UNIT II LOGICAL TIME AND GLOBAL STATE

Logical Time: Physical Clock Synchronization: NTP – A Framework for a System of Logical Clocks – Scalar Time – Vector Time; Message Ordering and Group Communication: Message Ordering Paradigms – Asynchronous Execution with Synchronous Communication – Synchronous Program Order on Asynchronous System – Group Communication – Causal Order – Total Order; Global State and Snapshot Recording Algorithms: Introduction – System Model and Definitions – SnapshotAlgorithms for FIFO Channels.

Logical Time

Definition

A system of logical clocks consists of a time domain T and a logical clock C. Elements of T form a partially ordered set over a relation <. This relation is usually called the happened before or causal precedence. Intuitively, this relation is analogous to the earlier than relation provided by the physical time. The logical clock C is a function that maps an event e in a distributed system to an element in the time domain T, denoted as C(*e*) and called the timestamp of *e*, and is defined as follows:

$$C : H \mapsto T,$$

such that the following property is satisfied: for two events e_i and e_j , $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$. This monotonicity property is called the clock consistency condition.

When T and C satisfy the following condition, for two events

$$e_i \text{ and } e_j ,\ e_i \rightarrow e_j \Leftrightarrow C(e_i) <$$

$C(e_j)$,the system of clocks is said to be strongly consistent.

Implementing logical clocks

Implementation of logical clocks requires addressing two issues: data structures local to every process to represent logical time and a protocol (set of rules) to update the data structures to ensure the consistency condition.

Each process p_i maintains data structures that allow it the following two capabilities:

1. A *local logical clock*, denoted by lc_i, that helps process p_i measure its own progress.

2. A *logical global clock*, denoted by gc_i, that is a representation of process p_i's local view of the logical global time. It allows this process to assign consistent timestamps to its local events. Typically, lc_i is a part of gc_i.

The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently. The protocol consists of the following two rules:

1. R1 This rule governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal).
2. R2 This rule governs how a process updates its global logical clock to update its view of the global time and global progress. It dictates what information about the logical time is piggybacked in a message and how this information is used by the receiving process to update its view of the global time.

Scalar time

Definition:

The scalar time representation was proposed by Lamport in 1978 as an attempt to totally order events in a distributed system. Time domain in this representation is the set of non-negative integers. The logical local clock of a process pi and its local view of the global time are squashed into one integer variable C_i.

Rules **R1** and **R2** to update the clocks are as follows:

1. **R1** Before executing an event (send, receive, or internal), process p_i executes the following:

$$C_i := C_i + d \qquad (d > 0)$$

In general, every time **R1** is executed, *d* can have a different value, and this value may be application-dependent. However, typically *d* is kept at 1 because this is able to identify the time of each event uniquely at a process, while keeping the rate of increase of *d* to its lowest level.

2. **R2** Each message piggybacks the clock value of its sender at sending time. When a process p_i receives a message with timestamp C_{msg}, it executes the following actions:

   1. $C_i := max(C_i, C_{msg},);$
   2. execute **R1**;
   3. deliver the message.

Figure shows the evolution of scalar time with d=1.

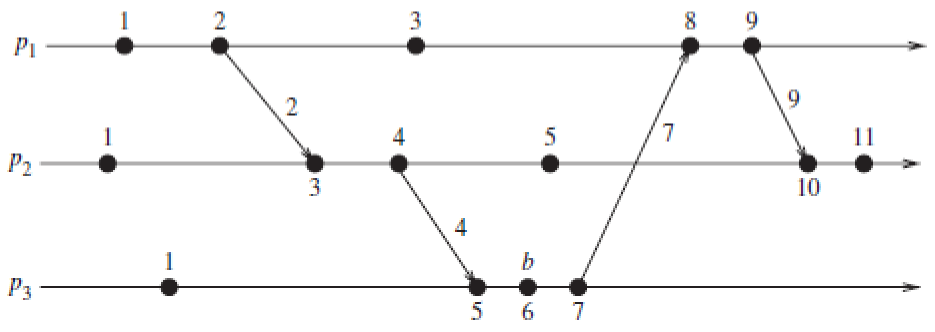


Figure: Evolution of scalar time

Basic properties

Consistency property

Clearly, scalar clocks satisfy the monotonicity and hence the consistency property:

for two events e_i and e_j, $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$.

Total Ordering

Scalar clocks can be used to totally order events in a distributed system. The main problem in totally ordering events is that two or more events at different processes may have an identical timestamp.

