UNIT II - MESSAGE ORDERING & SNAPSHOTS

Message ordering and group communication: Message ordering paradigms –Asynchronous execution with synchronous communication –Synchronous program order on an asynchronous system –Group communication – Causal order (CO) – Total order. Global state and snapshot recording algorithms: Introduction –System model and definitions –Snapshot algorithms for FIFO channels

Message Ordering and Group Communication

- For any two events a and b, where each can be either a send or a receive event, the notation
- a ~ b denotes that a and b occur at the same process, i.e., a ∈ E_i and b ∈ E_i for some process i. The send and receive event pair for a message called pair of *corresponding* events.
- For a given execution E, let the set of all send–receive event pairs be denoted as
- $\mathcal{T} = \{(s,r) \in E_i \times E_j \mid s \text{ corresponds to } r\}$.

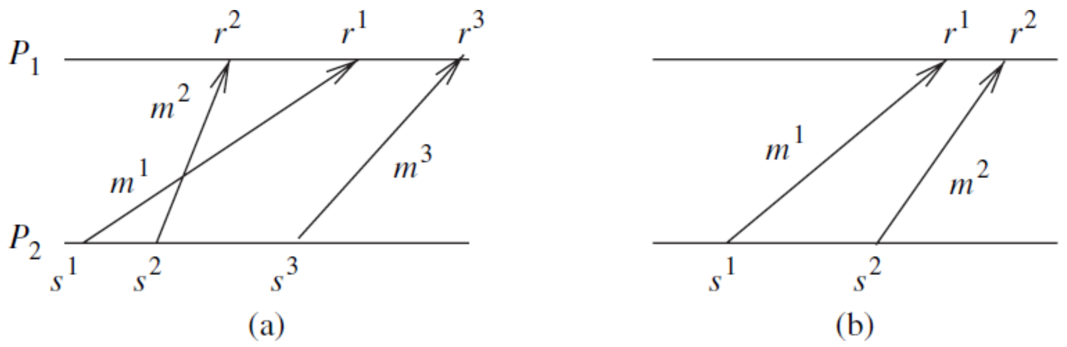
2.1 Message ordering paradigms

- Distributed program logic greatly depends on the order of delivery of messages.
- Several orderings on messages have been defined: (i) non-FIFO, (ii) FIFO, (iii) causal order, and (iv) synchronous order.

Asynchronous executions

Definition 6.1 (A-execution): An asynchronous execution (or A-execution) is an execution (E,≺) for which the causality relation is a partial order.

- On a logical link between two nodes (is formed as multiple paths may exist) in the system, if the messages are delivered in any order then it is known as non-FIFO executions. Example: IPv4.
- Each physical link delivers the messages sent on it in FIFO order due to the physical properties of the medium.



(Illustrating FIFO and non-FIFO executions. (a) An A-execution that is not a FIFO execution. (b) An A-execution that is also a FIFO execution.)

2.1.1 FIFO executions

Definition 6.2 (FIFO executions) :A FIFO execution is an A-execution in which, for all (s,r) and $(s',r') \in \mathcal{T}$, $(s \sim s' \text{ and } r \sim r' \text{ and } s \prec s') \Rightarrow r \prec r'$.

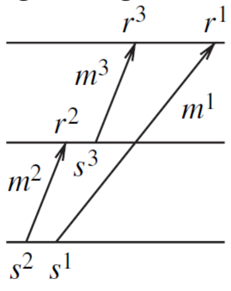
- In general on any logical link, messages are delivered in the order in which they are sent.
- To implement FIFO logical channel over a non-FIFO channel, use a separate numbering scheme to sequence the messages.
- The sender assigns and appends a <sequence_num, connection_id> tuple to each message. The receiver uses a buffer to order the incoming messages as per the sender's sequence numbers, and accepts only the "next" message in sequence.
- Figure 6.1(b) illustrates an A-execution under FIFO ordering.

2.1.2Causally ordered (CO) executions

Definition 6.3 (Causal order (CO)): A CO execution is an A-execution in which,

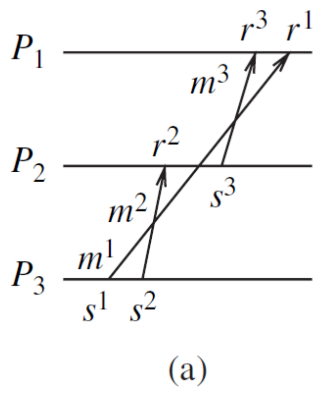
for all (s, r) and (s′, r′) $\in \mathcal{T}$, (r ∼ r′ and s ≺ s′) ⇒ r ≺ r′.

- If two send events s and s′ are related by causality ordering then their corresponding receive events r and r′ must occur in the same order at all common destinations.
- Figure 6.2 shows an execution that satisfies CO. s2 and s1 are related by causality but the destinations of the corresponding messages are different. Similarly for s2 ands3.



