CS 3551 DISTRIBUTED COMPUTING

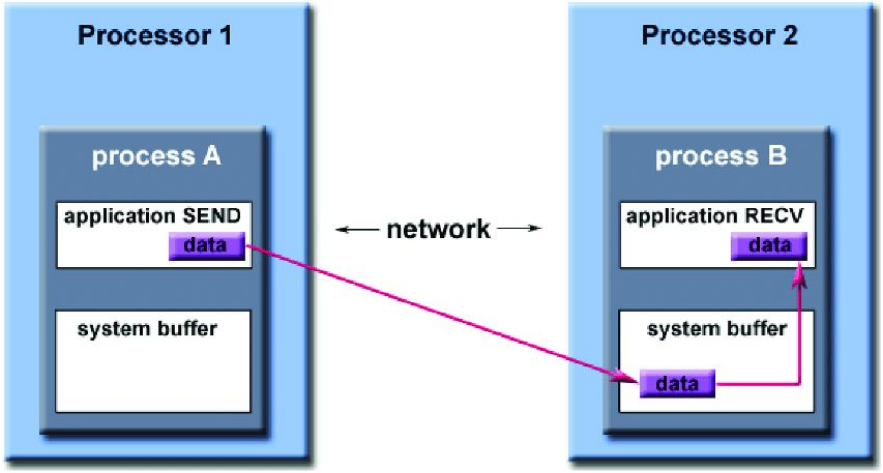
Logical Time: Physical Clock Synchronization: NTP – A Framework for a System of Logical Clocks – Scalar Time – Vector Time;

Message Ordering and Group Communication: Message Ordering Paradigms – Asynchronous Execution with Synchronous Communication – Synchronous Program Order on Asynchronous System – Group Communication – Causal Order – Total Order;

Global State and Snapshot Recording Algorithms: Introduction – System Model and Definitions – Snapshot Algorithms for FIFO Channels.

How interaction takes place in distributed system?

1 2 3 4 ⇒ 4, 2, 1, 3 ✗



✗ order ✗

① Non-FIFO
② FIFO
③ Causal

Notation

msg – m^i

send event – s^i

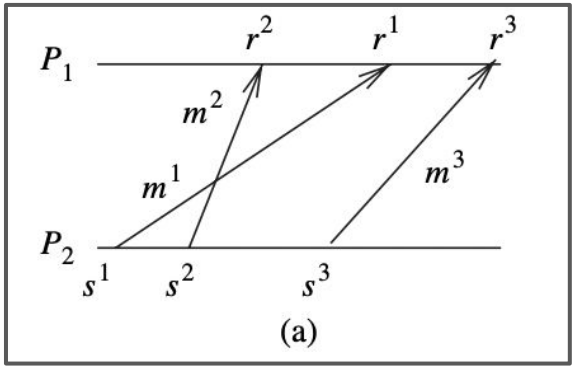
rec – r^i

$a \sim b \Rightarrow a \& b$ @ same process

$T \Rightarrow$ all send-rec pairs

Message Ordering Paradigms

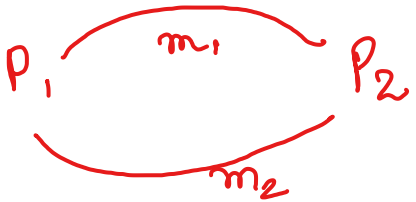
① Phy
② Log.



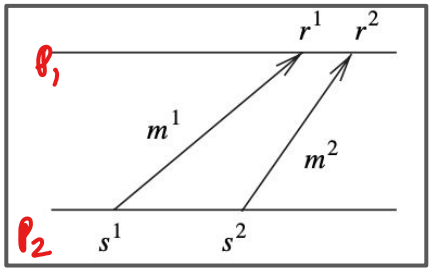
1. **Asynchronous executions** - **Non - FIFO**

Definition 6.1 (*A*-**execution**) An asynchronous execution (or *A*-execution) is an execution $(E, \prec)$ for which the causality relation is a partial order.

- **Non-FIFO transmission :** On any logical link between two nodes in the system, messages may be delivered in any order, not necessarily first-in first-out. Such executions are also known as non-FIFO executions.
- Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link.



Message Ordering Paradigms



2. FIFO Execution

Definition 6.2 (FIFO executions) A FIFO execution is an A-execution in which,
for all (s, r) and $(s', r') \in \mathcal{T}$, $(s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'$.