Event counting

If the increment value d is always 1, the scalar time has the following interesting property: if event e has a timestamp h, then h−1 represents the minimum logical duration, counted in units of events, required before producing the event e; we call it the height of the event e.

No strong consistency

The system of scalar clocks is not strongly consistent; that is, for two events e_i and

e_j, $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$

Vector time

Definition

The system of vector clocks was developed independently by Fidge, Mattern, and Schmuck. In the system of vector clocks, the time domain is represented by a set of n-dimensional non-negative integer vectors.

Each process p_i maintains a vector $vt_i[1\ldots n]$, where $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i. vt_i $[j]$ represents process p_i's latest knowledge of process p_j local time. If vt_i $[j] = x$, then process p_i knows that local time at process p_j has progressed till x. The entire vector vt_i constitutes p_i's view of the global logical time and is used to timestamp events.

Process p_i uses the following two rules R1 and R2 to update its clock:

1. R1 Before executing an event, process p_i updates its local logical time as follows:

$$vt_i[i] := vt_i[i] + d \qquad (d > 0).$$

2. R2 Each message m is piggybacked with the vector clock vt of the sender process at sending time. On the receipt of such a message *(m,vt)*, process p_i executes the following sequence of actions:
   1. update its global logical time as follows:

   $$1 \le k \le n \ : \ vt_i[k] := max(vt_i[k], vt[k]);$$

   2. execute R1;
   3. deliver the message *m*.

The timestamp associated with an event is the value of the vector clock of its process when the event is executed. Figure shows an example of vector clocks progress with the increment value d = 1. Initially, a vector clock is [0,0,0,….].

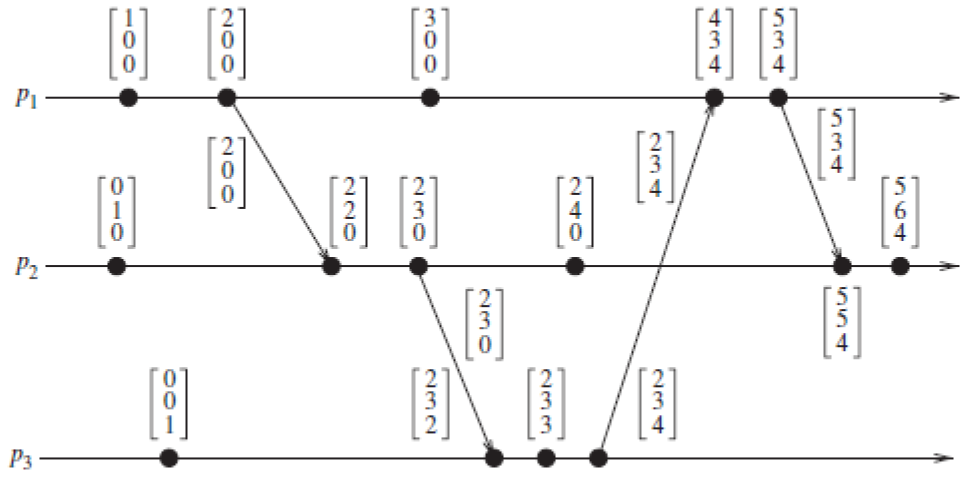


Figure: Evolution of vector time

The following relations are defined to compare two vector timestamps, *vh* and *vk*:

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \parallel vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh).$$

Basic Properties

Isomorphism

If events in a distributed system are timestamped using a system of vector clocks, we have the following property. If two events *x* and *y* have timestamps *vh* and *vk*, respectively, then

$$x \rightarrow y \Leftrightarrow vh < vk$$

$$x \parallel y \Leftrightarrow vh \parallel vk.$$

Thus, there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps. This is a very powerful, useful, and interesting property of vector clocks.

Strong consistency

The system of vector clocks is strongly consistent; thus, by examining the vector timestamp of two events, we can determine if the events are causally related.

Event counting

If *d* is always 1 in rule R1, then the *i*th component of vector clock at process *pi*, *vti[i]*, denotes the number of events that have occurred at pi until that instant. So, if an event e has timestamp *vh*, *vh*[j] denotes the number of events executed by process *pj* that causally precede *e*.

Applications

Since vector time tracks causal dependencies exactly, it finds a wide variety of applications. For example, they are used in distributed debugging, implementations of causal ordering communication and causal distributed shared memory, establishment of global breakpoints, and in determining the consistency of checkpoints in optimistic recovery.

