Distributed Systems

CS425/ECE428

02/28/2020

Today's agenda

- Review of relevant concepts for first midterm.
- Not meant to be an exhaustive review!
- Go over the slides for each class.
  - Refer to lecture videos and textbook to fill in gaps in understanding.

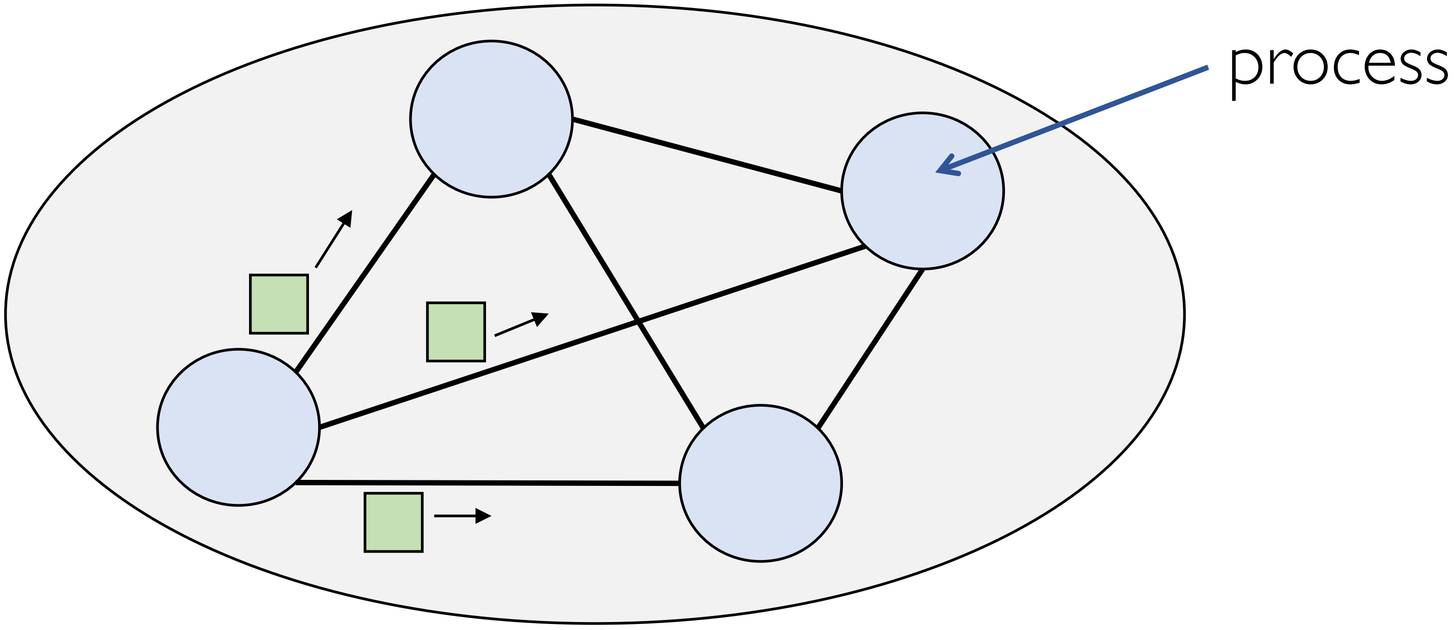
Topics for first midterm

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

Topics for first midterm

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

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.

