CMPSCI 677 Operating Systems **Spring 2013**

Lecture 13: March 6

Lecturer: Prashant Shenoy *Scribe: Yue Wang*

13.1 Logical Clocks

In last lecture we understood how to synchronize physical clocks on multiple machines. However, regardless of the algorithm we use, there is always some inaccuracy. One possible solution is to apply logical clocks.

When using logical clocks, users are just interested in the ordering of events instead of the real time of occurrence of every single event. The definition of "event" here depends on the application. In the programming assignment 1 as an example, an event may be the landing of the bird, the arrival of an alert message and so on. Its key idea comes from the following facts:

- Clock synchronization need not be absolute
- Synchronizing two machines that do not interact with each other is not necessary
- Processes should just agree on the order of events rather than the time at which they occurred

Now the problem is how to define an ordering of all events in a distributed system without global clock or synchronized physical local clocks.

13.1.1 Lamport's Logical Clocks

Leslie Lamport proposed an algorithm with the help of message exchange. Based on the fact that every message must be sent before received, one can easily order the send/receive event between two processes and therefore order more relevant events according to "Happened-Before Relation".

13.1.1.1 Happened-Before Relation

The happened-before relation (denoted by $\rightarrow$) describes the order of two events. $A \rightarrow B$ means event A happens before B. It is formally defined as:

- Rule 1: if events A and B occur on the same process and A is executed before B, $A \rightarrow B$
- Rule 2: if event A represents the sending of a message and event B is the reception of the same message, $A \rightarrow B$

The happened-before relation is obviously transitive ($\forall A, B, C$, if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$). In addition, one should notice that it gives only a partial order on events. For instance, if two processes do not change messages, the relation of their events is undefined.

Figure 13.1 and 13.2 illustrate the idea of happened-before relation. Suppose there are two machines M_1 and M_2. The program of M_1 reads a record (event A) and prints it (event B). As Figure 13.1 shows, A and

B occur on the same process sequentially, so there exists $A \rightarrow B$. However, the relation between (A, C) or (A, D) is unclear. In Figure 13.2, one message is passed from M_1 to M_2. Then relation $A \rightarrow D$ becomes true. And actually one can know that all events before *send* on M_1 happened before all events after *recv* on M_2. Nevertheless, the relation between B and C is still undefined.

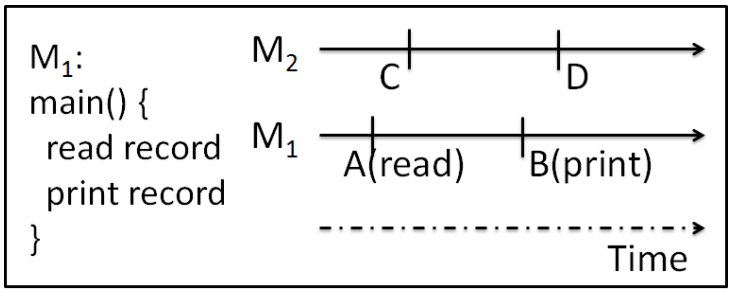

Figure 13.1: H-B Relation Rule 1

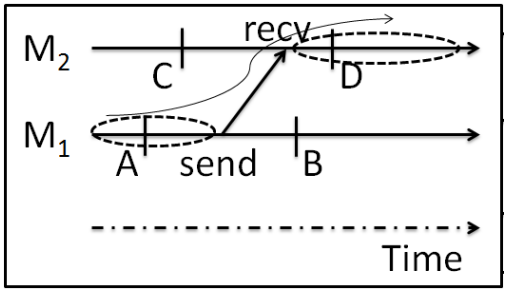

Figure 13.2: H-B Relation Rule 2

13.1.1.2 The Algorithm of Lamport's Clocks

Lamport's algorithm defines the notion of time of an event. The logical timestamp $C(A)$ of an event A is an integer such that:

- If $A \rightarrow B$, $C(A) < C(B)$
- If A and B are concurrent, $C(A) <, =$ or $> C(B)$

The algorithm is simple:

- Each process i maintains a logical clock LC_i
- Whenever an event occurs locally at i, $LC_i = LC_i + 1$
- When i sends a message to j, it "piggybacks" LC_i
- When j receives a message from i, $LC_j = max(LC_i, LC_j) + 1$

Figure 13.3 gives an example. M_1 and M_2 maintains two independent logical clocks initially. At time 3 on M_1, one message is passed from M_1 to M_2. The logical clock of M_2 is then set to be greater than both its own clock and the timestamp of the received message.