Size of vector clocks

- A vector clock provides the latest known local time at each other process. If this information in the clock is to be used to explicitly track the progress at every other process, then a vector clock of size *n* is necessary.
- A popular use of vector clocks is to determine the causality between a pair of events. Given any events *e* and *f,* the test for $e \prec f$ if and only if $T(e) < T(f)$, which requires a comparison of the vector clocks of *e* and *f*. Although it appears that the clock of size *n* is necessary, that is not quite accurate. It can be shown that a size equal to the dimension of the partial order $(E,\prec)$ is necessary, where the upper bound on this dimension is *n*.

Physical clock synchronization: NTP

In distributed systems, there is no global clock or common memory. Each processor has its own internal clock and its own notion of time. In practice, these clocks can easily drift apart by several seconds per day, accumulating significant errors over time. Also, because different clocks tick at different rates, they may not remain always synchronized although they might be synchronized when they start.

Some practical examples that stress the need for synchronization are listed below:

- In database systems, the order in which processes perform updates on a database is important to ensure a consistent, correct view of the database. To ensure the right ordering of events, a common notion of time between co-operating processes becomes imperative.

- It is quite common that distributed applications and network protocols use timeouts, and their performance depends on how well physically dispersed processors are time-synchronized. Design of such applications is simplified when clocks are synchronized.

Clock synchronization is the process of ensuring that physically distributed processors have a common notion of time. It has a significant effect on many problems like secure systems, fault diagnosis and recovery, scheduled operations, database systems, and real-world clock values.

Due to different clocks rates, the clocks at various sites may diverge with time, and periodically a clock synchronization must be performed to correct this clock skew in distributed systems. Clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time). Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed physical clocks.

Definitions and terminology

We provide the following definitions. C_a and C_b are any two clocks.

- **Time:** The time of a clock in a machine p is given by the function $C_p(t)$, where $C_p(t) = t$ for a perfect clock.
- **Frequency:** Frequency is the rate at which a clock progresses. The frequency at time t of clock C_a is $C_a'(t)$
- **Offset**: Clock offset is the difference between the time reported by a clock and the real time. The offset of the clock C_a is given by $C_a(t) - t$. The offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$.
- **Skew:** The skew of a clock is the difference in the frequencies of the clock and the perfect clock. The skew of a clock C_a relative to clock C_b at time t is $C_a'(t) - C_b'(t)$.

- **Drift (rate)**: The drift of clock C_a is the second derivative of the clock value with respect to time, namely, $C''_a(t)$. The drift of clock C_a relative to clock C_b at time t is $C''(t) - C''(t)$

  a b

Clock inaccuracies

Physical clocks are synchronized to an accurate real-time standard like UTC (Universal Coordinated Time).

However, due to the clock inaccuracy discussed above, a timer (clock) is said to be working within its specification if

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho,$$

where constant ρ is the maximum skew rate specified by the manufacturer.

Offset delay estimation method

The *Network Time Protocol (NTP)*, which is widely used for clock synchronization on the Internet, uses the the *offset delay estimation* method. The design of NTP involves a hierarchical tree of time servers. The primary server at the root synchronizes with the UTC. The next level contains secondary servers, which act as a backup to the primary server. At the lowest level is the synchronization subnet which has the clients.

Clock offset and delay estimation

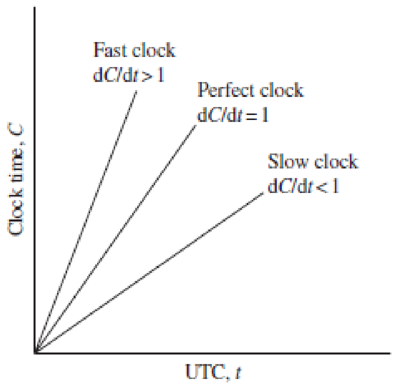


Figure: The behavior of fast, slow, and perfect clocks with respect to UTC

In practice, a source node cannot accurately estimate the local time on the target node due to varying message or network delays between the nodes. This protocol employs a very common practice of performing several trials and chooses the trial with the minimum delay. Recall that Cristian's remote clock reading method also relied on the same strategy to estimate message delay.