(Fig CO executions)

- Applications of Causal order: applications that requires update to shared data, to implement distributed shared memory, and fair resource allocation in distributed mutual exclusion.
- **Definition (causal order (CO) for implementations)** If send(m^1) ≺ send(m^2) then for each common destination d of messages m^1 and m^2, $deliver_d(m^1) ≺ deliver_d(m^2)$ must be satisfied.
- if m^1 and m^2 are sent by the same process, then property degenerates to FIFOproperty.
- In a FIFO execution, no message can be overtaken by another message between the same (sender, receiver) pair of processes.
- In a CO execution, no message can be overtaken by a chain of messages between the same (sender, receiver) pair of processes.
- ***Definition (Message order (MO))***: A MO execution is an A-execution in which, for all (s, r) and (s′, r′) $\in \mathcal{T}$, s ≺ s′ ⇒ ¬ (r′ ≺ r).
- Example: Consider any message pair, say m^1 and m^3 in Figure 6.2(a). s1 ≺ s3 but ¬ (r3 ≺ r1) is false. Hence, the execution does not satisfy MO.



(6.2 a) Not a CO execution.

- **Definition (Empty-interval execution)** An execution (E,≺) is an empty-interval (EI) execution if for each pair of events (s, r) ∈ $\mathcal{T}$, the open interval set {x ∈ E | s ≺ x ≺ r} in the partial order is empty.
- **Example** Consider any message, say m^2, in Figure 6.2(b). There does not exist any event x such that $s^2 \prec x \prec r^2$. This holds for all messages in the execution. Hence, the execution is EI.

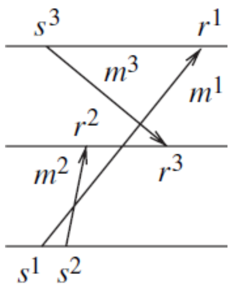


Figure: CO Execution

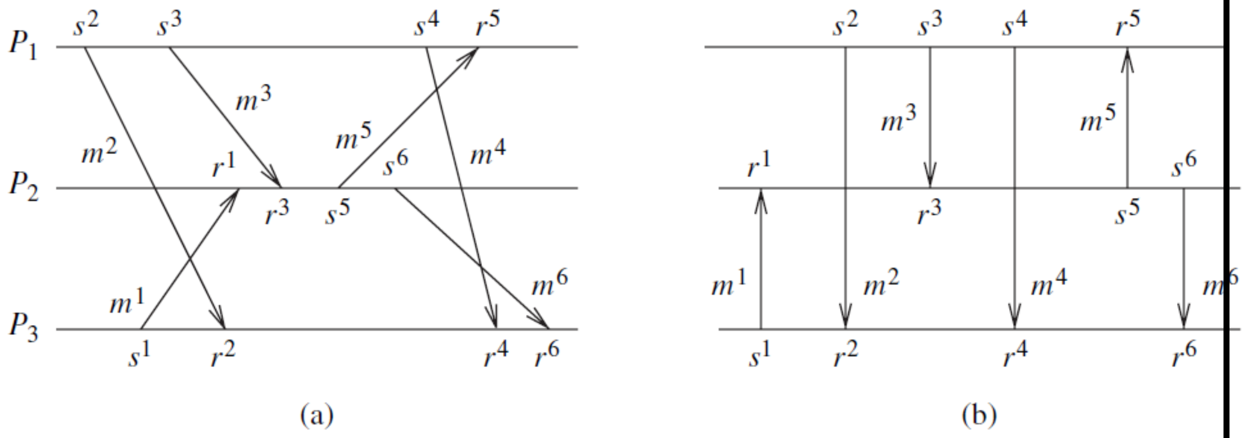
Corollary: An execution (E,≺) is CO if and only if for each pair of events (s,r) ∈ $\mathcal{T}$ and each event e ∈ E,

- *weak common past:* $e \prec r \Rightarrow \neg(s \prec e)$
- *weak common future:* $s \prec e \Rightarrow \neg(e \prec r)$.

2.2 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.
- In a timing diagram, the “instantaneous” message communication can be shown by bidirectional vertical message lines.
- The “instantaneous communication” property of synchronous executions requires that two events are viewed as being atomic and simultaneous, and neither event precedes the other.
- **Definition (Causality in a synchronous execution)** The synchronous causality relation ≪ on E is the smallest transitive relation that satisfies the following:

- S1: If x occurs before y at the same process, then x ≪ y.
- S2: If (s, r) ∈ *T*, then for all x ∈ E, [(x ≪ s ⇐⇒ x ≪ r) and (s ≪ x ⇐⇒ r ≪ x)].
- S3: If x ≪ y and y ≪ z, then x ≪ z.
- We can now formally define a synchronous execution.



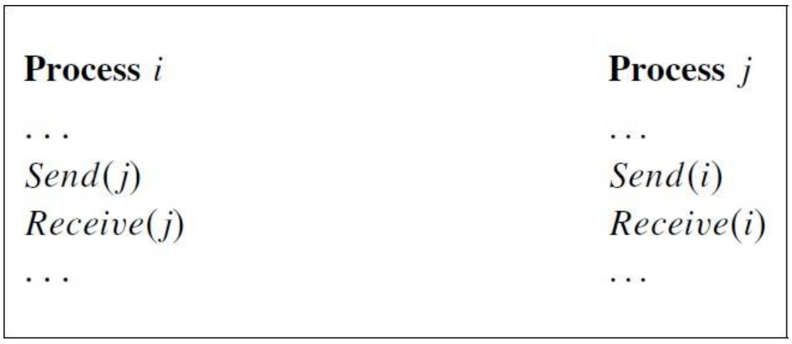
(Figure Illustration of a synchronous communication. (a) Execution in an asynchronous system. (b) Equivalent instantaneous communication.)

