Chapter 12

Message Ordering

12.1 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.

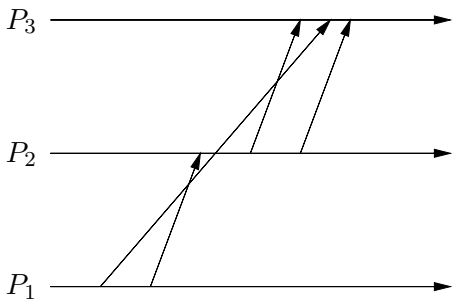

Figure 12.1: A FIFO computation that is not causally ordered

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 not be overtaken by a sequence of messages. For example, the computation in Figure 12.1 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 8, 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 happened-before order in which they are made. For another example of the usefulness of causal ordering, see Problem 12.1.

The relationship among various message orderings can be formally specified on the basis of 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). We also use $s_i \rightsquigarrow r_i$ to denote that r_i is the receive event corresponding to the send event s_i. Finally, we use $e \prec f$ to denote that e occurred before f in the same process.

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 the corresponding receive events. Then

$$s_1 \prec s_2 \Rightarrow \neg(r_2 \prec r_1) \qquad \text{(FIFO)}$$

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, this can be expressed as

$$s_1 \rightarrow s_2 \Rightarrow \neg(r_2 \prec r_1) \qquad \text{(CO)}$$

12.2 Causal Ordering

P_i::

var

M:array[1..N, 1..N] of integer initially $\forall j, k : M[j, k] = 0$;

To send a message to P_j:

$M[i, j] := M[i, j] + 1$;

piggyback M as part of the message;

To receive a message with matrix W from P_j

enabled if $(W[j, i] = M[j, i] + 1) \wedge (\forall k \neq j : M[k, i] \geq W[k, i])$

$M := max(M, W)$;

Figure 12.2: An algorithm for causal ordering of messages at P_i

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 $M[j, k]$ at P_i records the number of messages sent by process P_j to process P_k as known by process P_i. The algorithm for process P_i is given in Figure 12.2. Whenever a message is sent from P_i to P_j, first the entry $M[i, j]$ is incremented to reflect the fact that one more message is known to be sent from P_i to P_j. The matrix M is piggybacked with the message. 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. A message m from P_j is eligible to be received when

1. The entry $W[j,i]$ is one more than the entry $M[j,i]$ that records the number of messages received by P_i from P_j.

2. The number of messages sent from any other process $P_k (k \neq j)$ to P_i, as indicated by the matrix W in the message, is less than or equal to the number recorded in the matrix M. Formally, this condition is

$$\forall k \neq j : M[k,i] \geq W[k,i]$$

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

Whenever a message is accepted for delivery, the information at matrix M is updated with the matrix W received in the message.

12.3 Total Order for Multicast Messages

In applications where a message may be sent to multiple processes, it is often desirable that all messages be 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 be formally specified as follows:

There exists a total order on all messages in the system such that every process receives messages in an order that is consistent with the total order. (**Total Order**)

In this section we discuss algorithms for 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_1 followed by m_2. 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 now discuss centralized and distributed algorithms for causal total ordering of messages.

12.3.1 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 this centralized 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 through the use of a token. The token assigns sequence numbers to broadcasts, and messages are delivered only in this sequence order.

12.3.2 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 processes including itself. This step corresponds to requesting the critical section in the mutual exclusion algorithm.
- On receiving a broadcast message, the message and its timestamp are stored in the queue, and a timestamped acknowledgment is returned.
- A process can mark a message as deliverable if its timestamp is smallest in the request queue and the process has received acknowledgments from all other processes. This step corresponds to executing the critical section for the mutual exclusion algorithm.
- Once a process finds a message to be deliverable, it informs all other processes that the message is deliverable. This step corresponds to sending the *release* message in the mutual exclusion algorithm. On receiving this notification, the corresponding message is delivered.

