Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements: Indy Gupta and Nikita Borisov

Midterm exam: Feb 27-29

- Detailed instructions shared on CampusWire (post #126).
  - Go over them again.
  - Reserve a slot if you haven't already.
  - Submit your Letters of Accommodations to CBTF, if required.
  - Syllabus: everything covered in class upto and including Multicast.
  - Closed-book exam: cannot refer to any materials.
    - We will provide a cheatsheet over PrairieLearn.
  - CBTF will provide calculator and scratch paper.
  - Practice Midterm 1 has been released on PrairieLearn.

Midterm exam

- Syllabus:
  - everything up to and including Multicast.
- Exam duration: 50mins
  - Extra time to check-in and settle in.

PrairieLearn

- Exam format:
  - Multiple choice questions:
    - Single answer correct; True/False
    - Multiple answers may be correct.
  - Numerical questions
    - No step marking!
  - Ensure all your responses are "saved" and none are "invalid".

Practice Question		
Question 1	saved	5
Question 2	invalid	10

Today's agenda

- Exam Review

Disclaimer for our agenda today

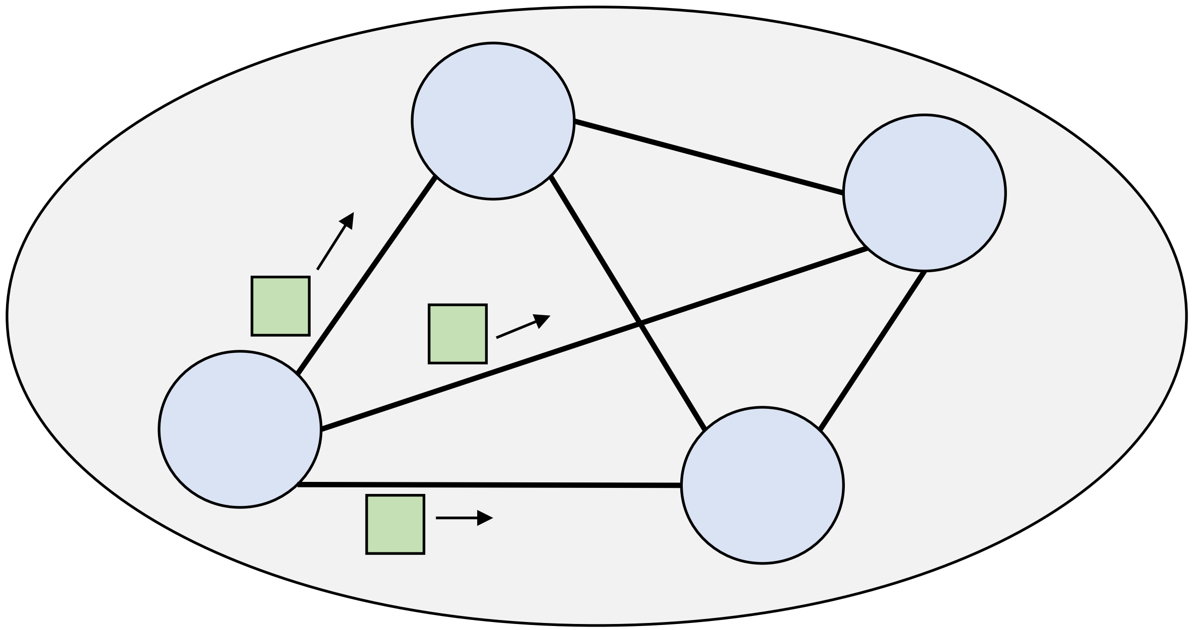
- Quick reminder of the relevant topics we covered in class, that are included in your midterm.
- Not meant to be an exhaustive review!
- Go over the slides for each class.
    - Refer to lecture videos and textbook to fill in gaps in understanding.

Topics for your midterm

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

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- Multicast

What is a distributed system?