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

- **Timestamping a synchronous execution:** An execution (E,≺) 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 $T(e_i) < T(e_i')$.

Asynchronous execution with synchronous communication

- When all the communication between pairs of processes is by using synchronous
- send and receive primitives, the resulting order is synchronous order.
- A distributed program that run correctly on an asynchronous system may not be executed by synchronous primitives. There is a possibility that the program may *deadlock*, as shown by the code in Figure 6.4.



Process *i*

. . .
Send(*j*)
Receive(*j*)
. . .

Process *j*

. . .
Send(*i*)
Receive(*i*)
. . .

Figure *A communication program for an asynchronous system deadlocks when using synchronous primitives.*

- **Examples:** In Figure 6.5(a-c) using a timing diagram, will deadlock if run with synchronous primitives.

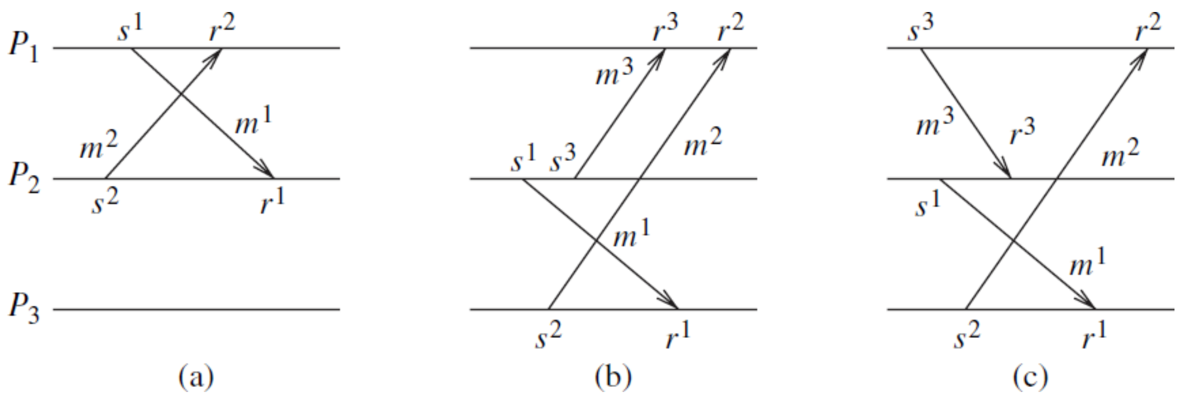


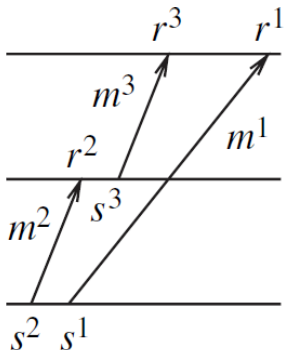
Figure 6.5 Illustrations of asynchronous executions and of crowns. (a) Crown of size 2. (b) Another crown of size 2. (c) Crown of size 3.

2.3 Executions realizable with synchronous communication (RSC)

- In an A-execution, the messages can be made to appear instantaneous if there exists a linear extension of the execution, such that each send event is immediately followed by its corresponding receive event. Such an A-execution that is realized under synchronous communication is called a *realizable with synchronous communication* (RSC) execution.

Non-separated linear extension: A non-separated linear extension of (E,≺) is a linear extension of (E,≺) such that for each pair (s, r) ∈ *T*, the interval { x ∈ E | s ≺ x ≺ r} is empty.

Example:



(CO Executions)

• In the above figure: <s^2, r^2, s^3, r^3, s^1, r^1> is a linear extension that is non separated.
<s^2, s^1, r^2, s^3, r^3, s^1> is a linear extension that is separated.

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

Crown : Let E be an execution. A crown of size k in E is a sequence
<(s^i, r^i), i ∈ {0, ..., k−1}> of pairs of corresponding send and receive events
such that: s0 ≺ r^1, s^1 ≺ r^2,... , s^{k-2} ≺ r^{k-1}, s^{k-1} ≺ r^0.

- On the set of messages *T*, we define an ordering ↪ such that m ↪ m' if and only if s ≺ r'.

2.4 Synchronous program order on an asynchronous system

There do not exist real systems with instantaneous communication that allows for synchronous communication to be naturally realized.

Non-determinism

- This suggests that the distributed programs are *deterministic*, i.e., repeated runs of the same program will produce the same partial order.
- In many cases, programs are *non-deterministic* in the following senses
  1. A receive call can receive a message from any sender who has sent a message, if the expected sender is not specified.
  2. 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, it results in a deadlock.
- However, there is no semantic dependency between the send and immediately following receive. If the receive call at one of the processes is scheduled before the send call, then there is no deadlock.
- **Rendezvous**