- On any logical link in the system, messages are necessarily delivered in the order in which they are sent. Although the logical link is inherently non-FIFO, most network protocols provide a connection-oriented service at the transport layer.

a b c
1 2 3

c a b
3 1 2

- **Design FIFO algo over non-FIFO channel:**
  - separate numbering scheme to sequence the messages on each logical channel.
  - The sender assigns and appends a sequence_num, connection_id tuple to each message.
  - The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.

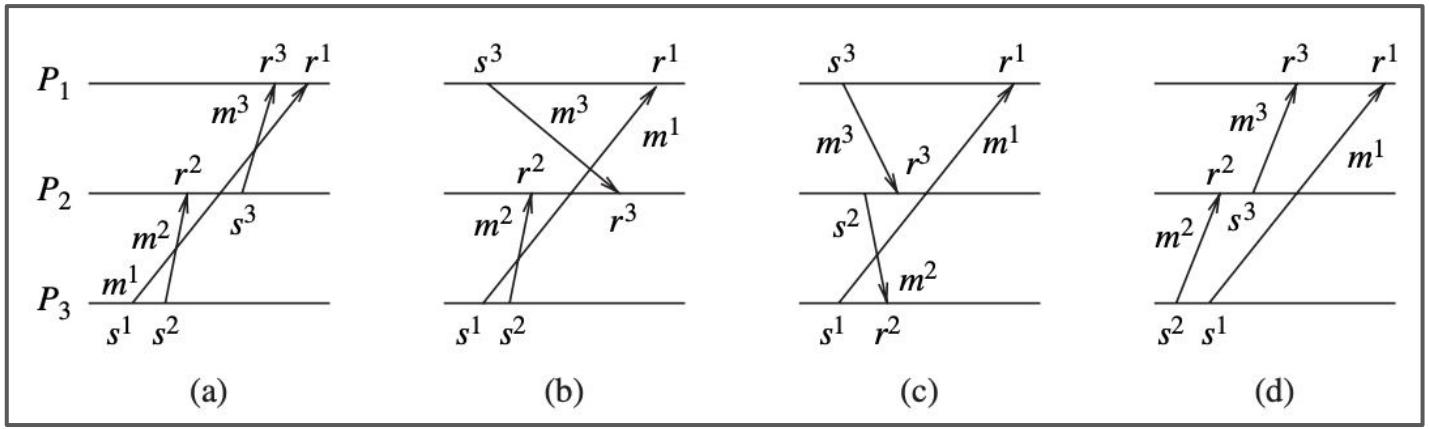
Message Ordering Paradigms

3. Causally Ordered

Definition 6.3 (Causal order (CO)) A CO execution is an A-execution in which,
for all (s, r) and $(s', r') \in \mathcal{T}$, $(r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'$.

- If two send events s and s' are related by causality ordering (not physical time ordering), then a causally ordered execution requires that their corresponding receive events r and r' occur in the same order at all common destinations.

3. Causally Ordered Examples



$S_1 < S_3$
$r_1 < r_3$

- **Figure 6.2(a)** shows an execution that violates CO because $s^1 \prec s^3$ and at the common destination P_1, we have $r^3 \prec r^1$.
- **Figure 6.2(b)** shows an execution that satisfies CO. Only s^1 and s^2 are related by causality but the destinations of the corresponding messages are different.
- **Figure 6.2(c)** shows an execution that satisfies CO. No send events are related by causality.
- **Figure 6.2(d)** shows an execution that satisfies CO. s^2 and s^1 are related by causality but the destinations of the corresponding messages are different. Similarly for s^2 and s^3.

Delivery Event

- To implement CO, we distinguish between the arrival of a message and its delivery.
- A message m that arrives in the local OS buffer at Pi may have to be delayed until the messages that were sent to Pi causally before m was sent (the “overtaken” messages) have arrived and are processed by the application.
- The delayed message m is then given to the application for processing.
- The event of an application processing an arrived message is referred to as a delivery event (instead of as a receive event) for emphasis.

Other Definitions

Definition 6.4 (Definition of causal order (CO) for implementations) If $send(m^1) \prec send(m^2)$ then for each common destination d of messages m^1 and m^2, $deliver_d(m^1) \prec deliver_d(m^2)$ must be satisfied.

Definition 6.5 (Message order (MO)) A MO execution is an A-execution in which,
for all (s, r) and $(s', r') \in \mathcal{T}$, $s \prec s' \Longrightarrow \neg(r' \prec r)$.

