UNIT II -Message ordering and group communication

Message ordering paradigms –Asynchronous execution with synchronous communication –Synchronous program order on an asynchronous system –Group communication – Causal order (CO) – Total order. Global state and snapshot recording algorithms: Introduction –System model and definitions –Snapshot algorithms for FIFO channels

MESSAGE ORDERING PARADIGMS

Notations

We model the distributed system as a graph (N, L). The following notation is used to refer to messages and events:

- When referring to a message without regard for the identity of the sender and receiver processes, we use m^i. For message m^i, its send and receive events are denoted as s^i and r^i, respectively.
- More generally, send and receive events are denoted simply as s and r. When the relationship between the message and its send and receive events is to be stressed, we also use M, send(M) , and receive(M) respectively.

For any two events a and b, where each can be either a send event or a receive event, the notation a ~ b denotes that a and b occur at the same process, i.e., a $\in E_i$ and b $\in E_i$ for some process i.

The send and receive event pair for a message is said to be **a pair of *corresponding* events**. The send event corresponds to the receive event, and vice-versa. For a given execution E, let the set of all send–receive event pairs be denoted as $T = \{(s,r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$.

Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of system executions because it determines the messaging behavior that can be expected by the distributed program.

Several orderings on messages have been defined:

(i) non-FIFO,

(ii) FIFO,

(iii) causal order, and

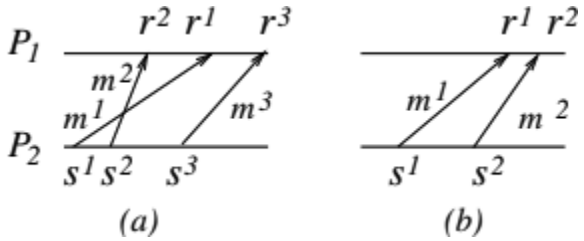
(iv) synchronous order.

Asynchronous and FIFO Executions

Definition (A-execution) : An asynchronous execution (or A-execution) is an execution E< for which the causality relation is a partial order.

- 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*. , e.g., network layer IPv4 connectionless service
- **All physical links obey FIFO**

(a) A-execution that is not FIFO (b) A-execution that is FIFO



FIFO executions

Definition (FIFO executions) A FIFO execution is an A-execution in which, for all (s, r) and (s', r')∈ *T*, (s ~ s'and r ~ r' and s < s') => r < r' .

- Logical link inherently non-FIFO
- Can assume connection-oriented service at transport layer, e.g., TCP
- To implement FIFO over non-FIFO link: use < seq num, conn id > per message. Receiver uses buffer to order messages.

Difference between Asynchronous and FIFO executions.

Asynchronous executions

- *A*-execution: $(E, \prec)$ for which the causality relation is a partial order.
- no causality cycles
- on any logical link, not necessarily FIFO delivery, e.g., network layer IPv4 connectionless service
- All physical links obey FIFO

FIFO executions

- an *A*-execution in which: for all (s, r) and $(s', r') \in \mathcal{T}$, $(s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Longrightarrow r \prec r'$
- Logical link inherently non-FIFO
- Can assume connection-oriented service at transport layer, e.g., TCP
- To implement FIFO over non-FIFO link: use $\langle$ *seq_num, conn_id* $\rangle$ per message. Receiver uses buffer to order messages.

Causal order (CO)

A CO execution is an a execution in which, for all (s,r) and (s' ,r') ∈ T , (r ~ r' and s ≺ s') =⇒ r ≺ r'

- **If send events s and s' are related by causality ordering (not physical time ordering), their corresponding receive events r and r' occur in the same order at all common destinations.**
- **If s and s' are not related by causality, then CO is vacuously satisfied.**

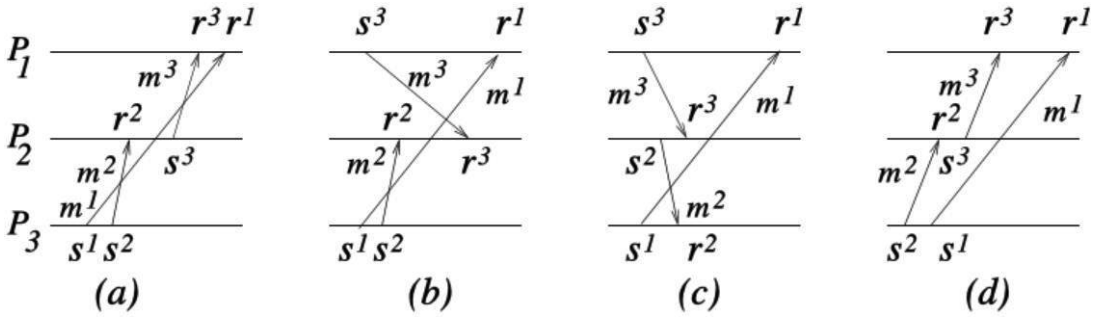


Fig (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO.No send events related by causality. (d) Satisfies CO.

Examples

- Figure (a) shows an execution that violates CO because $s^1 < s^3$ and at the common destination P_1, we have $r^3 < r^1$.
- Figure (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 (c) shows an execution that satisfies CO. No send events are related by causality.
- Figure (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.

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

Message arrival vs. Delivery

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 P_i may have to be delayed until the messages that were sent to P_i 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.
- No message overtaken by a chain of messages between the same (sender, receiver) pair. In Fig. (a), m_1 overtaken by chain $<m_2, m_3>$
- CO degenerates to FIFO when m1, m2 sent by same process

Listout the Uses of CO.

Causal order is useful for applications requiring **updates to shared data, implementing distributed shared memory, and fair resource allocation** such as granting of requests for distributed mutual exclusion ,**collaborative applications, event notification systems, distributed virtual environments**

Other Characterizations of Causal Order

(i) Definition (Message order (MO)) A MO execution is an execution in which, for all (s,r) and (s',r') $\in$ T , $s \prec s' \Rightarrow \neg(r' \prec r)$

Example Consider any message pair, say m^1 and m^3 in Figure (a). $s^1 < s^3$ but $\neg\, r^3 < r^1$ is false. Hence, the execution does not satisfy MO.

(ii) Another characterization of a CO execution in terms of the partial order E $<$ is known as **the empty-interval (EI) property.**

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

- Example: Consider any message, say m2, in Figure (b). There does not exist any event x such that $s2 \prec x \prec r2$. This holds for all messages in the execution. Hence, the execution is EI.
- For EI <s,r> there exists some linear extension $<$ such the corresp. interval $\{x \in E \mid s < x < r\}$ is also empty. (A linear extension of a partial order E $\prec$ is any total order E $<$ such that each ordering relation of the partial order is preserved.)
- An empty <s,r> interval in a linear extension implies s,r may be arbitrarily close; shown by vertical arrow in a timing diagram.
- An execution E is CO iff for each M, there exists some space-time diagram in which that message can be drawn as a vertical arrow.