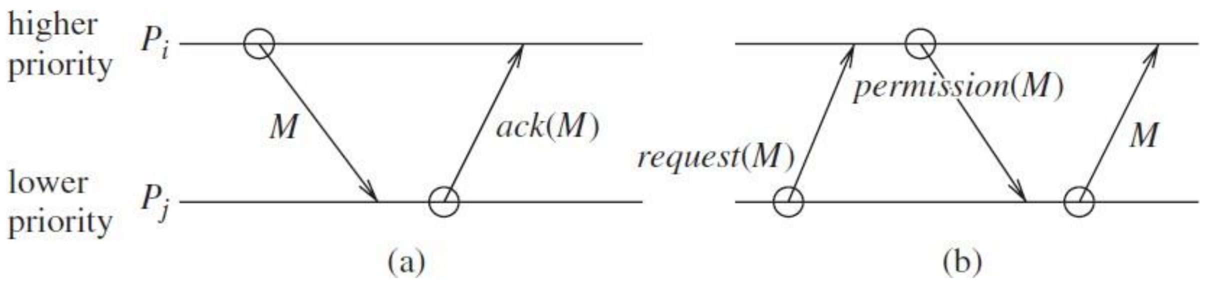
Rendezvous ("meet with each other")

- One form of group communication is called *multiway rendezvous*, which is a synchronous communication among an arbitrary number of asynchronous processes.
- Rendezvous between a pair of processes at a time is called *binary rendezvous* as opposed to the multiway rendezvous.
- Observations about synchronous communication under binary rendezvous:
  - For the receive command, the sender must be specified eventhough the multiple recieve commands exist.
  - Send and received commands may be individually disabled or enabled.
  - Synchronous communication is implemented by scheduling messages using asynchronous communication.
- Scheduling involves pairing of matching send and receive commands that are both enabled.
- The communication events for the control messages do not alter the partial order of execution.

  Algorithm for binary rendezvous
- Each process, has a set of tokens representing the current interactions that are enabled locally. If multiple interactions are enabled, a process chooses one of them and tries to "synchronize" with the partner process.
- The scheduling messages must satisfying the following constraints:
  - Schedule on-line, atomically, and in a distributed manner.
  - Schedule in a deadlock-free manner (i.e., crown-free), such that both the sender and receiver are enabled for a message when it is scheduled.

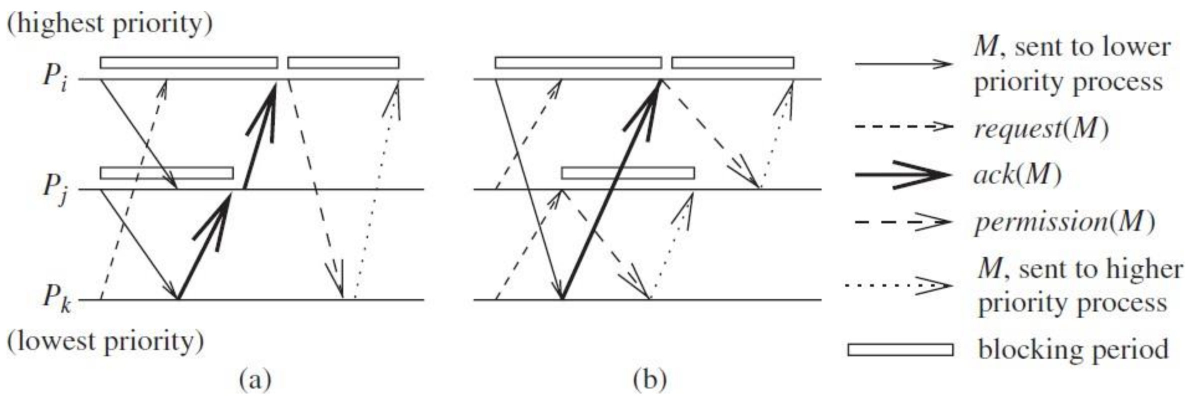
- Schedule to satisfy the progress property (i.e., find a schedule within a bounded number of steps) in addition to the safety (i.e., correctness) property.

- Additional features of a good algorithm are:
  (i) symmetry or some form of fairness, i.e., not favoring particular processes
  (ii) efficiency, i.e., using as few messages as possible
- A simple algorithm by Bagrodia, makes the following assumptions:
  1. Receive commands are forever enabled from all processes.
  2. A send command, once enabled, remains enabled until it completes.
  3. To prevent deadlock, process identifiers are used to break the crowns.
  4. Each process attempts to schedule only one send event at any time.
- The algorithm illustrates how crown-free message scheduling is achieved on-line.



Messages used to implement synchronous order. P_i has higher priority than P_j. (a) P_i issues SEND(M). (b) P_j issues SEND(M).

- The message types used are:
  (i) M – Message is the one i.e., exchanged between any two process during execution
  (ii) ack(M) – acknowledgment for the received message M ,
  (iii) request(M) – when low priority process wants to send a message M to the high priority process it issues this command.
  (iv) permission(M) – response to the request(M) to low priority process from the high priority process.



(Examples showing how to schedule messages sent with synchronous primitives)

- A cyclic wait is prevented because before sending a message M to a higher priority process, a lower priority process requests the higher priority process for permission to synchronize on M, in a non-blocking manner.
- While waiting for this permission, there are two possibilities:
  1. If a message M′ from a higher priority process arrives, it is processed by

a receive and ack(M′) is returned.Thus, a cyclic wait is prevented.
2. Also, while waiting for this permission, if a request(M′) from a lower priority process arrives, a permission(M′) is returned and the process blocks until M′ actually arrives.

