Chapter 6: Message Ordering and Group Communication

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Outline and Notations

- Outline
  - Message orders: non-FIFO, FIFO, causal order, synchronous order
  - Group communication with multicast: causal order, total order
  - Expected behaviour semantics when failures occur
  - Multicasts: application layer on overlays; also at network layer
- Notations
  - Network (N, L); event set $(E, \prec)$
  - message m^i: send and receive events s^i and r^i
  - send and receive events: s and r.
  - M, $send(M)$, and $receive(M)$
  - *Corresponding events*: $a \sim b$ denotes a and b occur at the same process
  - send-receive pairs $\mathcal{T} = \{(s, r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$

Asynchronous and FIFO Executions

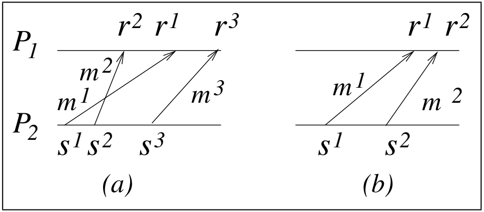


Figure 6.1: (a) *A*-execution that is FIFO (b) *A*-execution that is not FIFO

Asynchronous executions

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

FIFO executions

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

Causal Order: Definition

Causal order (CO)

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

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

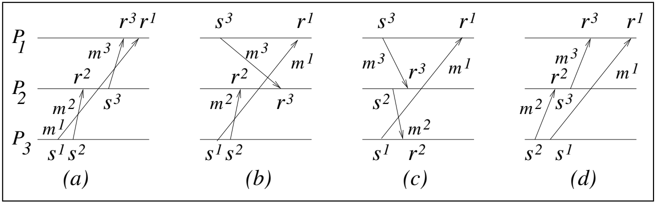


Figure 6.2: (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.

Causal Order: Definition from Implementation Perspective

CO alternate definition

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.

- Message arrival vs. delivery:
  - message m that arrives in OS buffer at P_i may have to be delayed until the messages that were sent to P_i causally before m was sent (the "overtaken" messages) have arrived!
  - The event of an application processing an arrived message is referred to as a *delivery* event (instead of as a *receive* event).
- no message overtaken by a chain of messages between the same (sender, receiver) pair. In Fig. 6.1(a), m_1 overtaken by chain $\langle m_2, m_3 \rangle$.
- CO degenerates to FIFO when $m1, m2$ sent by same process
- Uses: updates to shared data, implementing distributed shared memory, fair resource allocation; collaborative applications, event notification systems, distributed virtual environments

Causal Order: Other Characterizations (1)

Message Order (MO)

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

- Fig 6.2(a): $s^1 \prec s^3$ but $\neg(r^3 \prec r^1)$ is false $\Rightarrow$ MO not satisfied
- m cannot be overtaken by a chain

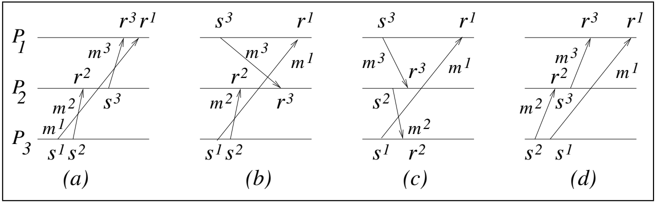


Figure 6.2: (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.

Causal Order: Other Characterizations (2)

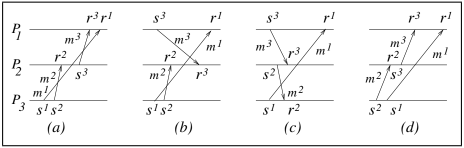


Figure 6.2: (a) Violates CO as $s^1 \prec s^3$; $r^3 \prec r^1$ (b) Satisfies CO. (c) Satisfies CO. No send events related by causality. (d) Satisfies CO.

Empty-Interval (EI) property

$(E, \prec)$ is an EI execution if for each $(s, r) \in \mathcal{T}$, the open interval set $\{x \in E \,|\, s \prec x \prec r\}$ in the partial order is empty.

- Fig 6.2(b). Consider M^2. No event x such that $s^2 \prec x \prec r^2$. Holds for all messages $\Rightarrow$ EI
- For EI $\langle s, r\rangle$, there exists some *linear* extension [1] $<$ | such the corresp. interval $\{x \in E \,|\, s < x < r\}$ is also empty.
- An empty $\langle s, r\rangle$ interval in a linear extension implies s, r may be arbitrarily close; shown by vertical arrow in a timing diagram.
- An execution E is CO iff for each M, there exists *some* space-time diagram in which that message can be drawn as a vertical arrow.

[1] A linear extension of a partial order $(E, \prec)$ is any total order $(E, <)$| each ordering relation of the partial order is preserved.

Causal Order: Other Characterizations (3)

- CO $\not\Longrightarrow$ *all* messages can be drawn as vertical arrows in the *same* space-time diagram (otherwise all $\langle s, r \rangle$ intervals empty in the *same* linear extension; synchronous execution).

Common Past and Future

An execution $(E, \prec)$ is CO iff for each pair $(s, r) \in \mathcal{T}$ and each event $e \in E$,

- Weak common past: $e \prec r \Longrightarrow \neg(s \prec e)$
- Weak common future: $s \prec e \Longrightarrow \neg(e \prec r)$

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

Synchronous Executions (SYNC)

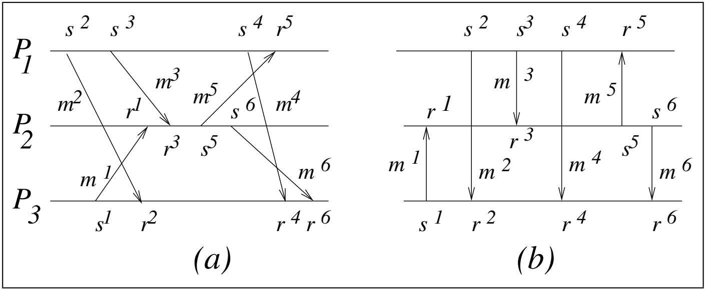


Figure 6.3: (a) Execution in an async system (b) Equivalent sync execution.

- Handshake between sender and receiver
- Instantaneous communication ⇒ modified definition of causality, where s, r are atomic and simultaneous, neither preceding the other.