(iii) Common Past and Future

Another characterization of CO executions is in terms of the causal past/future of a send event and its corresponding receive event.

***An execution* E $\prec$ *is CO if and only if for each pair of events* s r $\in$ *T and each event* e $\in$ E,**

- ***weak common past:*** $e \prec r = \neg\, s \prec e$ ***;***
- ***weak common future:*** $s \prec e = \neg\, e \prec r$ **.**

If the past of both the s and r events are identical (and analogously for the future), viz., $e \prec r \Rightarrow e \prec s$ and $s \prec e = r \prec e$, we get a subclass of CO executions, called ***synchronous executions.***

Synchronous execution (SYNC)

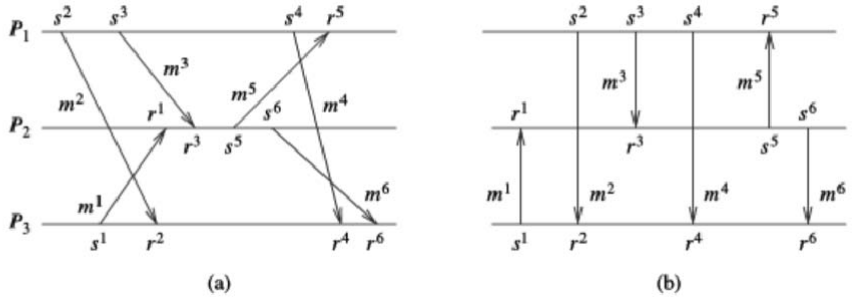


Figure 6.3 Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.

Definition (Casuality in a synchronous execution) The synchronous causality relation 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.

Synchronous execution (or S-execution).

An execution $(E, \ll)$ for which the causality relation $\ll$ is a partial order.

Timestamping a synchronous execution.

An execution $(E, \prec)$ is synchronous iff there exists a mapping from E to T (scalar timestamps) |

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

Asynchronous Execution with Synchronous Communication

Will a program written for an asynchronous system (A-execution) run correctly if run with synchronous primitives?

Process i	Process j
...	...
$Send(j)$	$Send(i)$
$Receive(j)$	$Receive(i)$
...	...

A-execution deadlocks when using synchronous primitives

An A-execution that is realizable under synchronous communication is a realizable with synchronous communication (RSC) execution.

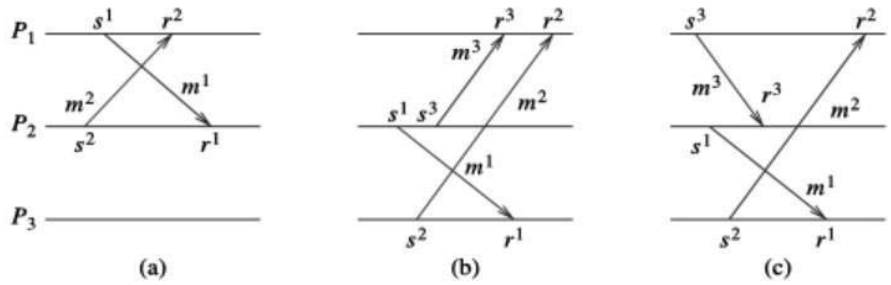


Figure 6.5 Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

RSC (Realizable with synchronous communication) Executions

Non-separated linear extension of (E, ≺)

A linear extension of (E, ≺) such that for each pair (s,r) ∈ T , the interval { x ∈ E | s ≺ x ≺ r } is empty.

Exercise: Identify a non-separated and a separated linear extension in Figs 6.2(d) and 6.3(b)

Examples

- **Figure** 6.2(d): $\langle s^2, r^2, s^3, r^3, s^1, r^1 \rangle$ is a linear extension that is non-separated. $\langle s^2, s^1, r^2, s^3, r^3, s^1 \rangle$ is a linear extension that is separated.
- **Figure** 6.3(b): $\langle s^1, r^1, s^2, r^2, s^3, r^3, s^4, r^4, s^5, r^5, s^6, r^6 \rangle$ is a linear extension that is non-separated. $\langle s^1, s^2, r^1, r^2, s^3, s^4, r^4, r^3, s^5, s^6, r^6, r^5 \rangle$ is a linear extension that is separated.

Defn : RSC execution An A-execution (E, ≺) is an RSC execution iff there exists a non-separated linear extension of the partial order (E, ≺).

- Checking for all linear extensions has exponential cost!
- Practical test using the crown characterization

Crown: Definition

Let E be an execution. A crown of size k in E is a sequence $s^i r^i$, i ∈ 0 k − 1 of pairs of corresponding send and receive events such that: $s^0 \prec r^1$, $s^1 \prec r^2$, , $s^{k-2} \prec r^{k-1}$, $s^{k-1} \prec r^0$.

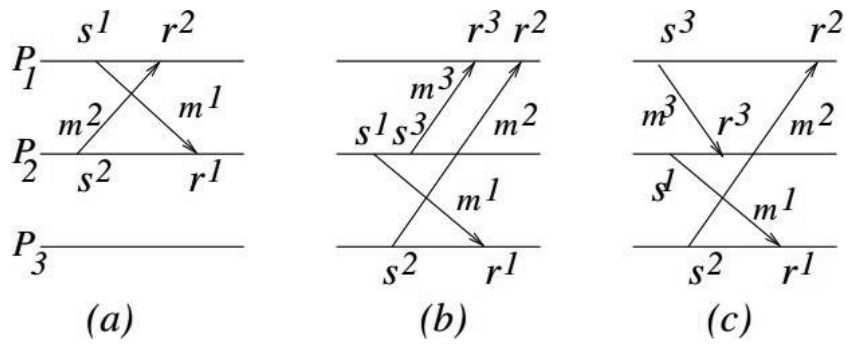


Figure 6.5: Illustration of non-RSC A-executions and crowns. .

(a) Crown of size 2.

(b) Another crown of size 2.

(c) Crown of size 3.

Fig 6.5(a): crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(b) (b) crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$
Fig 6.5(c): crown is $\langle (s^1, r^1), (s^3, r^3), (s^2, r^2) \rangle$ as we have $s^1 \prec r^3$ and $s^3 \prec r^2$ and $s^2 \prec r^1$
Fig 6.2(a): crown is $\langle (s^1, r^1), (s^2, r^2), (s^3, r^3) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^3$ and $s^3 \prec r^1$.

