ROHINI COLLEGE OF ENGINEERING & TECHNOLOGY
Near Anjugramam Junction, Kanyakumari Main Road, Palkulam, Variyoor P.O - 629401
Kanyakumari Dist, Tamilnadu., E-mail : admin@rcet.org.in, Website : www.rcet.org.in

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NAME OF THE SUBJECT : DISTRIBUTED SYSTEMS

Subject code : CS8603

Regulation : 2017

UNIT-II MESSAGE ORDERING & SNAPSHOTS

UNIT II CS8603
MESSAGE ORDERING & SNAPSHOTS

MESSAGE ORDERING & SNAPSHOTS
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

2.1 MESSAGE ORDERING AND GROUP COMMUNICATION

As the distributed systems are a network of systems at various physical locations, the coordination between them should always be preserved. The message ordering means the order of delivering the messages to the intended recipients. The common message order schemes are First in First out (FIFO), non FIFO, causal order and synchronous order. In case of group communication with multicasting, the causal and total ordering scheme is followed. It is also essential to define the behaviour of the system in case of failures. The following are the notations that are widely used in this chapter:

- Distributed systems are denoted by a graph (N, L).
- The set of events are represented by event set {E, $\prec$ }
- Message is denoted as m^i: send and receive events as s^i and r^i respectively.
- Send (M) and receive (M) indicates the message M send and received.
- a~b denotes a and b occurs at the same process
- The send receive pairs T={(s, r) εE_i x $E_j$corresponds to r}

2.1.1 Message Ordering Paradigms

The message orderings are

(i) non-FIFO
(ii) FIFO
(iii) causal order
(iv) synchronous order

There is always a trade-off between concurrency and ease of use and implementation.

Asynchronous Executions

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

- There cannot be any causal relationship between events in asynchronous execution.
- The messages can be delivered in any order even in non FIFO.
- Though there is a physical link that delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logicallink may be formed as a composite of physical links and multiple paths mayexist between the two end points of the logical link.

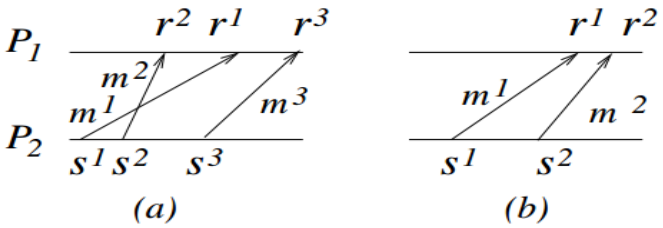


Fig 2.1: a) FIFO executions b) non FIFO executions

FIFO executions

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

- The logical link is non-FIFO.
- FIFO logical channels can be realistically assumed when designing distributed algorithms since most of the transport layer protocols follow connection oriented service.
- A FIFO logical channel can be created over a non-FIFO channel by using a 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.

Causally Ordered (CO) executions

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

- 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.
- If s and s' are not related by causality, then CO is vacuously(blankly)satisfied.
- Causal order is used in applications that update shared data, distributed shared memory, or fair resource allocation.
- 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.**
- No message overtaken by a chain of messages between the same (sender, receiver) pair.

If send(m^1) $\prec$ send(m^2) then for each common destination d of messages m^1 and m^2, deliverd(m^1) $\prec$deliverd(m^2) must be satisfied.

Other properties of causal ordering

1. **Message Order (MO):** A MO execution is an A-execution in which, for all

$$(s, r) \text{ and } (s', r') \in \mathcal{T}, s \prec s' \Longrightarrow \neg(r' \prec r).$$

2. **Empty Interval Execution:** An execution (E $\prec$) is an empty-interval (EI) execution if for each pair of events (s, r) $\in$ T, the open interval set

$$\{x \in E \mid s \prec x \prec r\}$$

in the partial order is empty.

