5.4 Logical Clocks

5.4.1 Lamport’s Logical Clock

If we are only interested in ordering events (not in time duration), then there is no need to use units like hour, minute or second. We might as well use dimensionless positive integers, with the understanding that smaller (larger) integers denote “earlier” (“later”) times. We also interpret the concept of “earlier” (“later”) differently from its conventional meaning. We will equate “earlier” with the “happened-before” (=“causally precedes”) relation among events.1 “Clocks” which keep such “logical time” are called logical clocks.

Let $C_i$ denote the logical clock associated with processor $i$. Namely, for any event $e$ which occurs in processor $i$, $C_i(e)$ denotes the logical time of its occurrence. The global time for event $e$, denoted $C(e)$, is defined by $C(e) = C_i(e)$ if $e$ occurs in processor $i$. Because of our interpretation of “earlier” and “later” as explained above, in assigning a logical time $C(e)$ to event $e$, $C()$ would have to satisfy certain conditions. Intuitively, if $a$ causally affects (i.e., happened before) $b$, we would want $C(a) < C(b)$ to hold, because an event wouldn’t affect another event that happened “earlier”.

**Clock condition:** if $a \rightarrow b$ then $C(a) < C(b)$

Fig. 5 shows an assignment of integer values to events in such a way that the clock condition is enforced. Those integer values were computed following the implementation rules given below (the clock at each site is initialized to 0). Note that $C(p_4) = 6$, even though $C(p_3) = 3$. This is because event $q_5$ with $C(q_5) = 5$ happened before $p_4$, and therefore we must have $C(p_4) > C(q_5)$.

![Figure 5: Lamport’s logical clock values.](image)

**Implementation Rules for $C_i()$**

---

1Note that our concept of “earlier” is a total order (i.e., it orders any pair of events), while “happened-before” is a partial order, so that two concurrent events are not related by the “happened-before” relationship.
R1: Increment $C_i$ whenever a new event occurs: the new event is considered to occur at the updated time.

R2: 1. When sending a message, attach timestamp $ts = C_i$ (after step R1), and
    2. Upon receiving a message, advance the local clock to $\text{Max}\{C_i, ts\} + 1$, and consider that the receiving event took place at this new time ($ts$ is the time stamp of the received message, i.e., the value of $C_j$ at message sending time if the sender is processor $j$).

If global clocks are implemented by rules R1 and R2, the local clocks at two different processors may diverge without bound, if they don’t exchange messages for a long time. If our only purpose of having global clock is to order events, then no harm is done. If it is desired to keep all local clocks more or less in pace, each processor can send special “tick messages” regularly or whenever an event occurs in that processor.

It is often desired to have a total order among all the events. The above implementation of logical clock defines a total order for each processor. Therefore, all we need is to order any pair of events from different processors. This can be done simply by assigning a processor id (a unique integer), $id(i)$, to each processor $i$, and using $C_i.id(i)$ as the clock value of processor $i$, where “.” is a decimal point. This way, if $C_i = C_j$, the decimal part will break the tie.

5.4.2 Holding back deliveries

You may want to design the communication subsystem (a part of the kernel) that delays the delivery of messages that arrived “too soon”, so that the delivered messages conform to the order you want to enforce. The basic idea is to hold back the delivery of message $M$ to process $P$ until there is a guarantee that no message $M’$ with $M’.ts < M.ts$ will arrive at $P$ in the future.

In the following message delivery scheme, we assume that the messages from a particular source arrive in the FIFO order. Each site (processor) maintains a set of message queues, one for each other site (processor), and follows the following steps:

1. When a message arrives, it is first placed in the corresponding queue.
2. As soon as all queues become non-empty, then compare the timestamps of the messages at the heads of the queues, and deliver the message with the oldest timestamp.

The above scheme works, provided all message queues always become non-empty. Unfortunately, there is normally no such guarantee. In the next section, therefore, we consider multicasts to solve this problem.