Algorithm 6.1 *A simplified implementation of synchronous order.*
Code shown is for process Pi , 1 ≤ i ≤ n.

(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)).
(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 a RECEIVE(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 and blocks waiting for the message M'. When M' arrives, the RECEIVE(M') event is executed.
(2c) When the permission(M) arrives, Pi knows partner Pj is synchronized and Pi executes send(M). The SEND(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 be always 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 a message arrival (as per rules 3 or 4).

2.5 Group communication

- Processes across a distributed system cooperate to solve a task. Hence there is need for *group communication*.
- A *message broadcast* is sending a message to all members.
- In *Multicasting* the message is sent to a certain subset, identified as a *group*.
- In *unicasting* is the point-to-point message communication.
- Broadcast and multicast is supported by the network protocol stack using variants of the spanning tree. This is an efficient mechanism for distributing information.

- However, the hardware or network layer protocol assisted multicast cannot efficiently provide the following 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.
- If a multicast algorithm requires the sender to be a part of the destination group, the multicast algorithm is said to be a *closed group* algorithm.
- If the sender of the multicast can be outside the destination group, then the multicast algorithm is said to be an *open group* algorithm.
- Open group algorithms are more general, and therefore more difficult to design and more expensive to implement, than closed group algorithms.
- Closed group algorithms cannot be used in in a large system like on-line reservation or Internet banking systems where client processes are short-lived and in large numbers.
- For multicast algorithms, the number of groups may be potentially exponential, i.e., $O(2^n)$.

2.6 Total order

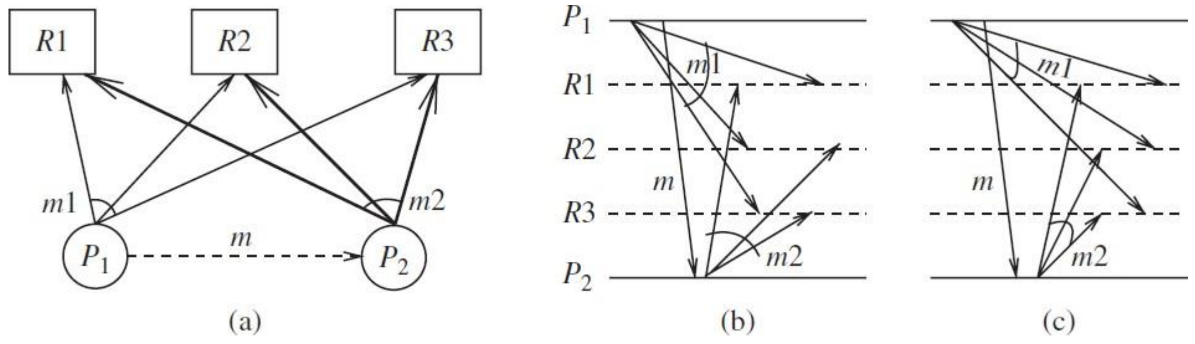
- For example of updates to replicated data would be logical only if all replicas see the updates in the same order.

Definition 6.14 (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.

Example

- The execution in Figure 6.11(b) does not satisfy total order. Even
- if the message m did not exist, total order would not be satisfied. The execution
- in Figure 6.11(c) satisfies total order.



Centralized algorithm for total order

- Algorithm Assumes all processes broadcast messages.
- It enforces total order and also the causal order in a system with FIFO channels.
- Each process sends the message it wants to broadcast to a centralized process.
- The centralized process relays all the messages it receives to every other process over FIFO channels.

Algorithm : centralized algorithm to implement total order & causal order of messages.

(1) When process Pi wants to multicast a message M to group G:

(1a) **send** M(i,G) to central coordinator.

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

(3) When M(i,G) arrives at Pj 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

Three-phase distributed algorithm

- It enforces total and causal order for closed groups.
- The three phases of the algorithm are defined as follows:

Sender

Phase 1

- A process multicasts the message M to the group members with the
  - a locally unique tag and
  - the local timestamp

Phase 2

- Sender process awaits for the reply from all the group members who respond with a tentative proposal for a revised timestamp for that message M.
- it is an non-blocking await i.e., any other messages received in the meanwhile are processed.
- Once all expected replies are received, the process computes the maximum of proposed timestamps for M, and uses the maximum as final timestamp.

Phase 3

- The process multicasts the final timestamp to the group members of phase 1.

***Algorithm:* D*istributed algorithm to implement total order & causal order of messages. Code at P_i, $1 \le i \le n$.*

```
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 Pi, with initial timestamp ts
PROPOSED_TS(j, i, tag, ts) // Phase 2 message sent by Pj , with revised timestamp, to $P_i$
FINAL_TS(i, tag, ts)          // Phase 3 message sent by Pi, with final timestamp
(1) When process Pi 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, x, clk) arrives from $P_j$ :
    (3a) Identify entry Q_e in temp_Q, where Q_e.tag = x;
    (3b) mark Q_e.deliverable as true;
    (3c) Update Q_e.timestamp to clk and re-sort temp_Q based on the timestamp field;
    (3d) if (head(temp_Q)).tag = Q_e.tag then
    (3e)     move Q_e from temp_Q to delivery_Q;
    (3f)      while (head(temp_Q)).deliverable is true do
    (3g)           dequeue head(temp_Q) and insert in delivery_Q.
(4) When Pi removes a message (M, tag, j, ts, deliverable) from head(delivery_$Q_i$):
    (4a) clock←max(clock, ts)+1.
```

Receivers

Phase 1

- The receiver receives the message with a tentative/proposed 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 (and the tag) back to the sender.
- The receiver then waits in a non-blocking manner for the final timestamp (correlated by the message tag).