Synchronous Executions: Definition

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 $x \in E$, $[(x \ll s \iff x \ll r)$ and $(s \ll x \iff r \ll x)]$

S3. If $x \ll y$ and $y \ll z$, then $x \ll z$

Synchronous execution (or S-execution).

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

Timestamping a synchronous execution.

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

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

Asynchronous Execution with Synchronous Communication

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

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

Figure 6.4: *A*-execution deadlocks when using synchronous primitives.

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

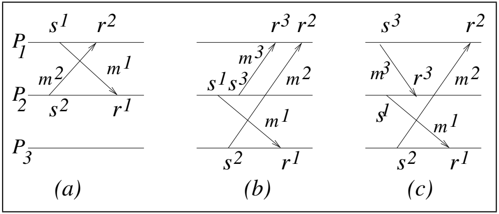


Figure 6.5: Illustration of non-RSC *A*-executions.

RSC Executions

Non-separated linear extension of $(E, \prec)$

A linear extension of $(E, \prec)$ such that for each pair $(s, r) \in \mathcal{T}$, the interval $\{ x \in E \mid s \prec x \prec r \}$ is empty.

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

RSC execution

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

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

Crown: Definition

Crown

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

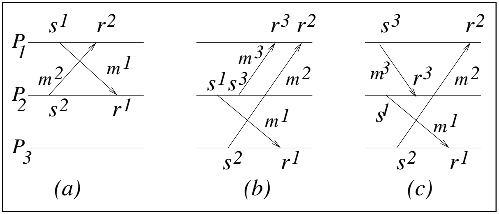


Figure 6.5: Illustration of non-RSC A-executions and crowns.

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

Fig 6.5(b) (b) crown is $\langle (s^1, r^1), (s^2, r^2) \rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$

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

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

Crown: Characterization of RSC Executions

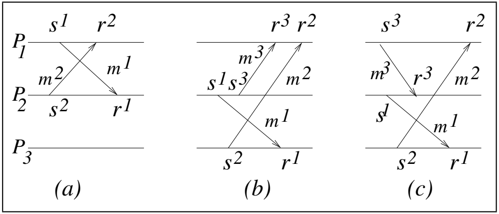


Figure 6.5: Illustration of non-RSC *A*-executions and crowns.

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

Fig 6.5(b) (b) crown is $\langle(s^1, r^1), (s^2, r^2)\rangle$ as we have $s^1 \prec r^2$ and $s^2 \prec r^1$

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

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

Some observations

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

Crown Test for RSC executions

1. Define the $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ relation on messages in the execution $(E, \prec)$ as follows. Let $\hookrightarrow([s,r],[s',r'])$ iff $s \prec r'$. Observe that the condition $s \prec r'$ (which has the form used in the definition of a crown) is implied by all the four conditions: (i) $s \prec s'$, or (ii) $s \prec r'$, or (iii) $r \prec s'$, and (iv) $r \prec r'$.
2. Now define a *directed* graph $G_{\hookrightarrow} = (\mathcal{T}, \hookrightarrow)$, where the vertex set is the set of messages $\mathcal{T}$ and the edge set is defined by $\hookrightarrow$.

   Observe that $\hookrightarrow: \mathcal{T} \times \mathcal{T}$ is a partial order iff $G_{\hookrightarrow}$ has no cycle, i.e., there must not be a cycle with respect to $\hookrightarrow$ on the set of corresponding (s, r) events.
3. Observe from the defn. of a crown that $G_{\hookrightarrow}$ has a directed cycle iff $(E, \prec)$ has a crown.

Crown criterion

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

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

Timestamps for a RSC execution

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

- for any message M, $T(s(M)) = T(r(M))$
- for each (a, b) in $(E \times E) \setminus \mathcal{T}$, $a \prec b \Longrightarrow T(a) < T(b)$

Hierarchy of Message Ordering Paradigms

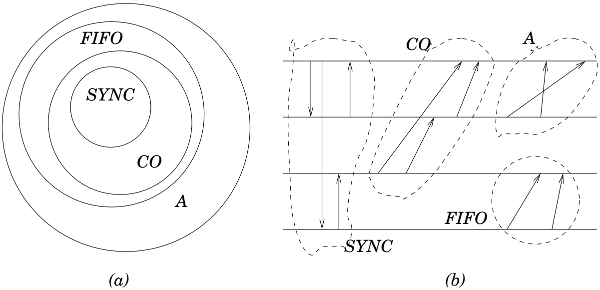


Figure 6.7: Hierarchy of message ordering paradigms. (a) Venn diagram (b) Example executions.

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

Simulations: Async Programs on Sync Systems

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

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

If A-execution is not RSC:

- partial order has to be changed; **or**
- model each $C_{i,j}$ by control process $P_{i,j}$ and use sync communication (see Fig 6.8)

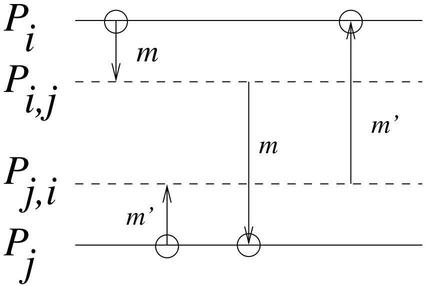


Figure 6.8: Modeling channels as processes to simulate an execution using asynchronous primitives on an synchronous system.

- Enables decoupling of sender from receiver.
- This implementation is expensive.

Simulations: Synch Programs on Async Systems

- Schedule msgs in the order in which they appear in S-program
- partial order of S-execution unchanged
- Communication on async system with async primitives
- When sync send is scheduled:
  - wait for ack before completion

Sync Program Order on Async Systems

Deterministic program: repeated runs produce same partial order

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

Nondeterminism (besides due to unpredictable message delays):

- Receive call does not specify sender
- Multiple sends and receives enabled at a process; can be executed in interchangeable order

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

Deadlock example of Fig 6.4

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

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

- Match send or receive with corresponding event

Binary rendezvous (implementation using tokens)

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

Bagrodia's Algorithm for Binary Rendezvous (1)

Assumptions

- Receives are always enabled
- Send, once enabled, remains enabled
- To break deadlock, PIDs used to introduce asymmetry
- Each process schedules *one* send at a time

Message types: $M, ack(M), request(M), permission(M)$

Process blocks when it knows it can successfully synchronize the current message

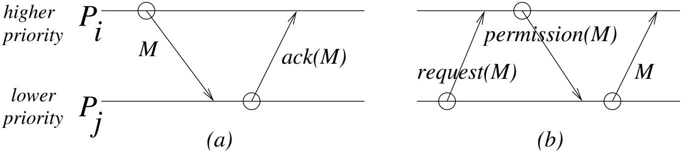


Fig 6.: Rules to prevent message cyles. (a) High priority process blocks. (b) Low priority process does not block.