Independent components or elements 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 main categories:
  - Client-server
  - Peer-to-peer

Two ways to model

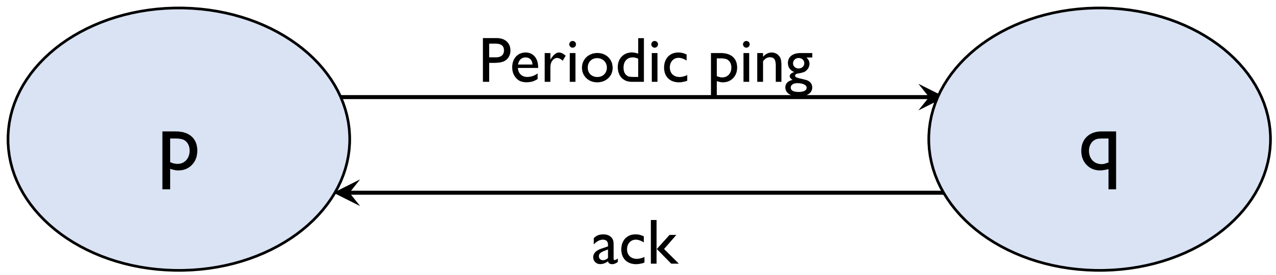
- Synchronous distributed systems:
  - Known upper and lower bounds on time taken by each step in a process.
  - Known bounds on message passing delays.
  - Known bounds on clock drift rates.
- Asynchronous distributed systems:
  - No bounds on process execution speeds.
  - No bounds on message passing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

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- Multicast

Types of failure

- **Omission:** when a process or a channel fails to perform actions that it is supposed to do.
    - Process may **crash**.
        - Detected using ping-ack or heartbeat failure detector.
        - Completeness and accuracy in synchronous and asynchronous systems.
        - Worst case failure detection time.
    - **Communication omission**: a message sent by process was not received by another.
        - Message drops (or omissions) can be mitigated by network protocols.

How to detect a crashed process?



p sends pings to q every T seconds.

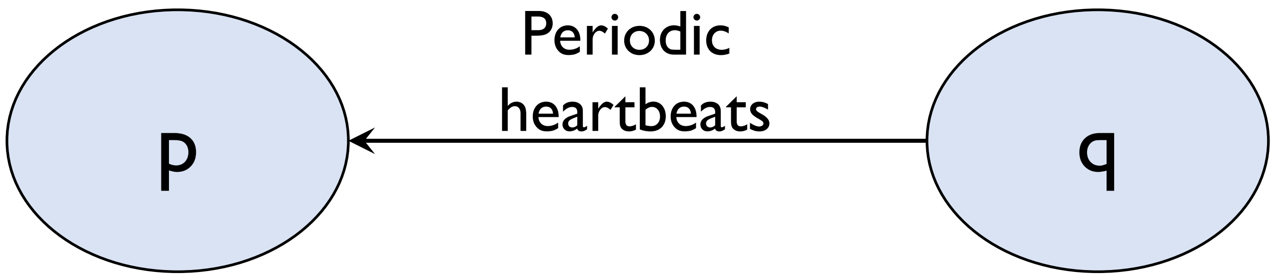
Δ_1 is the *timeout* value at p.

If Δ_1 time elapsed after sending ping, and no ack, report q crashed.

If synchronous, Δ_1 = 2(max network delay)

If asynchronous, Δ_1 = k(max observed round trip time)

How to detect a crashed process?



q sends heartbeats to p every T seconds.

$(T + \Delta_2)$ is the *timeout* value at p.

If $(T + \Delta_2)$ time elapsed since last heartbeat, report q crashed.

If synchronous, Δ_2 = max network delay – min network delay

If asynchronous, Δ_2 = k(observed delay)

Correctness of failure detection

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

Metrics for failure detection

- Worst case failure detection time
  - Ping-ack: $T + \Delta_1 - \Delta$ (where Δ is time taken for previous ping from p to reach q)
  - Heartbeat: $T + \Delta_2 + \Delta$ (where Δ is time taken for last heartbeat from q to reach p)
- Bandwidth usage:
  - Ping-ack: 2 messages every T units
  - Heartbeat: 1 message every T units.

Types of failure

- **Omission:** when a process or a channel fails to perform actions that it is supposed to do, e.g. process crash and message drops.
- **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.

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- 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).
    - Depends on change in the frequency of oscillation of a crystal in the hardware clock.
- Synchronous systems have bound on maximum drift rate.

Two forms of synchronization

- External synchronization
  - Synchronize time with an authoritative clock.
  - When accurate timestamps are required.