- In a crown, s^i and r^{i+1} may or may not be on same process
- Non-CO execution must have a crown
- CO executions (that are not synchronous) have a crown (see Fig 6.2(b))
- Cyclic dependencies of crown $\Rightarrow$ cannot schedule messages serially $\Rightarrow$ not RSC

Crown Test for RSC executions

1. Define the $\hookrightarrow$: $\mathcal{T} \times \mathcal{T}$ relation on messages in the execution $(E, \prec)$ as follows. Let $\hookrightarrow ([s, r], [s', r'])$ iff $s \prec r'$. Observe that the condition $s \prec r'$ (which has the form used in the definition of a crown) is implied by all the four conditions: (i) $s \prec s'$, or (ii) $s \prec r'$, or (iii) $r \prec s'$, and (iv) $r \prec r'$.
2. Now define a *directed* graph $G_{\hookrightarrow} = (\mathcal{T}, \hookrightarrow)$, where the vertex set is the set of messages $\mathcal{T}$ and the edge set is defined by $\hookrightarrow$.
   Observe that $\hookrightarrow$: $\mathcal{T} \times \mathcal{T}$ is a partial order iff $G_{\hookrightarrow}$ has no cycle, i.e., there must not be a cycle with respect to $\hookrightarrow$ on the set of corresponding (s, r) events.
3. Observe from the defn. of a crown that $G_{\hookrightarrow}$ has a directed cycle iff $(E, \prec)$ has a crown.

Crown criterion

An A-computation is RSC, i.e., it can be realized on a system with synchronous communication, iff it contains no crown.

Crown test complexity: O(|E|) (actually, # communication events)

Timestamps for a RSC execution

(E, $\prec$) is RSC iff there exists a mapping from E to T (scalar timestamps) such that

- for any message M, T(s(M)) = T(r (M))

- for each (a, b) in (E × E) \ T , a ≺ b ⇒ T(a) < T(b)

Hierarchy of Message Ordering Paradigms

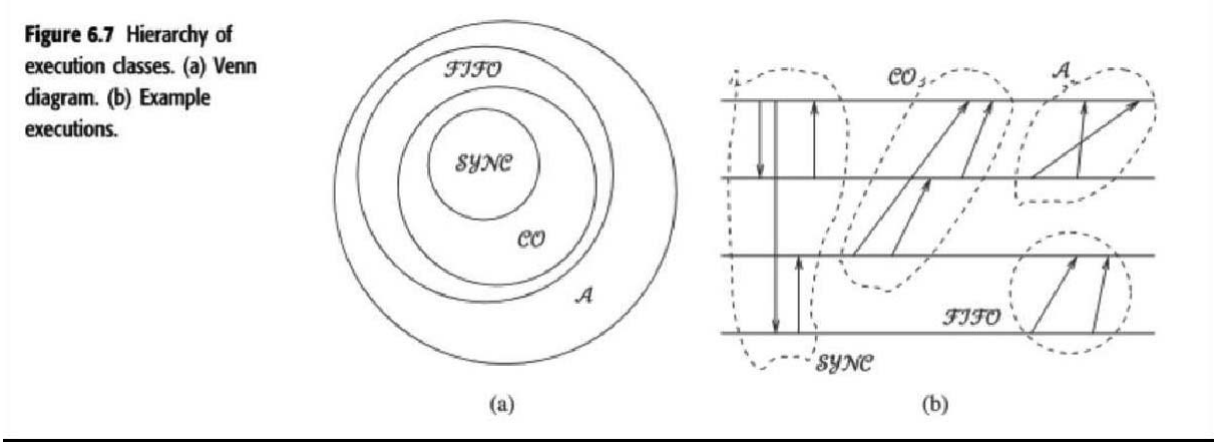


Figure 6.7 Hierarchy of execution classes. (a) Venn diagram. (b) Example executions.

- An A-execution is RSC iff A is an S-execution.
- RSC ⊂ CO ⊂ FIFO ⊂ A.
- More restrictions on the possible message orderings in the smaller classes. The degree of concurrency is most in A, least in SYN C.
- A program using synchronous communication easiest to develop and verify. A program using non-FIFO communication, resulting in an A-execution, hardest to design and verify.

Simulations:

Async Programs on Sync Systems

RSC execution: schedule events as per a non-separated linear extension

- adjacent (s,r) events sequentially
- partial order of original A-execution unchanged

If A-execution is not RSC:

- partial order has to be changed; or
- model each Ci,j by control process Pi,j and use sync communication (see Fig 6.8)
- Enables decoupling of sender from receiver.
- This implementation is expensive.

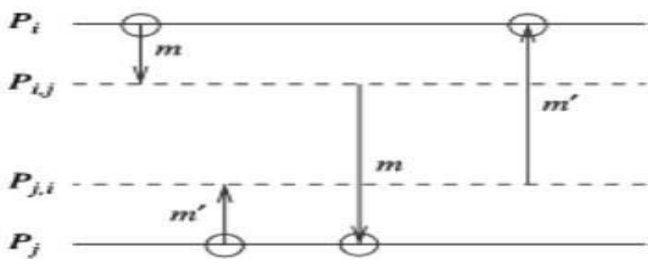


Figure 6.8 Modeling channels as processes to simulate an execution using asynchronous primitives on an synchronous system.

Simulations: Synch Programs on Async Systems

- Schedule msgs in the order in which they appear in S-program
- partial order of S-execution unchanged
- Communication on async system with async primitives

- When sync send is scheduled:
  - wait for ack before completion

2.3 Sync Program Order on Async Systems

Deterministic program: repeated runs produce same partial order

- Deterministic receive ⇒ deterministic execution ⇒ $(E, \prec)$ is fixed

Nondeterminism (besides due to unpredictable message delays):

- Receive call does not specify sender

Multiple sends and receives enabled at a process; can be executed in interchangeable order

Deadlock example of Fig 6.4

- If event order at a process is permuted, no deadlock!

How to schedule (nondeterministic) sync communication calls over async system?

- Match send or receive with corresponding event

Binary rendezvous (implementation using tokens)

- Token for each enabled interaction
- Schedule online, atomically, in a distributed manner
- Crown-free scheduling (safety); also progress to be guaranteed
- Fairness and efficiency in scheduling

Rendezvous

One form of group communication is called *multiway rendezvous*, which is a synchronous communication among an arbitrary number of asynchronous pro-cesses. All the processes involved "meet with each other," i.e., communicate "synchronously" with each other at one time. The solutions to this problem are fairly complex, and we will not consider them further as this model of syn-chronous communication is not popular. The **rendezvous between a pair of processes at a time, which is called *binary rendezvous* as opposed to the *multiway rendezvous*.**

Support for *binary rendezvous* communication was first provided by programming languages such as CSP and Ada. We consider here a subset of CSP. In these languages, the repetitive command (the $*$ operator) over the alternative command (the operator) on multiple guarded commands (each having the form $G_i \longrightarrow CL_i$) is used, as follows:

$$* [G_1 \longrightarrow CL_1 \quad G_2 \longrightarrow CL_2 \quad \cdots \quad G_k \longrightarrow CL_k]$$

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 P_k into local variable x is denoted as "P_k ? x." Some typical observations about synchronous communication under *binary rendezvous* are as follows:

- For the receive command, the sender must be specified. However, multiple recieve commands can exist. A type check on the data is implicitly performed.
- 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.