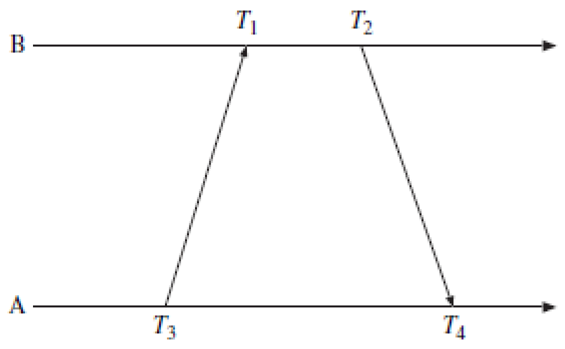


Figure: Offset and delay estimation

Figure shows how NTP timestamps are numbered and exchanged between peers A and B. Let T1, T2, T3, T4 be the values of the four most recent timestamps as shown. Assume that clocks A and B are stable and running at the same speed. Let a = T1 − T3 and b = T2 − T4. If the network delay difference from A to B and from B to A, called differential delay, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T4 are approximately given by the following:

$$\theta = \frac{a+b}{2}, \qquad \delta = a - b.$$

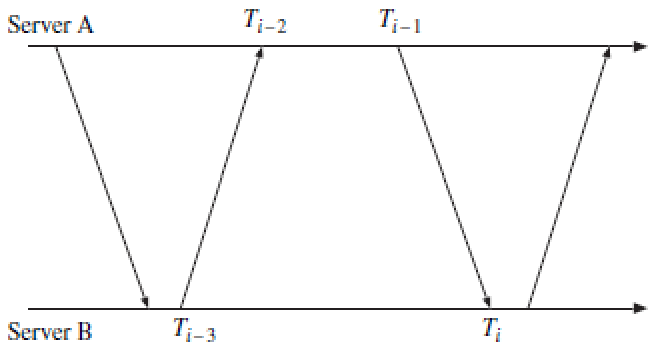


Figure: Timing diagram for the two servers

Each NTP message includes the latest three timestamps T1, T2, and T3, while T4 is determined upon arrival. Thus, both peers A and B can independently calculate delay and offset using a single bidirectional message stream as shown in Figure.

Message ordering and group communication

Message ordering paradigms

The order of delivery of messages in a distributed system is an important aspect of systemexecutions because it determines the messaging behavior that can be expected by the distributed program. Distributed program logic greatly depends on this order of delivery.

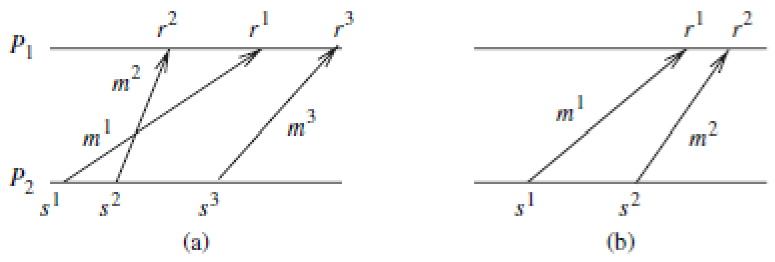
Several orderings on messages have been defined:

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

Asynchronous executions

An asynchronous execution (or A-execution) is an execution $(E, \prec)$ for which the causalityrelation 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. Although each physical link typically delivers the messages sent on it in FIFO order due to the physical properties of the medium, a logical link may be formed as a composite of physical links and multiple paths may exist between the two end points of the logical link. As an example, the mode of ordering at the Network Layer in connectionless networks such as IPv4 is non-FIFO. Thefollowing Figure (a) illustrates an A-execution under non-FIFO ordering.



a) An A-execution that is not a FIFO execution.

(b) An A-execution that is also a FIFO

FIFO executions

A FIFO execution is an A-execution in which,

for all (s, r) and $(s', r') \in$ T, $(s \sim s' and\ r \sim r' and\ s \prec$ s$') \Rightarrow r \prec$ r$'$

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

A simple algorithm to implement a FIFO logical channel over a non-FIFO channel woulduse 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 abuffer to order the incoming messages as per the sender's sequence numbers, and accepts only the"next" message in sequence. The above Figure (b) illustrates an A-execution under FIFO ordering.

Causally ordered (CO) executions

A CO execution is an A-execution in which,

for all (s, r) and $(s', r') \in$ T, $(r \sim r' and\ s \prec$ s$') \Rightarrow r \prec$ r$'$

If two send events s and s' are related by causality ordering (not physical time ordering), then acausally ordered execution requires that their corresponding receive events r and r' occur in thesame order at all common destinations. Note that if s and s' are not related by causality, then CO is vacuously satisfied because the antecedent of the implication is false.