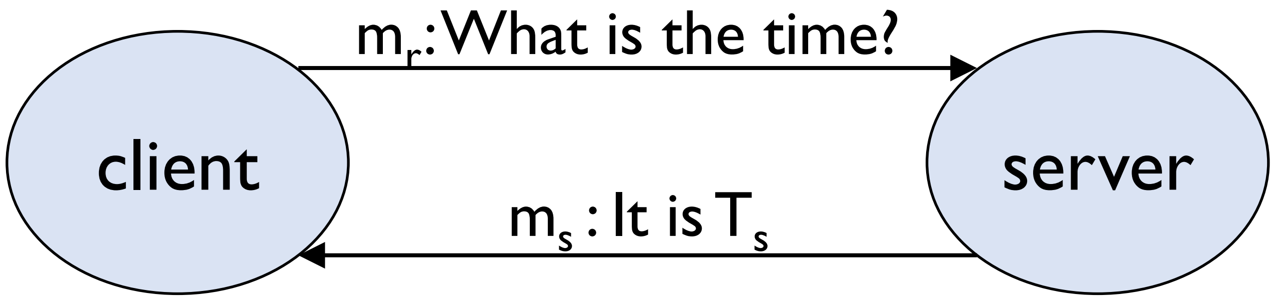
- Internal synchronization
  - Synchronize time internally between all processes in a distributed system.
  - When internally comparable timestamps are required.

- If all clocks in a system are externally synchronized, they are also internally synchronized.

Synchronization Bound

- 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.
    - Skew(A, B) < D during the time interval I.
    - A and B *agree* within a bound *D*.
  - If A is authoritative, D can also be called *accuracy bound*.
    - B is *accurate* within a bound of *D*.
- Synchronization/accuracy bound (D) at time 't'
  - worst-case skew between two clocks at time 't'
    - Skew(A, B) < D at time t

Synchronization in synchronous systems

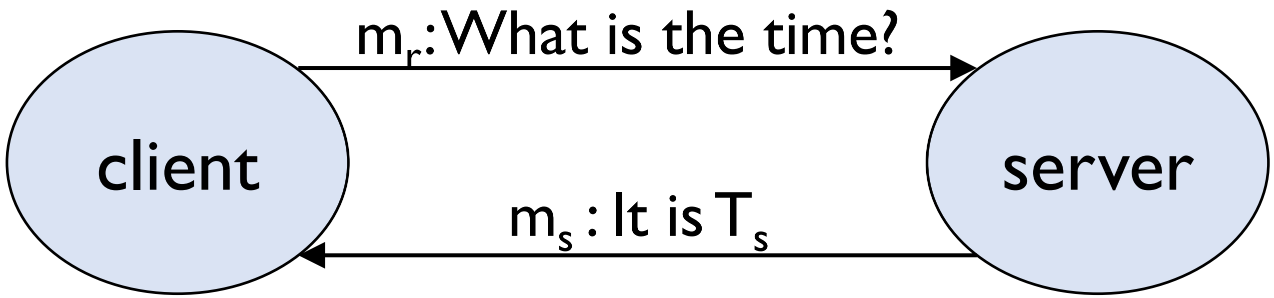


What time T_c should client adjust its local clock to after receiving m_s ?

Let *max* and *min* be maximum and minimum network delay.

If $\mathbf{T_c = (T_s + (min + max)/2)}$, skew(client,server) $\leq$ *(max – min)/2*

Cristian Algorithm



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

Client measures the round trip time (T_{round}).

