model inherently accounts for different execution schedules. For example, an execution that does not violate mutual exclusion in the time-based model may do so with a different execution schedule. This problem is avoided in the happened before model.

It is instructive to observe that a consistent global state is not simply a product of local states. To appreciate this, consider a distributed database for a banking application. Assume for simplicity that there are only two sites which keep the accounts for a customer. Also assume that the customer has $500 at the first site and $300 at the second site. In the absence of any communication between these sites, the total money of the customer can be easily computed to be $800. However, if there is a transfer of $200 from site A to site B, and a simple procedure is used to add up the accounts, we may falsely report that the customer has a total of $1000 in his accounts (to the chagrin of the bank). This happens when the value at the first site is used before the transfer and the value at the second site after the transfer. It is easily seen that these two states are not concurrent. Note that $1,000 cannot be justified even by the messages in transit (or, that “the check is in the mail”).

Figure 17.1: Consistent and inconsistent cuts

Figure 18.1 depicts a distributed computation. The dashed lines labeled $G_1$ and $G_2$ represent global states that consist of local states at $P_1$, $P_2$, and $P_3$ where $G_1$ and $G_2$ intersect. Because a global state can be visualized in such a figure as a cut across the computation, the term “cut” and “global snapshot” are used interchangeably with “global state.” The cut $G_1$ in this computation is not consistent because it records the message $m_2$ as having been received but not sent. This is clearly impossible. The cut $G_2$ is consistent. The message $m_3$ in this cut has been sent but not yet received. Thus it is a part of the channel from process $P_1$ to $P_3$. Our example of the distributed database illustrates the importance of recording only the consistent cuts.

17.2 Global Snapshot Algorithm

In this section, we describe an algorithm to take a global snapshot of a distributed system. The algorithm computes a consistent cut or a consistent subcut as desired. The computation of the snapshot is initiated by one or more processes. We assume that all channels are unidirectional and satisfy the FIFO property. Assuming that channels are unidirectional is not restrictive because a bidirectional channel can simply be modeled by using two unidirectional channels. The assumption that channels are FIFO is essential to the correctness of the algorithm as explained later.

The algorithm is shown in Figure 18.3. We associate with each process a variable called color that is either white or red. Intuitively, the computed global snapshot corresponds to the state of the system just before the processes turn red. All processes are initially white. After recording the local state, a process turns red. Thus the state of a local process is simply the state just before it turned red.

There are two difficulties in the design of rules for changing the color for the global snapshot algorithm. First, we need to ensure that the recorded local states are mutually concurrent. Second, we also need a mechanism to capture the state of the channels. To address these difficulties, the algorithm relies on a special message called a marker. Once a process turns red, it is required to send a marker along all its outgoing channels before it sends out any message. A process is required to turn red on receiving a marker if it has not already done so. Since channels
are FIFO, the above rule guarantees that no white process ever receives a message sent by a red process. This in turn guarantees that local states are mutually concurrent.

![Diagram of message classification](image)

Figure 17.2: Classification of messages

Now let us turn our attention to the problem of computing states of the channels. Figure 18.2 shows that messages in the presence of colors can be of four types:

1. **(ww messages)** These are the messages sent by a white process to a white process. These messages correspond to the messages sent and received before the global snapshot.

2. **(rr messages)** These are the messages sent by a red process to a red process. These messages correspond to the messages sent and received after the global snapshot.

3. **(rw messages)** These are the messages sent by a red process received by a white process. In the figure, they cross the global snapshot in the backward direction. The presence of any such message makes the global snapshot inconsistent. The reader should verify that such messages are not possible if a marker is used.

4. **(wr messages)** These are the messages sent by a white process received by a red process. These messages cross the global snapshot, or cut, in the forward direction and form the state of the channel in the global snapshot because they are in transit when the snapshot is taken.

To record the state of the channel, \( P_j \) starts recording all messages it receives from \( P_i \) after turning red. Since \( P_i \) sends a marker to \( P_j \) on turning red, the arrival of the marker at \( P_j \) from \( P_i \) indicates that there will not be any further white messages from \( P_i \) sent to \( P_j \). It can, therefore, stop recording messages once it has received the marker.

The program shown in Figure 18.3 uses \( chan[k] \) to record the state of the \( k^{th} \) incoming channel and \( closed[k] \) to stop recording messages along that channel. In the program, we say that \( P_j \) is a neighbor of \( P_i \) if there is a channel from \( P_i \) to \( P_j \).