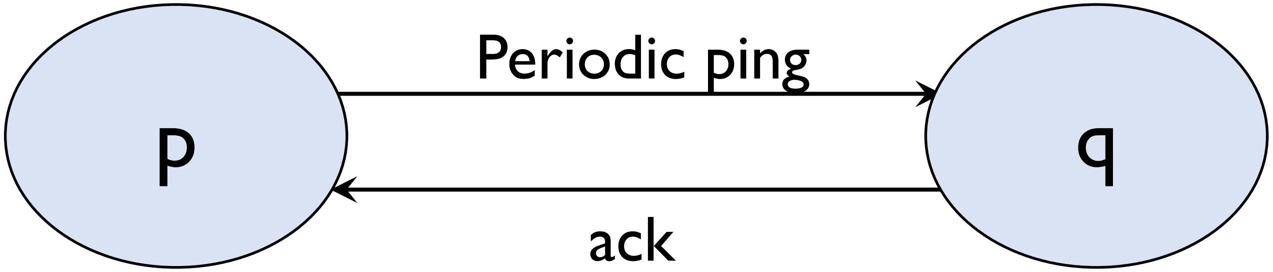
Types of failure

- **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. a 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
- Mutual Exclusion

How to detect a crashed process?



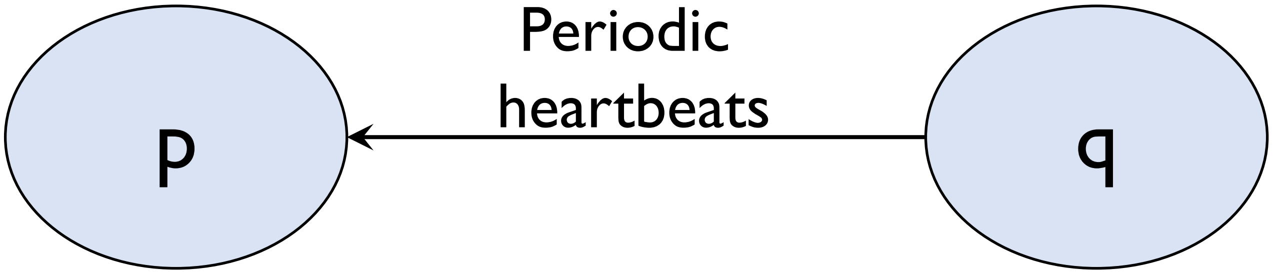
p

Periodic ping

q

ack

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



p

Periodic heartbeats

q

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

- **Completeness:** Every failed process is *eventually* detected.
- **Accuracy:** Every detected failure corresponds to a crashed process (no mistakes).
- Can we achieve completeness and accuracy in synchronous systems?
- What about asynchronous systems?

Metrics for evaluating failure detector

- **Completeness:** Every failed process is *eventually* detected.
- **Accuracy:** Every detected failure corresponds to a crashed process (no mistakes).
- What are the performance metrics?

Metrics for evaluating failure detector

- **Completeness:** Every failed process is *eventually* detected.
- **Accuracy:** Every detected failure corresponds to a crashed process (no mistakes).
- **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 a system of 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
- Mutual Exclusion

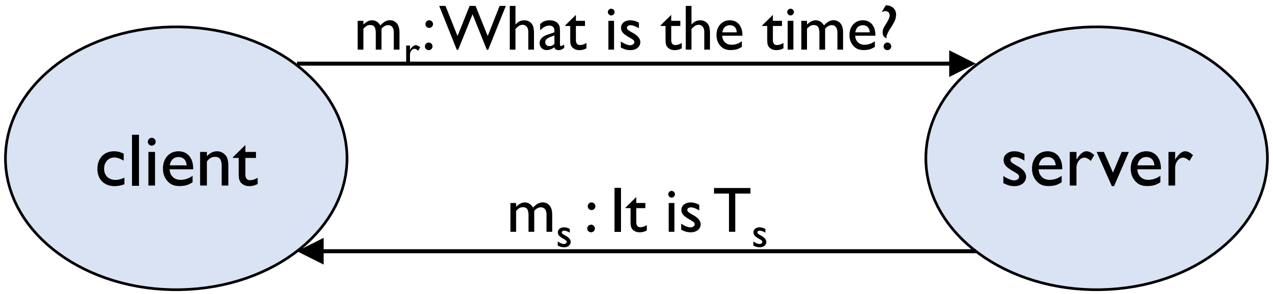
Clock Skew and Drift Rates

- 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.
  - Accuracy bound for external synchronization.