In this algorithm, the total order of messages delivered is given by the logical clock of send events of the broadcast messages. Since logical clocks preserve the happened-before order, the algorithm also satisfies the causal order.

12.3.3 Skeen's Algorithm

Lamport's algorithm is wasteful when most messages are multicast and not broadcast. Skeen's algorithm requires messages proportional to the number of recipients of a message and not the total number of processes in the system.

The distributed algorithm of Skeen also assumes that processes have access to Lamport's logical clock. The algorithm is given by the following rules:

- 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.
- On 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.

12.3.4 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 satisfies this property.

12.4 Implementation in Java

The structure of a causal message is shown in Figure 12.3, and the Java implementation of the causal ordering algorithm is shown in Figure 12.4. The causal ordering algorithm extends the class `Linker` to include the matrix in outgoing messages. The method `sendMsg` increments the entry $M[myId][destId]$ to account for this message and attaches the matrix M with it. The method `multicast` is used for sending a message to multiple sites. In this method, we first increment $M[myId][destId]$ for all $destId$ in the list of destinations. It is this matrix that is sent with every message.

The method `okayToReceive` determines whether a message can be delivered to the process. The method `receiveMsg` uses two `LinkedList` for storing messages. The `deliverQ` stores all messages that are deliverable to the application layer. The `pendingQ` stores all messages that are received but are not deliverable. When the application layer asks for a message, the `pendingQ` is traversed first to check whether some messages are deliverable. Deliverable messages are moved from the `pendingQ` to the `deliveryQ` by the method `checkPendingQ`. If `deliveryQ` is empty, then we wait for a message to arrive by calling the blocking method `super.receiveMsg`. On receiving this message, it is put in the `pendingQ` and the method `checkPendingQ` is invoked again. If `deliveryQ` is nonempty, the first message from that queue is delivered and the matrix M updated to record the delivery of this message.

```
1  public class CausalMessage {
2      Msg m;
3      int N;
4      int W[][];
5      public CausalMessage(Msg m, int N, int matrix[][]) {
6          this.m = m;
7          this.N = N;
8          W = matrix;
9      }
10     public int[][] getMatrix() {
11         return W;
12     }
13     public Msg getMessage() {
14         return m;
15     }
16 }
```

Figure 12.3: Structure of a causal message

```
1  import java.util.*; import java.net.*; import java.io.*;
2  public class CausalLinker extends Linker {
3      int M[][];
4      LinkedList deliveryQ = new LinkedList(); // deliverable messages
5      LinkedList pendingQ = new LinkedList(); // messages with matrix
6      public CausalLinker(String args[]) throws Exception {
7          super(args);
8          M = new int[N][N]; Matrix.setZero(M);
9      }
10     public synchronized void sendMsg(int destId, String tag, String msg){
11         M[myId][destId]++;
12         super.sendMsg(destId, "matrix", Matrix.write(M));
13         super.sendMsg(destId, tag, msg);
14     }
15     public synchronized void multicast(LinkedList<Integer> destIds,
16                                        String tag, String msg) {
17         for (int i: destIds)
18             M[myId][i]++;
19         for (int i: destIds) {
20             super.sendMsg(i, "matrix", Matrix.write(M));
21             super.sendMsg(i, tag, msg);
22         }
23     }
24     boolean okayToRecv(int W[][], int srcId) {
25         if (W[srcId][myId] > M[srcId][myId]+1) return false;
26         for (int k = 0; k < N; k++)
27             if ((k!=srcId) && (W[k][myId] > M[k][myId])) return false;
28         return true;
29     }
30     synchronized void checkPendingQ() {
31         ListIterator iter = pendingQ.listIterator(0);
32         while (iter.hasNext()) {
33             CausalMessage cm = (CausalMessage) iter.next();
34             if (okayToRecv(cm.getMatrix(), cm.getMessage().getSrcId())){
35                 iter.remove(); deliveryQ.add(cm);
36             }
37         }
38     }
39     // polls the channel given by fromId to add to the pendingQ
40     public Msg receiveMsg(int fromId) throws IOException {
41         checkPendingQ();
42         while (deliveryQ.isEmpty()) {
43             Msg matrix = super.receiveMsg(fromId);// matrix
44             int [][]W = new int[N][N];
45             Matrix.read(matrix.getMessage(), W);
46             Msg m1 = super.receiveMsg(fromId);//app message
47             pendingQ.add(new CausalMessage(m1, N, W));
48             checkPendingQ();
49         }
50         CausalMessage cm = (CausalMessage) deliveryQ.removeFirst();
51         Matrix.setMax(M, cm.getMatrix());
52         return cm.getMessage();
53     }
54 }
```