$r \prec r'$

Definition 6.6 (Empty-interval execution) An execution $(E, \prec)$ is an empty-interval (EI) execution if for each pair of events $(s, r) \in \mathcal{T}$, the open interval set $\{x \in E \mid s \prec x \prec r\}$ in the partial order is empty.

4. Synchronous execution (SYNC)

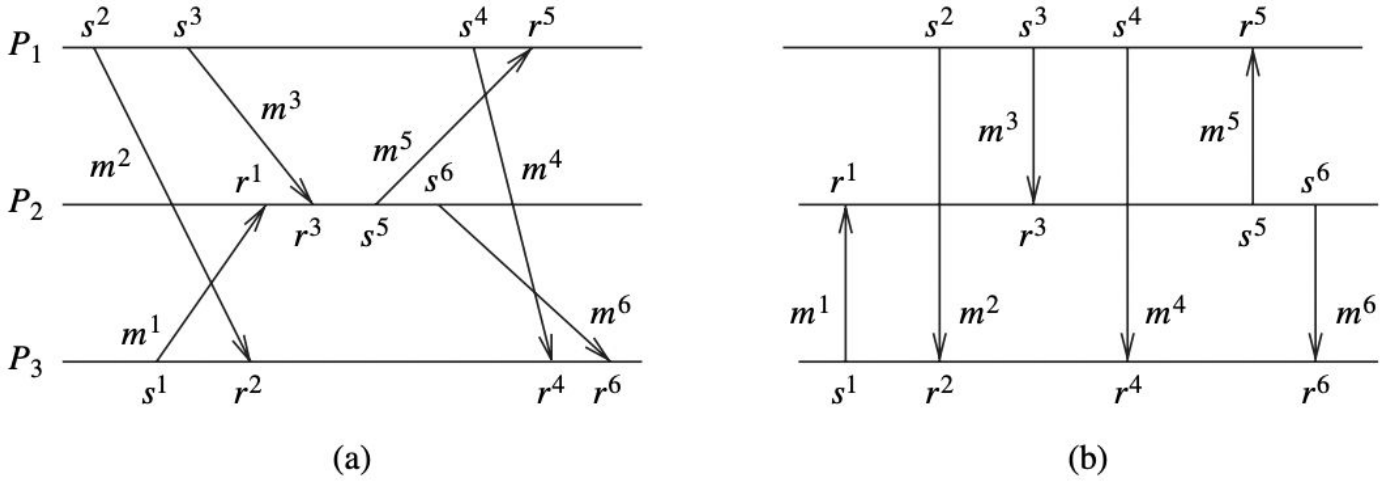


Figure 6.3 Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.

Key Points

- When all the communication between pairs of processes uses synchronous send and receive primitives, the resulting order is the synchronous order.
- As each synchronous communication involves a handshake between the receiver and the sender, the corresponding send and receive events can be viewed as occuring instantaneously and atomically

Definition 6.7 (Causality in a synchronous execution) The synchronous causality relation $\ll$ on E is the smallest transitive relation that satisfies the following:

S1: If x occurs before y at the same process, then $x \ll y$.
S2: If $(s, r) \in \mathcal{T}$, then for all $x \in E$, $[(x \ll s \iff x \ll r)$ and $(s \ll x \iff r \ll x)]$.
S3: If $x \ll y$ and $y \ll z$, then $x \ll z$.

We can now formally define a synchronous execution.

Definition 6.8 (Synchronous execution) A synchronous execution (or S-execution) is an execution $(E, \ll)$ for which the causality relation $\ll$ is a partial order.

Definition 6.9 (Timestamping a synchronous execution) An execution $(E, \prec)$ is synchronous if and only if there exists a mapping from E to T (scalar timestamps) such that

- for any message M, $T(s(M)) = T(r(M))$;
- for each process P_i, if $e_i \prec e_i'$ then $T(e_i) < T(e_i')$.

By assuming that a send event and its corresponding receive event are viewed atomically, i.e., $s(M) \prec r(M)$ and $r(M) \prec s(M)$, it follows that for any events e_i and e_j that are not the send event and the receive event of the same message, $e_i \prec e_j \implies T(e_i) < T(e_j)$.

LIKE

COMMENT

SHARE

SUBSCRIBE

Synchronous program order on an asynchronous system

How non-determinism occurs in DC?

1. A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.

2. Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.

Rendezvous

Multiway rendezvous:- synchronous communication among an arbitrary number of asynchronous processes.

Binary rendezvous:- synchronous communication between a pair of processes at a time.