3. 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:
   $$e \prec r \Longrightarrow \neg(s \prec e)$$
   - weak common future:
   $$s \prec e \Longrightarrow \neg(e \prec r).$$

Synchronous Execution

- When all the communication between pairs of processes uses synchronous send and receives primitives, the resulting order is the synchronous order.
- The synchronous communication always involves a handshake between the receiver and the sender, the handshake events may appear to be occurring instantaneously and atomically.
- The instantaneous communication property of synchronous executions requires a modified definition of the causality relation because for each (s, r) $\in$ T, the send event is not causally ordered before the receive event.
- The two events are viewed as being atomic and simultaneous, and neither event precedes the other.

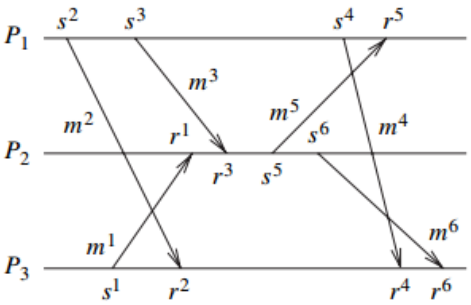


Fig 2.2 a) Execution in an asynchronous system

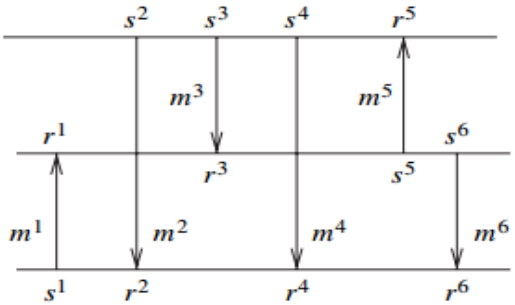


Fig 2.2 b) Equivalent synchronous communication

Causality 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 << y.

S2: If (s, r $\in$ T, then) for all x $\in$ E, [(x<< s $\Longleftrightarrow$ x<<r) and (s<< x $\Longleftrightarrow$ r<< x)].

S3: If x<<y and y<<z, then x<<z.

Synchronous execution: A synchronous execution or S-execution is an execution (E, <<) for which the causality relation << is a partial order.

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 $T(e_i) < T(e_i')$.***

2.2 Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous send and receive primitives, the resulting order is synchronous order. The algorithms run on asynchronous systems will not work in synchronous system and vice versa is also true**.**

Realizable Synchronous Communication (RSC)

A-execution can be realized under synchronous communication is called a realizable with synchronous communication (RSC).

- An execution can be modeled to give a total order that extends the partial order (E, ≺).
- In an A-execution, the messages can be made to appear instantaneous if there exist a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event in this linear extension.

Non-separated linear extension is an extension of (E, ≺) is a linear extension of (E, ≺) such that for each pair (s, r) ∈ T, the interval $\{ x \in E \ \ s \prec x \prec r \}$ is empty.

A A-execution (E, ≺) is an RSC execution if and only if there exists a non-separated linear extension of the partial order (E, ≺).

- In the non-separated linear extension, if the adjacent send event and its corresponding receive event are viewed atomically, then that pair of events shares a common past and a common future with each other.

Crown

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

The crown is $<(s^1, r^1) (s^2, r^2)>$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$. Cyclic dependencies may exist in a crown. The crown criterion states that an A-computation is RSC, i.e., it can be realized on a system with synchronous communication, if and only if it contains no crown.

Timestamp criterion for RSC execution

An execution (E, ≺) is RSC 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 (a, b) in $(E \times E) \setminus \mathcal{T}$, $a \prec b \Longrightarrow T(a) < T(b)$

2.2.1 Hierarchy of ordering paradigms

The orders of executions are:

- Synchronous order (SYNC)
- Causal order (CO)
- FIFO order (FIFO)

- Non FIFO order (non-FIFO)

The Execution order have the following results

- For an A-execution, A is RSC if and only if A is an S-execution.
- RSC ⊂ CO ⊂ FIFO ⊂ A
- This hierarchy is illustrated in Figure 2.3(a), and example executions of each class are shown side-by-side in Figure 2.3(b)
- The above hierarchy implies that some executions belonging to a class X will not belong to any of the classes included in X. The degree of concurrency is most in A and least in SYNC.
- A program using synchronous communication is easiest to develop and verify.
- A program using non-FIFO communication, resulting in an A execution, is hardest to design and verify.