Phase 3

- In the third phase, 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.

- 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 in that order.

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.

Example An example execution to illustrate the algorithm is given in Figure 6.14. Here, A and B multicast to a set of destinations and C and D are the common destinations for both multicasts.

- Figure 6.14a. The main sequence of steps is as follows:

1. A sends a *REVISE_TS*(7) message, having timestamp 7. B sends a *REVISE_TS*(9) message, having timestamp 9.
2. C receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 7. C then sends *PROPOSED_TS*(7) message to A.
3. D receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 9. D then sends *PROPOSED_TS*(9) message to B.
4. C receives B's *REVISE_TS*(9), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 9. C then sends *PROPOSED_TS*(9) message to B.
5. D receives A's *REVISE_TS*(7), enters the corresponding message in *temp_Q*, and marks it as undeliverable; *priority* = 10. D assigns a tentative timestamp value of 10, which is greater than all of the timestamps on *REVISE_TS*s seen so far, and then sends *PROPOSED_TS*(10) message to A.

The state of the system is as shown in the figure.

• **Figure 6.14(b)** The main steps is as follows:

6. When A receives *PROPOSED_TS*(7) from C and *PROPOSED_TS*(10) from D, it computes the final timestamp as max(7, 10) = 10, and sends *FINAL_TS*(10) to C and D.
7. When B receives *PROPOSED_TS*(9) from C and *PROPOSED_TS*(9) from D, it computes the final timestamp as max(9, 9)= 9, and sends *FINAL_TS*(9) to C and D.
8. C receives *FINAL_TS*(10) from A, updates the corresponding entry in *temp_Q* with the timestamp, resorts the queue, and marks the message as deliverable. As the message is not at the head of the queue, and some entry ahead of it is still undeliverable, the message is not moved to *delivery_Q*.
9. D receives *FINAL_TS*(9) from B, updates the corresponding entry in *temp_Q* by marking the corresponding message as deliverable, and resorts the queue. As the message is at the head of the queue, it is moved to *delivery_Q*.
10. When C receives *FINAL_TS*(9) from B, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*, and the next message (of A), which is also deliverable, is also moved to the *delivery_Q*.

11. When D receives *FINAL_TS*(10) from A, it will update the corresponding entry in *temp_Q* by marking the corresponding message as deliverable. As the message is at the head of the queue, it is moved to the *delivery_Q*.