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 distributedmutual exclusion.

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 thatwere 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 deliveryevent (instead of as a receive event) for emphasis.

Definition of causal order (CO) for implementations

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

Observe that if the definition of causal order is restricted so that m^1 and m^2 are sent by thesame process, then the property degenerates into the FIFO property. In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes. The FIFO property which applies on a per-logical channel basis can be extended globally to give the CO property. In a CO execution, no message can be overtaken by a chain of messages between the same (sender, receiver) pair of processes.

Message order (MO)

A MO execution is an A-execution in which,

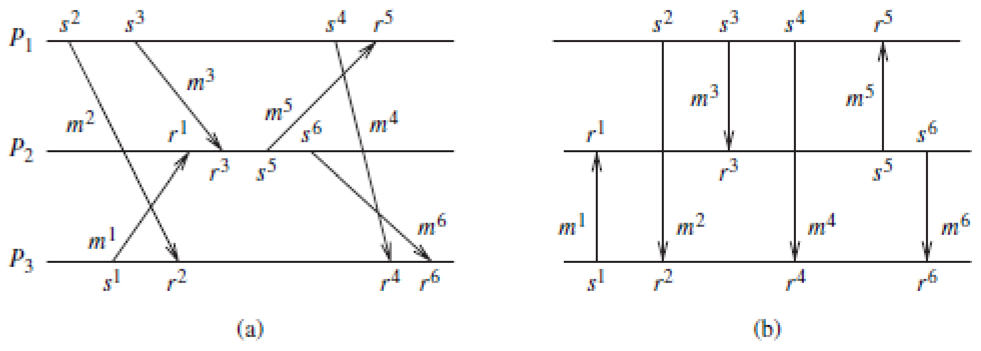
$$\text{for all } (s, r) \text{ and } (s', r') \in T \; s \prec s' \Rightarrow \neg(r \prec r')$$

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 $\{x \in E | s \prec x \prec r\}$ in the partial order is empty.

Synchronous execution (SYNC)

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



a) Execution in an Asynchronous system

b) Equivalent instantaneous communication

In a timing diagram, the "instantaneous" message communication can be shown by bidirectional vertical message lines. Figure (a) shows a synchronous execution on an asynchronoussystem. Figure (b) shows the equivalent timing diagram with the corresponding instantaneous message communication.

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.

Causality in a synchronous execution

The synchronous causality relation $\ll$ on E is the smallest transitive relation that satisfies the following:

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

Synchronous execution

A synchronous execution (or S-execution) is an execution (E, $\ll$) for which the causalityrelation $\ll$ 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 $T(e_i) < T(e_i')$.

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

Asynchronous execution with synchronous communication

When all the communication between pairs of processes is by using synchronous sendand 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, $\prec$).
- 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 followedby its corresponding receive event in this linear extension.

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

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

- 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,..., 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 berealized on a system with synchronous communication, if and only if it contains no crown.

Timestamp criterion for RSC execution

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

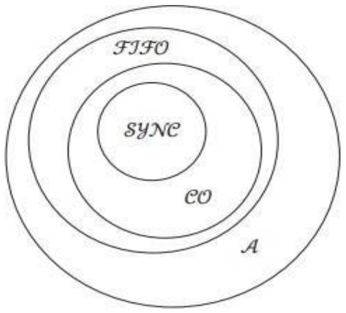
Hierarchy of ordering paradigms

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 belongto any of the classes included in X. The degree of concurrency is most in A andleast inSYNC.
- 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.



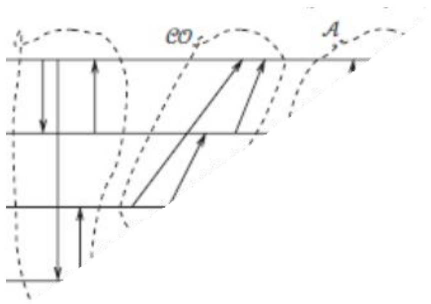


Fig (a) **Fig (b)**

Fig 2.3: Hierarchy of execution classes

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 essentialin asynchronous systems.

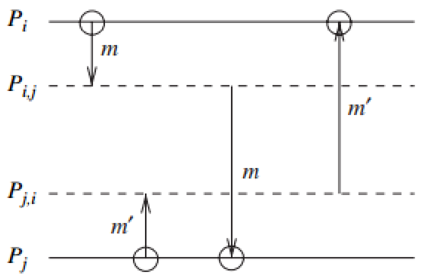


Fig 2.4: Modeling channels as processes to simulate an execution using asynchronous primitives on synchronous system

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.