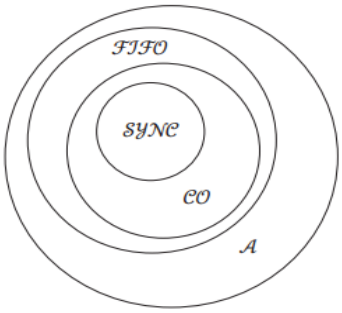


Fig (a)

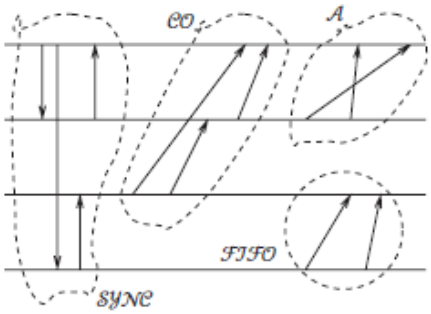


Fig (b)

Fig 2.3: Hierarchy of execution classes

2.2.3 Simulations

- The events in the RSC execution are scheduled as per some non-separated linear extension, and adjacent (s, r) events in this linear extension are executed sequentially in the synchronous system.
- The partial order of the asynchronous execution remains unchanged.
- If an A-execution is not RSC, then there is no way to schedule the events to make them RSC, without actually altering the partial order of the given A-execution.
- However, the following indirect strategy that does not alter the partial order can be used.
- Each channel $C_{i,j}$ is modeled by a control process $P_{i,j}$ that simulates the channel buffer.
- An asynchronous communication from i to j becomes a synchronous communication from i to $P_{i,j}$ followed by a synchronous communication from $P_{i,j}$ to j.
- This enables the decoupling of the sender from the receiver, a feature that is essential in asynchronous systems.

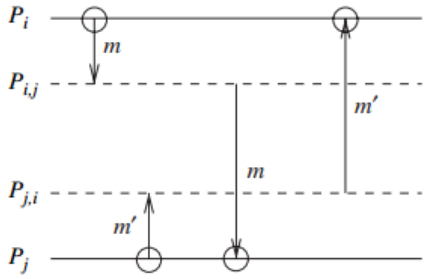


Fig 2.4: Modeling channels as processes to simulate an execution using asynchronous primitives on synchronous system

Synchronous programs on asynchronous systems

- A (valid) S-execution can be trivially realized on an asynchronous system by scheduling the messages in the order in which they appear in the S-execution.
- The partial order of the S-execution remains unchanged but the communication occurs on an asynchronous system that uses asynchronous communication primitives.
- Once a message send event is scheduled, the middleware layer waits for acknowledgment; after the ack is received, the synchronous send primitive completes.

2.3 SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

Non deterministic programs

The partial ordering of messages in the distributed systems makes the repeated runs of the same program will produce the same partial order, thus preserving deterministic nature. But sometimes the distributed systems exhibit non determinism:

- A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.
- Multiple send and receive calls which are enabled at a process can be executed in an interchangeable order.
- If i sends to j, and j sends to i concurrently using blocking synchronous calls, there results a deadlock.
- There is no semantic dependency between the send and the immediately following receive at each of the processes. If the receive call at one of the processes can be scheduled before the send call, then there is no deadlock.

2.3.1 Rendezvous

Rendezvous systems are a form of synchronous communication among an arbitrary number of asynchronous processes. All the processes involved meet with each other, i.e., communicate synchronously with each other at one time. Two types of rendezvous systems are possible:

- Binary rendezvous: When two processes agree to synchronize.
- Multi-way rendezvous: When more than two processes agree to synchronize.

Features of binary rendezvous:

- 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 receives commands that are both enabled. The communication events for the control messages under the covers do not alter the partial order of the execution.