Figure 12.4: CausalLinker for causal ordering of messages

12.4.1 Application: Causal Chat

To illustrate an application of causal ordering, we consider a chat application in which a user can send messages to multiple other users. This simple program, shown in Figure 12.5, takes as input from the user a message and the list of destination process identifiers. This message is then multicast to all the process identifiers in the list.

```
1  import java.io.*; import java.util.*;
2  public class Chat extends Process {
3      public Chat(Linker initComm) {
4          super(initComm);
5      }
6      public synchronized void handleMsg(Msg m, int src, String tag){
7          if (tag.equals("chat"))
8              println("Message from " + src +":" + m.getMessage());
9      }
10     public static void main(String[] args) throws Exception {
11         Linker comm = null;
12         if (args[3].equals("simple"))
13             comm = new Linker(args);
14         else if (args[3].equals("causal"))
15             comm = new CausalLinker(args);
16         else if (args[3].equals("synch"))
17             comm = new SynchLinker(args);
18         Chat c = new Chat(comm);
19         c.startListening();
20         BufferedReader din = new BufferedReader(
21         new InputStreamReader(System.in));
22         while (true) {
23              System.out.println("Type your message in a single line:");
24              String chatMsg = din.readLine();
25              if (chatMsg.equals("quit")) break;
26             println("Type in destination pids on one line: ");
27             LinkedList<Integer> destIds = new LinkedList<Integer>();
28             String s = din.readLine();
29             Util.readList(s, destIds);
30             if (args[3].equals("synch"))
31                 comm.sendMsg(destIds.get(0), "chat", chatMsg);
32             else
33                 comm.multicast(destIds, "chat", chatMsg);
34         }
35     }
36 }
```

Figure 12.5: A chat program

The application takes as an argument the message ordering to be used. The user can verify that if the plain `Linker` class were used in this application, then the following scenario would be possible. If P_0 sends a query to both P_1 and P_2, and P_1 sends a reply to the query to both P_0 and P_2, then P_2 may receive the reply before the query. On the other hand, if the class `CausalLinker` is used, then such a scenario is not possible.

12.5 Problems

12.1. 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 that have that file. Show that if the communication is guaranteed to be causally ordered, then the scheme described above will ensure that all updates at all sites happen in the same order.

12.2. 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.

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

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

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

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

where s_1 and s_2 are sends of any two messages and r_1 and r_2 are the corresponding receives. Note that a computation satisfies a delivery condition if and only if the condition is true for all pairs of messages.

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

12.5. 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.

12.6. In this chapter, we have used the happened-before model for modeling the 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.

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

12.8. Prove the correctness of Lamport's algorithm for providing causal total ordering of messages.

12.9. Prove the correctness of Skeen's algorithm for providing total ordering of messages.

12.10. Build a multiuser *Chat* application in Java that guarantees that all users see all messages in the same order.

12.6 Bibliographic Remarks

Causal ordering was first proposed by Birman and Joseph [BJ87]. The algorithm for causal ordering described in this chapter is essentially the same as that described by Raynal, Schiper, and Toueg [RST91]. For a discussion on total ordering of messages, see the article by Birman and Joseph [BJ87]. The distributed algorithm for causal total ordering of messages is implicit in the replicated state machine construction described by Lamport [Lam78]. Skeen's algorithm is taken from the reference [Ske82].