Clock Synchronization



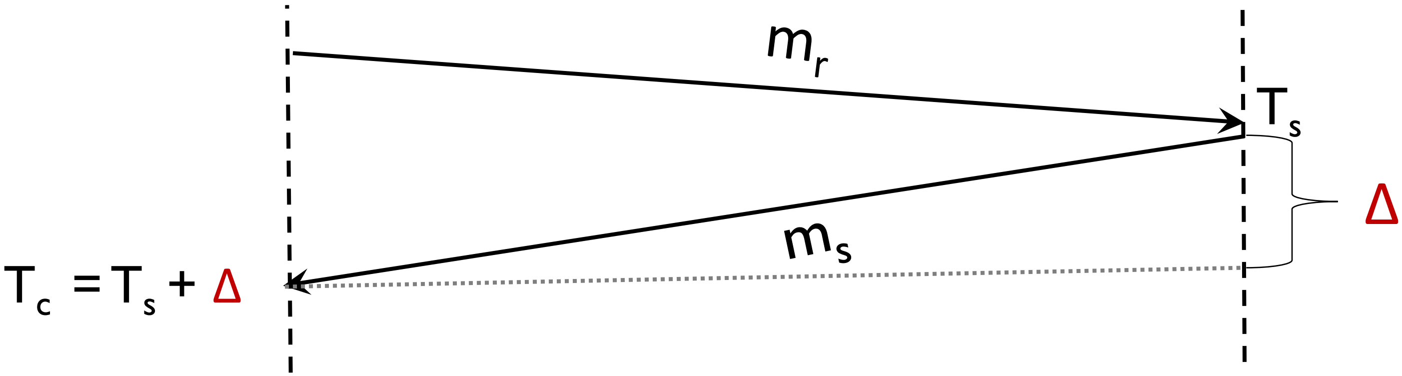
m_r: What is the time?

client

server

m_s : It is T_s

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



m_r

T_s

Δ

m_s

$T_c = T_s + \Delta$

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
- Mutual Exclusion

Happened-Before Relationship

- *Happened-before* (HB) relationship denoted by →.
  - **e** → **e'** means **e** *happened before* **e'**.
  - **e** $\rightarrow_i$ **e'** means **e** *happened before* **e'**, as observed by $\mathbf{p_i}$.

- HB rules:
  - If ∃ $\mathbf{p_i}$, **e** $\rightarrow_i$ **e'** then **e** → **e'**.
  - For any message m, **send(m)** → **receive(m)**
  - If **e** → **e'** and **e'** → **e''** then **e** → **e''**

- Also called *"potentially causal"* or "*causal*" ordering.

Lamport's Logical Clock

- Logical timestamp for each event that captures the *happened-before* relationship.

- *Algorithm:* Each process $\mathbf{p_i}$
  1. initializes local clock $\mathbf{L_i = 0}$.
  2. increments $\mathbf{L_i}$ before timestamping each event.
  3. piggybacks $\mathbf{L_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{L_i = max(t, L_i)}$
     - increments $\mathbf{L_i}$ (as per point 2).

- If $\mathbf{e \rightarrow e'}$ then $\mathbf{L(e) < L(e')}$.
- What can we conclude if $\mathbf{L(e) < L(e')}$?

Vector Clocks

- Each event associated with a vector timestamp.
- Each process maintains vector of clocks $\mathbf{V_i}$
  - $\mathbf{V_i[j]}$ is the clock for process $\mathbf{p_j}$

- Algorithm: each process $\mathbf{p_i}$:
  1. initializes local clock $\mathbf{V_i[j] = 0}$
  2. increments $\mathbf{V_i[i]}$ before timestamping each event.
  3. piggybacks $\mathbf{V_i}$ when sending a message.
  4. upon receiving a message with clock value $\mathbf{t}$
     - sets $\mathbf{V_i[j] = max(V_i[j], t[j])}$ for all $j=1\ldots n$.
     - increments $\mathbf{V_i[i]}$ (as per point 2).