In the algorithm, any change in the value of color must be reported to all neighbors. On receiving any such notification, a process is required to update its own color. This may result in additional messages because of the semantics of the event turn_red. The net result is that if one process turns red, all processes that can be reached directly or indirectly from that process also turn red. Thus if any process turns red, all other processes will also eventually turn red.

The above algorithm requires that a marker be sent along all channels. Thus it has an overhead of \( E \) messages where \( E \) is the number of unidirectional channels in the system. We have not discussed the overhead required to combine local snapshots into a global snapshot. A simple method would be for all processes to send their local snapshots to a predetermined process, say \( P_0 \).
class camera {
    int myColor = WHITE;
    static final int WHITE = 0, RED = 1;
    boolean closed[];
    int numProc;
    int myID;
    Linker comm;

    public void camera(int N, int id, Linker initComm) {
        numProc = N;
        myID = id;
        comm = initComm;
        closed = new boolean[numProc];
        for(int i=0; i < numProc; i++) closed[i] = false;
        closed[myID] = true;
    }

    public void turnRed() {
        myColor = RED;
        // recordState();
        for(int i=0; i < numProc; i++) // send Markers
            if (i != myID) comm.sendMessage(i, "marker", 0);
    }

    public void markerHandler(int source_id) {
        if (myColor == WHITE) turnRed();
        closed[source_id] = true;
        checkDone();
    }

    public void checkDone() {
        boolean done = true;
        if (myColor == WHITE) done = false;
        for(int i=0; i < numProc; i++) {
            if (closed[i] == false) done = false;
        }
        if (done) System.out.println("done");
    }

    public void recordChannel(int logID, int event, int msg, int len) {
        if (closed[logID] == false)
            chan[logID] = chan[logID] + event + msg;
    }
}

Figure 17.3: Chandy and Lamport’s snapshot algorithm
17.3. UNSTABLE PREDICATE DETECTION

17.2.1 Application: Detecting Stable Properties

Computation of a global snapshot is useful in many contexts. It can be be used to detect a stable property of a distributed computation. To define stable predicates we use the notion of the reachability of one global state from another. For two global states \( G \) and \( H \), we say that \( G \leq H \) if \( H \) is reachable from \( G \). A predicate \( B \) is stable iff

\[
\forall G, H : G \leq H : B(G) \Rightarrow B(H)
\]

In other words, a property \( B \) is stable if once it becomes true, it stays true. Some examples of stable properties are deadlock, termination, and loss of a token. Once a system has deadlocked or terminated, it remains in that state. A simple algorithm to detect a stable property is as follows. Compute a global snapshot \( S \). If the property \( B \) is true in the state \( S \), then we are done. Otherwise, we repeat the process after some delay. It is easily seen that if the stable property ever becomes true, the algorithm will detect it. Conversely, if the algorithm detects that some stable property \( B \) is true, then the property must have become true in the past (and is therefore also true currently).

Formally, if the global snapshot computation was started in the global state \( G_i \), and the recorded state is \( G_s \), then the following is true.

1. \( B(G_s) \Rightarrow B(G_f) \)
2. \( \neg B(G_s) \Rightarrow \neg B(G_f) \)

Note that the converse of (1) and (2) may not hold.

The global snapshot algorithm can also be used for providing fault tolerance in distributed systems. On failure, the system can be restarted from the last snapshot. Finally, snapshots can also be used for distributed debugging. Inspection of intermediate snapshots may sometimes reveal the source of an error.

At this point it is important to observe some limitations of the snapshot algorithm for detection of global properties.

- The algorithm is not useful for unstable predicates. An unstable predicate may turn true only between two snapshots.
- In many applications (such as debugging), it is desirable to compute the least global state that satisfies some given predicate. The snapshot algorithm cannot be used for this purpose.
- The algorithm may result in an excessive overhead depending on the frequency of snapshots. A process in Chandy and Lamport’s algorithm is forced to take a local snapshot upon receiving a marker even if it knows that the global snapshot that includes its local snapshot cannot satisfy the predicate being detected. For example, suppose that the property being detected is termination. Clearly, if a process is not terminated then the entire system could not have terminated. In this case, computation of the global snapshot is a wasted effort.

17.3 Unstable Predicate Detection

In this section, we discuss an algorithm to detect unstable predicates. We will assume that the given global predicate, say \( B \), is constructed from local predicates using boolean connectives. We first show that \( B \) can be detected using an algorithm that can detect \( q_i \) where \( q \) is a pure conjunction of local predicates. The predicate \( B \) can be rewritten in its disjunctive normal form. Thus,

\[
B = q_1 \lor \ldots \lor q_k \quad k \geq 1
\]

where each \( q_i \) is a pure conjunction of local predicates. Next, observe that a global cut satisfies \( B \) if and only if it satisfies at least one of the \( q_i \)'s. Thus the problem of detecting \( B \) is reduced to solving \( k \) problems of detecting \( q \), where \( q \) is a pure conjunction of local predicates.