2.3.2 Binary rendezvous algorithm

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.
- Schedule in a deadlock-free manner (i.e., crown-free).
- Schedule to satisfy the progress property in addition to the safety property.

Steps in Bagrodia algorithm

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 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 message (M) types used are: M, ack(M), request(M), and permission(M). 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 control messages. The messages request(M), ack(M), and permission(M) use M's unique tag; the message M is not included in these messages.

(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) Pi seeks permission from Pj 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 messageM'. WhenM' 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 Pj 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).

Fig 2.5: Bagrodia Algorithm

2.4 GROUP COMMUNICATION

Group communication is done by broadcasting of messages. A message broadcast is the sending of a message to all members in the distributed system. The communication may be

- **Multicast:** A message is sent to a certain subset or a group.
- **Unicasting:** A point-to-point message communication.

The network layer protocol cannot provide the following functionalities:

- 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.
- The multicast algorithms can be open or closed group.

Differences between closed and open group algorithms:

Closed group algorithms	Open group algorithms
If sender is also one of the receiver in the multicast algorithm, then it is closed group algorithm.	If sender is not a part of the communication group, then it is open group algorithm.
They are specific and easy to implement.	They are more general, difficult to design and expensive.
It does not support large systems where client processes have short life.	It can support large systems.

2.5 CAUSAL ORDER (CO)

In the context of group communication, there are two modes of communication: *causal order and total order*. Given a system with FIFO channels, causal order needs to be explicitly enforced by a protocol. 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. 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.

2.5.1 The Raynal–Schiper–Toueg algorithm

- Each message M should carry a log of all other messages sent causally before M's send event, and sent to the same destination dest(M).
- The Raynal–Schiper–Toueg algorithm canonical algorithm is a representative of several algorithms that reduces the size of the local space and message space overhead by various techniques.
- This log can then be examined to ensure whether it is safe to deliver a message.
- All algorithms aim to reduce this log overhead, and the space and time overhead of maintaining the log information at the processes.
- To distribute this log information, broadcast and multicast communication is used.
- The hardware-assisted or network layer protocol assisted multicast cannot efficiently provide features:
  - 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

2.6 Causal Order (CO)

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 messageM 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.

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.

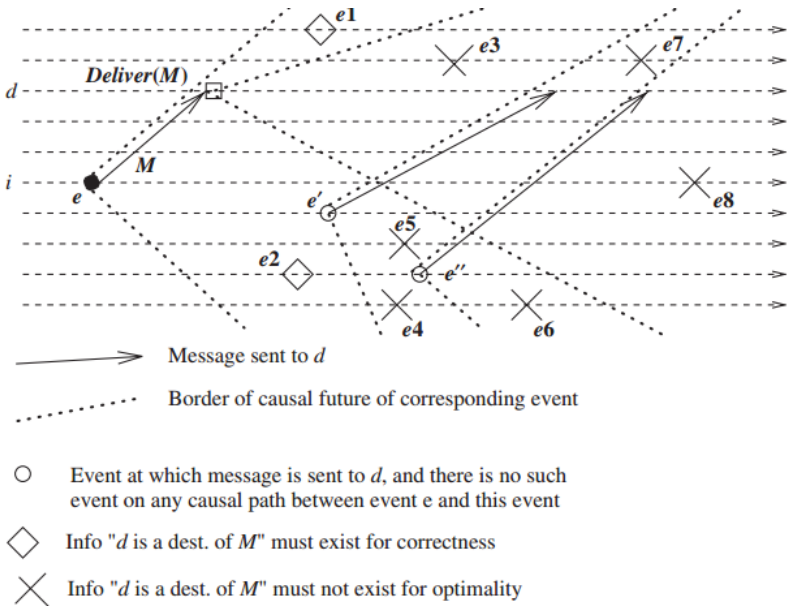


Fig 2.6: Conditions for causal ordering

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:

- not in the causal future of $Deliver_d(M_1, a)$
- not in the causal future of $e_{k,c}$ where $d \in M_{k,c}Dests$ and there is no other message sent causally between M_i,a and $M_{k,c}$ to the same destination d.