SYNCHRONOUS PROGRAM ORDER ON AN ASYNCHRONOUS SYSTEM

Non deterministic programs

The partial ordering of messages in the distributed systems makes the repeated runs ofthe 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.

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 systemsare 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 notalter the partial order of the execution.

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), andpermission(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) Pi wants to execute SEND(M) to a lower priority process Pj:

Pi executes *send*(M) and blocks until it receives *ack*(M) from Pj . 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) Pi wants to execute SEND(M) to a higher priority process Pj:

(2a) Pi seeks permission from Pj by executing *send*(*request*(M)).

// to avoid deadlock in which cyclically blocked processes queue //

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

(i) If a message M' arrives from a higher priority process Pk, Pi accepts M' by scheduling aRECEIVE(M') event and then executes *send*(*ack*(M')) to Pk.

(ii) If a *request*(M') arrives from a lower priority process Pk, Pi executes *send*(*permission*(M')) to Pk andblocks waiting for the messageM'. WhenM' arrives, the RECEIVE(M') event is executed.

(2c) When the *permission*(M) arrives, Pi knows partner Pj is synchronized and Pi executes *send*(M). TheSEND(M) now completes.

(3) *request*(M) arrival at Pi from a lower priority process Pj:

At the time a *request*(M) is processed by Pi, process Pi 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 Pi from a higher priority process Pj:

At the time a message *M* is processed by Pi, process Pi executes RECEIVE(M) (which is assumed to bealways enabled) and then *send*(*ack*(M)) to Pj .

(5) Processing when Pi is unblocked:

When Pi is unblocked, it dequeues the next (if any) message from the queue and processes it as amessage arrival (as per rules 3 or 4).

Fig 2.5: Bagrodia Algorithm

GROUP COMMUNICATION

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

- **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 andexpensive.
It does not support large systems where clientprocesses have short life.	It can support large systems.

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 arrivesat 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 ofa message is transparent to the application process. The delivery event corresponds tothe 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

- 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 overheadby 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

Causal Order (CO)

An optimal CO algorithm stores in local message logs and propagates on messages, informationof 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.

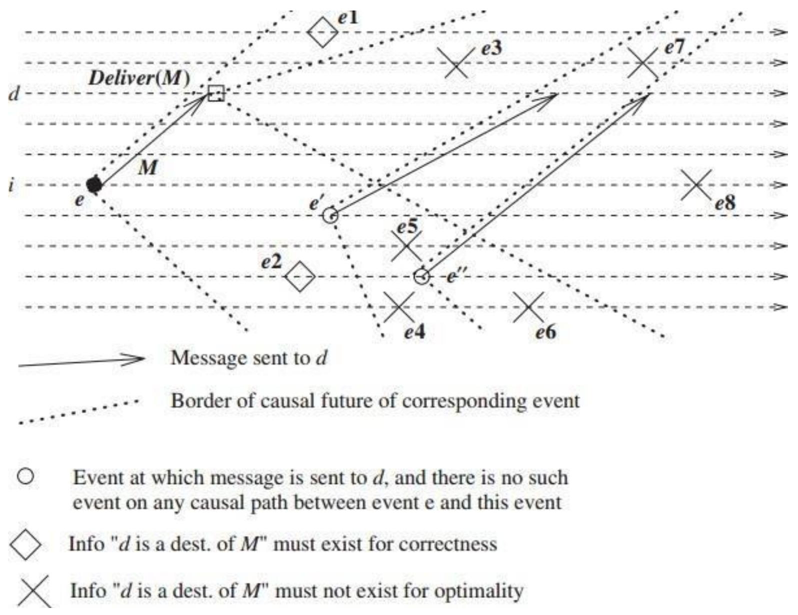

e1
e3
e7
Deliver(M)
d
M
i
e
e'
e8
e5
e2
e''
e4
e6
Message sent to d
Border of causal future of corresponding event
Event at which message is sent to d, and there is no such event on any causal path between event e and this event
Info "d is a dest. of M" must exist for correctness
Info "d is a dest. of M" must not exist for optimality

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:

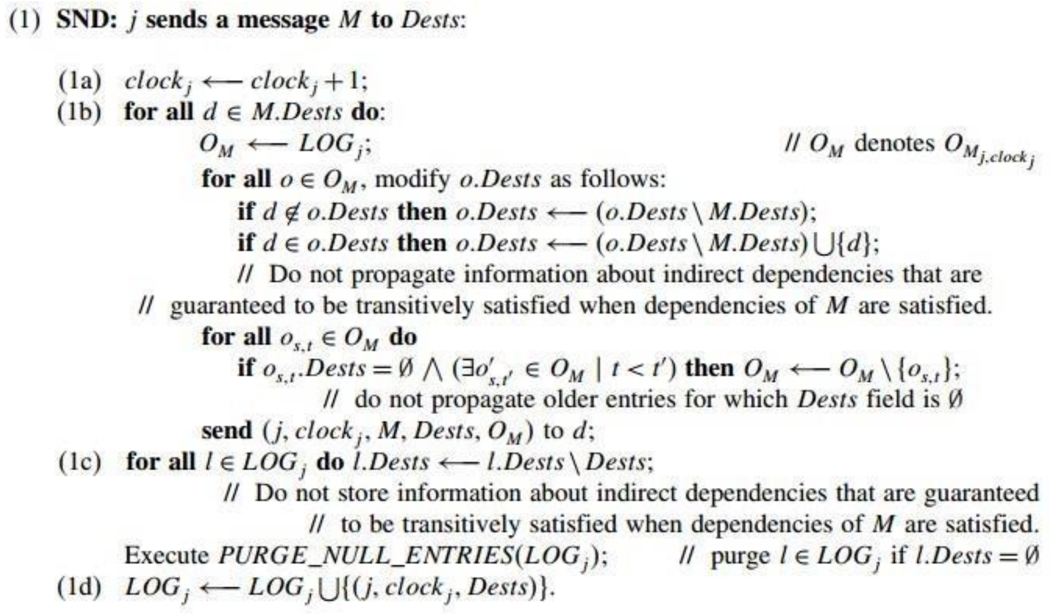
- not in the causal future of Deliver$_d$(M_1, a)
- not in the causal future of e $_{k,\ c}$ where d ∈$M_{k,c}$Dests and there is no othermessage sent causally between $M_{i,a}$ and $M_{k,\ c}$ to the same destination d.

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

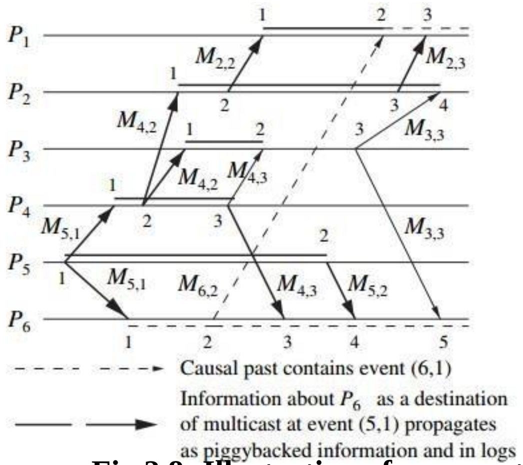
The data structures maintained are sorted row–major and then column–major:

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 deliveredin 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 andis 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.


P_1
P_2
P_3
P_4
P_5
P_6
$M_{2,2}$
$M_{2,3}$
$M_{4,2}$
$M_{3,3}$
$M_{4,3}$
$M_{5,1}$
$M_{6,2}$
$M_{5,2}$
Causal past contains event (6,1)
Information about P_6 as a destination of multicast at event (5,1) propagates as piggybacked information and in logs

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 M5,1and M4,1

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 M4,3

At event (4, 3), the information P6 ∈$M_{5,1}$.Dests in Log_4 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 guaranteedto 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 ∈ M5,1.Dests in the piggybacked information, and hence marks information P6 ∈ M5,1Dests for deletion from Log1. Simultaneously, M5,1Dests = P6 in Log1 implies the implicit information that M5,1has been delivered or is guaranteed to be delivered in causal order to P4.Thus, P1 also learns that the explicit piggybacked information M5,1.Dests = P4 is outdated. M5,1.Dests in Log1 is set to ∅.

- The information "P6 ∈M5,1.Dests piggybacked on M2,3,which arrives at P 1, is inferred to be outdated usingthe implicit knowledge derived from M5,1.Dest= ∅" in Log1.

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 deliveredto 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, whichrelays 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 messagesin a system of n processes.

Drawbacks: A centralized algorithm has a single point of failure and congestion, and isnot an elegant solution.