- Synchronous communication is implemented by *scheduling* messages under the covers using asynchronous communication. Scheduling involves pairing of matching send and receive commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

The concept underlying *binary rendezvous*, which provides synchronous communication, differs from the concept underlying the classification of synchronous send and receive primitives as blocking or non-blocking. *Binary rendezvous* explicitly assumes that multiple send and receives are enabled. Any send or receive event that can be "matched" with the corresponding receive or send event can be scheduled. This is dynamically scheduling the ordering of events and the partial order of the execution.

Algorithm for binary rendezvous

These algorithms typically share the following features

- At each process, there is a set of tokens representing the current interactions that are enabled locally.
- If multiple interactions are enabled, a process chooses one of them and tries to "synchronize" with the partner process.

The problem reduces to one of scheduling messages satisfying the following constraints:

- Schedule on-line, atomically, and in a distributed manner, i.e., the schedul-ing 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.
- Additional features of a good algorithm are: (i) symmetry or some form of fairness, i.e., not favoring particular processes over others during scheduling, and (ii) efficiency, i.e., using as few messages as possible, and involving as low a time overhead as possible.

We now outline a simple algorithm by Bagrodia that makes the following **assumptions:**

1. **Receive commands are forever enabled from all processes.**
2. **A send command, once enabled, remains enabled until it completes, i.e., it is not possible that a send command gets disabled (by its guard getting falsified) before the send is executed.**
3. **To prevent deadlock, process identifiers are used to introduce asymmetry to break potential crowns that arise.**
4. **Each process attempts to schedule only one send event at any time.**

The algorithm illustrates how crown-free message scheduling is achieved on-line.

The message types used are: (i) M, (ii) *ack*(M), (iii) *request*(M), and (iv) *permission*(M). A process blocks when it knows that it can successfully synchronize the current message with the partner process. Each process maintains a queue that is processed in FIFO order only when the process is unblocked. When a process is blocked waiting for a particular message that it is currently synchronizing, any other message that arrives is queued up.

Execution events in the synchronous execution are only the *send* of the message M and *receive* of the message M. The send and receive events for the other message types – *ack*(M), *request*(M), and *permission*(M) which are con-trol messages – are under the covers, and are not included in the synchronous execution. The messages *request*(M), *ack*(M), and *permission*(M) use M's unique tag; the message M is not included in these messages. We use cap-ital SEND(M) and RECEIVE(M) to denote the primitives in the application execution, the lower case send and receive are used for the control messages.

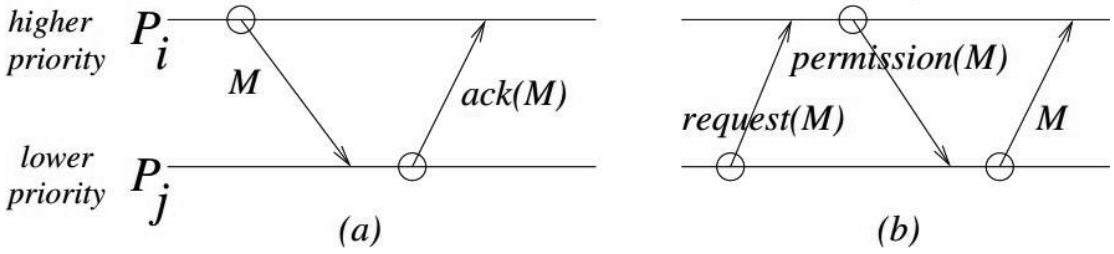
The algorithm to enforce synchronous order is given in Algorithm 6.1. The key rules to prevent cycles among the messages are summarized as follows and illustrated in Figure 6.9:

To send to a lower priority process, messages M and ack(M) are involved in that order. The sender issues send(M) and blocks until ack(M) arrives. 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, messages request(M), permission(M), and M are involved, in that order. The sender issues send(request(M)), does not block, and awaits permission. When permission(M) arrives, the sender issues send(M).

Rules to prevent message cyles.

(a) High priority process blocks. **(b) Low priority process does not block.**



Bagrodia's Algorithm for Binary Rendezvous: Code

(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.

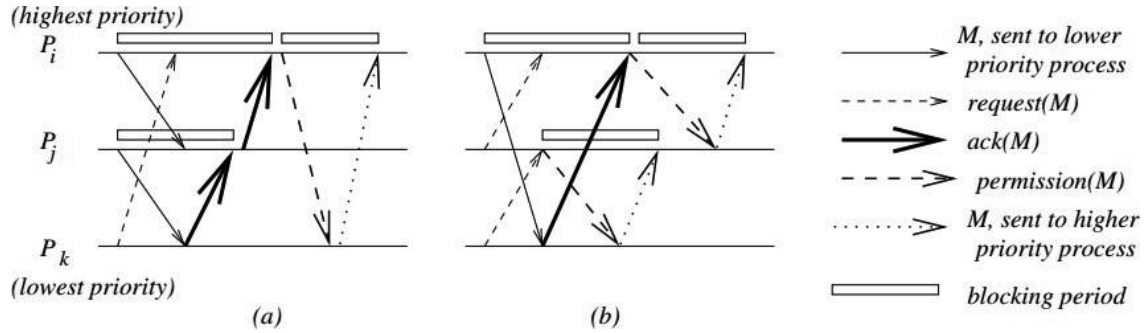
2. **P_i wants to execute SEND(M) to a higher priority process P_j:**
   1. P_i seeks permission from P_j by executing *send(request(M))*.
      // to avoid deadlock in which cyclically blocked processes queue messages.
   2. While P_i is waiting for permission, it remains unblocked.
      1. If a message 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.
      2. 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.
   3. 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).

Figure:Scheduling messages with sync communication.



Higher prio P_i blocks on lower prio P_j to avoid cyclic wait (whether or not it is the intended sender or receiver of msg being scheduled)

- Before sending M to P_i , P_j requests permission in a nonblocking manner.

1. If a message M from a higher priority process arrives, it is processed by a receive (assuming receives are always enabled) and *ack*(M) is returned. Thus, a cyclic wait is prevented.

2. Also, while waiting for this permission, if a *request*(M) from a lower priority process arrives, a *permission*(M) is returned and the process blocks until M actually arrives.

