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 inan interchangeable order.
- If i sends to j, and j sends to i concurrently using blocking synchronous calls, thereresults 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 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 receivecommands 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.

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 execute 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 Pi is waiting for permission, it remains unblocked.

(i) If a message M' arrives from a higher priority process P_k, Pi accepts M' by scheduling a RECEIVER(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 Pk and blocks waiting for the message M'. when M' arrives, the RECEIVER (M') event is executed.

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

(3) request(M) arrival at Pi from a higher priority process P_j:

At the time a request(M) is processed by pi process pi executes send(permission(M)) toPj and blocks waiting for the message M. when M arrives the RECEIVE(M) event isexecuted 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, proess P_i executed RECEIVE(M) (which is assumed to be always enabled) and then send(ack(M)) to P_j.

(5) Processing when Pi is unblocked:

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

Fig : Bagrodia Algorithm

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.

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.

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

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.

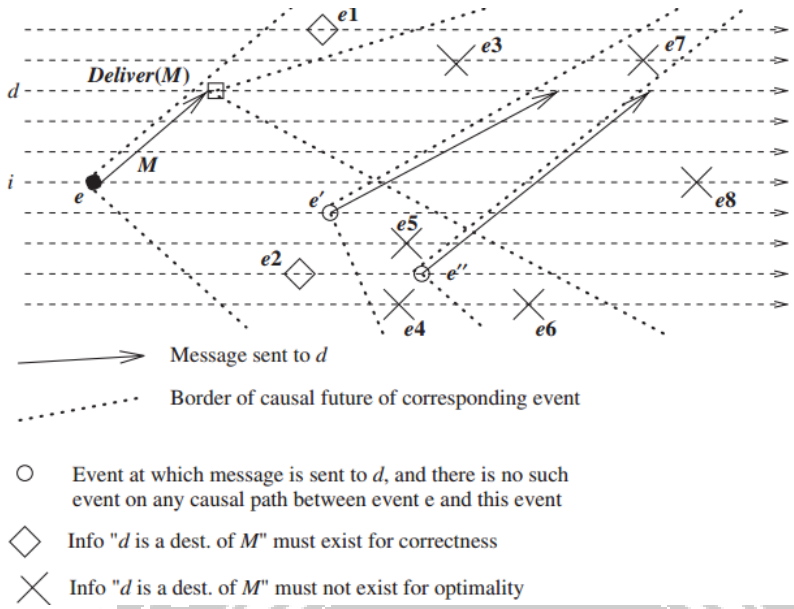


Fig : Conditions for causal ordering

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

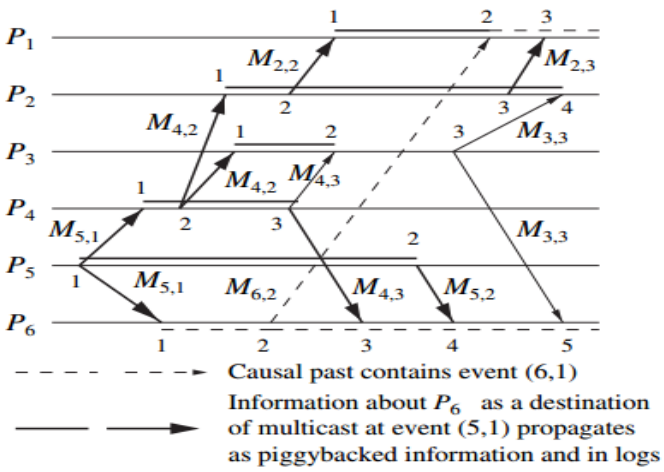
The data structures maintained are sorted row–major and then column–major:

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 : 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 $P4 \in M_{5,1}$.Dests is stored in Log_6 as $M_{5,1}$.Dests ={P4} information about $P6 \in M_{5,1}$.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 $P6 \in M_{5,1}$ .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 $P6 \in M_{5,1}$.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 $P6 \in M_{5,1}$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 information$P6 \in M_{5,1}$.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.