As an example, consider a distributed program in which \( x, y \) and \( z \) are in three different processes. Then,

\[
even(x) \land ((y < 0) \lor (z > 6))
\]
can be rewritten as

$$(\text{even}(x) \land (y < 0)) \lor (\text{even}(x) \land (z > 6))$$

where each disjunct is a conjunctive predicate.

Note that even if the global predicate is not a boolean expression of local predicates, but is satisfied by a finite number of possible global states, then it can also be rewritten as a disjunction of conjunctive predicates. For example, consider the predicate \((x = y)\), where \(x\) and \(y\) are in different processes. \((x = y)\) is not a local predicate because it depends on both processes. However, if we know that \(x\) and \(y\) can only take values \(\{0, 1\}\), then the above expression can be rewritten as

$$((x = 0) \land (y = 0)) \lor ((x = 1) \land (y = 1)).$$

Each of the disjuncts in this expression is a conjunctive predicate.

We have emphasized conjunctive predicates and not disjunctive predicates. The reason is that disjunctive predicates are quite simple to detect. To detect a disjunctive predicate \(l_1 \lor l_2 \lor \ldots \lor l_n\), where \(l_i\) denotes a local predicate in the process \(P_i\), it is sufficient for the process \(P_i\) to monitor \(l_i\). If any of the processes finds its local predicate true, then the disjunctive predicate is true.

Formally, we define a weak conjunctive predicate (WCP) to be true for a given run if and only if there exists a consistent global cut in that run in which all conjuncts are true. Intuitively, detecting a weak conjunctive predicate is generally useful when one is interested in detecting a combination of states that is unsafe. For example, violation of mutual exclusion for a two process system can be written as: “\(P_1\) is in the critical section and \(P_2\) is in the critical section.” We first give a necessary and sufficient condition for a weak conjunctive predicate to be true in a computation \((S_1, S_2, \ldots, S_N, \leadsto)\) where \(S_i\) is the set of states on process \(P_i\). Let \(l_i(s)\) denote that the predicate \(l_i\) is true in the state \(s\).

Our aim is to detect whether \((l_1 \land l_2 \land \ldots l_n)\) holds for a given computation. We can assume \(n \leq N\) (the total number of processes in the system) because \(l_i \land l_j\) is just another local predicate if \(l_i\) and \(l_j\) belong to the same process. From the definition of the consistent state, we get that

\[(l_1 \land l_2 \land \ldots l_n)\] is in a consistent global state of a computation iff for all \(1 \leq i \leq n\), \(\exists s_i \in S_i\) such that \(l_i\) is true in state \(s_i\), and \(s_i\) and \(s_j\) are concurrent for \(i \neq j\).

Thus it is necessary and sufficient to find a set of incomparable states in which local predicates are true to detect a weak conjunctive predicate. We now present an algorithm to do so. This algorithm finds the first consistent cut for which a WCP is true.

In this algorithm, one process serves as a checker. All other processes involved in detecting the WCP are referred to as application processes. Each application process checks for local predicates. It also maintains the vector clock algorithm. Whenever the local predicate of a process becomes true for the first time since the most recently sent message (or the beginning of the trace), it generates a debug message containing its local timestamp vector and sends it to the checker process.

Note that a process is not required to send its vector clock every time the local predicate is detected. If two local states, say \(s\) and \(t\), on the same process are separated only by internal events, then they are indistinguishable to other processes so far as consistency is concerned, that is, if \(u\) is a local state on some other process, then \(s || u\) if and only if \(t || u\). Thus it is sufficient to consider at most one local state between two external events and the vector clock need not be sent if there has been no message activity since the last time the vector clock was sent.

The checker process is responsible for searching for a consistent cut that satisfies the WCP by considering a sequence of candidate cuts. If the candidate cut either is not a consistent cut or does not satisfy some term of the WCP, the checker can efficiently eliminate one of the states along the cut. The eliminated state can never be part of a consistent cut that satisfies the WCP. The checker can then advance the cut by considering the successor to one of the eliminated states on the cut. If the checker finds a cut for which no state can be eliminated, then that cut satisfies the WCP and the detection algorithm halts. The algorithm for the checker process is shown in Figure 18.4.