Support for binary rendezvous communication was first provided by programming languages such as CSP and Ada. We consider here a subset of CSP.

Simple Command with Notation

repetative → $*[G_1 \longrightarrow CL_1 \;||\; G_2 \longrightarrow CL_2 \;||\; \cdots \;||\; G_k \longrightarrow CL_k].$ ← alternative

guard

- Each communication command may be a part of a guard G_i, and may also appear within the statement block CL_i. A guard G_i is a boolean expression.
- If a guard G_i evaluates to true then CL_i is said to be enabled, otherwise CL_i is said to be disabled.
- A send command of local variable x to process P_k is denoted as “x $!P_k$.”
- A receive from process Pk into local variable x is denoted as “P_k ? x.”

Observations about synchronous communication under binary rendezvous

1. For the receive command, the sender must be specified.
2. However, multiple receive commands can exist.
3. A type check on the data is implicitly performed.
4. Send and received commands may be individually disabled or enabled. A command is disabled if it is guarded and the guard evaluates to false. The guard would likely contain an expression on some local variables.
5. Synchronous communication is implemented by scheduling messages under the covers using asynchronous communication.
6. Scheduling involves pairing of matching send and receive commands that are both enabled.
7. The communication events for the control messages under the covers do not alter the partial order of the execution.

Algorithm for binary rendezvous

Constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the scheduling code at any process does not know the application code of other processes.
- Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.
- Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.

Bagrodia's Algorithm for Binary Rendezvous (1)

Assumptions

- Receives are always enabled
- Send, once enabled, remains enabled
- To break deadlock, PIDs used to introduce asymmetry
- Each process schedules *one* send at a time

Message types: M, $ack(M)$, $request(M)$, $permission(M)$

Process blocks when it knows it can successfully synchronize the current message

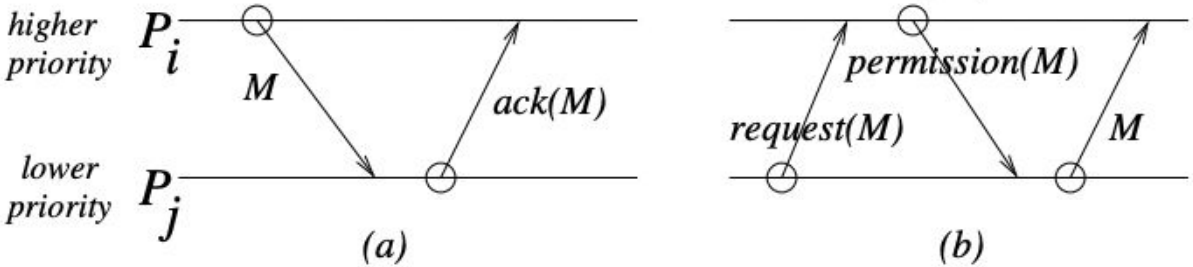
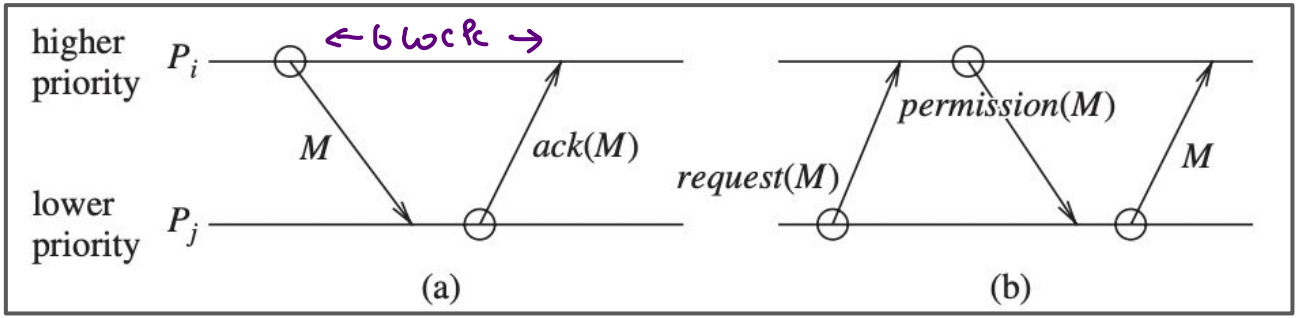


Fig 6.: Rules to prevent message cyles. (a) High priority process blocks. (b) Low priority process does not block.