There is another problem with the above scheme. With Lamport’s clock, $C(a) < C(b)$ does not necessarily imply $a \rightarrow b$. Therefore, a message $M_1$ with a smaller timestamp may not necessarily causally affect another message $M_2$ with a larger timestamp, and hence $M_2$ could be delivered before $M_1$, without violating causality. However, by employing Lamport’s clock values, we may unnecessarily delay some messages. We address this issue in the

\footnote{Recall that the “happened-before” relation is a partial order.}
next subsection, by introducing a clock whose values exactly represent the happened-before relation.

5.4.3 Representing partial order by vector logical clock

When comparing two \( n \)-vectors, \( V \) and \( W \), we write \( V < W \) if

1. For each \( k(1 \leq k \leq n) \), \( V[k] \leq W[k] \) holds, where \( V[k] \) denotes the \( k \)th component of \( V \).
2. \( V[j] < W[j] \) holds for at least one \( j \).

If only condition 1 holds, then we write \( V \leq W \).

As we saw above, it is desirable to have a clock \( VC \) with the following property:

\[
\text{a \rightarrow b if and only if } VC(a) < VC(b),
\]

so that, by comparing \( VC(a) \) and \( VC(b) \), we could tell if \( a \) and \( b \) are causally related. Clearly, we must use a vector for \( VC(a) \), since scalar quantities cannot represent a partial order. There is a straightforward way of generalizing Lamport’s logical clock to the vector logical clock which satisfies the above condition. In this subsection, we assume each processor (or a unique process that represents it) maintains its own local clock.

For each processor \( P_i \), its local (vector) clock \( V_i() \) has \( n \) components, where \( n \) is the total number of processors. Its \( i \)th component, is just the event counter at \( P_i \), which is incremented every time an event occurs in \( P_i \). For each local event \( e \), processor \( P_i \) maintains \( V_i(e) \) as follows:

1. Initialize \( V_i(e_0) = [0, \ldots, 0] \), where \( e_0 \) is the hypothetical “initializing event”.
2. For each event \( e \) in \( P_i \), increment the \( i \)th component of \( V_i() \) by 1, i.e., \( V_i(e)[i] = V_i(\text{prev})[i] + 1 \), where \( \text{prev} \) is the last event of \( P_i \) prior to \( e \).
3. To send a message (let \( m \) be its sending event), attach \( V_i(m) \) as the timestamp.
4. If \( m \) is the receiving event of a message with timestamp \( ts[\ ] \), update \( V_i() \) as follows:
   - For each component \( k (\neq i) \), compute \( V_i(m)[k] = \max\{V_i(\text{prev})[k], ts[k]\} \).

We now prove that \( a \rightarrow b \) if and only if \( V_i(a) < V_j(b) \), where \( a \) is an event of \( P_i \) and \( b \) is an event of \( P_j \).

First consider the case where both \( a \) and \( b \) are “adjacent” events of a process \( P_i \) such that \( a \rightarrow b \). Then it is clear that \( V_i(a)[i] < V_i(b)[i] \) and \( V_i(a) < V_j(b) \). Similarly, if \( a \) is the sending event of a message \( M \) (by \( P_i \)) and \( b \) is the receiving event of \( M \) (by \( P_j \)), then we clearly have \( V_i(a) < V_j(b) \). This is because, on receiving \( M \), \( P_j \) updates \( V_j \) so that \( V_i(a)[j] < V_j(b)[j] \) and \( V_i(a)[k] \leq V_j(b)[k] \) for all \( k \neq j \).

In general, if \( a \rightarrow b \), then we have \( a \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow b \), such that \( a \) and \( e_1 \) are related as in i) or ii) (so, \( V(a) < V(e_1) \)), \( e_1 \) and \( e_2 \) are related as in i) or ii) (so, \( V(e_1) < V(e_2) \)), etc. From the transitivity of “<”, it follows that if \( a \rightarrow b \) then we have \( V_i(a) < V_j(b) \).