Comparing 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 $V \leq V'$ & $\exists\, j$ such that $(V[j] < V'[j])$

- $e \rightarrow e'$ iff $V < V'$
  - ($V < V'$ implies $e \rightarrow e'$) and ($e \rightarrow e'$ implies $V < V'$)
- $e \parallel e'$ iff ($V \nless V'$ and $V' \nless V$)

Topics for first midterm

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

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 $e_i^0, e_i^1, \ldots$ occur:

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

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

  s_i^k : p_i's state immediately after kth event.

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

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

  a **cut** $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_3}$

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

  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

$$\forall \mathbf{e} \in \mathbf{C} \ (\text{if } \mathbf{f} \rightarrow \mathbf{e} \text{ then } \mathbf{f} \in \mathbf{C})$$

- 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 $\mathbf{h}_i$'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).

Chandy-Lamport Algorithm

- Initiating process records its state and sends a marker to all other processes.

- When a process receives a marker, its records its state and sends a marker to all other processes.

- Channel state recorded by the receiving process:
  - set of messages received from the channel between when the process records its state to when it receives a marker on that channel.

- Algorithm terminates when each process receives a marker from all other processes.

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:
  - Guarantee that a distributed computation will terminate.
  - "Completeness" in failure detectors.
  - All processes eventually decide on a value.
- A global state S_0 satisfies a **liveness** property P iff:
  - liveness($P(S_0)$) $\equiv \forall L \in$ linearizations from S_0, L passes through a S_L & $P(S_L)$ = true
  - For any linearization starting from S_0, P(s) is true for some state S_L reachable from S_0.
  - For any linearization starting from S_0, (not P(S)) is false 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.
    - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
    - safety(P(S_0)) ≡ $\forall$S reachable from S_0, P(S) = true.
    - For all states S reachable from S_0, P(S) is true.
    - For all states S reachable from S_0, (not P(S)) is false.

Stable Global Predicates

- Stable = once true, stays true forever afterwards.

- Stable liveness examples
  - Computation has terminated.