Algorithm in Simple Steps



- To send to a lower priority process
  a. The sender issues send(M)
  b. blocks until ack(M) arrives
  c. Thus, when sending to a lower priority process, the sender blocks waiting for the partner process to synchronize and send an acknowledgement.
- To send to a higher priority process
  a. The sender issues send(request(M)), does not block, and awaits permission.
  b. When permission(M) arrives, the sender issues send(M).

(message types)
M, $ack(M)$, $request(M)$, $permission(M)$

(1) **P_i wants to execute SEND(M) to a lower priority process P_j:**
P_i executes $send(M)$ and blocks until it receives $ack(M)$ from P_j. The send event SEND(M) now completes.

Any M' message (from a higher priority processes) and $request(M')$ request for synchronization (from a lower priority processes) received during the blocking period are queued.

(2) **P_i wants to execute SEND(M) to a higher priority process P_j:**

(2a) P_i seeks permission from P_j by executing $send(request(M))$.
// to avoid deadlock in which cyclically blocked processes queue
// messages.

(2b) While P_i is waiting for permission, it remains unblocked.

(i) If a message M' arrives from a higher priority process P_k, P_i accepts M' by scheduling a RECEIVE(M') event and then executes $send(ack(M'))$ to P_k.

(ii) If a $request(M')$ arrives from a lower priority process P_k, P_i executes $send(permission(M'))$ to P_k and blocks waiting for the message M'. When M' arrives, the RECEIVE(M') event is executed.

(2c) When the $permission(M)$ arrives, P_i knows partner P_j is synchronized and P_i executes $send(M)$. The SEND(M) now completes.

(3) ***request*(M) arrival at P_i from a lower priority process P_j:**
At the time a *request*(M) is processed by P_i, process P_i executes *send*(*permission*(M)) to P_j and blocks waiting for the message M. When M arrives, the RECEIVE(M) event is executed and the process unblocks.

(4) **Message M arrival at P_i from a higher priority process P_j:**
At the time a message M is processed by P_i, process P_i executes RECEIVE(M) (which is assumed to be always enabled) and then *send*(*ack*(M)) to P_j.

(5) **Processing when P_i is unblocked:**
When P_i is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per rules 3 or 4).

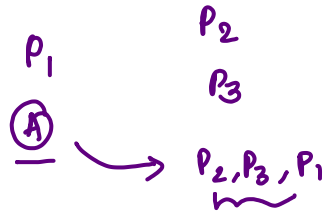
LIKE

COMMENT

SHARE

SUBSCRIBE

Group communication

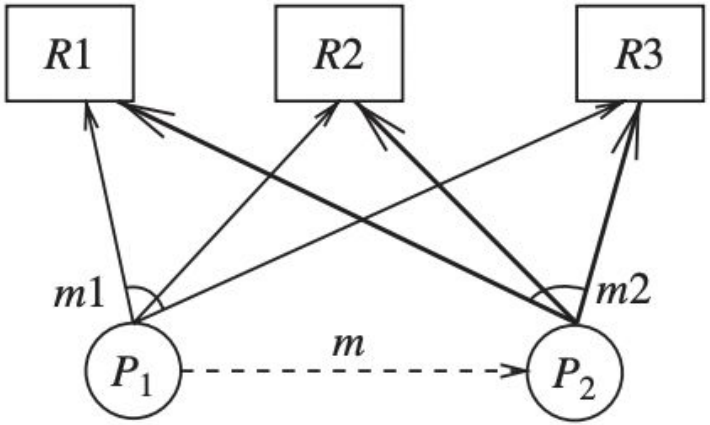


- Processes across a distributed system cooperate to solve a joint task. Often, they need to communicate with each other as a group, and therefore there needs to be support for group communication.
- A message broadcast is the sending of a message to all members in the distributed system.
- If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a closed group algorithm.
- If the sender of the multicast can be outside the destination group, the multicast algorithm is said to be an open group algorithm