\(^{3}\)Note that 1 is not added to the max value, unlike in Lamport’s clock.
We now want to prove that if \( V_i(a) < V_j(b) \) then \( a \rightarrow b \) by showing that if \( a \not\rightarrow b \) then \( V_i(a) \not< V_j(b) \). Pay attention to the \( i \)th components, \( V_i(a)[i] \) and \( V_j(b)[i] \). (For example, consider which events have vector clock values not larger than \( V_1(E_1) = [200] \) in Fig. 6.) \( a \not\rightarrow b \) implies that, in the space-time diagram, there is no directed path from \( a \) to \( b \). Therefore, the \( i \)th component of \( V_i(a) \) is larger than the \( i \)th component of \( V_j(b) \), i.e., \( V_i(a)[i] > V_j(b)[i] \), hence \( V_i(a) \not< V_j(b) \). \( \Box \)

Example:

![Diagrams illustrating vector logical clocks of three processors.](image)

Figure 6: Vector logical clocks of three processors: (a) Event \( E_1 \) happened before the events surrounded by dashed lines; (b) The events surrounded by dashed lines happened before event \( E_2 \).

As in the previous section, the global vector clock (without subscript) is defined by \( V(a) = V_i(a) \) if \( a \) is an event of processor \( P_i \). Clearly, two events \( a \) and \( b \) are concurrent if and only if neither \( V(a) < V(b) \) nor \( V(b) < V(a) \) holds.

5.5 Group Communication

5.5.1 Multicasting

One-to-one communication is often called a unicast. In a multicast, a message is sent to all the members of a group. Some of its possible applications are

1. Sending a video stream to a set of customers,

2. Implementing a chat program for more than two participants,

3. Sending updates to a group of replica managers, etc.

Internet protocol IPv4 defines Class D as the multicast addresses. They all start with the bit sequence 1110, i.e., they range from 224.0.0.1 to 239.255.255.255, which provide about \( 2^{28} \approx 268 \) million addresses. Some of these addresses are set aside for specific purposes. For
example, 224.0.0.1 is for all systems on “this” subnet, and 224.2.0.0 ~ 224.2.127.253 are for multimedia conference calls.

Java provides API’s for multicasting in the java.net.* package. Given below are simple programs for a multicast sender and receiver. For an additional example, see Fig. 4.17 of the Textbook.

Java multicast sender:
The following program multicasts a command-line message to a multicast group and quits. It does not join the group.

```java
import java.io.*;
import java.net.*;
public class mcSender {
    public static void main (String[] args) {
        MulticastSocket s;
        InetAddress group;
        try {
            group = InetAddress.getByName("239.1.2.3");
            s = new MulticastSocket(3456);
            s.setTimeToLive(1);
            String msg = args[0]; //Msg in command line should be in quotes.
            DatagramPacket packet =
                new DatagramPacket(msg.getBytes(), msg.length(), group, 3456);
            s.send(packet);
            s.close();
        }
        catch (Exception ex) { ex.printStackTrace();
        } //end catch
    } //end main
} //end class
```

`s.setTimeToLive()` in the above program specifies the number of hops within the range [0,255] that the message should traverse before being discarded. It should be set to 0 if the multicast is restricted to processes on the same host. It should be set to 1 if the multicast is restricted to processes on the same subnet. A larger value will generate a lot of network traffic.

Java multicast receiver:
The following program receives one multicast message, prints it and quits.

```java
import java.io.*;
import java.net.*;
public class mcReceiver {
    public static void main (String[] args) {
```
MulticastSocket s;
InetAddress group;
try {
    group = InetAddress.getByName("239.1.2.3");
s = new MulticastSocket(3456);
s.joinGroup(group);
    byte[ ] buf = new byte[100];
    DatagramPacket recv =
        new DatagramPacket(buf, buf.length);
    s.receive(recv);
    System.out.println(new String(buf));
    s.close();
}
    catch (Exception ex) { ex.printStackTrace();
}   //end catch
}//end main
}//end class

In order to run, first start a few receivers; they will block on s.receive(). Then start the sender; it will send the message, "This is a test msg.", and stop. Then the receivers will get unblocked and receive the message and stop.