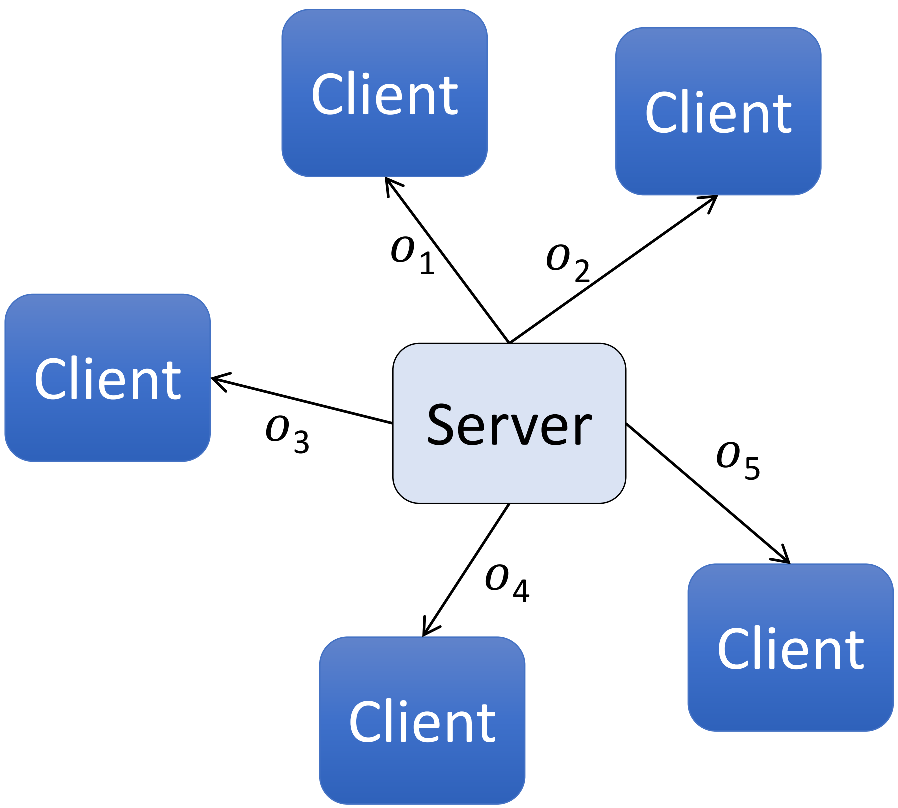
$$T_c = T_s + (T_{round} / 2)$$

$$\begin{aligned} \text{skew} &\leq (T_{round} / 2) - \mathit{min} \\ &\leq (T_{round} / 2) \end{aligned}$$

(*min* is minimum one way network delay which is atleast zero).

Berkeley Algorithm

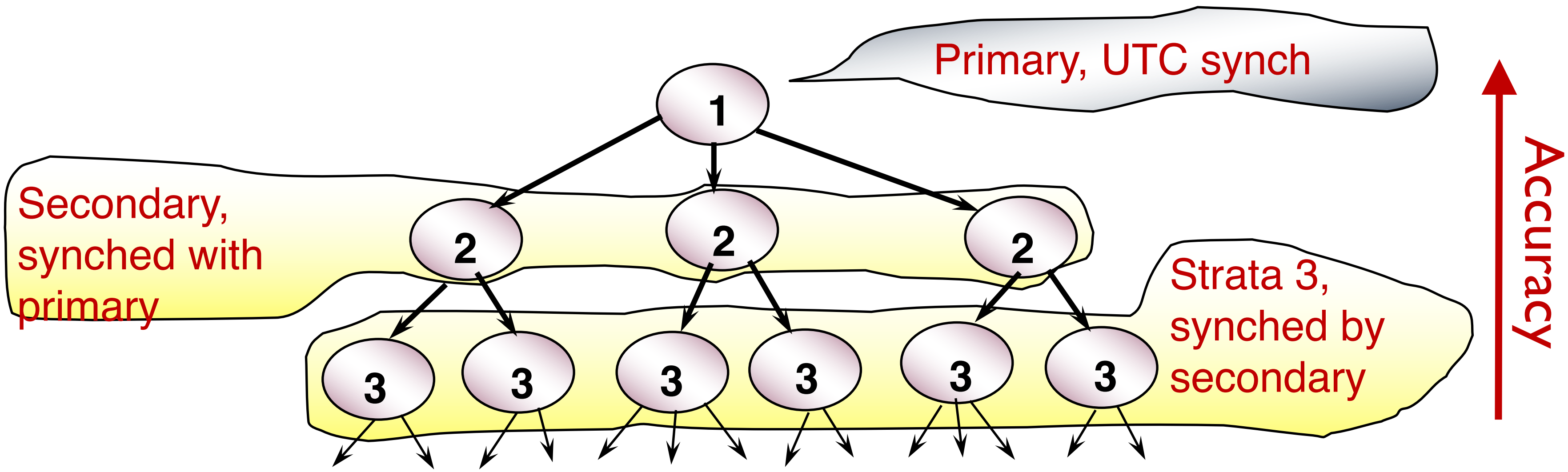
Only supports internal synchronization.



1. Server periodically polls clients: *"what time do you think it is?"*
2. Each client responds with its local time.
3. Server uses Cristian algorithm to estimate local time at each client.
4. Average all local times (including its own) – use as updated time.
5. Send the offset (amount by which each clock needs adjustment).

Network Time Protocol

Time service over the Internet for synchronizing to UTC.