The checker receives local snapshots from the other processes in the system. These messages are used by the checker to create and maintain data structures that describe the global state of the system for the current cut. The data structures are divided into two categories: queues of incoming messages and those data structures that describe the state of the processes.
17.3. UNSTABLE PREDICATE DETECTION

```
var
  cut: array[1..n] of struct
  v: array[1..n] of integer;
  color: {red, green};
endstruct initially (\forall i : cut[i].color = red);
detect: boolean initially false;

while (\exists i : (cut[i].color = red)) do
  if (q[i] = null) and P_i terminated then return false;
  else cut[i],v := receive(q[i]);// advance the cut
  paintState(i);
endwhile;

detect := true;
```

Figure 17.4: WCP detection algorithm—checker process.

The queue of incoming messages is used to hold incoming local snapshots from application processes. We require that messages from an individual process be received in FIFO order. We abstract the message passing system as a set of \( n \) FIFO queues, one for each process. We use the notation \( q[1...n] \) to label these queues in the algorithm.

The checker also maintains information describing one state from each process \( P_i \). The collection of this information is organized into a vector \( \text{cut} \) which is an array of structure consisting of the vector \( v \) and \( \text{color} \). The color of a state is either red or green and indicates whether the state has been eliminated in the current cut. A state is green only if it is concurrent with all other green states. A state is red only if it cannot be part of a consistent cut that satisfies the WCP.

The aim of advancing the cut is to find a new candidate cut. However, we can advance the cut only if we have eliminated at least one state along the current cut and if a message can be received from the corresponding process. The data structures for the processes are updated to reflect the new cut. This is done by the procedure \( \text{paintState} \). This procedure is shown in Figure 18.5. The parameter \( i \) is the index of the process from which a local snapshot was most recently received. The color of \( \text{cut}[i] \) is temporarily set to green. It may be necessary to change some green states to red to preserve the property that all green states are mutually concurrent. Hence, we must compare the vector clock of \( \text{cut}[i] \) to each of the other green states. Whenever the states are comparable, the smaller of the two is painted red.

```
paintState(i)
  cut[i].color := green;
  for j := 1 to n do
    if (cut[j].color = green) then
      if (cut[i],v < cut[j],v) then cut[i].color := red;
      else if (cut[j],v < cut[i],v) then cut[j].color := red;
  endfor
```

Figure 17.5: Procedure \( \text{paintState} \).

17.3.1 Overhead Analysis

Let \( n \) denote the number of processes involved in the WCP, and \( m \) denote the maximum number of messages sent or received by any process.
CHAPTER 17. DETECTING GLOBAL PREDICATES

The main time complexity is involved in detecting the local predicates and time required to maintain vector clocks. In the worst case, one debug message is generated for each program message sent, so the worst case message complexity is \( O(m) \). In addition, program messages have to include time vectors. Every process sends \( O(m) \) local snapshots to the checker process. With the same assumptions as made for space complexity, it follows that \( O(mn) \) bits are sent by each process.

The main space requirement of the checker process is the buffer for the local snapshots. Each local snapshot consists of a vector clock that requires \( O(n) \) space. Since there are at most \( O(mn) \) local snapshots, \( O(n^2m) \) total space is required to hold the component of local snapshots devoted to vector clocks. Therefore, the total amount of space required by the checker process is \( O(n^2m) \).

We now discuss the time complexity of the checker process. Note that it takes only two comparisons to check whether two vectors are concurrent. Hence, each invocation of \( paintState \) requires at most \( n \) comparisons. This function is called at most once for each state, and there are at most \( mn \) states. Therefore, at most \( n^2m \) comparisons are required by the algorithm.

17.4 Problems

17.1. Chandy and Lamport’s algorithm requires the receiver to record the state of the channel. Since messages in real channels may get lost, it may be advantageous for senders to record the state of the channel. Give an algorithm to do so. Try to make your algorithm as practical as possible. Assume that control messages can be sent over unidirectional channels even in the reverse direction.

17.2. The original algorithm proposed by Chandy and Lamport does not require FIFO but a condition weaker than that. Specify the condition formally.

17.3. How can you use Lamport’s logical clock to compute a consistent global snapshot?

17.4. Assume that the given global predicate is a simple conjunction of local predicates. Further assume that the global predicate is stable. In this scenario, both Chandy and Lamport’s algorithm and the weak conjunctive algorithm can be used to detect the global predicate. What are the advantages and disadvantages of using each of them?
Chapter 18

Message Ordering

18.1 Introduction

Distributed programs are difficult to design and test because of their nondeterministic nature, that is, a distributed program may exhibit multiple behaviors on the same external input. This nondeterminism is caused by reordering of messages in different executions. It is sometimes desirable to control this nondeterminism by restricting the possible message ordering in a system.