Information about messages:
(i) not known to be delivered
(ii) not guaranteed to be delivered in CO, is explicitly tracked by the algorithm using (source, timestamp, destination) information.
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. The algorithm for the send and receive operations is given in Fig. 2.7 a) and b). Procedure SND is executed atomically. Procedure RCV is executed atomically except for a possible interruptionin line 2a where a non-blocking wait is required to meet the Delivery Condition.

(1) **SND:** j **sends a message** M **to** $Dests$:

(1a) $clock_j \longleftarrow clock_j + 1$;
(1b) **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;
(1c) **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 $l.Dests = \emptyset$
(1d) $LOG_j \longleftarrow LOG_j \bigcup \{(j, clock_j, Dests)\}$.

Fig 2.7 a) Send algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

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

(2a) // 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** $t_m \leq SR_j[m]$;
(2b) Deliver M; $SR_j[k] \longleftarrow t_k$;
(2c) $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
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
(2d) // 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 non-redundant information of O_M into LOG_j
(2e) $PURGE\_NULL\_ENTRIES(LOG_j)$. // 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

Fig 2.7 b) Receive algorithm by Kshemkalyani–Singhal to optimally implement causal ordering

The data structures maintained are sorted row–major and then column–major:

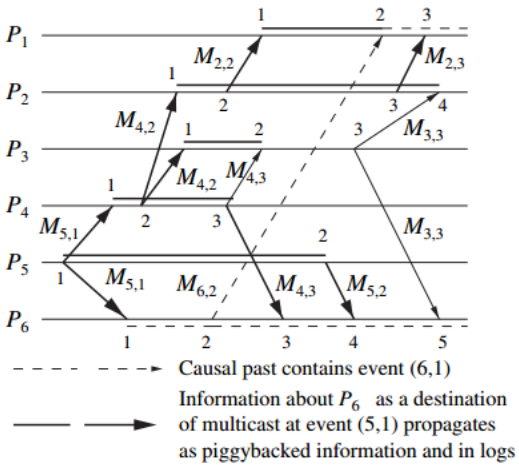
1. Explicit tracking:

- Tracking of (source, timestamp, destination) information for messages (i) not known to be delivered and (ii) not guaranteed tobe delivered in CO, is done explicitly using the I.Dests field of entries inlocal logs at nodes and o.Dests field of entries in messages.
- Sets $l_{i,a}$Destsand $o_{i,a.}$ Dests contain explicit information of destinations to which $M_{i,a}$is not guaranteed to be delivered in CO and is not known to be delivered.
- The information about d $\in M_{i,a}$ .Destsis propagated up to the earliestevents on all causal paths from (i, a) at which it is known that $M_{i,a}$ isdelivered to d or is guaranteed to be delivered to d in CO.

2. Implicit tracking:

- Tracking of messages that are either (i) already delivered, or (ii) guaranteed to be delivered in CO, is performed implicitly.

- The information about messages (i) already delivered or (ii) guaranteed tobe delivered in CO is deleted and not propagated because it is redundantas far as enforcing CO is concerned.
- It is useful in determiningwhat information that is being carried in other messages and is being storedin logs at other nodes has become redundant and thus can be purged.
- Thesemantics are implicitly stored and propagated. This information about messages that are (i) already delivered or (ii) guaranteed to be delivered in CO is tracked without explicitly storing it.
- The algorithm derives it from the existing explicit information about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, by examining only $o_{i,a}$Dests or $l_{i,a}$Dests, which is a part of the explicit information.