Bagrodia's Algorithm for Binary Rendezvous: Code

(message types)
M, *ack(M)*, *request(M)*, *permission(M)*

1. **P_i wants to execute SEND(M) to a lower priority process P_j:**
   P_i executes *send(M)* and blocks until it receives *ack(M)* from P_j. The send event SEND(M) now completes.
   Any *M'* message (from a higher priority processes) and *request(M')* request for synchronization (from a lower priority processes) received during the blocking period are queued.

2. **P_i wants to execute SEND(M) to a higher priority process P_j:**
   1. P_i seeks permission from P_j by executing *send(request(M))*.
      // to avoid deadlock in which cyclically blocked processes queue messages.
   2. While P_i is waiting for permission, it remains unblocked.
      1. If a message M' arrives from a higher priority process P_k, P_i accepts M' by scheduling a RECEIVE(M') event and then executes *send(ack(M'))* to P_k.
      2. If a *request(M')* arrives from a lower priority process P_k, P_i executes *send(permission(M'))* to P_k and blocks waiting for the message M'. When M' arrives, the RECEIVE(M') event is executed.
   3. When the *permission(M)* arrives, P_i knows partner P_j is synchronized and P_i executes *send(M)*. The SEND(M) now completes.

3. **Request(M) arrival at P_i from a lower priority process P_j:**
   At the time a *request(M)* is processed by P_i, process P_i executes *send(permission(M))* to P_j and blocks waiting for the message M. When M arrives, the RECEIVE(M) event is executed and the process unblocks.

4. **Message M arrival at P_i from a higher priority process P_j:**
   At the time a message M is processed by P_i, process P_i executes RECEIVE(M) (which is assumed to be always enabled) and then *send(ack(M))* to P_j.

5. **Processing when P_i is unblocked:**
   When P_i is unblocked, it dequeues the next (if any) message from the queue and processes it as a message arrival (as per Rules 3 or 4).

Bagrodia's Algorithm for Binary Rendezvous (2)

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

- Before sending M to P_i, P_j requests permission in a nonblocking manner. While waiting:
  - M' arrives from another higher prio process. $ack(M')$ is returned
  - $request(M')$ arrives from lower prio process. P_j returns $permission(M')$ and blocks until M' arrives.
- Note: $receive(M')$ gets permuted with the $send(M)$ event

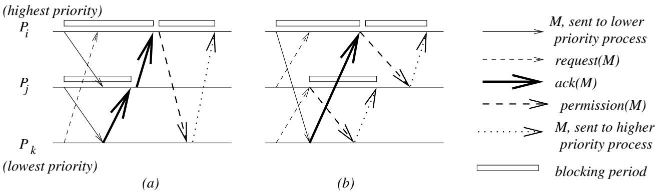


Figure 6.10: Scheduling messages with sync communication.

Group Communication

- Unicast vs. multicast vs. broadcast
- Network layer or hardware-assist multicast cannot easily provide:
  - Application-specific semantics on message delivery order
  - Adapt groups to dynamic membership
  - Multicast to arbitrary process set at each send
  - Provide multiple fault-tolerance semantics
- Closed group (source part of group) vs. open group
- # groups can be $O(2^n)$

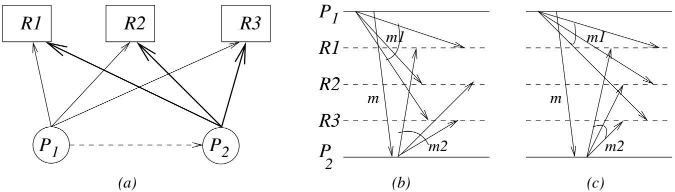


Figure 6.11: (a) Updates to 3 replicas. (b) Causal order (CO) and total order violated. (c) Causal order violated.

If *m* did not exist, (b,c) would not violate CO.

Raynal-Schiper-Toueg (RST) Algorithm

(local variables)
array of int $SENT[1 \ldots n, 1 \ldots n]$
array of int $DELIV[1 \ldots n]$ // $DELIV[k] = \#$ messages sent by k that are delivered locally

(1) **send event**, where P_i wants to send message M to P_j:
(1a) **send** $(M, SENT)$ to P_j;
(1b) $SENT[i, j] \longleftarrow SENT[i, j] + 1$.

(2) **message arrival,** when (M, ST) arrives at P_i from P_j:
(2a) **deliver** M to P_i when **for each** process x,
(2b) $DELIV[x] \geq ST[x, i]$;
(2c) $\forall x, y$, $SENT[x, y] \longleftarrow \max(SENT[x, y], ST[x, y])$;
(2d) $DELIV[j] \longleftarrow DELIV[j] + 1$.

How does algorithm simplify if all msgs are broadcast?

Assumptions/Correctness

- FIFO channels.
- Safety: Step (2a,b).
- Liveness: assuming no failures, finite propagation times

Complexity

- n^2 ints/ process
- n^2 ints/ msg
- Time per send and rcv event: n^2

Optimal KS Algorithm for CO: Principles

$M_{i,a}$: a^{th} multicast message sent by P_i

Delivery Condition for correctness:

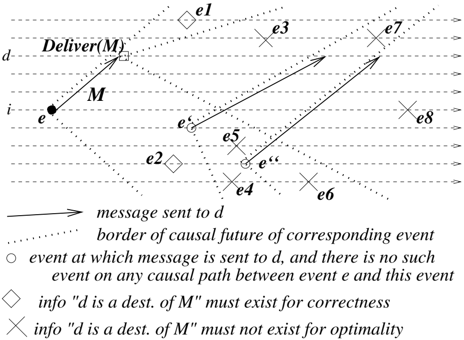
Msg M^* that carries information "$d \in M.Dests$", where message M was sent to d in the causal past of $Send(M^*)$, is not delivered to d if M has not yet been delivered to d.