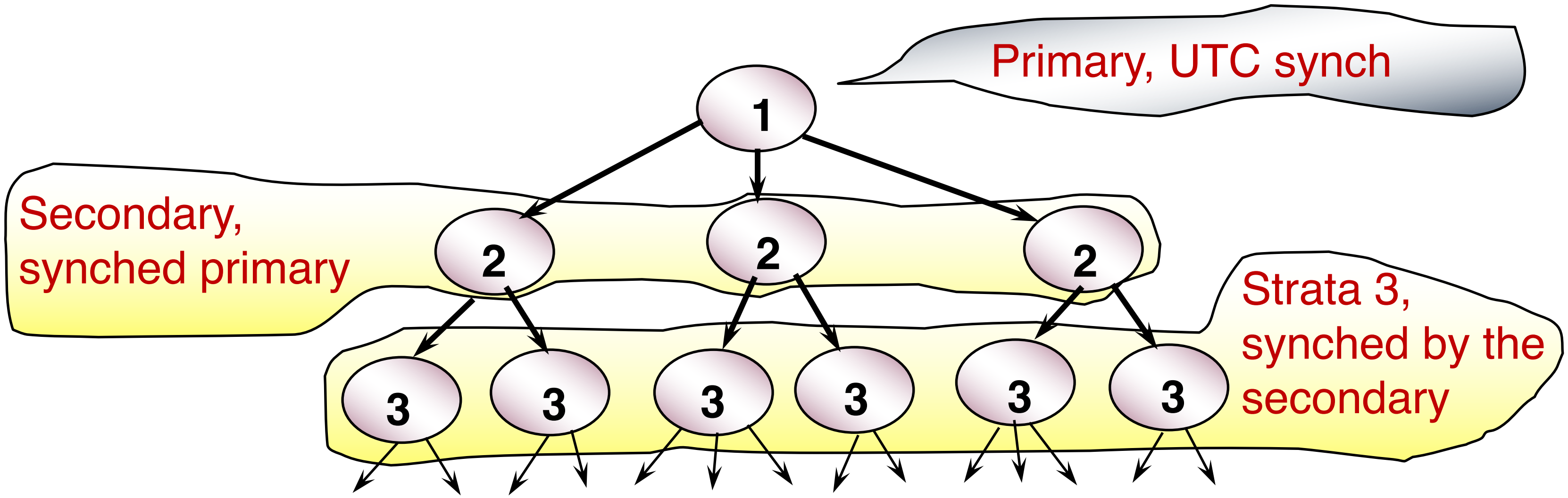
Hierarchical structure for *scalability*.

Multiple lower strata servers for *robustness*.

Authentication mechanisms for *security*.

Statistical techniques for better *accuracy*.

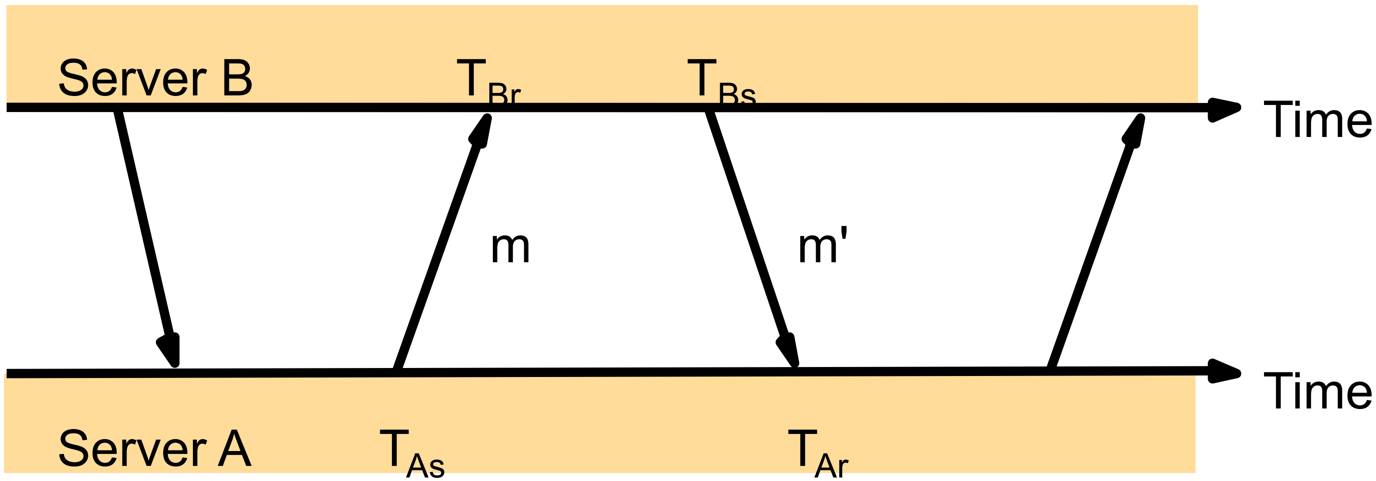
Network Time Protocol



How clocks get synchronized:

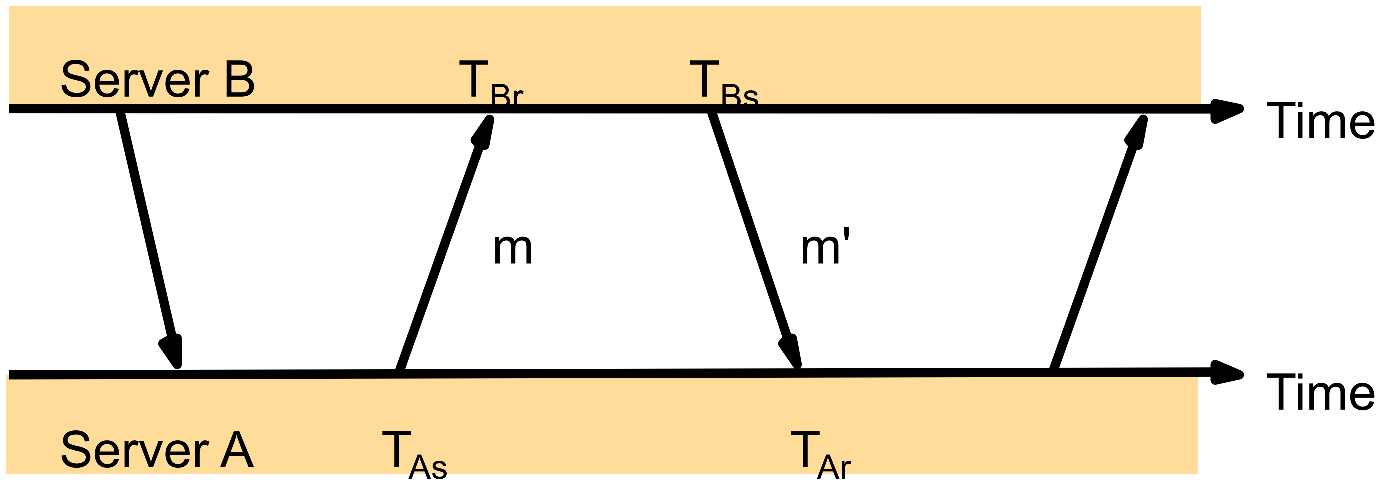
- Servers may *multicast* timestamps within a LAN. Clients adjust time assuming a small delay. *Low accuracy*.
- *Procedure-call* (Cristian algorithm). *Higher accuracy.*
- *Symmetric mode* used to synchronize lower strata servers. *Highest accuracy*.

NTP Symmetric Mode



- A and B exchange messages and record the send and receive timestamps.
  - T_{Br} and T_{Bs} are local timestamps at B.
  - T_{Ar} and T_{As} are local timestamps at A.
  - A and B exchange their local timestamp with eachother.
- Use these timestamps to compute offset with respect to one another.

NTP Symmetric Mode



- t and t': actual transmission times for m and m'(unknown)
- o: <u>true</u> offset of clock at B relative to clock at A (unknown)
- o_i: <u>estimate</u> of actual offset between the two clocks
- d_i: estimate of <u>accuracy</u> of o_i ; total transmission times for m and m'. d_i=t+t'