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 $P_{2,6}$	{ }

Fig 2.8: Illustration of propagation constraints

Multicasts $M_{5,1}$and $M_{4,1}$

Message $M_{5,1}$ sent to processes P4 and P6 contains the piggybacked information $M_{5,1}$. Dest= {P4, P6}. Additionally, at the send event (5, 1), the information $M_{5,1}$.Dests = {P4,P6} is also inserted in the local log Log_5. When $M_{5,1}$ is delivered to P6, the (new) piggybacked information P4 ∈ $M_{5,1}$ .Dests is stored in Log_6 as $M_{5,1}$.Dests ={P4} information about P6 ∈ $M_{5,1}$.Dests which was needed for routing, must not be stored in Log_6 because of constraint I. In the same way when $M_{5,1}$ is delivered to process P4 at event (4, 1), only the new piggybacked information P6 ∈ $M_{5,1}$ .Dests is inserted in Log_4 as $M_{5,1}$.Dests =P6which is later propagated duringmulticast $M_{4,2}$.

Multicast $M_{4,3}$

At event (4, 3), the information P6 ∈$M_{5,1}$.Dests in Log4 is propagated onmulticast $M_{4,3}$only to process P6 to ensure causal delivery using the DeliveryCondition. The piggybacked information on message $M_{4,3}$sent to process P3must not contain this information because of constraint II. As long as any future message sent to P6 is delivered in causal order w.r.t. $M_{4,3}$sent to P6, it will also be delivered in causal order w.r.t. $M_{5,1}$. And as $M_{5,1}$ is already delivered to P4, the information $M_{5,1}$Dests = ∅ is piggybacked on $M_{4,3}$ sent to P 3. Similarly, the information P6 ∈ $M_{5,1}$Dests must be deleted from Log4 as it will no longer be needed, because of constraint II. $M_{5,1}$Dests = ∅ is stored in Log4 to remember that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to all its destinations.

Learning implicit information at P2 and P3

When message $M_{4,2}$is received by processes P2 and P3, they insert the (new) piggybacked information in their local logs, as information $M_{5,1}$.Dests = P6. They both continue to store this in Log2 and Log3 and propagate this information on multicasts until they learn at events (2, 4) and (3, 2) on receipt of messages $M_{3,3}$and $M_{4,3}$, respectively, that any future message is expected to be delivered in causal order to process P6, w.r.t. $M_{5,1}$sent toP6. Hence by constraint II, this information must be deleted from Log2 andLog3. The flow of events is given by;

- When $M_{4,3}$ with piggybacked information $M_{5,1}$Dests = ∅ is received byP3at (3, 2), this is inferred to be valid current implicit information aboutmulticast $M_{5,1}$because the log Log3 already contains explicit informationP6 ∈$M_{5,1}$.Dests about that multicast. Therefore, the explicit informationin Log3 is inferred to be old and must be deleted to achieve optimality. $M_{5,1}$Dests is set to ∅ in Log3.
- The logic by which P2 learns this implicit knowledge on the arrival of $M_{3,3}$is identical.

Processing at P6

When message $M_{5,1}$ is delivered to P6, only $M_{5,1}$.Dests = P4 is added to Log6. Further, P6 propagates only $M_{5,1}$.Dests = P4 on message $M_{6,2}$, and this conveys the current implicit information $M_{5,1}$ has been delivered to P6 by its very absence in the explicit information.

- When the information P6 ∈ $M_{5,1}$Dests arrives on $M_{4,3}$, piggybacked as $M_{5,1}$.Dests = P6 it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 (constraint I) – further, the presence of $M_{5,1}$.Dests = P4 in Log6 implies the implicit information that $M_{5,1}$ has already been delivered to P6. Also, the absence of P4 in $M_{5,1}$.Dests in the explicit piggybacked information implies the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to P4, and, therefore, $M_{5,1}$. Dests is set to ∅ in Log6.
- When the information P6 ∈ $M_{5,1}$.Dests arrives on $M_{5,2}$ piggybacked as $M_{5,1}$. Dests = {P4, P6} it is used only to ensure causal delivery of $M_{4,3}$ using the Delivery Condition, and is not inserted in Log6 because Log6 contains $M_{5,1}$.Dests = ∅, which gives the implicit information that $M_{5,1}$ has been delivered or is guaranteed to be delivered in causal order to both P4 and P6.