Necessary and Sufficient Conditions for Optimality:

- For how long should the information "$d \in M_{i,a}.Dests$" be stored in the log at a process, and piggybacked on messages?
- *as long as* and *only as long as*
  - (*Propagation Constraint I:*) it is not known that the message $M_{i,a}$ is delivered to d, and
  - (*Propagation Constraint II:*) it is not known that a message has been sent to d in the causal future of $Send(M_{i,a})$, and hence it is not guaranteed using a reasoning based on transitivity that the message $M_{i,a}$ will be delivered to d in CO.

- $\Rightarrow$ if either (I) or (II) is false, "$d \in M.Dests$" must *not* be stored or propagated, even to remember that (I) or (II) has been falsified.

Optimal KS Algorithm for CO: Principles



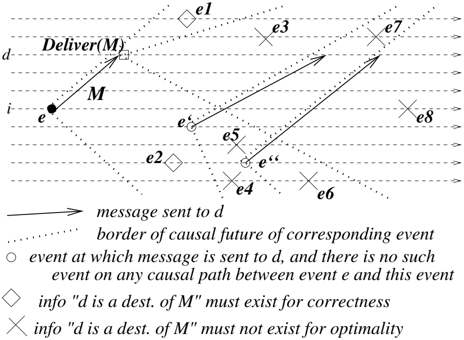
"$d \in M_{i,a}.Dests$" must be available in the

causal future of event $e_{i,a}$, but

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

- In the causal future of $Deliver_d(M_{i,a})$, and $Send(M_{k,c})$, the information is redundant; elsewhere, it is necessary.
- Information about what messages have been delivered (or are guaranteed to be delivered without violating CO) is necessary for the Delivery Condition.
  - For optimality, this cannot be stored. Algorithm infers this using set-operation logic.

Optimal KS Algorithm for CO: Principles



"$d \in M.Dests$"

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

- Info about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is *explicitly* tracked using (*source, ts, dest*).
- Must be deleted as soon as either (i) or (ii) becomes false.
- Info about messages already delivered and messages guaranteed to be delivered in CO is *implicitly* tracked without storing or propagating it:
  - derived from the explicit information.
  - used for determining when (i) or (ii) becomes false for the explicit information being stored/piggybacked.

Optimal KS Algorithm for CO: Code (1)

(local variables)
$clock_j \longleftarrow 0$; // local counter clock at node j
$SR_j[1 \ldots n] \longleftarrow \bar{0}$; // $SR_j[i]$ is the timestamp of last msg. from i delivered to j
$LOG_j = \{(i, clock_i, Dests)\} \longleftarrow \{\forall i, (i, 0, \emptyset)\}$;
// Each entry denotes a message sent in the causal past, by i at $clock_i$. $Dests$ is the set of remaining destinations
// for which it is not known that $M_{i,clock_i}$ (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

SND: j sends a message M to Dests:

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

Optimal KS Algorithm for CO: Code (2)

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

1. // Delivery Condition; ensure that messages sent causally before M are delivered.
   for all $o_{m,t_m} \in O_M$ **do**
   if $j \in o_{m.t_m}.Dests$ **wait until** $t_m \leq SR_j[m]$;

2. Deliver M; $SR_j[k] \longleftarrow t_k$;

3. $O_M \longleftarrow \{(k, t_k, Dests)\} \bigcup O_M$;
   for all $o_{m,t_m} \in O_M$ **do** $o_{m,t_m}.Dests \longleftarrow o_{m,t_m}.Dests \setminus \{j\}$;
   // delete the now redundant dependency of message represented by o_{m,t_m} sent to j

4. // Merge O_M and LOG_j by eliminating all redundant entries.
   // Implicitly track "already delivered" & "guaranteed to be delivered in CO" messages.
   for all $o_{m,t} \in O_M$ **and** $l_{s,t'} \in LOG_j$ **such that** $s = m$ **do**
   if $t < t' \bigwedge l_{s,t} \notin LOG_j$ **then** mark $o_{m,t}$;
   // $l_{s,t}$ had been deleted or never inserted, as $l_{s,t}.Dests = \emptyset$ in the causal past
   if $t' < t \bigwedge o_{m,t'} \notin O_M$ **then** mark $l_{s,t'}$;
   // $o_{m,t'} \notin O_M$ because $l_{s,t'}$ had become $\emptyset$ at another process in the causal past
   Delete all marked elements in O_M and LOG_j; // delete entries about redundant information
   for all $l_{s,t'} \in LOG_j$ **and** $o_{m,t} \in O_M$, **such that** $s = m \bigwedge t' = t$ **do**
   $l_{s,t'}.Dests \longleftarrow l_{s,t'}.Dests \bigcap o_{m,t}.Dests$; // delete destinations for which Delivery
   // Condition is satisfied or guaranteed to be satisfied as per $o_{m,t}$
   Delete $o_{m,t}$ from O_M; // information has been incorporated in $l_{s,t'}$
   $LOG_j \longleftarrow LOG_j \bigcup O_M$; // merge nonredundant information of O_M into LOG_j

5. $PURGE\_NULL\_ENTRIES(LOG_j)$. // Purge older entries l for which $l.Dests = \emptyset$

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

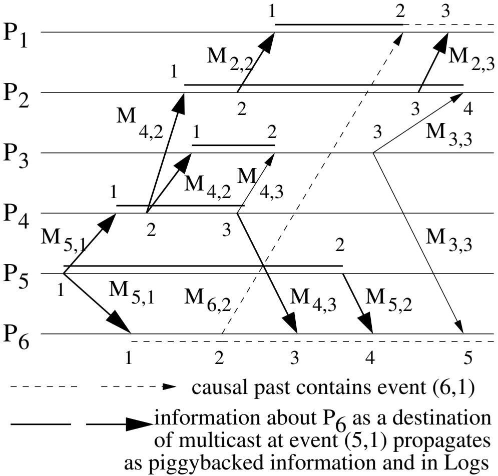
for all $l_{s,t} \in Log_j$ **do**

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

Optimal KS Algorithm for CO: Information Pruning

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

Optimal KS Algorithm for CO: Example



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

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

Total Message Order

Total order

For each pair of processes P_i and P_j and for each pair of messages M_x and M_y that are delivered to both the processes, P_i is delivered M_x before M_y if and only if P_j is delivered M_x before M_y.