13.1.2 Vector Clock

Vector clocks generalize Lamport's clocks. Lamport's clocks provides that if $A \rightarrow B$, $C(A) < C(B)$. However, it cannot guarantee the reverse that if $C(A) < C(B)$, $A \rightarrow B$. Vector clocks solve this by using a vector of logical clocks.

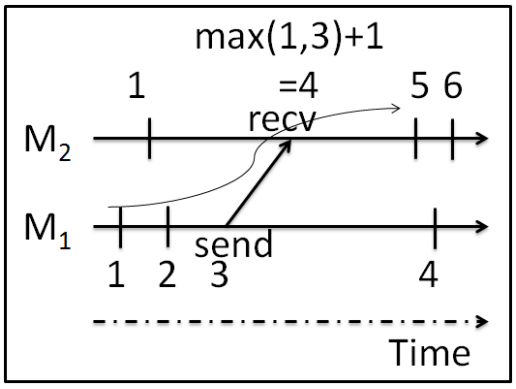

Figure 13.3: Lamport's Clocks

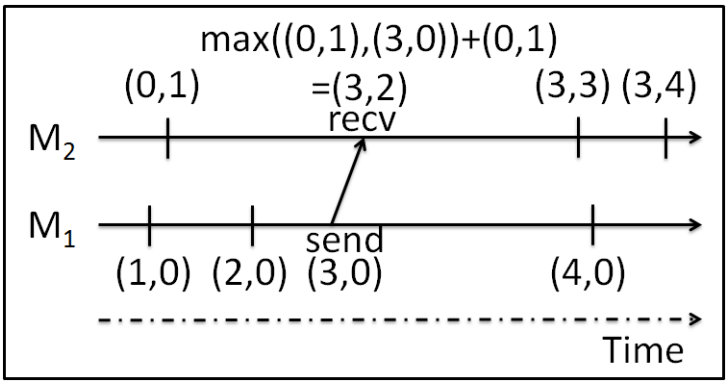

Figure 13.4: Vector Clocks

13.1.2.1 The Algorithm of Vector Clocks

In vector clocks, the logical timestamp $V(A)$ of an event A is a vector of integers. It guarantees that $(A \to B) \Leftrightarrow (V(A) < V(B))$. The algorithm is similar to the one of Lamport's clocks:

- Each process i maintains a vector V_i; $V_i[i]$ represents the number of events occurred at i and $V_i[j]$ means the number of events that i knows have occurred at process j
- Whenever an event occurs locally at i, $V_i[i] = V_i[i] + 1$
- When i sends a message to j, it "piggybacks" the entire vector V_i
- When j receives a message from i, $\forall k, V_j[k] = max(V_j[k], V_i[k])$, and then $V_j[j] = V_j[j] + 1$

One example is illustrated in Figure 13.4. Every process maintains a vector and only increases its own element when local event occurs. When M_2 receives a message from M_1, it updates its own vector and increases its own element by one. It is obvious in the figure that $A \to B$ if and only if $V(A) < V(B)$.

13.1.2.2 Causal Delivery

One application of vector clocks is to provide causal delivery. The causality relation "$A \to B$" means A is causally related to B, which is similar to happened-before relation. Causal delivery is a property guarantees that $(send(m) \to send(n)) \Rightarrow (deliver(m) \to deliver(n))$, where m and n are messages.

Causal delivery order is useful in totally-ordered multicasting cases. Suppose there is a distributed database for a bank. Its two replicas M and N are at different locations. Now an update m comes to M and another update n comes to N. All updates should be multicasted and applied on both M and N. However, because of network latency, M receives m first while N receives n first. This may cause inconsistency when the ordering matters. For example, m is "deduct 100 dollars from Bob's account" and n is "add the monthly interest to Bob's account". There will be no problem if causal delivery of m and n can be guaranteed.