![Diagram of message ordering](image)

Figure 18.1: A FIFO computation that is not causally ordered

A fully asynchronous computation does not have any restriction on the message ordering. It permits maximum concurrency, but algorithms based on fully asynchronous communication can be difficult to design because they are required to work for all ordering of the messages. Therefore, many systems restrict message delivery to a FIFO order. This results in simplicity in design of distributed algorithms based on the FIFO assumption. For example, we used the FIFO assumption in Lamport’s algorithm for mutual exclusion and Chandy and Lamport’s algorithm for a global snapshot.

A FIFO-ordered computation is implemented generally by using sequence numbers for messages. However, observe that by using FIFO ordering, a program loses some of its concurrency. When a message is received out of order, its processing must be delayed.

A stronger requirement than FIFO is that of causal ordering. Intuitively, causal ordering requires that a single message should not be overtaken by a sequence of messages. For example, the computation in Figure 19.1 satisfies FIFO ordering of messages but does not satisfy causal ordering. A sequence of messages from $P_1$ to $P_2$ and from $P_2$ to $P_3$ overtakes a message from $P_1$ to $P_3$ in this example. Causal ordering of messages is useful in many contexts. In Chapter 16, we considered the problem of mutual exclusion. Assume that we use a centralized coordinator for
granting requests to the access of the critical section. The fairness property requires that the requests be honored in the order they are made (and not in the order they are received). It is easy to see that if the underlying system guaranteed a causal ordering of messages, then the order in which requests are received cannot violate the order in which they are made. For another example of the usefulness of causal ordering, see Problem 19.1.

The relationship among various message orderings can be formally specified based on the happened before relation. For convenience, we denote the receive event corresponding to the send event $s_i$ by $r_i$ and vice versa. The message is represented as $(s_i, r_i)$. Thus $s_i \sim r_i$ for any $i$.

Now, FIFO and causally ordered computations can be defined as follows.

**FIFO**: Any two messages from a process $P_i$ to $P_j$ are received in the same order as they were sent. Formally, let $s_1$ and $s_2$ be any two send events and $r_1$ and $r_2$ be corresponding receive events. Then,

$$s_1 \prec s_2 \implies \neg(r_2 \prec r_1).$$

(CIF)

**Causally Ordered**: Let any two send events $s_1$ and $s_2$ in a distributed computation be related such that the first send happened before the second send. Then, the second message cannot be received before the first message by any process. Formally,

$$s_1 \rightarrow s_2 \implies \neg(r_2 \prec r_1).$$

(CO)

### 18.2 Algorithm

We now describe an algorithm to ensure causal ordering of messages. We assume that a process never sends any message to itself. Each process maintains a matrix $m$ of integers. The entry $s.m[i, j]$ records the number of messages sent by process $P_j$ to process $P_i$ as known by process $P_j$ in the state $s$. The algorithm for process $P_i$ is given in Figure 19.2. Whenever a message is sent from $P_i$ to $P_j$, the matrix $m$ is piggybacked with the message. The entry $m[i, j]$ is incremented to reflect the fact that one more message has been sent from $P_i$ to $P_j$. Whenever messages are received by the communication system at $P_i$, they are first checked for eligibility before delivery to $P_i$. If a message is not eligible it is simply buffered until it becomes eligible. In the following discussion, we identify a message with the state it is sent from. A message $u$ is eligible to be received at state $s$ when the number of messages sent from any process $P_k$ to $P_i$, as indicated by the matrix $u.m$ in the message, is less than or equal to the number recorded in the matrix $s.m$. Formally, this condition is

$$\forall k : s.m[k, i] \geq u.m[k, i]$$

If for some $k$, $u.m[k, i] > s.m[k, i]$, then there is a message that was sent in the causal history of the message $u$ and has not arrived by the state $s$. Therefore, $P_i$ must wait for that message to be delivered before it can accept the message $u$.

### 18.3 Synchronous and Total Ordering

Synchronous ordering is a stronger requirement than causal ordering. A computation satisfies synchronous ordering of messages if it is equivalent to a computation in which all messages are logically instantaneous. Figure 19.3 gives an example of a synchronously ordered computation and Figure 19.4 an example of a computation that does not satisfy synchronous ordering.

