for first midterm

- System model and Failures
- Failure Detection
- Clock Synchronization
- Event ordering and Logical Timestamps
- Global Snapshot
- Multicast

Good luck!