Same order seen by all

Solves coherence problem

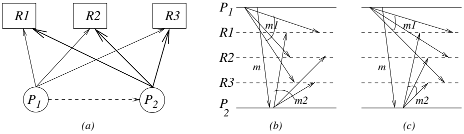


Fig 6.11: (a) Updates to 3 replicas. (b) Total order violated. (c) Total order not violated.

Centralized algorithm

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

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

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

Time Complexity: 2 hops/ transmission

Message complexity: n

Total Message Order: 3-phase Algorithm Code

```
record Q_entry
        M: int;                          // the application message
        tag: int;                        // unique message identifier
        sender_id: int;                  // sender of the message
        timestamp: int;                  // tentative timestamp assigned to message
        deliverable: boolean;            // whether message is ready for delivery
(local variables)
queue of Q_entry: temp_Q, delivery_Q
int: clock                               // Used as a variant of Lamport's scalar clock
int: priority                            // Used to track the highest proposed timestamp
(message types)
REVISE_TS(M, i, tag, ts)                 // Phase 1 message sent by P_i, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)               // Phase 2 message sent by P_j, with revised timestamp, to P_i
FINAL_TS(i, tag, ts)                     // Phase 3 message sent by P_i, with final timestamp


(1) When process P_i wants to multicast a message M with a tag tag:
(1a) clock = clock + 1;
(1b) send REVISE_TS(M, i, tag, clock) to all processes;
(1c) temp_ts = 0;
(1d) await PROPOSED_TS(j, i, tag, ts_j) from each process P_j;
(1e) ∀j ∈ N, do temp_ts = max(temp_ts, ts_j);
(1f) send FINAL_TS(i, tag, temp_ts) to all processes;
(1g) clock = max(clock, temp_ts).
(2) When REVISE_TS(M, j, tag, clk) arrives from P_j:
(2a) priority = max(priority + 1, clk);
(2b) insert (M, tag, j, priority, undeliverable) in temp_Q;     // at end of queue
(2c) send PROPOSED_TS(i, j, tag, priority) to P_j.
(3) When FINAL_TS(j, tag, clk) arrives from P_j:
(3a) Identify entry Q_entry(tag) in temp_Q, corresponding to tag;
(3b) mark q_tag as deliverable;
(3c) Update Q_entry.timestamp to clk and re-sort temp_Q based on the timestamp field;
(3d) if head(temp_Q) = Q_entry(tag) then
(3e)     move Q_entry(tag) from temp_Q to delivery_Q;
(3f)     while head(temp_Q) is deliverable do
(3g)         move head(temp_Q) from temp_Q to delivery_Q.
(4) When P_i removes a message (M, tag, j, ts, deliverable) from head(delivery_Q_i):
(4a) clock = max(clock, ts) + 1.
```

Total Order: Distributed Algorithm: Example and Complexity

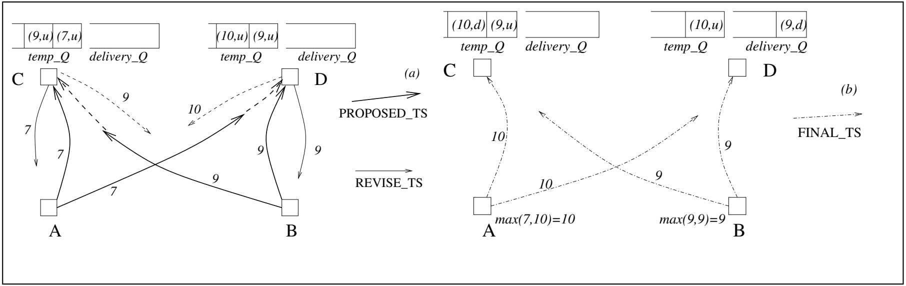
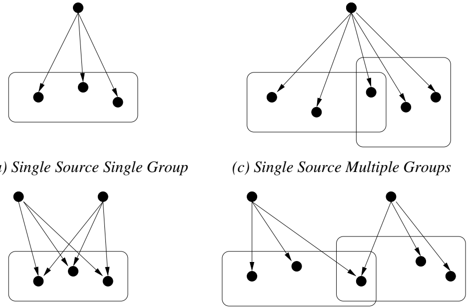


Figure 6.14: (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages.

Complexity:

- Three phases
- $3(n-1)$ messages for $n-1$ dests
- Delay: 3 message hops
- Also implements causal order

A Nomenclature for Multicast



(a) Single Source Single Group (c) Single Source Multiple Groups

(b) Multiple Sources Single Group (d) Multiple Sources Multiple Groups

Fig 6.15 : Four classes of source-dest relationships for open-group multicasts. For closed-group multicasts, the sender needs to be part of the recipient group.

4 classes of source-dest relns for open groups:

- SSSG: Single source and single dest group
- MSSG: Multiple sources and single dest group
- SSMG: Single source and multiple, possibly overlapping, groups
- MSMG: Multiple sources and multiple, possibly overlapping, groups

SSSG, SSMG: easy to implement
MSSG: easy. E.g., Centralized algorithm

MSMG: Semi-centralized *propagation tree* approach

Propagation Trees for Multicast: Definitions

- set of groups $\mathcal{G} = \{G_1 \dots G_g\}$
- set of *meta-groups* $\mathcal{MG} = \{MG_1, \dots MG_h\}$ with the following properties.
  - Each process belongs to a single meta-group, and has the exact same group membership as every other process in that meta-group.
  - No other process outside that meta-group has that exact group membership.
- MSMG to groups $\longrightarrow$ MSSG to meta-groups
- A distinguished node in each meta-group acts as its manager.
- For each user group G_i, one of its meta-groups is chosen to be its *primary meta-group (PM)*, denoted $PM(G_i)$.
- All meta-groups are organized in a *propagation forest/tree* satisfying:
  - For user group G_i, $PM(G_i)$ is at the lowest possible level (i.e., farthest from root) of the tree such that all meta-groups whose destinations contain any nodes of G_i belong to subtree rooted at $PM(G_i)$.
- `Propagation tree is not unique!`
  - `Exercise:` How to construct propagation tree?
  - Metagroup with members from more user groups as root $\Rightarrow$ low tree height

Propagation Trees for Multicast: Properties

1. The primary meta-group $PM(G)$ is the ancestor of all the other meta-groups of G in the propagation tree.
2. $PM(G)$ is uniquely defined.
3. For any meta-group MG, there is a unique path to it from the PM of any of the user groups of which the meta-group MG is a subset.
4. Any $PM(G_1)$ and $PM(G_2)$ lie on the same branch of a tree or are in disjoint trees. In the latter case, their groups membership sets are disjoint.