5.6 ISIS vector clock

As we saw earlier, the vector logical clocks exactly incorporate the happened-before (i.e., causally-precedes) relation. Thus, upon reception of a message \( M \) with vector timestamp \( M.ts \), process \( P_i \) can determine if it has already received a message which is causally preceded by \( M \). If \( V_i > M.ts \), it has indeed received such a message and a causality violation has occurred.

For a concrete example, see Fig. 7, where processor \( P \) decides to migrate an object \( O \) to processor \( Q \). In the meantime, process \( R \) is looking for \( O \); it sends an enquiry message to \( P \) and is told by message \( M_2 \) that \( O \) is now at \( Q \), but discovers by sending message \( M_3 \) to \( Q \) that \( Q \) doesn’t have it. What’s gone wrong? The problem here is that message \( M_1 \) that causally precedes message \( M_3 \) arrived at \( Q \) after \( M_3 \). It’s like you get an official letter firing you from your boss after you have heard a rumour that you have been fired.

Note that causality violations can occur even if all communication channels are FIFO as in Fig. 7. To relieve the application programmers from the problems caused by causality violations, we want to design the communication subsystem that delays the delivery of messages that arrived “too soon”, in such a way that no causality violation can occur. Let \( M = M_1 \) in Fig. 7. We want to hold back the delivery of message \( M \) to process \( P \) until there is a guarantee that no message \( M' \) with \( M'.ts < M.ts \) will arrive at \( P \) in the future. The ISIS system uses multicasting to provide such a guarantee.

4Here we identify a processor with the process running on it.
In ISIS, only the sending events advance the local component of the vector clock. Thus, we need to modify the definition of the vector time, so that the $i$th component of the vector time is the sequence number of messages generated by processor $P_i$. Each processor $P_i$ maintains the \textbf{local} vector clock $L_i = \langle L[1], \ldots, L[n] \rangle$. We have explained above how its local component $L[i]$ is advanced. For $j = 1, 2, \ldots, n$, $j \neq i$, $L[j]$ is the sequence number of the most recent message from $P_j$ that has arrived \textbf{and} delivered; it does not reflect those messages from $P_j$, if any, currently in the holdback queue. Initially, $L[j] = 0$ for all $j$.

Suppose a new message $M$ multicast by $P_j$ arrives at $P_i$. $P_i$ compares the vector timestamp of the message $M.ts$ (call it $V$) and its own local vector clock $L_i$ and delivers $M$ if the following two conditions are satisfied:

1. $V[j] = L[j] + 1$ ($M$ is the next expected message from $P_j$),
2. $V[k] \leq L[k]$ for all $k \neq j$.

Condition 2. above means, intuitively, that the receiver is as up-to-date as the sender of $M$ about the current values of the event counters at the other processors. If $V[k] > L[k]$ for some $k \neq j$, then it implies that there is a message $M'$ from $P_k$ that the sender $P_j$ knows about but $P_i$ doesn’t know about. Therefore, $P_i$ must wait until $M'$ arrives before delivering $M$.

If $M$ is delivered, $L[j]$ is incremented by one, so that $L[j] = V[j]$ will hold. Otherwise, $M$ is held back until the arrival and delivery of other messages (such as $M'$ in the previous paragraph) make the above conditions satisfied for $M$. In Fig. 8(a), when $M_1$ arrives with timestamp $M_1.ts = \langle 110 \rangle$, $P_3$ delays its delivery because $L_3 = \langle 000 \rangle$ and $M_1.ts = \langle 110 \rangle$ violate condition 2. for component $k = 1$. Note that $P_3$’s vector time $L_3$ remains $\langle 000 \rangle$.

When $M_2$ arrives at $P_3$, as shown in Fig. 8(b), $P_3$ delivers it and updates its time to $L_3 = \langle 100 \rangle$, and reexamines the timestamp $M_1.ts$. The condition is now satisfied and $M_1$ is also delivered.