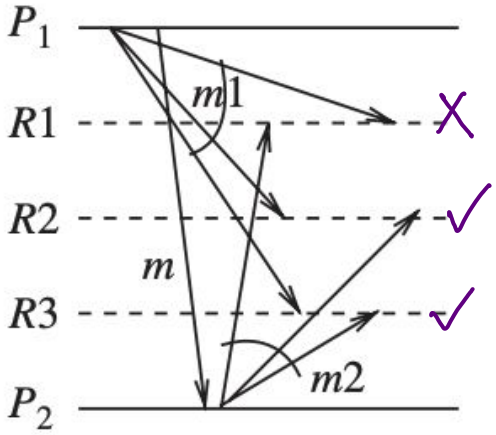
Causal Order

$P_1 < P_2$

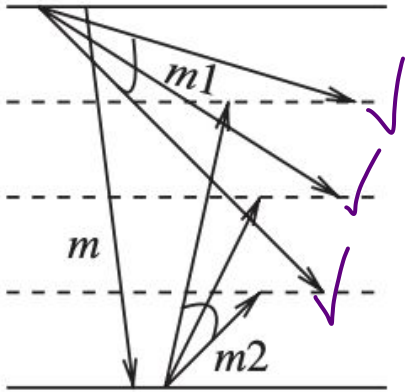
PB instances



(a)



(b)



(c)

A ∝ B B ∝ C ABC ⇒ P2 C B A X - delay

Criteria must be met by a causal ordering protocol

- **Safety** In order to prevent causal order from being violated, a message M that arrives at a process may need to be buffered until all systemwide messages sent in the causal past of the $send(M)$ event to that same destination have already arrived.
  Therefore, we distinguish between the arrival of a message at a process (at which time it is placed in a local system buffer) and the event at which the message is given to the application process (when the protocol deems it safe to do so without violating causal order). The arrival of a message is transparent to the application process. The delivery event corresponds to the *receive* event in the execution model.
- **Liveness** A message that arrives at a process must eventually be delivered to the process.

Raynal-Schiper-Toueg (RST) Algorithm

(local variables)
array of int $SENT[1 \ldots n, 1 \ldots n]$
array of int $DELIV[1 \ldots n]$ // $DELIV[k]$ = # messages sent by k that are delivered locally

(1) **send event**, where P_i wants to send message M to P_j:
(1a) **send** $(M, SENT)$ to P_j;
(1b) $SENT[i,j] \longleftarrow SENT[i,j] + 1$.

(2) **message arrival,** when (M, ST) arrives at P_i from P_j:
(2a) **deliver** M to P_i when **for each** process x,
(2b) $DELIV[x] \geq ST[x,i]$;
(2c) $\forall x, y$, $SENT[x,y] \longleftarrow \max(SENT[x,y], ST[x,y])$;
(2d) $DELIV[j] \longleftarrow DELIV[j] + 1$.

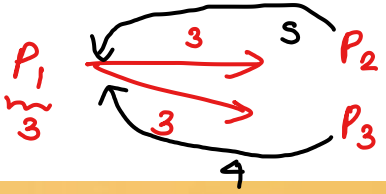
How does algorithm simplify if all msgs are broadcast?

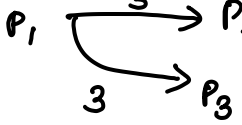
Assumptions/Correctness

- FIFO channels.
- Safety: Step (2a,b).
- Liveness: assuming no failures, finite propagation times

Complexity

- n^2 ints/ process
- n^2 ints/ msg
- Time per send and rcv event: n^2





3-phase

Sender

1) M + tag + ts (3) ⇒ to all process (non-block)

2) Reply of revised ts S,4 from everyone.

final ts = max(ts_j, ts) S//

3) Send final TS to all ✓

Receiver

1) Receive tentative ts.

revised ts = max(pri+1, ts)

insert into queue 1) sender 2) tag 3) ts 4) flag

2) Send back revised ts ✓

3) Receive F TS ✓

Identify in queue.

Make it deliverable ✓

Sort & dequeue.

ts//

Total Message Order

Total order

For each pair of processes P_i and P_j and for each pair of messages M_x and M_y that are delivered to both the processes, P_i is delivered M_x before M_y if and only if P_j is delivered M_x before M_y.

Same order seen by all

Solves coherence problem

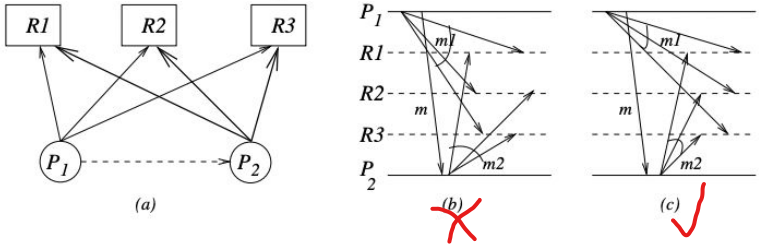


Fig 6.11: (a) Updates to 3 replicas. (b) Total order violated. (c) Total order not violated.