Key idea: Multicasts to G_i are sent first to the meta-group $PM(G_i)$ as only the subtree rooted at $PM(G_i)$ can contain the nodes in G_i. The message is then propagated down the subtree rooted at $PM(G_i)$.

- MG_1 *subsumes* MG_2 if MG_1 is a subset of each user group G of which MG_2 is a subset.
- MG_1 *is joint with* MG_2 if neither subsumes the other and there is some group G such that $MG_1, MG_2 \subset G$.

Propagation Trees for Multicast: Example

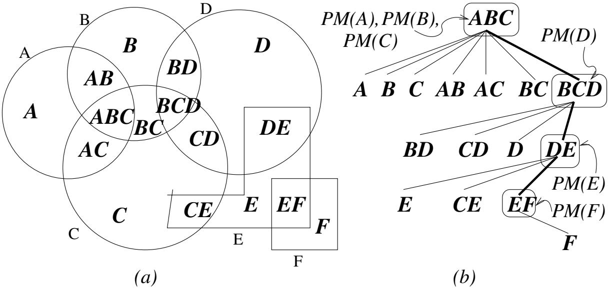


Fig 6.16: Example illustrating a propagation tree. Meta-groups in boldface. (a) Groups A, B, C, D, E and F, and their meta-groups. (b) A *propagation tree*, with the primary meta-groups labeled.

- $\langle ABC \rangle$, $\langle AB \rangle$, $\langle AC \rangle$, and $\langle A \rangle$ are meta-groups of user group $\langle A \rangle$.
- $\langle ABC \rangle$ is $PM(A), PM(B), PM(C)$. $\langle B, C, D \rangle$ is $PM(D)$. $\langle D, E \rangle$ is $PM(E)$. $\langle E, F \rangle$ is $PM(F)$.
- $\langle ABC \rangle$ is joint with $\langle CD \rangle$. Neither subsumes the other and both are a subset of C.
- Meta-group $\langle ABC \rangle$ is the primary meta-group $PM(A), PM(B), PM(C)$. Meta-group $\langle BCD \rangle$ is the primary meta-group $PM(D)$. A multicast to D is sent to $\langle BCD \rangle$.

Propagation Trees for Multicast: Logic

- Each process knows the propagation tree
- Each meta-group has a distinguished process (*manager*)
- $SV_i[k]$ at each P_i: # msgs multicast by P_i that will traverse $PM(G_k)$. Piggybacked on each multicast by P_i.
- $RV_{manager(PM(G_z))}[k]$: # msgs sent by P_k received by $PM(G_z)$
- At $manager(PM(G_z))$: process M from P_i if $SV_i[z] = RV_{manager(PM(G_z))}[i]$; else buffer M until condition becomes true
- At manager of non-primary meta-group: msg order already determined, as it never receives msg directly from sender of multicast. Forward (2d-2g).

`Correctness for Total Order:` Consider $MG_1, MG_2 \subset G_x, G_y$

- $\Rightarrow PM(G_x), PM(G_y)$ both subsume MG_1, MG_2 and lie on the same branch of the propagation tree to either MG_1 or MG_2
- order seen by the "lower-in-the-tree" primary meta-group (+ FIFO) = order seen by processes in meta-groups subsumed by it

Propagation Trees for Multicast (CO and TO): Code

(local variables)
array of integers: $SV[1 \ldots h]$; //kept by each process. h is #(primary meta-groups), $h \leq |\mathcal{G}|$
array of integers: $RV[1 \ldots n]$; //kept by each primary meta-group manager. n is #(processes)
set of integers: PM_set; //set of primary meta-groups through which message must traverse

(1) When process P_i wants to multicast message M to group G:
(1a) **send** $M(i, G, SV_i)$ to manager of $PM(G)$, primary meta-group of G;
(1b) $PM\_set \longleftarrow \{$ primary meta-groups through which M must traverse $\}$;
(1c) **for all** $PM_x \in PM\_set$ **do**
(1d) $SV_i[x] \longleftarrow SV_i[x] + 1$.

(2) When P_i, the manager of a meta-group MG receives $M(k, G, SV_k)$ from P_j:
// Note: P_i may not be a manager of any meta-group
(2a) **if** MG is a primary meta-group **then**
(2b) **buffer** the message **until** $(SV_k[i] = RV_i[k])$;
(2c) $RV_i[k] \longleftarrow RV_i[k] + 1$;
(2d) **for each** child meta-group that is subsumed by MG **do**
(2e) **send** $M(k, G, SV_k)$ to the manager of that child meta-group;
(2f) **if** there are no child meta-groups **then**
(2g) **send** $M(k, G, SV_k)$ to each process in this meta-group.

Propagation Trees for Multicast: Correctness for CO

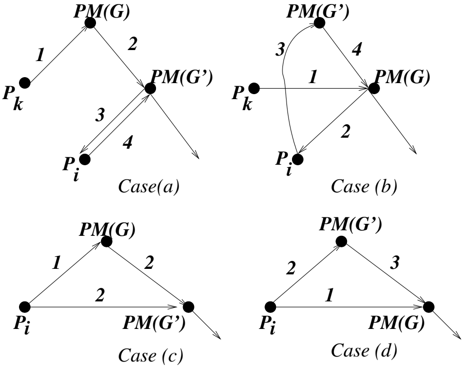


Fig 6.17: The four cases for the correctness of `causal ordering`. The sequence numbers indicate the order in which the msgs are sent.

