2.5 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.

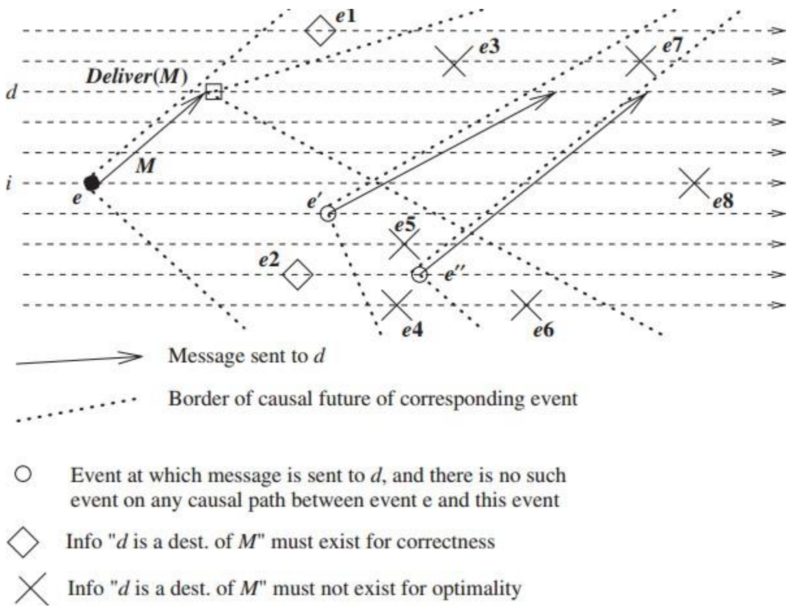


Fig 2.6: Conditions for causal ordering

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:

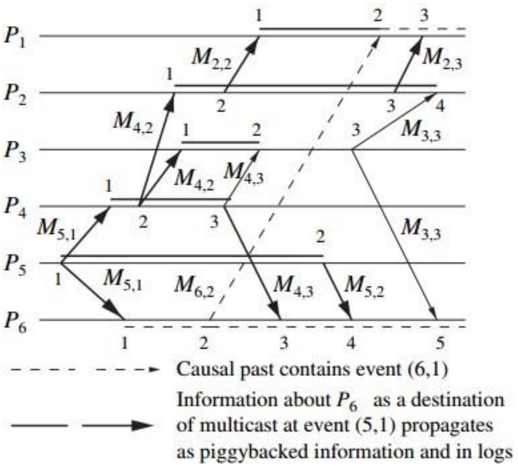
1. Explicit tracking:

- 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 ∈$M_{i,a}$ .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.

2.6 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) $clock \leftarrow clock + 1$;
(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 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