- Note: receive(M^0) gets permuted with the send(M) event

6.4 Group communication

A *message broadcast* is the sending of a message to all members in the distributed system. The notion of a system can be confined only to those sites/processes participating in the joint application. Refining the notion of *broadcasting*, there is *multicasting* wherein a message is sent to a certain subset, identified as a *group*, of the processes in the system. At the other extreme is *unicasting*, which is the familiar point-to-point message communication.

Broadcast and multicast support can be provided by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information. However, the hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features such as the following:

- Application-specific ordering semantics on the order of delivery of messages.
- Adapting groups to dynamically changing membership.
- Sending multicasts to an arbitrary set of processes at each send event.
- Providing various fault-tolerance semantics.

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. Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms. Closed group algorithms cannot be used in several scenarios such as in a large system (e.g., on-line reservation or Internet banking systems) where client processes are short-lived and in large numbers. It is also worth noting that, for multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$, and algorithms that have to explicitly track the groups can incur this high overhead.

Two popular orders for the delivery of messages were proposed in the context of group communication: *causal order* and *total order*.

6.5 Causal order (CO)

Causal order has many applications such as updating replicated data, allo-cating requests in a fair manner, and synchronizing multimedia streams.

The use of causal order in updating replicas of a data item in the system.

Consider Figure 6.11(a), which shows two processes P_1 and P_2 that issue updates to the three replicas R1 d , R2 d , and R3 d of data item d. Message m creates a causality between send m1 and send m2 . If P_2 issues its update causally after P_1 issued its update, then P_2's update should be seen by the replicas after they see P_1's update, in order to preserve the semantics

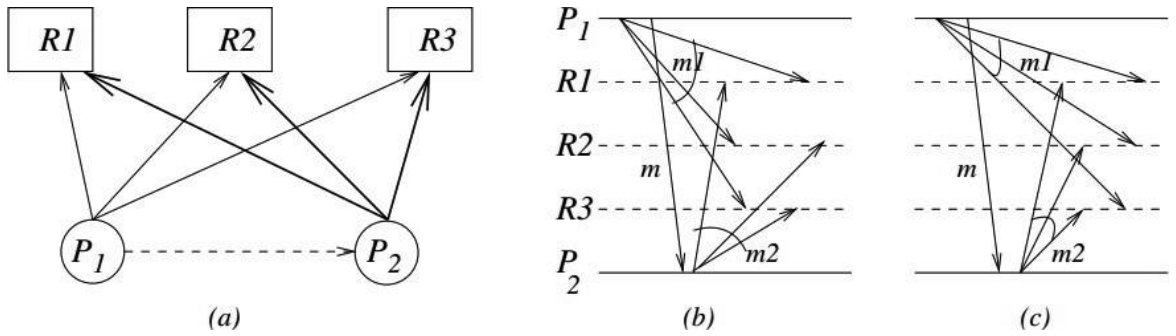


Figure 6.11: (a) Updates to 3 replicas. (b) Causal order (CO) and total order violated. (c) Causal order violated.

of the application. (In this case, CO is satisfied.) However, this may happen at some, all, or none of the replicas. Figure 6.11(b) shows that R1 sees P_2's update first, while R2 and R3 see P_1's update first. Here, CO is violated. Figure 6.11(c) shows that all replicas see P_2's update first. However, CO is still violated. If message m did not exist as shown, then the executions shown in Figure 6.11(b) and (c) would satisfy CO.

The following two 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 system wide 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.

The Raynal–Schiper–Toueg algorithm (RST)

(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) $DELIV[j] \longleftarrow DELIV[j] + 1$.

Assumptions/Correctness	Complexity
• FIFO channels. • Safety: Step (2a,b). • Liveness: assuming no failures, finite propagation times	• n^2 ints/ process • n^2 ints/ msg • Time per send and rcv event: n^2

Optimal KS Algorithm for CO: Principles

Delivery Condition for correctness:

Msg M that carries information "d ∈M.Dests", where message M was sent to d in the causal past of Send(M*), is not delivered to d if M has not yet been delivered to d .

Necessary and Sufficient Conditions for Optimality:

An optimal CO algorithm stores in local message logs and propagates on messages, information of the form "d is a destination of M" about a message sent in the causal past, *as long as* and *only as long as*:

(Propagation Constraint I) it is not known that the message M is delivered to d, and

(Propagation Constraint II) it is not known that a message has been sent to d in the causal future of Send M , and hence it is not guaranteed using a reasoning based on transitivity that the message M will be delivered to d in CO.

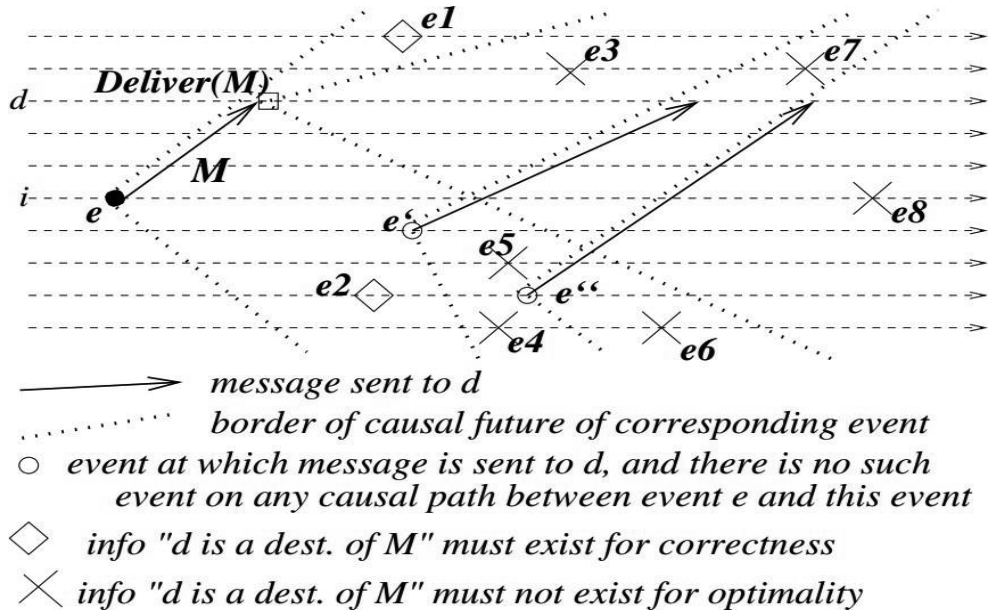
The Propagation Constraints also imply that if either (I) or (II) is false, the information "d ∈ M Dests" must *not* be stored or propagated, even to remember that (I) or (II) has been falsified.

