Distributed Transaction Management Note 3
Logical Clocks

Peter Thanisch and Jyrki Nummenmaa

October 27, 2003

1 Logical clocks

This section is based on Chapter 9 of the book by Chow and Johnson.

As we will probably see on some of our future lectures, there are problems with ordering events using physical clocks. If we give up the idea of using physical clocks, then we should use something like a logical order of events. Ordering events logically on one host is not a problem (but then again, neither is using a clock). Ordering events from different hosts logically needs some events between the hosts. As the hosts communicate exchanging messages, it would be natural to use the exchange of messages to order events.

If we know that event e_1 happened before event e_2, we write $e_1 < e_2$. We write $e_1 <_p e_2$, if e_1 and e_2 occurred at the same processor, and e_1 happened before e_2. Further, if e_1 is the sending of some message m, and e_2 is the receipt of m, then we write $e_1 <_m e_2$.

We define the *happens-before* relation to be the transitive closure of the processor orderings and the message passing orderings:

1. If $e_1 <_p e_2$, then $e_1 <_H e_2$.
2. If $e_1 <_m e_2$, then $e_1 <_H e_2$.
3. If $e_1 <_H e_2$ and $e_2 <_H e_3$, then $e_1 <_H e_3$.

The happens-before relation is a partial order but not a total one, since it might well happen that for some events e_1 and e_2 we have neither $e_1 <_H e_2$ nor $e_2 <_H e_1$. If the happens-before relation does not order two events, they are said to be concurrent.

We can get a directed acyclic graph, called *happens-before* DAG, from the events in such a way that there is an edge (e_1, e_2) if $e_1 <_p e_2$ or $e_1 <_m e_2$. From the happens-before DAG it is, e.g., possible to generate all possible execution orders of the events using graph-theoretical algorithms.

1.1 Lamport timestamps

Lamport's timestamp protocol produces timestamps, which preserve the happens-before relation. It can be used to produce a total order for the events, if, when the timestamps are otherwise identical, we use something like processor addresses to order the timestamps.

1. Initially, assing 0 to my_TS.
2. If event e is the receipt of a message m, then:
   Assign $max(m.TS, my\_TS)$ to my_TS.
   Add 1 to my_TS.
   Assign my_TS to $e.TS$.
3. If event e is the sending of a message m, then:
   Add 1 to my_TS.
   Assign my_TS to both $e.TS$ and $m.TS$.

Lamport's timestamp algorithm guarantees that if $e <_H e'$, then $e.TS < e'.TS$ (Exercise 1). However, there is no guarantee that if $e.TS < e'.TS$, then $e <_H e'$ (why?).

1.2 Vector timestamps

Sometimes we would like to have messages delivered in the same order as in which they are sent. We will denote sending of a message m by $s(m)$ and receipt of a message m by $r(m)$. A *causality violation* happens if for some messages m_1, m_2, we have $s(m_1) <_H s(m_2)$ but $r(m_2) <_p r(m_1)$.

To be able to detect causality violations, we can use *vector timestamps.* The vector timestamp is an array of M integers, where M is the number of processors. We define the order on vector timestamps as

- $e.VT \leq_V e'.VT$, if and only if $e.VT[i] \leq e'.VT[i]$, $1 \leq i \leq M$.
- $e.VT <_V e'.VT$, if and only if $e.VT[i] \leq_V e'.VT[i]$, and $e.VT \neq e'.VT$.

It is left as an exercise to show that $e.VT <_V e'.VT$ if and only if $e <_H e'$. The following algorithm assigns vector timestamps to events.

1. Initially, assign $[0, ..., 0]$ to my_VT.
2. If event e is the receipt of a message m, then:
   For $i = 1, ..., M$ assign $max(m.VT[i], my\_VT[i])$ to $my\_VT[i]$.
   Add 1 to $my\_VT[self]$.
   Assign my_VT to $e.VT$.