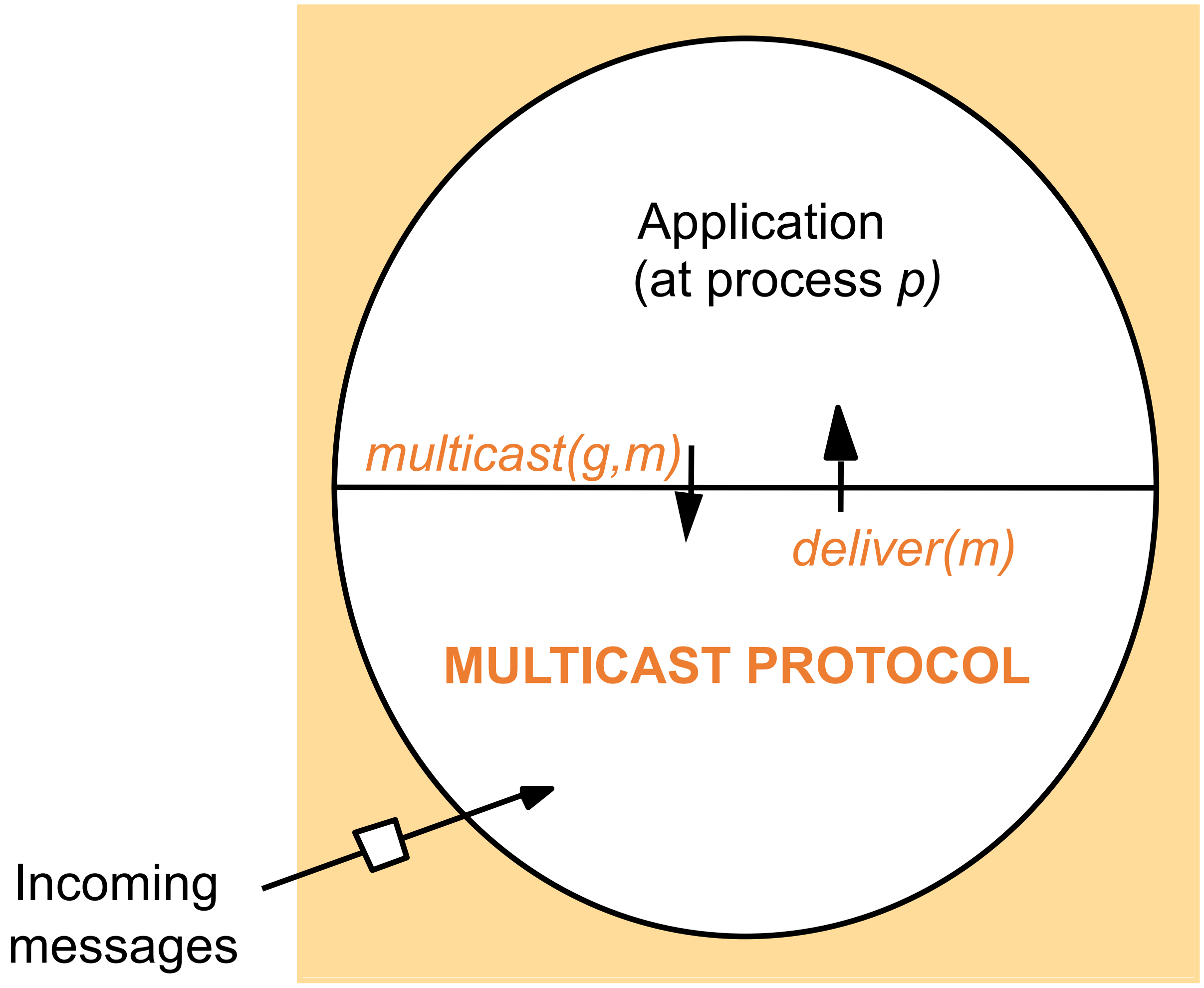
- Stable non-safety examples
  - There is a deadlock.