Stated differently, the information "d ∈ $M_{i,a}$ Dests" must be available in the causal future of event $e_{i\,a}$, but:

- not in the causal future of $Deliver_d$ $M_{i\,a}$, and
- not in the causal future of $e_{k\,c}$, where d ∈ $M_{k,\,c}$ Dests and there is no other message sent causally between $M_{i,a}$ and $M_{k,\,c}$ to the same destination d.

In the causal future of $Deliver_d$ ($M_{i,a}$) , and Send($M_{k,c}$), the information is redundant; elsewhere, it is necessary. Additionally, to maintain optimality, no other information should be stored, including information about what messages have been delivered.

As information about what messages have been delivered (or are guaranteed to be delivered without violating causal order) is necessary for the Delivery Condition, this information is inferred using a set-operation based logic.



The message M is sent by process i at event e to process d. The information **"d ∈ M Dests":**

- **must exist at e1 and e2 because (I) and (II) are true;**
- **must not exist at e3 because (I) is false;**
- **must not exist at e4 e5 e6 because (II) is false;**
- **must not exist at e7 e8 because (I) and (II) are false.**

- Information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is ***explicitly* tracked** by the algorithm using (*source, timestamp, destination*) information.
- The information must be deleted as soon as either (i) or (ii) becomes false. The key problem in designing an optimal CO algorithm is to identify the events at which (i) or (ii) becomes false.
- Information about messages already delivered and messages guaranteed to be delivered in CO is ***implicitly* tracked** without storing or propagating it, and is derived from the explicit information.
- Such implicit information is used for determining when (i) or (ii) becomes false for the explicit information being stored or carried in messages.

Optimal KS Algorithm for CO: Code (1)

(local variables)

$clock_j \longleftarrow 0$; // local counter clock at node j

$SR_j[1 \ldots n] \longleftarrow \overline{0}$; // $SR_j[i]$ is the timestamp of last msg. from i delivered to j

$LOG_j = \{(i, clock_i, Dests)\} \longleftarrow \{\forall i, (i, 0, \emptyset)\}$;

// Each entry denotes a message sent in the causal past, by i at $clock_i$. $Dests$ is the set of remaining destinations
// for which it is not known that $M_{i,clock_i}$ (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

SND: j sends a message M to Dests:

1. $clock_j \longleftarrow clock_j + 1$;
2. **for all** $d \in M.Dests$ **do**:
   $O_M \longleftarrow LOG_j$; // O_M denotes $O_{M_{j,clock_j}}$
   for all $o \in O_M$, modify $o.Dests$ as follows:
   if $d \notin o.Dests$ **then** $o.Dests \longleftarrow (o.Dests \setminus M.Dests)$;
   if $d \in o.Dests$ **then** $o.Dests \longleftarrow (o.Dests \setminus M.Dests) \bigcup \{d\}$;
   // Do not propagate information about indirect dependencies that are
   // guaranteed to be transitively satisfied when dependencies of M are satisfied.
   for all $o_{s,t} \in O_M$ **do**
   if $o_{s,t}.Dests = \emptyset \bigwedge (\exists o'_{s,t'} \in O_M \mid t < t')$ **then** $O_M \longleftarrow O_M \setminus \{o_{s,t}\}$;
   // do not propagate older entries for which $Dests$ field is $\emptyset$
   send $(j, clock_j, M, Dests, O_M)$ to d;
3. **for all** $l \in LOG_j$ **do** $l.Dests \longleftarrow l.Dests \setminus Dests$;
   // Do not store information about indirect dependencies that are guaranteed
   // to be transitively satisfied when dependencies of M are satisfied.
   Execute $PURGE\_NULL\_ENTRIES(LOG_j)$; // purge $l \in LOG_j$ if $l.Dests = \emptyset$
4. $LOG_j \longleftarrow LOG_j \bigcup \{(j, clock_j, Dests)\}$.

Optimal KS Algorithm for CO: Code (2)

RCV: j **receives a message** $(k, t_k, M, Dests, O_M)$ **from** k**:**

1. // Delivery Condition; ensure that messages sent causally before M are delivered.
   for all $o_{m,t_m} \in O_M$ **do**
   if $j \in o_{m.t_m}.Dests$ **wait until** $t_m \leq SR_j[m]$;
2. Deliver M; $SR_j[k] \longleftarrow t_k$;
3. $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
   for all $o_{m,t_m} \in O_M$ **do** $o_{m,t_m}.Dests \longleftarrow o_{m,t_m}.Dests \setminus \{j\}$;
   // delete the now redundant dependency of message represented by o_{m,t_m} sent to j
4. // Merge O_M and LOG_j by eliminating all redundant entries.
   // Implicitly track "already delivered" & "guaranteed to be delivered in CO" messages.
   for all $o_{m,t} \in O_M$ **and** $l_{s,t'} \in LOG_j$ **such that** $s = m$ **do**
   if $t < t' \bigwedge l_{s,t} \notin LOG_j$ **then** mark $o_{m,t}$;
   // $l_{s,t}$ had been deleted or never inserted, as $l_{s,t}.Dests = \emptyset$ in the causal past
   if $t' < t \bigwedge o_{m,t'} \notin O_M$ **then** mark $l_{s,t'}$;
   // $o_{m,t'} \notin O_M$ because $l_{s,t'}$ had become $\emptyset$ at another process in the causal past
   Delete all marked elements in O_M and LOG_j ; // delete entries about redundant information
   for all $l_{s,t'} \in LOG_j$ **and** $o_{m,t} \in O_M$, **such that** $s = m \bigwedge t' = t$ **do**
   $l_{s,t'}.Dests \longleftarrow l_{s,t'}.Dests \bigcap o_{m,t}.Dests$; // delete destinations for which Delivery
   // Condition is satisfied or guaranteed to be satisfied as per $o_{m,t}$
   Delete $o_{m,t}$ from O_M; // information has been incorporated in $l_{s,t'}$
   $LOG_j \longleftarrow LOG_j \bigcup O_M$; // merge nonredundant information of O_M into LOG_j
5. $PURGE\_NULL\_ENTRIES(LOG_j)$. // Purge older entries l for which $l.Dests = \emptyset$

PURGE_NULL_ENTRIES(Log_j)**:** // Purge older entries l for which $l.Dests = \emptyset$ is implicitly inferred