Processing at P1

- When $M_{2,2}$arrives carrying piggybacked information $M_{5,1}$.Dests = P6 this (new) information is inserted in Log1.
- When $M_{6,2}$arrives with piggybacked information $M_{5,1}$.Dests ={P4}, P1learns implicit information $M_{5,1}$has been delivered to P6 by the very absence of explicit information P6 ∈ $M_{5,1}$.Dests in the piggybacked information, and hence marks information P6 ∈ $M_{5,1}$Dests for deletion from Log1. Simultaneously, $M_{5,1}$Dests = P6 in Log1 implies the implicit information that $M_{5,1}$has been delivered or is guaranteed to be delivered in causal order to P4.Thus, P1 also learns that the explicit piggybacked information $M_{5,1}$.Dests = P4 is outdated. $M_{5,1}$.Dests in Log1 is set to ∅.
- The information "P6 ∈$M_{5,1}$.Dests piggybacked on $M_{2,3}$,which arrives at P 1, is inferred to be outdated usingthe implicit knowledge derived from $M_{5,1}$.Dest= ∅" in Log1.

2.7 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.

Centralized Algorithm for total ordering

Each process sends the message it wants to broadcast to a centralized process, which relays all the messages it receives to every other process over FIFO channels.

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

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

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

Complexity: Each message transmission takes two message hops and exactly n messages in a system of n processes.

Drawbacks: A centralized algorithm has a single point of failure and congestion, and is not an elegant solution.

Three phase distributed algorithm

Three phases can be seen in both sender and receiver side.

Sender side

Phase 1

- In the first phase, a process multicasts the message M with a locally unique tag and the local timestamp to the group members.

Phase 2

- 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 is non-blocking.

Phase 3

- The process multicasts the final timestamp to the group.

```
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).
```

Fig 2.9: Sender side of three phase distributed algorithm

Receiver Side

Phase 1

- The receiver receives the message with a tentative timestamp. It updates the variable priority that tracks the highest proposed timestamp, 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. In the queue, the entry is marked as undeliverable.

Phase 2

- The receiver sends the revised timestamp back to the sender. The receiver then waits in a non-blocking manner for the final timestamp.

Phase 3

- The final timestamp is received from the multicaster. The corresponding message entry in temp_Q is identified using the tag, and is marked as deliverable after the revised timestamp is overwritten by the final timestamp.
- The queue is then resorted using the timestamp field of the entries as the key. 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.

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

2.8 GLOBAL STATE AND SNAPSHOT RECORDING ALGORITHMS

- A distributed computing system consists of processes that do not share a common memory and communicate asynchronously with eachother by message passing.
- Each component ofhas a local state. The state of the process is the local memory and a history of its activity.
- The state of achannel is characterized by the set of messages sent along the channel lessthe messages received along the channel. The global state of a distributedsystem is a collection of the local states of its components.
- If shared memory were available, an up-to-date state of the entire systemwould be available to the processes sharing the memory.
- The absence ofshared memory necessitates ways of getting a coherent and complete view ofthe system based on the local states of individual processes.
- A meaningfulglobal snapshot can be obtained if the components of the distributed systemrecord their local states at the same time.
- This would be possible if thelocal clocks at processes were perfectly synchronized or if there were aglobal system clock that could be instantaneously read by the processes.
- If processes read time froma single common clock, various indeterminatetransmission delays during the read operation will cause the processes toidentify various physical instants as the same time.

2.8.1 System Model

- The system consists of a collection of n processes, p_1, p_2,…,p_n that are connected by channels.
- Let C_{ij} denote the channel from process p_i to process p_j.
- Processes and channels have states associated with them.
- The state of a process at any time is defined by the contents of processor registers, stacks, local memory, etc., and may be highly dependent on the local context of the distributed application.
- The state of channel C_{ij}, denoted by SC_{ij}, is given by the set of messages in transit in the channel.
- The events that may happen are: internal event, send (send (m_{ij})) and receive (rec(m_{ij})) events.
- The occurrences of events cause changes in the processstate.
- A **channel** is a distributed entity and its state depends on the local states of the processes on which it is incident.