$$T_{Br} = T_{As} + t + o$$
$$T_{Ar} = T_{Bs} + t' - o$$

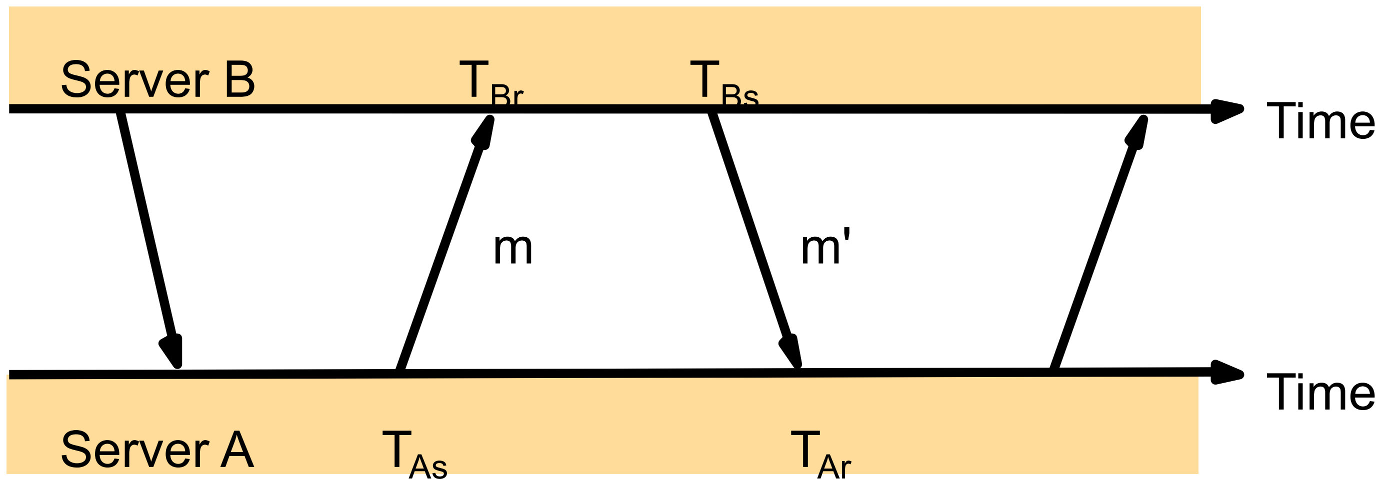
$$o = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}) + (t' - t))/2$$
$$\mathbf{o_i = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}))/2}$$
$$o = o_i + (t' - t)/2$$
$$\mathbf{d_i = t + t' = (T_{Br} - T_{As}) + (T_{Ar} - T_{Bs})}$$

$o = o_i + t'/2$ (if $t \to 0$)

$o = o_i - t/2$ (if $t' \to 0$)

$d_i = t + t'$

NTP Symmetric Mode



- t and t': actual transmission times for m and m'(unknown)
- o: true offset of clock at B relative to clock at A (unknown)
- o_i: estimate of actual offset between the two clocks
- d_i: estimate of accuracy of o_i ; total transmission times for m and m'. d_i=t+t'

$$T_{Br} = T_{As} + t + o$$
$$T_{Ar} = T_{Bs} + t' - o$$

$$o = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}) + (t' - t))/2$$
$$\mathbf{o_i = ((T_{Br} - T_{As}) - (T_{Ar} - T_{Bs}))/2}$$
$$o = o_i + (t' - t)/2$$
$$\mathbf{d_i = t + t' = (T_{Br} - T_{As}) + (T_{Ar} - T_{Bs})}$$

$$(o_i - d_i / 2) \leq o \leq (o_i + d_i / 2) \quad \text{given } t, t' \geq 0$$

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- 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 $\mathbf{p_i}$.

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

- Also called *"causal" or "potentially 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 $\mathbf{L_i}$ before timestamping the receive event (as per step 2).

Vector Clocks

- Each event associated with a vector timestamp.
- Each process p_i maintains vector of clocks V_i
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process p_j as maintained by p_i
- Algorithm: each process p_i:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks V_i when sending a message.
  4. upon receiving a message with vector clock value v
     - sets $V_i[j] = \max(V_i[j], v[j])$ for all j=1…n.
     - increments $V_i[i]$ before timestamping receive event (as per step 2).

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- Multicast

Some more 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 k^{th} event.

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

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

  global state: $S = \cup_i (s_i^{c_i})$

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

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

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

Consistent cuts and snapshots

- A cut **C** is **consistent** if and only if

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

- A global state **S** is consistent if and only if it corresponds to a consistent cut.

Chandy-Lamport Algorithm

- Goal:
  - Record a global snapshot
    - Process state (and channel state) for a set of processes.
  - The recorded global state is consistent.
- Identifies a consistent cut.
- Records corresponding state locally at each process.

Chandy-Lamport Algorithm

- *System model and assumptions:*
  - System of **n** processes: $<p_1, p_2, p_3, \ldots., p_n>$.
  - There are 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).
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.

- *Requirements:*
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages.
  - Any process may initiate algorithm.