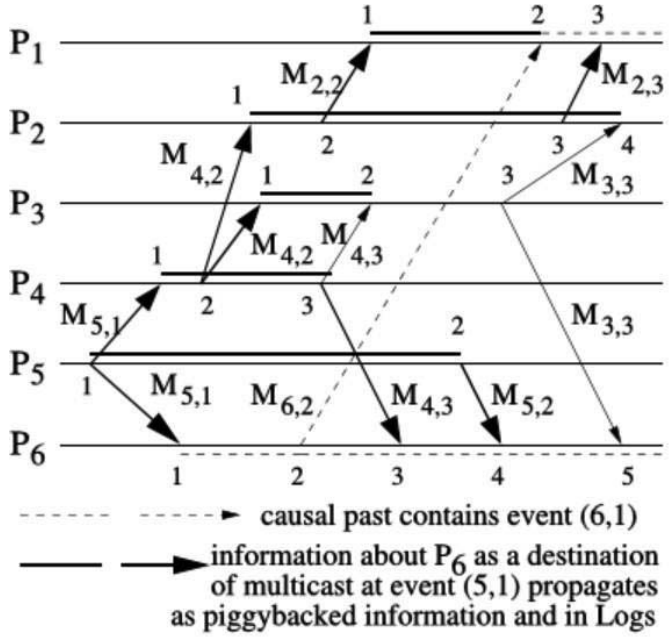
for all $l_{s,t} \in Log_j$ **do**

if $l_{s,t}.Dests = \emptyset \bigwedge (\exists l'_{s,t'} \in Log_j \mid t < t')$ **then** $Log_j \longleftarrow Log_j \setminus \{l_{s,t}\}$.

Information Pruning

- Explicit tracking of $(s, ts, dest)$ per multicast in Log and O_M
- Implicit tracking of msgs that are (i) delivered, or (ii) guaranteed to be delivered in CO:
  - (Type 1:) $\exists d \in M_{i,a}.Dests \mid d \notin l_{i,a}.Dests \bigvee d \notin o_{i,a}.Dests$
    - ⋆ When $l_{i,a}.Dests = \emptyset$ or $o_{i,a}.Dests = \emptyset$?
    - ⋆ Entries of the form l_{i,a_k} for $k = 1, 2, \ldots$ will accumulate
    - ⋆ Implemented in Step (2d)
  - (Type 2:) if $a_1 < a_2$ and $l_{i,a_2} \in LOG_j$, then $l_{i,a_1} \in LOG_j$. (Likewise for messages)
    - ⋆ entries of the form $l_{i,a_1}.Dests = \emptyset$ can be inferred by their absence, and should not be stored
    - ⋆ Implemented in Step (2d) and PURGE_NULL_ENTRIES

Example



Message to dest.	piggybacked $M_{5,1}$.Dests
$M_{5,1}$ to P_4,P_6	{P_4,P_6}
$M_{4,2}$ to P_3,P_2	{P_6}
$M_{2,2}$ to P_1	{P_6}
$M_{6,2}$ to P_1	{P_4}
$M_{4,3}$ to P_6	{P_6}
$M_{4,3}$ to P_3	{ }
$M_{5,2}$ to P_6	{P_4,P_6}
$M_{2,3}$ to P_1	{P_6}
$M_{3,3}$ to P_2,P_6	{ }

Figure 6.13: Tracking of information about $M_{5,1}.Dests$

2.6 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

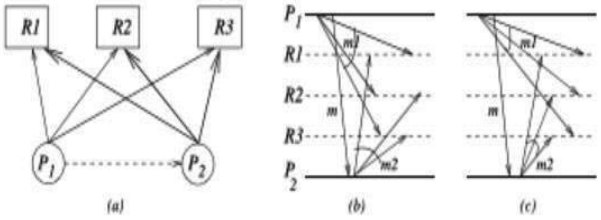
Solves coherence problem

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

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

Three-phase Algorithm

A distributed algorithm to implement total order and causal order of messages

The three phases of the algorithm are first described from the viewpoint of the sender, and then from the viewpoint of the receiver.

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).