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

- The transit function records the state of the channel C_{ij}.
- In the FIFO model, each channel acts as a first-in first-out message queue and, thus, message ordering is preserved by a channel.
- In the non-FIFO model, achannel acts like a set in which the sender process adds messages and thereceiver process removes messages from it in a random order.

2.8.2 A consistent global state

The global state of a distributed system is a collection of the local states ofthe processes and the channels. The global state is given by:

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

The two conditions for global state are:

C1: $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$

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

Condition 1 preserves **law of conservation of messages.**Condition C2 states that in the collected global state, for everyeffect, its cause must be present.

Law of conservation of messages: Every messagem_{ij}that is recorded as sent in the local state of a process p_i must be capturedin the state of the channel C_{ij} or in the collected local state of the receiver process p_j.

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

2.8.3 Interpretation of cuts

- Cuts in a space–time diagram provide a powerful graphical aid in representingand reasoning about the global states of a computation. A cut is a line joiningan arbitrary point on each process line that slices the space–time diagraminto a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every messagereceived in the PAST of the cut has been sent in the PAST of that cut. Sucha cut is known as a consistent cut.
- In a consistent snapshot, all the recorded local states of processes are concurrent; that is, the recorded local state of no process casuallyaffects the recorded local state of any other process.

2.8.4 Issues in recording global state

The non-availability of global clock in distributed system, raises the following issues:

Issue 1:

How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?

Answer:

- 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, mustnot be recorded in the global snapshot (from C2).

Issue 2:

How to determine the instant when a process takes its snapshot?

The answer

Answer:

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.

2.9 SNAPSHOT ALGORITHMS FOR FIFO CHANNELS

Each distributed application has number of processes running on different physical servers. These processes communicate with each other through messaging channels.

A snapshot captures the local states of each process along with the state of each communication channel.

Snapshots are required to:

- Checkpointing
- Collecting garbage
- Detecting deadlocks
- Debugging

2.9.1Chandy–Lamport algorithm

- The algorithm will record a global snapshot for each process channel.
- The Chandy-Lamport algorithm uses a control message, called a marker.
- Aftera site has recorded its snapshot, it sends a marker along all of its outgoingchannels before sending out any more messages.
- Since channels are FIFO, amarker separates the messages in the channel into those to be included in the snapshot from those not to be recorded inthe snapshot.
- This addresses issue I1. The role of markers in a FIFO systemis to act as delimiters for the messages in the channels so that the channelstate recorded by the process at the receiving end of the channel satisfies thecondition C2.

Marker sending rule for process p_i

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

Marker receiving rule for process p_j
On receiving a marker along channel C:
if p_j has not recorded its state **then**
Record the state of C as the empty set
Execute the "marker sending rule"
else
Record the state of C as the set of messages received along C after $p_{j's}$ state was recorded and before p_j received the marker along C

Fig 2.10: Chandy–Lamport algorithm

Initiating a snapshot

- Process P_i initiates the snapshot
- P_i records its own state and prepares a special marker message.
- Send the marker message to all other processes.
- Start recording all incoming messages from channels C_{ij} for j not equal to i.

Propagating a snapshot

- For all processes $P_j$consider a message on channel C_{kj}.

- If marker message is seen for the first time:
  - $P_j$records own sate and marks C_{kj} as empty
  - Send the marker message to all other processes.
  - Record all incoming messages from channels C_{lj} for 1 not equal to j or k.
  - Else add all messages from inbound channels.

Terminating a snapshot

- All processes have received a marker.
- All process have received a marker on all the N-1 incoming channels.
- A central server can gather the partial state to build a global snapshot.

Correctness of the algorithm

- Since a process records its snapshot when itreceives the first marker on any incoming channel, no messages that followmarkers on the channels incoming to it are recorded in the process's snapshot.
- A process stops recording the state of an incoming channel whena marker is received on that channel.
- Due to FIFO property of channels, itfollows that no message sent after the marker on that channel is recorded inthe 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: ifprocess 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 thediameter of the network.

2.9.2 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.