Three phase distributed algorithm

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

Sender side

Phase 1

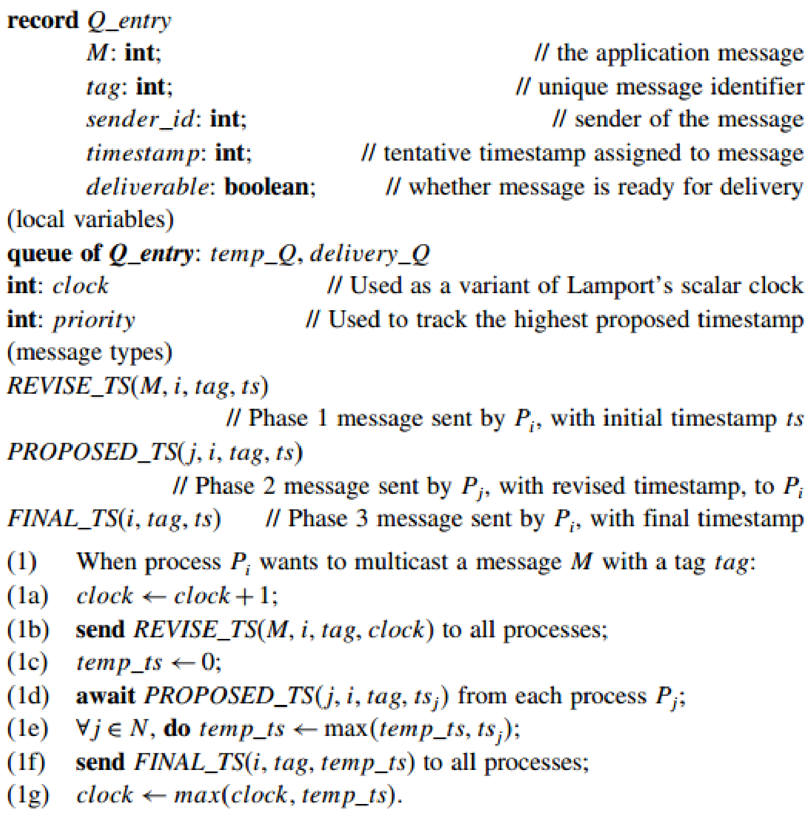
- In the first phase, a process multicasts the message M with a locally unique tag and the local timestamp to the group members.

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)   $clock \leftarrow clock + 1$;
(1b)   send REVISE_TS(M, i, tag, clock) to all processes;
(1c)   $temp\_ts \leftarrow 0$;
(1d)   await PROPOSED_TS(j, i, tag, $ts_j$) from each process $P_j$;
(1e)   $\forall j \in N$, do $temp\_ts \leftarrow \max(temp\_ts, ts_j)$;
(1f)   send FINAL_TS(i, tag, temp_ts) to all processes;
(1g)   $clock \leftarrow max(clock, temp\_ts)$.
```

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 waitsin 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

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 ahistory 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 isa 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 ifthere 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.

System Model

- The system consists of a collection of n processes, $p_1, p_2, \ldots, p_n$ that are connectedby 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 thedistributed 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 theprocesses 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 addsmessages and thereceiver process removes messages from it in a random order.

A consistent global state

The global state of a distributed system is a collection of the local states ofthe processesand 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:

$$\textbf{C1:}\ send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$$
$$\textbf{C2:}\ send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j.$$

Condition 1 preserves **law of conservation of messages.**Condition C2 states that in thecollected 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.

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 anyother process.

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 notto be recorded?

Answer:

2.8.4.1 Any message that is sent by a process before recording its snapshot,must berecorded in the global snapshot (from C1).

2.8.4.2 Any message that is sent by a process after recording its snapshot, mustnot berecorded in the global snapshot (from C2).

Issue 2:

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

Answer:

A process pj must record its snapshot before processing a message mij that was sent byprocess pi after recording its snapshot.

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

Chandy–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

outgoingchannelsbefore sending out any more messages.

- Since channels are FIFO, amarker separates the messages in the channel into those tobe 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 forthe 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 Pi initiates the snapshot
- Pi 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 Cij for j not equal to i.

Propagating a snapshot

- For all processes Pjconsider a message on channel Ckj.
- If marker message is seen for the first time:
  - Pjrecords own sate and marks Ckj as empty
  - Send the marker message to all other processes.
  - Record all incoming messages from channels Clj 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 asfollows: 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 thenetwork.

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 sentare 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 recordedglobal snapshot.

Therefore, a recorded global state is useful in detecting stable properties.