3. If event e is the sending of a message m, then:
   Add 1 to $my\_VT[self]$.
   Assign my_VT to both $e.VT$ and $m.VT$.

1.3 Causal communication

This subject is dealt with in a very short manner and it is not really a part of the distributed transaction management course module. However, you may read on, if you are interested.

Suppose we would want to eliminate causal violations. Then, as a message m arrives we may have to store it and only deliver it when we are sure that each such message m' that $s(m') <_H s(m)$ has arrived and been delivered.

It is easy to eliminate causal violations within point-to-point communication (which is what e.g. TCP does) by simply numbering the messages.

Eliminating causal violations in general is more tricky. It is possible to do it with vector timestamps by checking before delivery of a message m that we can not receive any messages with earlier vector timestamps. However, it would reduce waiting if we could see all messages (so that we would know if they were to arrive to us or not). For this reason, it can be practical to multicast the messages to all processors.

1.4 Distributed snapshots

We call the state of a process at one given time a *snapshot*. From the discussion above it should be clear that there is no way we can take simultanous snapshots of all processors (in order to construct a global state).

However, some kind of a global snapshot would be very useful for some purposes. For instance, a distributed deadlock detection algorithm examining the local waits-for graphs may draw incorrect conclusions because the global state changes (local waits-for graphs may change) while the algorithm is running.

Assume we have m processors and each of them has a state (like contents of their memory) s_i, $1 \leq i \leq M$. Let $S = \{s_1, ..., s_M\}$ be the global state of all processors. In addition to the global state of the processors, the global state has to include the messages that are travelling between processors. We assume that between each pair (p_i, p_j) of processors there is a reliable order-preserving communication channel $C_{i,j}$, whose contents is a list of messages $L_{i,j} = m_1, m_2, ..., m_k$. Let $L = \{L_{i,j}\}$ be the collection of all message lists. We say that the pair $G = (S, L)$ is the global state of the system.

We say that a state is *consistent cut*, if for all pairs of events e, e' in S, e and e' are concurrent.

We must somehow record the states and wait for arrival of travelling messages and record them appropriately. For this, we use *snapshot tokens*.

As some processor p_i initiates the distributed snaphsot collection, it stops sending messages, records its state and sends a snapshot token to all other processors with which it communicates. After that it can start sending messages again.

When processor p_j receives the snapshot token from p_i, it must stop receiving and sending messages. Then p_j records its state and sends a snapshot token to all processors with which it communicates. After that it can start receiving and sending messages normally.

Suppose that the processor p_j receives a snapshot token for the first time and it receives it from processor p_k. Then, since messages are delivered in order, it has received all messages that are sent from p_k before the snapshot token and none are on their way, therefore $L_{k,j}$ is empty.

Suppose now that p_j has received a token from p_k but it has also received a token earlier on and it has recorded its state at that time. Let L' be the list of messages it has received between recording its state and receiving the snapshot token from p_k. Then $L_{k,j} = L'$ (Showing this may still need a little work).

Once the lists and states have been recorded, a snapshot can be composed of these. There are further complications in the actual implementation if we consider the possibility that there may be several snapshots being collected at the same time.

2 Extra exercises for your enjoyment

In Exercises 1 and 2 you may use a fixed number of fixed participants and a fixed set of messages. You can also simulate possible decisions by random numbers or user interaction (which you can also implement using the sockets, if you want to).

1. Suppose that the following happens on processors $P_1, P2$ and P_3.
   P_1: Send m_1 to P_2. Receive m_2 from P_2. Receive m_3 from P_3.
   P_2: Receive m_1 from P_1. Send m_2 to P_1. Receive m_4 from P_3.
   P_3: Send m_3 to P_1. Send m_4 to P_2.
   Give 4 possible execution orders.

2. How many different execution orders exist for Exercise 1?

3. Show that the algorithm in this note for giving vector timestamps works correctly.