Centralized algorithm

(1) When P_i wants to multicast M to group G:
(1a) **send** $M(i, G)$ to coordinator.

(2) When $M(i, G)$ arrives from P_i at coordinator:
(2a) **send** $M(i, G)$ to members of G.

(3) When $M(i, G)$ arrives at P_j from coordinator:
(3a) **deliver** $M(i, G)$ to application.

Time Complexity: 2 hops/ transmission

Message complexity: n

Total Message Order: 3-phase Algorithm Code

```
record Q_entry
        M: int;                                   // the application message
        tag: int;                                 // unique message identifier
        sender_id: int;                           // sender of the message
        timestamp: int;                           // tentative timestamp assigned to message
        deliverable: boolean;                     // whether message is ready for delivery
(local variables)
queue of Q_entry: temp_Q, delivery_Q
int: clock                                        // Used as a variant of Lamport's scalar clock
int: priority                                     // Used to track the highest proposed timestamp
(message types)
REVISE_TS(M, i, tag, ts)                          // Phase 1 message sent by P_i, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)                        // Phase 2 message sent by P_j, with revised timestamp, to P_i
FINAL_TS(i, tag, ts)                              // Phase 3 message sent by P_i, with final timestamp

(1) When process P_i wants to multicast a message M with a tag tag:
(1a) clock = clock + 1;
(1b) send REVISE_TS(M, i, tag, clock) to all processes;
(1c) temp_ts = 0;
(1d) await PROPOSED_TS(j, i, tag, ts_j) from each process P_j;
(1e) ∀j ∈ N, do temp_ts = max(temp_ts, ts_j);
(1f) send FINAL_TS(i, tag, temp_ts) to all processes;
(1g) clock = max(clock, temp_ts).
(2) When REVISE_TS(M, j, tag, clk) arrives from P_j:
(2a) priority = max(priority + 1, clk);
(2b) insert (M, tag, j, priority, undeliverable) in temp_Q;        // at end of queue
(2c) send PROPOSED_TS(i, j, tag, priority) to P_j.
(3) When FINAL_TS(j, tag, clk) arrives from P_j:
(3a) Identify entry Q_entry(tag) in temp_Q, corresponding to tag;
(3b) mark q_tag as deliverable;
(3c) Update Q_entry.timestamp to clk and re-sort temp_Q based on the timestamp field;
(3d) if head(temp_Q) = Q_entry(tag) then
(3e)     move Q_entry(tag) from temp_Q to delivery_Q;
(3f)     while head(temp_Q) is deliverable do
(3g)         move head(temp_Q) from temp_Q to delivery_Q.
(4) When P_i removes a message (M, tag, j, ts, deliverable) from head(delivery_Q_i):
(4a) clock = max(clock, ts) + 1.
```

sender

receivers

finalTS

Sender

Phase 1 In the first phase, a process multicasts (line 1b) the message M with a locally unique tag and the local timestamp to the group members.

Phase 2 In the second phase, the sender process awaits a reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M. The **await** call in line 1d is non-blocking,

i.e., any other messages received in the meanwhile are processed. Once all expected replies are received, the process computes the maximum of the proposed timestamps for M, and uses the maximum as the final timestamp.

Phase 3 In the third phase, the process multicasts the final timestamp to the group in line (1f).

Receivers

Phase 1 In the first phase, the receiver receives the message with a tentative/proposed timestamp. It updates the variable *priority* that tracks the highest proposed timestamp (line 2a), then revises the proposed timestamp to the *priority*, and places the message with its tag and the revised timestamp at the tail of the queue *temp_Q* (line 2b). In the queue, the entry is marked as undeliverable.

Phase 2 In the second phase, the receiver sends the revised timestamp (and the tag) back to the sender (line 2c). The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

Phase 3 In the third phase, the final timestamp is received from the multicaster (line 3). The corresponding message entry in *temp_Q* is identified using the tag (line 3a), and is marked as deliverable (line 3b) after the revised timestamp is overwritten by the final timestamp (line 3c). The queue is then resorted using the timestamp field of the entries as the key (line 3c). As the queue is already sorted except for the modified entry for the message under consideration, that message entry has to be placed in its sorted position in the queue. If the message entry is at the head of the *temp_Q*, that entry, and all consecutive subsequent entries that are also marked as deliverable, are dequeued from *temp_Q*, and enqueued in *deliver_Q* in that order (the loop in lines 3d–3g).