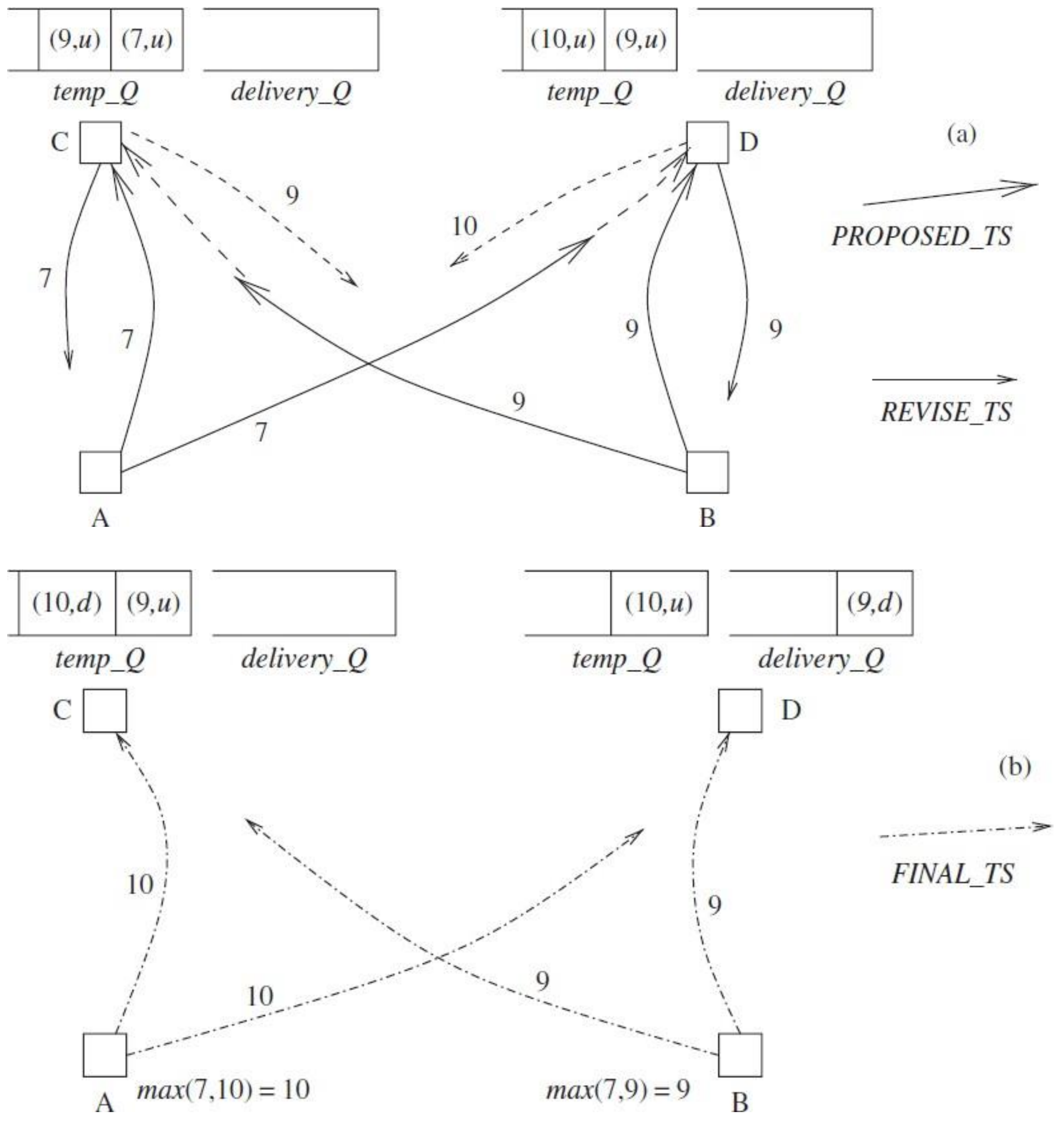


Figure *An example to illustrate the three-phase total ordering algorithm. (a) A snapshot for PROPOSED_TS and REVISE_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL_TS messages in the example.*

Global state and snapshot recording Algorithms

2.7Introduction

distributed computing system consists of spatially separated processes that do not share a common memory and communicate asynchronously with each other by message passing over communication channels.

- Each component of a distributed system has a local state. The state of a process is the state of its local memory and a history of its activity.
- The state of a channel is the set of messages in the transit.
- The global state of a distributed system is the collection of states of the process and the channel.
- Applications that use the global state information are :
- deadlocks detection
- failure recovery,
- for debugging distributed software
- If shared memory is available then an up-to-date state of the entire system is available to the processes sharing the memory.
- The absence of shared memory makes difficult to have the coherent and complete view of the system based on the local states of individual processes.
- A global snapshot can be obtained if the components of distributed system record their local states at the same time. This is possible if the local clocks at processes were perfectly synchronized or a global system clock that is instantaneously read by the processes.
- However, it is infeasible to have perfectly synchronized clocks at various sites as the clocks are bound to drift. If processes read time from a single common clock (maintained at one process), various indeterminate transmission delays may happen.
- In both cases, collection of local state observations is not meaningful, as discussed below.
- **Example:**
  - Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by C12 and C21, respectively.
- Consider the following sequence of actions, which are also illustrated in the timing
- diagram of Figure 4.1:
- Time t0: Initially, Account A=$600, Account B=$200, C12 =$0, C21=$0.
- Time t1: Site S1 initiates a transfer of $50 from A to B. Hence,
  A= $550, B=$200, C12=$50, C21=$0.
- Time t2: Site S2 initiates a transfer of $80 from Account B to A. Hence,
  A= $550,B=$120, C12 =$50, C21=$80.
- Time t3: Site S1 receives the message for a $80 credit to Account A. Hence,
  A=$630, B=$120, C12 =$50, C21 =$0.
- Time t4: Site S2 receives the message for a $50 credit to Account B. Hence,
  A=$630, B=$170, C12=$0, C21=$0.
- Suppose the local state of Account A is recorded at time t0 which is $600 and the local state of Account B and channels C12 and C21 are recorded at time t2 are $120, $50, and $80, respectively.
- Then the recorded global state shows $850 in the system. An extra $50 appears in the system.

- Reason: Global state recording activities of individual components must be coordinated.

2.8System model and definitions

System model

- The system consists of a collection of n processes, p_1, p_2,…, p_n, that are connected by channels.
- There is no globally shared memory and processes communicate solely by passing messages (send and receive) asynchronously i.e., delivered reliably with finite but arbitrary time delay.
- There is no physical global clock in the system.
- The system can be described as a directed graph where vertices represents processes and edges represent unidirectional communication channels.
- Let C_{ij} denote the channel from process p_i to process p_j .
- Processes and channels have states associated with them.
- *Process State*: is the contents of processor registers, stacks, local memory, etc., and dependents on the local context of the distributed application.
- *Channel State of* C_{ij}: is SC_{ij} , is the set of messages in transit of the channel.
- The actions performed by a process are modeled as three types of events,
- internal events – affects the state of the process.
- message send events, and
- message receive events.
- For a message m_{ij} that is sent by process pi to process p_j, let send(m_{ij}) and rec(m_{ij}) denote its send and receive events affects state of the channel, respectively.
- The events at a process are linearly ordered by their order of occurrence.
- At any instant, the state of process pi, denoted by LS_i, is a result of the sequence of all the events executed by p_i up to that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process pi to state LS_i.
- For a channel C_{ij} , the following set of messages will be:
- **Transit** : $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j \}$
- There are several models of communication among processes.
- In the FIFO model, each channel acts as a first-in first-out message queue hence, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- In causal delivery of messages satisfies the following property:
  "for any two messages m_{ij} and m_{kj},
  if $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$."
- Causally ordered delivery of messages implies FIFO message delivery.