Three-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.
```

Example and Complexity

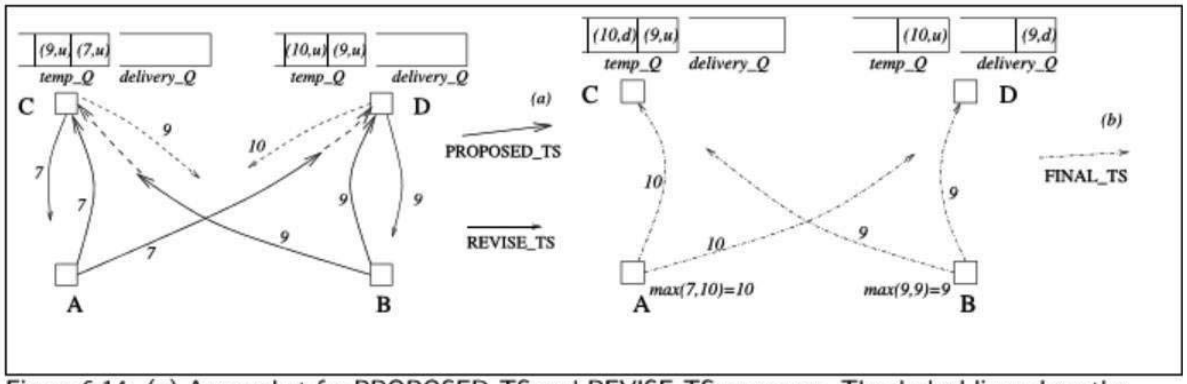


Figure 6.14: (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages.

Complexity:

This algorithm uses three phases, and, to send a message to n− 1 processes, **it uses 3(n− 1) messages and incurs a delay of three message hops.**

6.7 Global state and snapshot recording algorithms

Introduction

- **Recording the global state of a distributed system on-the-fly is an important**
- **paradigm.**
- **The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.**

System model

- The system consists of a collection of n processes p_1, p_2, ..., p_n that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- C_{ij} denotes the channel from process p_i to process p_j and its state is denoted by SC_{ij}.
- The actions performed by a process are modeled as three types of events: Internal events,the message send event and the message receive event.
- For a message m_{ij} that is sent by process p_i to process p_j, let $send(m_{ij})$ and $rec(m_{ij})$ denote its send and receive events.

- At any instant, the state of process p_i, denoted by LS_i, is a result of the sequence of all the events executed by p_i till that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process p_i to state LS_i.
- For a channel C_{ij}, the following set of messages can be defined based on the local states of the processes p_i and p_j

Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j \}$

Models of communication

Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- In FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: "For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \longrightarrow send(m_{kj})$, then $rec(m_{ij}) \longrightarrow rec(m_{kj})$".

Consistent global state

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state GS is defined as,

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij} \}$$

- A global state GS is a *consistent global state* iff it satisfies the following two conditions :

C1: $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$. ($\oplus$ is Ex-OR operator.)

C2: $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$.

- In a consistent global state, every message that is recorded as received is also recorded as sent. Such a global state captures the notion of causality that a message cannot be received if it was not sent.
- Consistent global states are meaningful global states and inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.

Interpretation in terms of cuts

- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
- Such a cut is known as a *consistent cut*.
- For example, consider the space-time diagram for the computation illustrated in Figure 4.1.
- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel C_{21}.

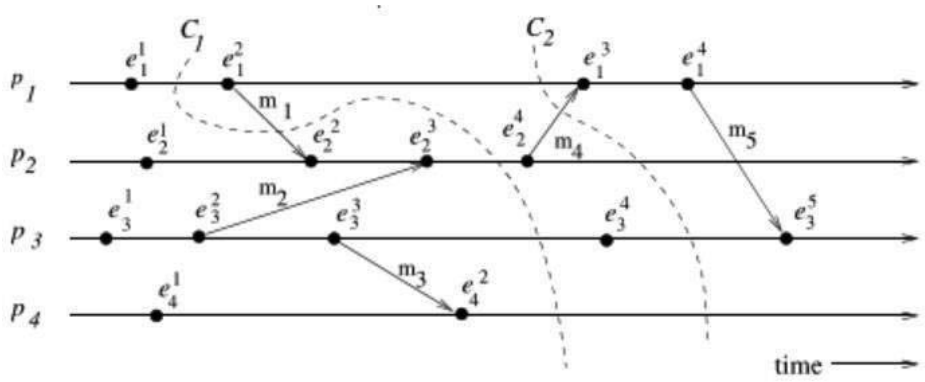


Figure 4.1: An Interpretation in Terms of a Cut.

Issues in recording a global state

The following two issues need to be addressed:

I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

-Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
-Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot.

-A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

6.8 Snapshot algorithms for FIFO channels

Chandy Lamport Algorithm

Chandy-Lamport algorithm

- The Chandy-Lamport algorithm uses a control message, called a *marker* whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a *marker*, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

- The algorithm can be initiated by any process by executing the "Marker Sending Rule" by which it records its local state and sends a marker on each outgoing channel.
- A process executes the "Marker Receiving Rule" on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the "Marker Sending Rule" to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

The algorithm

Marker Sending Rule for process *i*

1. Process *i* records its state.
2. For each outgoing channel C on which a marker has not been sent, *i* sends a marker along C before *i* sends further messages along C.

Marker Receiving Rule for process *j*

On receiving a marker along channel C:

if *j* has not recorded its state **then**
    Record the state of C as the empty set
    Follow the "Marker Sending Rule"
else
    Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

Correctness and Complexity

Correctness

- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij}, it acts as follows: If process p_j has not taken its snapshot yet, then it includes m_{ij} in its recorded snapshot. Otherwise, it records m_{ij} in the state of the channel C_{ij}. Thus, condition **C1** is satisfied.

Complexity

- The recording part of a single instance of the algorithm requires $O(e)$ messages and $O(d)$ time, where e is the number of edges in the network and d is the diameter of the network.

Properties of the recorded global state

- The recorded global state may not correspond to any of the global states that occurred during the computation.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
  - Therefore, a recorded global state is useful in detecting stable properties.

The recorded global state may not correspond to any of the global states that occurred during the computation. Consider two possible executions of the snapshot algorithm (shown in Figure 4.3) for the money transfer example of Figure 4.2:

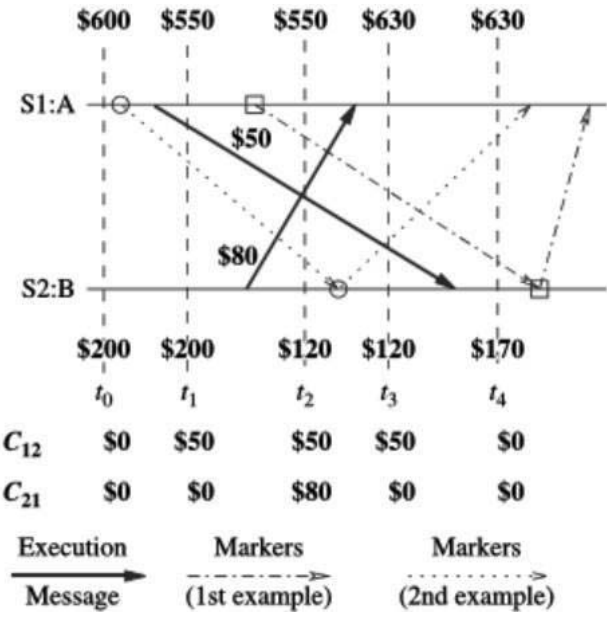


Figure 4.3 Timing diagram of two possible executions of the banking example.

(Markers shown using dashed-and-dotted arrows.) Let site S1 initiate the algorithm just after t_1. Site S1 records its local state (account A = $550) and sends a marker to site S2. The marker is received by site S2 after t_4. When site S2 receives the marker, it records its local state (account B = $170), the state of channel C_{12} as $0, and sends a marker along channel C_{21}. When site S1 receives this marker, it records the state of channel C_{21} as $80. The $800 amount in the system is conserved in the recorded global state,

$$A= \$550\ B = \$170\ C_{12} = \$0\ C_{21} = \$80$$

(Markers shown using dotted arrows.) Let site S1 initiate the algorithm just after t_0 and before sending the $50 for S2. Site S1 records its local state (account A = $600) and sends a marker to site S2. The marker is received by site S2 between t_2 and t_3. When site S2 receives the marker, it records its local state (account B = $120), the state of channel C_{12} as $0, and sends a marker along channel C_{21}. When site S1 receives this marker, it records the state of channel C_{21} as $80. The $800 amount in the system is conserved in the recorded global state,

$$A= \$600\ B = \$120\ C_{12} = \$0\ C_{21} = \$80$$

In both these possible runs of the algorithm, the recorded global states never occurred in the execution. This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.

A physical interpretation of the collected global state is as follows: consider the two instants of recording of the local states in the banking example. If the cut formed by these instants is viewed as being an elastic band and if the elastic band is stretched so that it is vertical, then recorded states of all processes occur simultaneously at one physical instant, and the recorded global state occurs in the execution that is depicted in this modified space– time diagram. This is called the **rubber-band criterion**.

QUESTIONS:

1. Explain Asynchronous execution with synchronous communication
2. Discuss Synchronous program order on an asynchronous system
3. Explain the Algorithm for binary rendezvous (or) Bagrodia's Algorithm.
4. Dicuss the Raynal–Schiper–Toueg algorithm (RST) (2.5.1)
5. Explain group communication in detail.
6. Explain Optimal KS Algorithm for CO: (2.5.2)
7. Explain the distributed algorithm to implement total order and causal order of messages (or) Three-phase Algorithm
8. Explain Snapshot algorithms for FIFO channels or Chandy Lamport Algorithm