- *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**
- Mutual Exclusion

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 **exactly** 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 the 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 ∈ g is included as destination)
On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received):
        Received := Received ∪ {m};
        if (q ≠ p): B-multicast(g,m);
        R-deliver(m)
```

Ordered Multicast

- **FIFO ordering:** If a correct process issues multicast(*g*,*m*) and then multicast(*g*,*m'*), then every correct process that delivers *m'* will have already delivered m.

- **Causal ordering:** If multicast(*g*,*m*) → multicast(*g*,*m'*) then any correct process that delivers *m'* will have already delivered *m*.
  - Note that → counts messages **delivered** to the application, rather than all network messages.

- **Total ordering**: If a correct process delivers message *m* before *m'* (independent of the senders), then any other correct process that delivers *m'* will have already delivered *m*.

Implementing FIFO order multicast

- Each process maintains a per-process sequence number
  - Processes P*1* through P*N*
  - P*i* maintains a vector of sequence numbers S*i*[1…N] (initially all zeroes)
  - Si[i], is the no. of messages Pi multicast (and delivered to itself).
  - Si[j] is the latest sequence number P*i* has received from P*j.*
- Pi sends value Si[i] along with its multicast message.
- Receiving process Pj delivers Pi's message only if its sequence number is the next expected value (Sj[i] + 1) and increments Sj[i].
  - Otherwise buffer it until the condition is satisfied.

Implementing causal order multicast

- Each process maintains a per-process sequence number
  - Processes P_1 through P_N
  - P_i maintains a vector of sequence numbers $S_i[1 \ldots N]$ (initially all zeroes)
  - $S_i[i]$, is the no. of messages P_i multicast (and delivered to itself).
  - $S_i[j]$ is the latest sequence number P_i has received from P_j.
- P_i sends the entire vector S_i along with its multicast message.
- Receiving process P_j delivers P_i's message (with sequence vector S) if:
    - It the next expected value ($S[i] = S_j[i] + 1$)
    - For all $k \neq i$: $S[k] \leq S_i[k]$

  It then sets S_j to S.
  - Otherwise buffer it until the condition is satisfied.

Implementing total order multicast

- Central sequencer-based approach:
  - Sequencer maintains a global (total) sequence number counter.
  - Each process multicasts a message to the group and the sequencer.
  - Sequencer assigns a sequence number to the received message, multicasts this sequence number (and message id) to other processes in the group, and increments its sequence number counter.
  - A process waits for the sequencer to send the sequence number of a message before delivering it, and delivers messages in the order of their sequence numbers.

Implementing total order multicast

- ISIS algorithm:
  - Sender multicasts message to everyone.
  - Receiving processes:
    - reply with *proposed* priority (sequence no.)
      - larger than all observed *agreed* priorities
      - larger than any previously proposed (by self) priority
    - store message in *priority queue*
      - ordered by priority (proposed or agreed)
    - mark message as undeliverable
  - Sender chooses *agreed* priority, re-multicasts message with agreed priority
    - maximum of all proposed priorities
  - Upon receiving agreed (final) priority
    - reorder messages based on final priority.
    - mark the message as deliverable.
    - deliver any deliverable messages at front of priority queue.

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
- **Mutual Exclusion**

Problem Statement for mutual exclusion

- ***Critical Section* Problem:**
  - Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.

- Each process can call three functions
  - enter() to enter the critical section (CS)
  - AccessResource() to run the critical section code
  - exit() to exit the critical section

Mutual Exclusion Requirements

- Need to guarantee 3 properties:
  - Safety (essential):
    - At most one process executes in CS (Critical Section) at any time.
  - Liveness (essential):
    - Every request for a CS is granted eventually.
  - Ordering (desirable):
    - Requests are granted in the order they were made.

Performance metrics

- **Bandwidth**:
  - the total number of messages sent in each *enter* and *exit* operation.
- **Client delay**:
  - the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - *We will focus on the client delay for the enter operation.*
- **Synchronization delay**:
  - the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting).

Mutual exclusion in distributed systems

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

Central server based

- A client process:
  - sends request to the central server when it wants to enter CS.
  - enters CS only after receiving a token from the server.
  - releases the token back to the server upon exiting CS.
- Server grants token to only one process at a time.

- Does it guarantee safety, liveness, and ordering?

- What is its bandwidth usage, client delay, and synchronization delay?

Ring based

- A single token moves around a logical ring of processes.
- A process holds the token while executing CS, and releases it when done.
  - It simply forwards the token if it does not want to enter CS.

- Does it guarantee safety, liveness, and ordering?

- What is its bandwidth usage, client delay, and synchronization delay?

Ricart-Agrawala Algorithm

- Send request to all processes and wait for reply from all.
- A process always replies back to a request, except when:
    - It is currently executing CS (in HELD state)
    - It wants to enter CS (in WANTED state) and deserves to enter it sooner.
        - The Lamport timestamp of its own request is smaller than the Lamport timestamp of the received request.
        - Use process ID to break ties.

- Does it guarantee safety, liveness, and ordering?
- What is its bandwidth usage, client delay, and synchronization delay?

Maekawa Algorithm

- Each process has a voting set consisting of a subset of processes.
- Intersection of voting set of any two processes must be non-zero.
- Send request to all processes in the voting set and wait for reply from all of them.
- A process replies back to a request only if it has not replied to (or voted for) a request from another process.

- Does it guarantee safety, liveness, and ordering?
- What is its bandwidth usage, client delay, and synchronization delay?

Topics for first midterm

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

Summary

- Review of relevant concepts for first midterm.
- Not meant to be an exhaustive review!
- Go over the slides for each class.
  - Refer to lecture videos and textbook to fill in gaps in understanding.

Good luck!