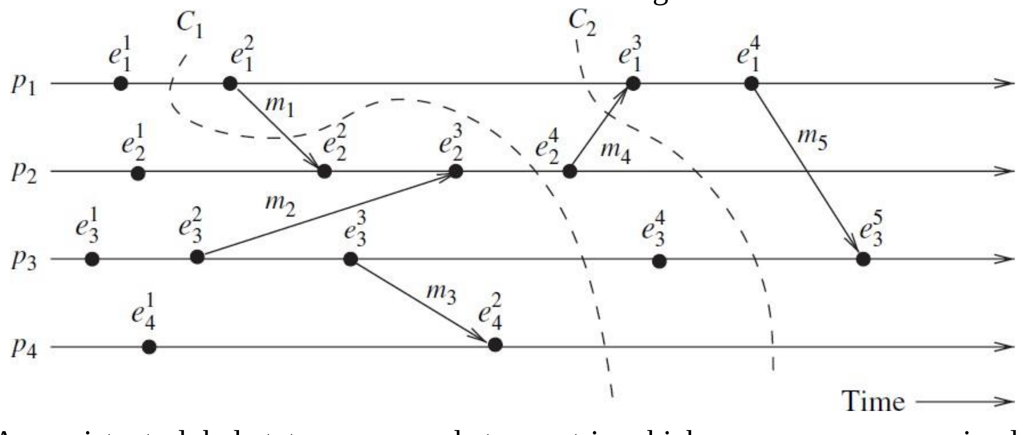
- The causal ordering model is useful in developing distributed algorithms and may simplify the design of algorithms.

A consistent global state

- The global state of a distributed system is a collection of the local states of
- the processes and the channels. Notationally, global state GS is defined as GS $=\{\cup_i LS_i, \cup_{i,j} SC_{ij}\}$.
- A global state GS is a *consistent global state* iff it satisfies the following two conditions:
  C1: *send*$(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus$ *rec*$(m_{ij}) \in LS_j$ ($\oplus$ is the Ex-OR operator).
  C2: *send*$(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge$ *rec*$(m_{ij}) \notin LS_j$.
- In a consistent global state, every message that is recorded as received is also recorded as sent. These are meaningful global states.
- The inconsistent global states are not meaningful ie., without send if receive of the respective message exists.

Interpretation in terms of cuts

- Cuts is a zig-zag line that connects a point in the space–time diagram at some arbitrary point in the process line.
- Cut is a powerful graphical aid for representing and reasoning about the global states of a computation.
- Left side of the cut is referred as PAST event and right is referred as FUTURE event.



- A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut. Such a cut is known as a *consistent cut*. Example: Cut C2 in the above figure.
- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state.
- If the flow is from the FUTURE to the PAST is inconsistent. Example: Cut C1.

Issues in recording a global state

- If a global physical clock is used then the following simple procedure is used to record a consistent global snapshot of a distributed system.
  - Initiator of the snapshot decides a future time at which the snapshot is to be taken and broadcasts this time to every process.
  - All processes take their local snapshots at that instant in the global time.

  - The snapshot of channel C_{ij} includes all the messages that process p_j receives after taking the snapshot and whose timestamp is smaller than the time of the snapshot.

- However, a global physical clock is not available in a distributed system. Hence the following two issues need to be addressed to record a consistent global snapshot.
- **I1:** *How to distinguish between the messages to be recorded in the snapshot from those not to be recorded?*
- Any message i.e., sent by a process before recording its snapshot, must be recorded in the global snapshot. (from **C1**).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).
- **I2:** How to determine the instant when a process takes its snapshot.
- A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.
- These algorithms use two types of messages: computation messages and control messages. The former are exchanged by the underlying application and the latter are exchanged by the snapshot algorithm.

2.9 Snapshot algorithms for FIFO channels

Chandy–Lamport algorithm

- This algorithm uses a control message, called a *marker*.
- After a site has recorded its snapshot, it sends a *marker* along all of its outgoing channels before sending out any more messages.
- Since channels are FIFO, marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot. This addresses issue **I1**.
- The role of markers in a FIFO system is to act as delimiters for the messages in the channels so that the channel state recorded by the process at the receiving end of the channel satisfies the condition **C2**.
- Since all messages that follow a marker on channel C_{ij} have been sent by process p_i after p_i has taken its snapshot, process p_j must record its snapshot if not recorded earlier and record the state of the channel that was received along the marker message. This addresses issue **I2**.
- **The algorithm**
- The algorithm is initiated by any process by executing the *marker sending rule*.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- **Algorithm 4.1** The Chandy–Lamport algorithm.

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 pj
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
```

Correctness

- To prove the correctness of the algorithm, it is shown that a recorded snapshot satisfies conditions **C1** and **C2**.
- Since a process records its snapshot when it receives the first marker on any incoming channel, no messages that follow markers on the channels incoming to it are recorded in the process’s snapshot.
- Moreover, a process stops recording the state of an incoming channel when a marker is received on that channel.
- Due to FIFO property of channels, it follows that no message sent after the marker on that channel is recorded in the channel state. Thus, condition **C2** is satisfied.
- When a process p_j receives message m_{ij} that precedes the marker on channel C_{ij}, it acts as follows:
- If process 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 the diameter of the network.