Chandy-Lamport Algorithm

- First, initiator $\mathbf{p_i}$:
  - records its own state.
  - creates a special **marker** message.
  - for *j=1 to n* except *i*
    - $\mathbf{p_i}$ sends a **marker** message on outgoing channel $\mathbf{c}_{ij}$
    - starts recording the incoming messages on each of the incoming channels at $\mathbf{p_i}$: $\mathbf{c}_{ji}$ (for *j=1 to n* except *i*).

Chandy-Lamport Algorithm

Whenever a process p_i receives a **marker** message on an incoming channel c_{ki}

- if (this is the first **marker** p_i is seeing)
  - p_i records its own state first
  - marks the state of channel c_{ki} as "empty"
  - for j=1 to n except i
    - p_i sends out a **marker** message on outgoing channel c_{ij}
  - starts recording the incoming messages on each of the incoming channels at p_i : c_{ji} (for *j=1 to n* except *i and k*).
- else // already seen a **marker** message
  - mark the state of channel c_{ki} as all the messages that have arrived on it since recording was turned on for c_{ki}

Chandy-Lamport Algorithm

The algorithm terminates when

- All processes have received a **marker**
  - To record their own state
- All processes have received a **marker** on all the (n-1) incoming channels
  - To record the state of all channels

More 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.

Global State Predicates

- A global-state-predicate is a property that is *true* or *false* for a global state.
  - Is there a deadlock?
  - Has the distributed algorithm terminated?
- Two ways of reasoning about predicates (or system properties) as global state gets transformed by events.
  - Liveness
  - Safety

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 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.
  - No two processes decide on different values.
- A global state S_0 satisfies a **safety** property P iff:
  - $\text{safety}(P(S_0)) \equiv \forall S$ 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 for a state S, stays true for all states reachable from S (for stable liveness)
- once false for a state S, stays false for all states reachable from S (for stable non-safety)
- Stable liveness examples (once true, always true)
  - Computation has terminated.
- Stable non-safety examples (once false, always false)
  - There is no deadlock.
  - An object is not orphaned.
- *All stable global properties can be detected using the Chandy-Lamport algorithm.*

- System Model
- Failure Detection
- Time and Clocks
- Logical Clocks and Timestamps
- Global State
- **Multicast**

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?*
  - *What does this mean?*

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* to 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 $\in$ g is included as destination)

On B-deliver(m) at process q in g = group(m)

    if (m $\notin$ 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 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

- On FO-multicast(g,m) at process P*j*:

  set P*j*[*j*] = P*j*[*j*] + 1

  piggyback P*j*[*j*] with m as its sequence number.

  B-multicast(g,{m, P*j*[*j*]})

- On B-deliver({m, S}) at Pi from Pj: *If Pi receives a multicast from Pj with sequence number S in message*

  if (S == P*i*[*j*] + 1) then

    FO-deliver(m) to application

    set P*i*[*j*] = P*i*[*j*] + 1

  else buffer this multicast until above condition is true

Implementing causal order multicast

- *CO-multicast(g,m) at Pj:*
  set $Pj[j] = Pj[j] + 1$
  piggyback entire vector $Pj[1…N]$ with m as its sequence no.
  B-multicast(g,{m, $Pj[1…N]$})
- *On B-deliver({m, V[1..N]}) at Pi from Pj:* If Pi receives a multicast from Pj with sequence vector V[1…N], buffer it until both:
  1. This message is the next one P*i* is expecting from P*j*, i.e.,
     $V[j] = Pi[j] + 1$
  2. All multicasts, anywhere in the group, which happened-before m have been received at P*i*, i.e.,
     For all $k \neq j$: $V[k] \leq Pi[k]$

  When above two conditions satisfied,
  CO-deliver(m) and set $Pi[j] = V[j]$

Sequencer based total ordering

- Special process elected as leader or sequencer.
- TO-multicast(g,m) at P*i*:
  - Send multicast message m to group g *and the sequencer*
- Sequencer:
  - Maintains a global sequence number S (initially 0)
  - When a multicast message m is B-delivered to it:
    - sets S = S + 1, and B-multicast(g,{"order", m, S})
- Receive multicast at process P*i*:
  - P*i* maintains a local received global sequence number S*i* (initially 0)
  - On B-deliver(m) at P*i* from P*j*, it buffers it until both conditions satisfied
    1. B-deliver({"order", m, S}) at P*i* from sequencer, and
    2. S*i* + 1 = S
    - Then TO-deliver(m) to application and set S*i* = S*i* + 1

ISIS algorithm for total ordering

- 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.

Topics for your midterm

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

Good luck!