Figure 13.5 illustrates how vector clocks provide causal delivery. P_0 broadcasts message m at $VC_0 = (1, 0, 0)$ first. After m reaches P_1, P_1 updates its vector clock and then broadcasts another message $m*$ at $VC_1 = (1, 1, 0)$. P_2 may be close to P_1 and receives $m*$ before m comes. Then P_2 checks the timestamp $V(m*) = (1, 1, 0)$ and notices there should be another message from P_1 on the way, because the first element of $V(m*)$ is 1 which is greater than the first element of its own vector clock $VC_2 = (0, 0, 0)$. So it holds $m*$ until it receives m, which guarantees causal delivery.

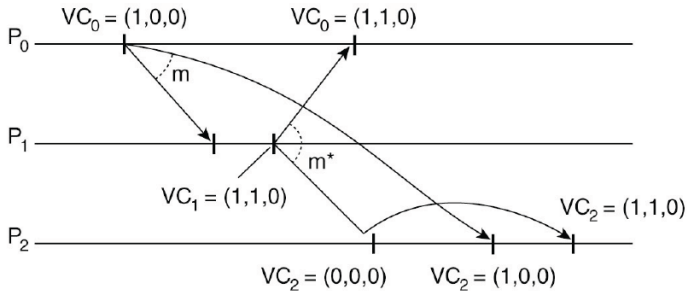

Figure 13.5: Vector Clocks in Causal Delivery

13.2 Distributed Snapshot Algorithm

Distributed snapshot of a distributed system is important in many applications such as failure recovery, distributed deadlock detection and so on. It captures a consistent global state of the system, which includes local state of each process and messages that are sent but not received.

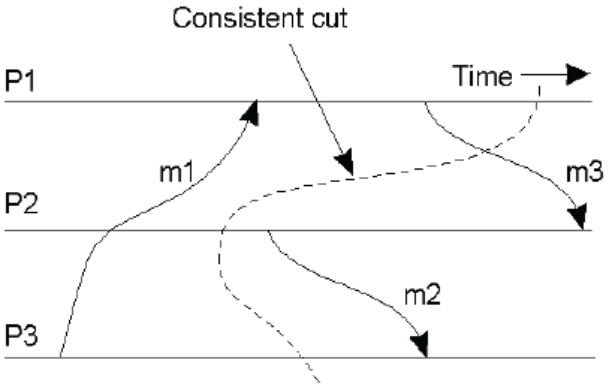

Figure 13.6: Consistent State

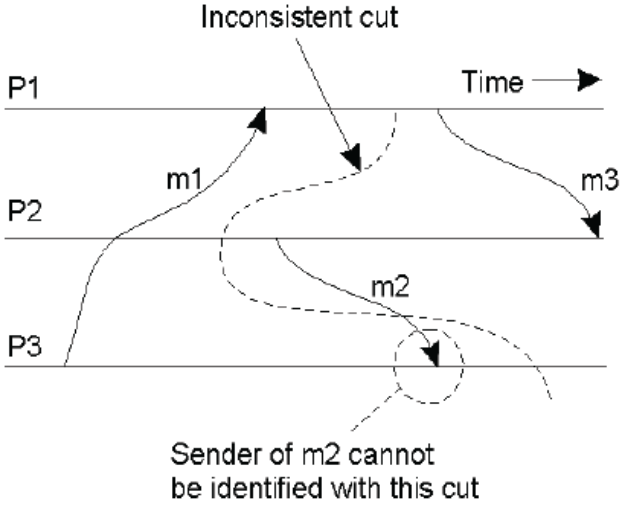

Figure 13.7: Inconsistent State

Figure 13.6 and 13.7 show a consistent state and an inconsistent state. The dashed lines are "cuts" indicates the time each process capture its local state. In the consistent state, all messages are either completely before the cut or sent before the cut. In the inconsistent state, one message m_2 is received before the cut, but sent after the cut. It will cause problems when the system is recovered from the cut because P_2 will resend the message m_2 and impact P_3 again, even though P_3's local state has already be impacted before the cut.

The detailed algorithm will be introduced in next lecture.