Total Order: Distributed Algorithm: Example and Complexity

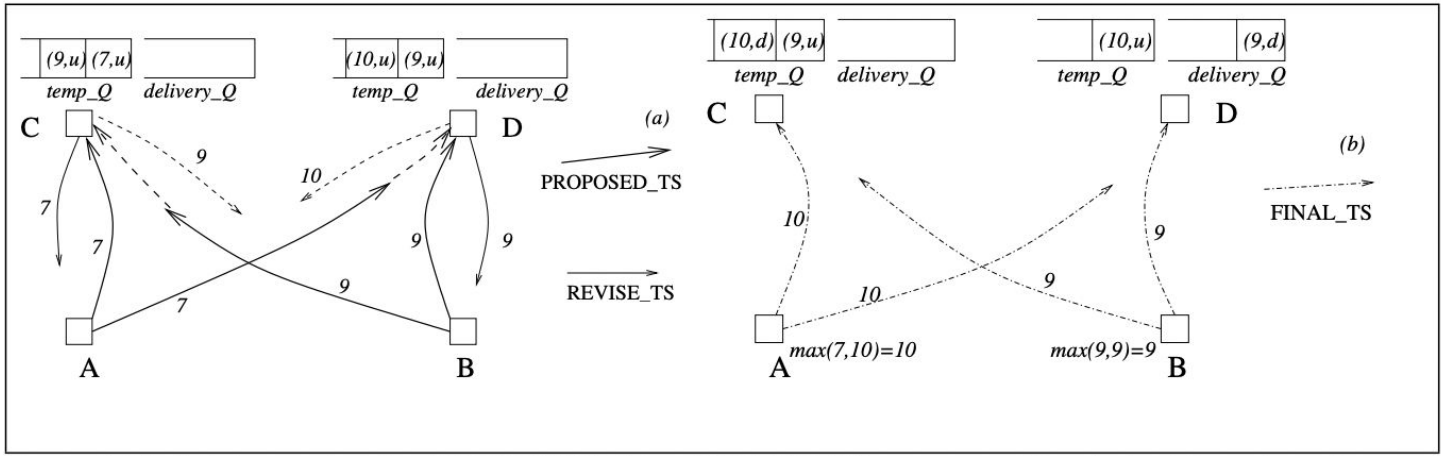


Figure 6.14: (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages.

Complexity:

- Three phases
- $3(n-1)$ messages for $n-1$ dests
- Delay: 3 message hops
- Also implements causal order

- **Figure 6.14(a)** The main sequence of steps is as follows:
  1. A sends a *REVISE_TS*(7) message, having timestamp 7. B sends a *REVISE_TS*(9) message, having timestamp 9.
  2. C receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* $= 7$. C then sends *PROPOSED_TS*(7) message to A.
  3. D receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* $= 9$. D then sends *PROPOSED_TS*(9) message to B.
  4. C receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* $= 9$. C then sends *PROPOSED_TS*(9) message to B.
  5. D receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* $= 10$. D assigns a tentative timestamp value of 10, which is greater than all of the timestamps on *REVISE_TS*s seen so far, and then sends *PROPOSED_TS*(10) message to A.

- **Figure 6.14(b)** The continuing sequence of main steps is as follows:
  6. When A receives *PROPOSED_TS*(7) from C and *PROPOSED_TS*(10) from D, it computes the final timestamp as $max(7, 10) = 10$, and sends *FINAL_TS*(10) to C and D.

7. When B receives *PROPOSED_TS*(9) from C and *PROPOSED_TS*(9) from D, it computes the final timestamp as $max(9, 9) = 9$, and sends *FINAL_TS*(9) to C and D.
8. C receives *FINAL_TS*(10) from A, updates the corresponding entry in *temp_Q* with the timestamp, resorts the queue, and marks the message as deliverable. As the message is not at the head of the queue, and some entry ahead of it is still undeliverable, the message is not moved to *delivery_Q*.
9. D receives *FINAL_TS*(9) from B, updates the corresponding entry in *temp_Q* by marking the corresponding message as deliverable, and resorts the queue. As the message is at the head of the queue, it is moved to *delivery_Q*.

This is the system snapshot shown in Figure 6.14(b). The following further steps will occur:

10. When C receives *FINAL_TS*(9) from B, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*, and the next message (of A), which is also deliverable, is also moved to the *delivery_Q*.
11. When D receives *FINAL_TS*(10) from A, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*.

LIKE

COMMENT

SHARE

SUBSCRIBE