Algorithms for synchronous systems are easier to design than those for causally ordered systems. The model of synchronous message passing lets us reason about a distributed program under the assumption that messages are instantaneous or “points” rather than “intervals” (i.e., we can always draw the time diagrams for the distributed programs with the message arrows being vertical). If we assume messages as points instead of intervals, we can order the messages as a partial order and therefore, we can have vector clocks with respect to messages. One of the applications for synchronous ordering of messages is that it enables us to reason about distributed objects as if they were centralized. Assume that a process invokes an operation on a remote object by sending a message. If
synchronous ordering of messages is assumed, then all operations on the objects can be ordered based on when the messages are sent because messages can be considered instantaneous.

Total ordering of messages is useful when messages may be multicast. One of the applications of total ordering of messages is in implementing an object that is replicated across multiple sites. If different processes invoke different operations on the object, the sequence of operations applied at each site must be identical. Implementing this constraint in a distributed system is significantly easier if there is a total ordering of messages.

18.3.1 Synchronous Ordering

A computation is synchronous if its time diagram can be drawn such that all message arrows are vertical, that is, all external events can be assigned a timestamp such that time increases within a single process and for any message its send and receive are assigned the same timestamp. Formally, let $\mathcal{E}$ be the set of all external events. Then, a computation is synchronous iff there exists a mapping $T$ from $\mathcal{E}$ to the set of natural numbers such that for all $s, r, e, f \in \mathcal{E}$:

$$s \sim r \Rightarrow T(s) = T(r)$$

and

$$e \prec f \Rightarrow T(e) < T(f).$$
18.3.2 Relationship Among Message Orderings

We show that the hierarchy associated with the various message orderings is

Synchronous \subseteq Causally Ordered \subseteq FIFO \subseteq Asynchronous.

\textit{FIFO} \subseteq \textit{Asynchronous} is obvious. A causally Ordered computation satisfies FIFO because

\[ s_1 \prec s_2 \Rightarrow s_1 \rightarrow s_2. \]

We only need to show that if a computation is synchronous then it is also causally ordered. Because the communication is synchronous, there exists a function \( T \) satisfying SYNC.

For any set of send events \( s_1, s_2 \) and receive events \( r_1, r_2 \) such that \( s_1 \sim r_1, s_2 \sim r_2 \) and \( s_1 \rightarrow s_2 \):

\[ T(s_1) = T(r_1), \quad T(s_2) = T(r_2), \quad \text{and} \quad T(s_1) < T(s_2). \]

It follows that \( T(r_1) < T(r_2) \). Therefore, (18.1) implies

\[ \neg(r_2 \rightarrow r_1). \]

18.3.3 Algorithm

The algorithm implements the desired ordering using control messages. Note that control messages are not required to satisfy synchronous ordering. Thus \( \mathcal{E} \) includes the send and receive events only of application messages. It does not include send and receive of control messages sent by the algorithm to ensure synchronous ordering.

The algorithm shown in Figure 19.5 is for the process \( P_i \). All processes have the same algorithm. Observe that the protocol to implement synchronous message ordering cannot be completely symmetric. If two processes desire to send messages to each other, then there is no symmetric synchronous computation that allows this—one of them must succeed before the other. To introduce asymmetry, we use process numbers to totally order all processes. Each send event, \( s_i \) is either from a bigger process (denoted by \( \text{big}(s) \)) or from a smaller process (denoted by \( \text{small}(s) \)). We assume that processes do not send messages to themselves.

In our algorithm, a process can be in two states—\textit{active} or \textit{passive}. Every process is initially active. We first consider the algorithm for a send event corresponding to \( \text{big}(s) \). A process is allowed to send a message to a smaller process only when it is active. After sending the message it turns passive until it receives an \textit{ack} message from the receiver of the message. While passive, it cannot send any other message nor can it accept any other message. Note that the protocol for a message from a bigger process requires only one control message (\textit{ack}).
To send a message to a bigger process, say \( P_j \), \( P_i \) first needs permission from \( P_j \). It can request the permission at any time. \( P_j \) can grant permission only when it is active. Furthermore, after granting the permission, \( P_j \) turns passive until it receives the message for which it has granted the permission. Thus the protocol for a message from a smaller process requires two control messages (request and permission).