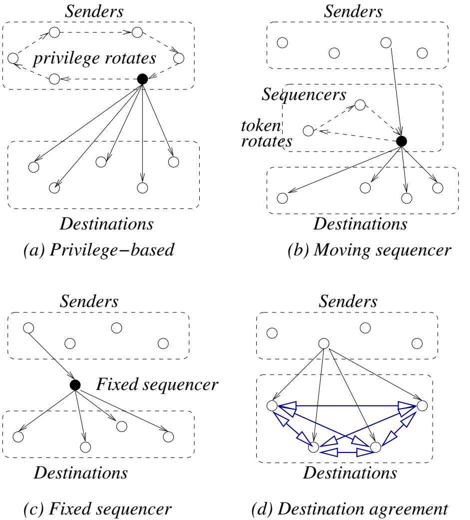
M and M' multicast to G and G', resp.
Consider $G \cap G'$

- Senders of M, M' are different.
  P_i in G receives M, then sends M'.
  $\Rightarrow \forall MG_q \in G \cap G'$, $PM(G)$, $PM(G')$ are both ancestors of metagroup of P_i
  - (a) $PM(G')$ processes M before M'
  - (b) $PM(G)$ processes M before M'

  FIFO $\Rightarrow$ CO guaranteed for all in $G \cap G'$
- P_i sends M to G, then sends M' to G'.
  Test in lines (2a)-(2c) $\Rightarrow$
  - $PM(G')$ will not process M' before M
  - $PM(G)$ will not process M' before M

  FIFO $\Rightarrow$ CO guaranteed for all in $G \cap G'$

Classification of Application-Level Multicast Algorithms



- Communication-history based: RST, KS, Lamport, NewTop
- Privilege-based: Token-holder multicasts
  - processes deliver msgs in order of *seq_no*.
  - Typically closed groups, and CO & TO.
  - E.g., Totem, On-demand.
- Moving sequencer: E.g., Chang-Maxemchuck, Pinwheel
  - Sequencers' token has *seq_no* and list of msgs for which *seq_no* has been assigned (these are sent msgs).
  - On receiving token, sequencer assigns *seq_no*s to received but unsequenced msgs, and sends the newly sequenced msgs to dests.
  - Dests deliver in order of *seq_no*
- Fixed Sequencer: simplifies moving sequencer approach. E.g., propagation tree, ISIS, Amoeba, Phoenix, Newtop-asymmetric
- Destination agreement:
  - Dests receive limited ordering info.
  - (i) Timestamp-based (Lamport's 3-phase)
  - (ii) Agreement-based, among dests.

Semantics of Fault-Tolerant Multicast (1)

- Multicast is non-atomic!
- Well-defined behavior during failure ⇒ well-defined recovery actions
- if one correct process delivers M, what can be said about the other correct processes and faulty processes being delivered M?
- if one faulty process delivers M, what can be said about the other correct processes and faulty processes being delivered M?
- For causal or total order multicast, if one correct or faulty process delivers M, what can be said about other correct processes and faulty processes being delivered M?
- *(Uniform)* specifications: specify behavior of faulty processes (benign failure model)

Uniform Reliable Multicast of M.

Validity. If a correct process multicasts M, then all correct processes will eventually deliver M.

(*Uniform*) Agreement. If a correct (*or faulty*) process delivers M, then all correct processes will eventually deliver M.

(*Uniform*) Integrity. Every correct (*or faulty*) process delivers M at most once, and only if M was previously multicast by $sender(M)$.

Semantics of Fault-Tolerant Multicast (2)

(*Uniform*) FIFO order. If a process broadcasts M before it broadcasts M', then no correct (*or faulty*) process delivers M' unless it previously delivered M.

(*Uniform*) Causal Order. If a process broadcasts M causally before it broadcasts M', then no correct (*or faulty*) process delivers M' unless it previously delivered M.

(*Uniform*) Total Order. If correct (*or faulty*) processes a and b both deliver M and M', then a delivers M before M' if and only if b delivers M before M'.

Specs based on global clock or local clock (needs clock synchronization)

(*Uniform*) Real-time Δ-Timeliness. For some known constant Δ, if M is multicast at real-time t, then no correct (*or faulty*) process delivers M after real-time $t + \Delta$.

(*Uniform*) Local Δ-Timeliness. For some known constant Δ, if M is multicast at local time t_m, then no correct (*or faulty*) process delivers M after its local time $t_m + \Delta$.

Reverse Path Forwarding (RPF) for Constrained Flooding

Network layer multicast exploits topology, e.g., bridged LANs use spannint trees for learning dests and distributing information, IP layer
RPF approximates DVR/ LSR-like algorithms at lower cost

- Broadcast gets curtailed to approximate a spanning tree
- Approx. to rooted spanning tree is identified without being computed/stored
- # msgs closer to $|N|$ than to $|L|$

(1) When P_i wants to multicast M to group *Dests*:
(1a) **send** $M(i, Dests)$ on all outgoing links.

(2) When a node i receives $M(x, Dests)$ from node j:
(2a) **if** $Next\_hop(x) = j$ **then** // this will necessarily be a new message
(2b) **forward** $M(x, Dests)$ on all other incident links besides (i, j);
(2c) **else** ignore the message.

Steiner Trees

Steiner tree

Given a weighted graph (N, L) and a subset $N' \subseteq N$, identify a subset $L' \subseteq L$ such that (N', L') is a subgraph of (N, L) that connects all the nodes of N'.
A *minimal Steiner tree* is a minimal weight subgraph (N', L').

NP-complete $\Rightarrow$ need heuristics
Cost of routing scheme R:

- Network cost: $\sum$ cost of Steiner tree edges
- Destination cost: $\frac{1}{N'} \sum_{i \in N'} cost(i)$, where $cost(i)$ is cost of path (s, i)

Kou-Markowsky-Berman Heuristic for Steiner Tree

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where N' is the set of Steiner points

1. Construct the complete undirected distance graph $G' = (N', L')$ as follows. $L' = \{(v_i, v_j) \mid v_i, v_j \text{ in } N'\}$, and $wt(v_i, v_j)$ is the length of the shortest path from v_i to v_j in (N, L).
2. Let T' be the minimal spanning tree of G'. If there are multiple minimum spanning trees, select one randomly.
3. Construct a subgraph G_s of G by replacing each edge of the MST T' of G', by its corresponding shortest path in G. If there are multiple shortest paths, select one randomly.
4. Find the minimum spanning tree T_s of G_s. If there are multiple minimum spanning trees, select one randomly.
5. Using T_s, delete edges as necessary so that all the leaves are the Steiner points N'. The resulting tree, $T_{Steiner}$, is the heuristic's solution.

- Approximation ratio = 2 (even without steps (4) and (5) added by KMB)
- Time complexity: Step (1): $O(|N'| \cdot |N|^2)$, Step (2): $O(|N'|^2)$, Step (3): $O(|N|)$, Step (4): $O(|N|^2)$, Step (5): $O(|N|)$. Step (1) dominates, hence $O(|N'| \cdot |N|^2)$.