The ISIS scheme works even if the message channel between two processors is not FIFO.

\textbf{Proof of Correctness}
Figure 8: Message delivery: (a) Delivery of $M_1$ is delayed; (b) $M_2$ is delivered first then $M_1$.

In general, to prove a protocol correct, we need to show the following two properties:

1. **Safety** (bad things don’t happen): No causality violation in the context of message delivery.

2. **Liveness** (good things keep happening): Livelock-free (starvation-free), i.e., no message will wait forever in the hold-back Q.

**Safety:**

Consider two messages $M_1$ (from $P_j$) and $M_2$ (from $P_k$) received by $P_i$ such that the sending event $m_1$ of $M_1$ precedes the sending event $m_2$ of $M_2$ ($m_1 \rightarrow m_2$), i.e.,

\[ M_1.ts = IC_j(m_1) < M_2.ts = IC_k(m_2) \]  \(1\)

where $IC_j(a)$ denotes the ISIS vector clock value at $P_j$ for event $a$. (Before this proof, we used a simpler notation, $L_j$, to denote $IC_j$.)

We assume that $P_i$ delivered $M_2$ before $M_1$ and derive a contradiction. Just before $P_i$ delivered $M_1$, by condition 1,

\[ IC_i[j] = IC_j(m_1)[j] - 1 \text{ hence } IC_i[j] < IC_j(m_1)[j]. \]  \(2\)

However, the delivery of $M_2$ by $P_i$ prior to that would have resulted in (condition 2),

\[ IC_i^*[j] \geq IC_k(m_2)[j]. \]  \(3\)

Since $IC_i^*[j] < IC_i[j]$ ($M_2$ was delivered before $M_1$), from Eqs. (2) and (3), we have $IC_k(m_2)[j] < IC_j(m_1)[j]$. However, this contradicts Eq. (1).

**Liveness:** You never get into a situation where one or more messages get stuck in holdback queues and are not delivered.

Informally, the delivery conditions (1) and (2) for msg $M$ are satisfied if and only if the receiver ($P_i$) of $M$ has received and delivered all messages that the sender ($P_j$) had delivered and sent before sending $M$. This will eventually happen, because they will all arrive at the receiver. Let $M$ be a msg in the holdback Q, which is not preceded by any other msg in Q. Then this $M$ satisfies the above condition, and would have been delivered, a contradiction.
More formally, the delivery condition (1) is satisfied when all the messages sent by $P_j$ before $M$ arrive and are delivered. Condition (2) is satisfied when all the message that sender $P_j$ know about when it sent $M$ also arrive and are delivered. When they all arrive eventually, they will be delivered, since every message in the holdback Q is tested against the delivery conditions each time a new message arrives and is delivered. So, if there exists a delivery order, it will be discovered and carried out. Note that a greedy delivery is OK, since advancing any component of $IC_i$ is beneficial for condition (2) to be satisfied.

To proceed more formally, note that

1. All messages sent from one site are linearly ordered by the “precedes” relation.
2. The set of the timestamps of all messages is a partially ordered set, and therefore any subset of the set is also partially ordered.

Suppose the subset corresponding to the messages in the holdback Q of a process $P_i$ is non-empty. This Q contains at least one message $M$ (from, say, $P_j$) that is not preceded by any other message in the Q. Thus, all msgs $M'$ with $M'.ts \leq M.ts$, i.e., all msgs $P_j$ had delivered prior to sending $M$, have already been delivered by $P_i$. This implies that $IC_i[k] \geq M.ts[k]$ for all $k \neq i$, i.e., delivery condition (2) is satisfied.

Since $P_i$ has delivered all messages from $P_j$ that precede $M$, $IC_i[j] + 1 = M.ts$ (this is delivery condition (1)). Therefore, $M$ is deliverable. This contradicts the assumption that $M$ was in Q.

\qed