\[
P_i ::\begin{align*}
\text{var} & \quad state : \{\text{active, passive}\} \text{ initially active;} \\
\text{To send } m \text{ to } P_j, \ (j < i) & \quad \text{\textbf{enabled} if (state} = \text{active):} \\
& \quad \text{send } m \text{ to } P_j \\
& \quad \text{state} := \text{passive;} \\
\text{Upon receive } m \text{ from } P_j, \ (j > i) & \quad \text{\textbf{enabled} if (state} = \text{active):} \\
& \quad \text{send } ack \text{ to } P_j; \\
\text{Upon receive } ack & \quad \text{state} := \text{active;} \\
\text{To send a message } (message_{id}, m) \text{ to } P_j, \ (j > i) & \quad \text{send request}(message_{id}) \text{ to } P_j; \\
\text{Upon receive request}(message_{id}) \text{ from } P_j, \ (j < i) & \quad \text{\textbf{enabled} if (state} = \text{active):} \\
& \quad \text{send permission}(message_{id}) \text{ to } P_j \\
& \quad \text{state} := \text{passive;} \\
\text{Upon receive permission}(message_{id}) \text{ from } P_j, \ (j > i) & \quad \text{\textbf{enabled} if (state} = \text{active):} \\
& \quad \text{send } m \text{ to } P_j; \\
\text{Upon receive } m \text{ from } P_j, \ (j < i) & \quad \text{state} := \text{active;}
\end{align*}
\]

Figure 18.5: The algorithm at \( P_i \) for synchronous ordering of messages

### 18.4 Total Order for Multicast Messages

So far we have assumed that messages were point-to-point. In many applications, where a message may be sent to multiple processes, it is desirable that all messages are delivered in the same order at all processes. For example, consider a server that is replicated at multiple sites for fault tolerance. If a client makes a request to the server, then all copies of the server should handle requests in the same order. The total ordering of messages can formally be specified as:

For all messages \( x \) and \( y \) and all processes \( P \) and \( Q \), if \( x \) is received at \( P \) before \( y \), then \( y \) is not received before \( x \) at \( Q \). (Total Order)
CHAPTER 18. MESSAGE ORDERING

We require that $y$ not be received before $x$, rather than that $x$ be received before $y$, to address the case where $x$ is not sent to $Q$. Observe that we do not require that a message be broadcast to all processes.

In this section we discuss algorithms that will provide the total ordering of messages. Observe that the property of total order of messages does not imply causal or even FIFO property of messages. Consider the case when $P$ sends messages $m_2$ followed by $m_1$. If all processes receive $m_2$ before $m_1$, then the total order is satisfied even though FIFO is not. If messages satisfy causal order in addition to the total order, then we will call this ordering of messages causal total order.

The algorithms for ensuring total order are very similar to mutual exclusion algorithms. After all, mutual exclusion algorithms ensure that all accesses to the critical section form a total order. If we ensure that messages are received in the “critical section” order, then we are done. We will discuss a centralized algorithm and a distributed algorithm for causal total ordering of messages.

18.4.1 Centralized Algorithm

We first modify the centralized algorithm for mutual exclusion to guarantee causal total ordering of messages. We assume that channels between the coordinator process and other processes satisfy the FIFO property. A process that wants to multicast a message simply sends it to the coordinator. This step corresponds to requesting the lock in the mutual exclusion algorithm. Furthermore, in that algorithm, the coordinator maintains a request queue and whenever a request by a process becomes eligible, it sends the lock to that process. In the algorithm for total ordering of messages, the coordinator will simply multicast the message corresponding to the request instead of sending the lock. Since all multicast messages originate from the coordinator, and the channels are FIFO, the total order property holds.

In the above algorithm, the coordinator has to perform more work than the other nodes. One way to perform load balancing over time is by suitably rotating the responsibility of the coordinator among processes. This can be achieved using the notion of a token. The token assigns sequence numbers to broadcasts, and messages are delivered only in the sequence order.

18.4.2 Lamport’s Algorithm for Total Order

We modify Lamport’s algorithm for mutual exclusion to derive an algorithm for total ordering of messages. As in that algorithm, we assume FIFO ordering of messages. We also assume that a message is broadcast to all processes. To simulate multicast, a process can simply ignore a message that is not meant for it. Each process maintains a logical clock (used for timestamps) and a queue (used for storing undelivered messages). The algorithm is given by the following rules.

- To send a broadcast message, a process sends a timestamped message to all other processes including itself. This step corresponds to requesting the critical section in mutual exclusion algorithm.

- On receiving a broadcast message, the message and its timestamp are stored in the queue and an acknowledgment is returned.

- A process can deliver the message from the request queue with the smallest timestamp, $t$, if it has received a message from every other process with timestamp greater than $t$. This step corresponds to executing the critical section for the mutual exclusion algorithm.

In this algorithm, the total order of messages delivered is given by the logical clock of send events of the broadcast messages.