Constrained (Delay-bounded) Steiner Trees

- $\mathcal{C}(l)$ and $\mathcal{D}(l)$: cost, integer delay for edge $l \in L$

Definition

For a given delay tolerance Δ, a given source s and a destination set $Dest$, where $\{s\} \cup Dest = N' \subseteq N$, identify a spanning tree T covering all the nodes in N', subject to the constraints below.

- $\sum_{l \in T} C(l)$ is minimized, subject to
- $\forall v \in N'$, $\sum_{l \in path(s,v)} D(l) < \Delta$, where $path(s, v)$ denotes the path from s to v in T.

- *constrained cheapest path* between x and y is the cheapest path between x and y having delay $< \Delta$.
- its cost and delay denoted $\mathcal{C}(x, y)$, $\mathcal{D}(x, y)$, resp.

Constrained (Delay-Bounded) Steiner Trees: Algorithm

$\mathcal{C}(l)$, $\mathcal{D}(l)$; // cost, delay of edge l
$\mathcal{T}$; // constrained spanning tree to be constructed
$P(x, y)$; // path from x to y
$\mathcal{P}_C(x, y)$, $\mathcal{P}_D(x, y)$; // cost, delay of constrained cheapest path from x to y
$\mathcal{C}_d(x, y)$; // cost of the cheapest path with delay exactly d

Input: weighted graph $G = (N, L)$, and $N' \subseteq N$, where N' is the set of Steiner points and source s; Δ is the constraint on delay.

1. Compute the closure graph G' on (N', L), to be the complete graph on N'. The closure graph is computed using the all-pairs constrained cheapest paths using a dynamic programming approach analogous to Floyd's algorithm. For any pair of nodes $x, y \in N'$:
   - $\mathcal{P}_c(x, y) = \min_{d<\Delta} \mathcal{C}_d(x, y)$ This selects the cheapest constrained path, satisfying the condition of Δ, among the various paths possible between x and y. The various $\mathcal{C}_d(x, y)$ can be calculated using DP as follows.
   - $\mathcal{C}_d(x, y) = \min_{z \in N}\{\mathcal{C}_{d-\mathcal{D}(z,y)}(x, z) + \mathcal{C}(z, y)\}$ For a candidate path from x to y passing through z, the path with weight exactly d must have a delay of $d - \mathcal{D}(z, y)$ for x to z when the edge (z, y) has delay $\mathcal{D}(z, y)$.

   In this manner, the complete closure graph G' is computed. $\mathcal{P}_D(x, y)$ is the constrained cheapest path that corresponds to $\mathcal{P}_C(x, y)$.
2. Construct a constrained spanning tree of G' using a greedy approach that sequentially adds edges to the subtree of the constrained spanning tree T (thus far) until all the Steiner points are included. The initial value of T is the singleton s. Consider that node u is in the tree and we are considering whether to add edge (u, v).
   The following two edge selection criteria (heuristics) can be used to decide whether to include edge (u, v) in the tree.
   - Heuristic CST_{CD}: $f_{CD}(u, v) = \begin{cases} \frac{\mathcal{C}(u,v)}{\Delta-(\mathcal{P}_D(s,u)+\mathcal{D}(u,v))}, & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise} \end{cases}$
     The numerator is the "incremental cost" of adding (u, v) and the denominator is the "residual delay" that could be afforded. The goal is to minimize the incremental cost, while also maximizing the residual delay by choosing an edge that has low delay.
   - Heuristic CST_C: $f_c = \begin{cases} \mathcal{C}(u, v), & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise} \end{cases}$
     Picks the lowest cost edge between the already included tree edges and their nearest neighbour, provided total delay $< \Delta$.

   The chosen node v is included in T. This step 2 is repeated until T includes all $|N'|$ nodes in G'.
3. Expand the edges of the constrained spanning tree T on G' into the constrained cheapest paths they represent in the original graph G. Delete/break any loops introduced by this expansion.

Constrained (Delay-Bounded) Steiner Trees: Example

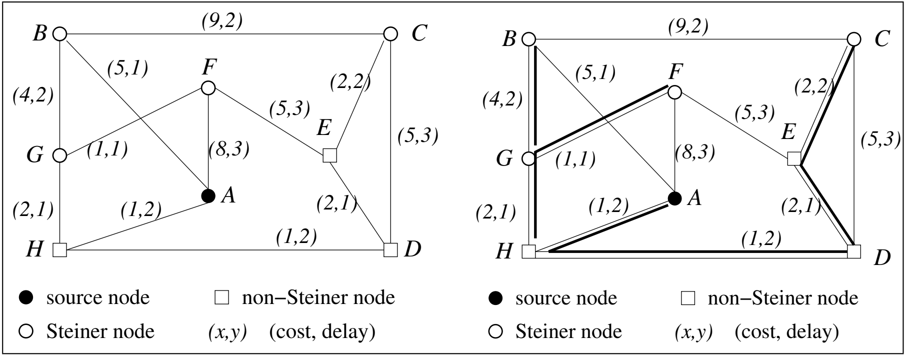


Figure 6.19: (a) Network graph. (b,c) MST and Steiner tree (optimal) are the same and shown in thick lines.

Constrained (Delay-Bounded) Steiner Trees: Heuristics, Time Complexity

Heuristic CST_{CD}: Tries to choose low-cost edges, while also trying to maximize the remaining allowable delay.

Heuristic CST_C: Minimizes the cost while ensuring that the delay bound is met.

- step (1) which finds the constrained cheapest shortest paths over all the nodes costs $O(n^3\Delta)$.
- Step (2) which constructs the constrained MST on the closure graph having k nodes costs $O(k^3)$.
- Step (3) which expands the constrained spanning tree, involves expanding the k edges to up to $n-1$ edges each and then eliminating loops. This costs $O(kn)$.
- Dominating step is step (1).

Core-based Trees

Multicast tree constructed dynamically, grows on demand.
Each group has a *core* node(s)

1. A node wishing to join the tree as a receiver sends a unicast `join` message to the core node.
2. The `join` marks the edges as it travels; it either reaches the core node, or some node already part of the tree. The path followed by the `join` till the core/multicast tree is grafted to the multicast tree.
3. A node on the tree multicasts a message by using a flooding on the core tree.
4. A node not on the tree sends a message towards the core node; as soon as the message reaches any node on the tree, it is flooded on the tree.