18.4.3 Skeen’s Algorithm

The distributed algorithm of Skeen is given by the following rules. It also assumes that processes have access to Lamport’s logical clock.
18.5. **Problem**

- To send a multicast message, a process sends a timestamped message to all the destination processes.
- On receiving a message, a process marks it as *undeliverable* and sends the value of the logical clock as the proposed timestamp to the initiator.
- When the initiator has received all the proposed timestamps, it takes the maximum of all proposals and assigns that timestamp as the final timestamp to that message. This value is sent to all the destinations.
- Upon receiving the final timestamp of a message, it is marked as deliverable.
- A deliverable message is delivered to the site if it has the smallest timestamp in the message queue.

In this algorithm, the total order of message delivery is given by the final timestamps of the messages.

### 18.4.4 Application: Replicated State Machines

Assume that we are interested in providing a fault-tolerant service in a distributed system. The service is expected to process *requests* and provide *outputs*. We would also like the service to tolerate up to $t$ faults where each fault corresponds to a crash of a processor. We can build such a service using $t+1$ processors in a distributed system as follows. We structure our service as a *deterministic* state machine. This means that if each nonfaulty processor starts in the same initial state and executes the requests in the same order, then each will produce the same output. Thus by combining outputs of the collection we can get a $t$ fault-tolerant service. The key requirement for implementation is that all state machines process all requests in the same order. The total ordering of messages (for example, Lamport’s algorithm) satisfies this property.

### 18.5 Problem

18.1. Assume that you have replicated data for fault tolerance. Any file (or a record) may be replicated at more than one site. To avoid updating two copies of the data, assume that a token-based scheme is used. Any site possessing the token can update the file and broadcast the update to all sites which have that file. Show that if the communication is guaranteed to be causally ordered, then the above scheme will ensure that all updates at all sites happen in the same order.

18.2. Let $M$ be the set of messages in a distributed computation. Given a message $x$, we use $x.s$ to denote the send event and $x.r$ to denote the receive event. We say that a computation is *causally* ordered if

$$
\forall x, y \in M : (x.s \rightarrow y.s) \Rightarrow \neg(y.r \rightarrow x.r).
$$

We say that a computation is *mysteriously* ordered if

$$
\forall x, y \in M : (x.s \rightarrow y.r) \Rightarrow \neg(y.s \rightarrow x.r).
$$

(a) Prove or disprove that every causally ordered computation is also mysteriously ordered.
(b) Prove or disprove that every mysteriously ordered computation is also causally ordered.

18.3. Show the relationship between conditions (C1), (C2), and (C3) on message delivery of a system.

(C1) $s_1 \rightarrow s_2 \Rightarrow \neg(r_2 \rightarrow r_1)$

(C2) $s_1 \prec s_2 \Rightarrow \neg(r_2 \rightarrow r_1)$

(C3) $s_1 \rightarrow s_2 \Rightarrow \neg(r_2 \prec r_1)$

where $s_1$ and $s_2$ are sends of any two messages and $r_1$ and $r_2$ are corresponding receives. Note that a computation satisfies a delivery condition if and only if the condition is true for all pairs of messages.
18.4. How will the algorithm for causal ordering change if messages can be multicast instead of point to point?

18.5. Assume that all messages are broadcast messages. How can you simplify the algorithm for guaranteeing causal ordering of messages under this condition?

18.6. Consider a system of \( N+1 \) processes \( \{ P_0, P_1, \ldots, P_N \} \) in which processes \( P_1 \) through \( P_N \) can only send messages to \( P_0 \) or receive messages from \( P_0 \). Show that if all channels in the system are FIFO, then any computation on this system is causally ordered.

18.7. In this chapter, we have used the happened before model for modeling dependency of one message to the other. Thus all messages within a process are totally ordered. For some applications, messages sent from a process may be independent. Give an algorithm to ensure causal ordering of messages when the send events from a single process do not form a total order.

18.8. Suppose that the system is composed of nonoverlapping groups such that any communication outside the group is always through the group leader, that is, only group leaders are permitted to send or receive messages from outside the group. How will you exploit this structure to reduce the overhead in causal ordering of messages?

18.9. Design an algorithm for synchronous ordering for point-to-point messages that does not use a static priority scheme. (Hint: Impose an acyclic directed graph on processes. The edge from \( P_i \) to \( P_j \) means that \( P_i \) is bigger than \( P_j \) for the purpose of sending messages. Give a rule by which the direction of edges is reversed, such that acyclicity of the graph is maintained.)

18.10. Prove the correctness of Lamport’s algorithm for providing causal total ordering of messages.

18.11. Prove the correctness of Skeen’s algorithm for providing total ordering of messages.