CMPSCI 677 Operating Systems **Spring 2019**

Lecture 13: March 19

Lecturer: Prashant Shenoy *Scribe:* ***Chenghao Lyu***

Announcements

- Midterm on Friday (3.22) in ILC S140. The exam will cover first 13 lectures plus the guest lecture. Last year's exam has posted and can be used for reviewing. Please focus on the slides and notes.
- Lab 2 has been put out. Please start early.

Q: Will you post the solution for the past exam?
A: Yes, I will
Q: Is the guest lecture a part of this exam?
A: Yes.

13.1 Logical and Vector Clocks

13.1.1 Recap from last lecture

Last lecture, we talked about (1) clock synchronization (2) Network Time Protocol (NTP), and (3) logical clocks or Lamport's clock. For the logical clocks, the problem we address is how to reason about the ordering of events in a distributed environment without the specific times at which events occurred.

According to the last lecture, we can actually use the concept of a logical time, which depends on message exchanging among processes, to figure out the event order. A message involves two events, the sending of a message on the first process and the receipt of the message in the second process. Since the message has to be sent before it is received, those two events are always ordered. This allows us to order two events across machines. In addition, local events within a process are also ordered based on the execution order. According to the transitive property of events ordering, events across processes are ordered.

It is worth noting that, Lamport's logical clock is a partial ordering of events. Specifically, the events that occur after the send cannot order w.r.t. another process unless there are more messages exchanged. Similarly, events that before receipt of a message also cannot be ordered w.r.t. other processes unless there are other prior messages that have been received.

Another disadvantage of logical clocks is that it does not have causality. If event A has occurred before event B, then the clock value of A is less than the clock value of B. However, if a clock value is numerically less than the other, we cannot say the first event has happened before the second one.

13.1.2 Total Order

To convert a partial order into a total order, we can impose an arbitrary order by appending '.' and a process' id to a logical time value in each process. As a result, the process id can be used to break ties.

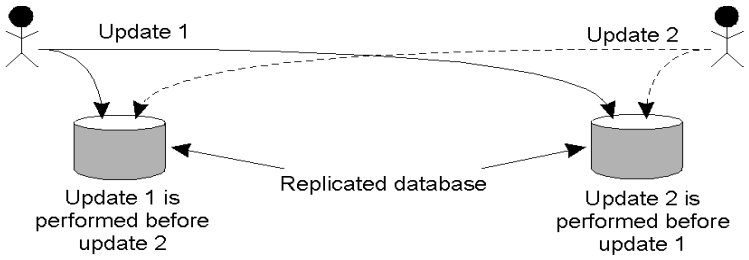

Figure 13.1: Example: Totally-Ordered Multicasting

What requires attention is that the total order is just a tie-breaking rule to assign an order for the events, so it does not actually tell us the real order of events.

This is an approach that many system designers used to take Lamport's clock which gives us a partial order and convert it into a set of events that gives us a total order.

13.1.3 Example: Totally-Ordered Multicasting

Here we use the example of a banking database to show the idea of total orders.

Assume a bank maintains the accounts in a database and the database is replicated on two sites. So every transaction that goes to this database and modifies it should be performed at both of the replicas for consistency reasons. Specifically, not only every transaction needs to be performed at all replicas, the order in which you perform the transaction has to be the same on both replicas.

To ensure the ordering is the same on each replica, we can use Lamport's clock to order all the transactions by a totally-ordered multicasting approach. Basically, when a set of transactions coming in to any replica, you are going to use a logical value to clock them similarly with the idea of above total order.

In the example of the replicated bank databases, whenever a message is sent to one database, it is also sent to all the other databases. All queries and transactions are replicated. So messages are multicast to all the bank database replicas as shown in Figure 13.1. Like the total order example, the logical time in each machine can be appended with "." and its machine id. So we can break ties for the logical clock values across different machines, thus each database replica will respect to a consistent order.

The details of this algorithm are not expended too much in the lecture, but essentially, the totally-ordered multicasting approach can ensure each replica receives all the transactions and executes them in a consistent order.

Q: Is it possible that there is too many transactions and a logical clock overflows?
A: I did say the logical clock is an integer, but a conceptual integer never overflows and can grow infinitely. The concept of a logical clock does not depend on an integer with a finite number of bits. However, when you are implementing the system, you are going to use a variable that is an integer with 32-bit or 64-bit and that integer can actually overflow if lots of events appear. So you need to make sure that whatever used to represent a logical clock has sufficient bits that exceeds the lifetime of your application. This is an implementation detail and nothing to do with the concept of a logical clock.
Q: what will happen if there is a network problem, e.g., in Figure 13.1, one update (message) only goes to one replica?
A: The algorithm actually handles the problem. The algorithm will process a transaction only it has been acknowledged by every process. If a message never even reaches a client, it is not going to acknowledge that

message so the message is going to sit in the queue forever and never get processed. In this way, safety property is preserved. But everything will be blocked if network problem happens. Thus, on the one hand, we do need a reliable network like TPC or reliable multicast that will do re-transmit when problem happens to guarantee the aliveness. On the other hand, it will not do anything bad when the network is not reliable.
Q: Does it guarantee a real-world ordering or does it guarantee consistency?
A: In this case, since we are not synchronized, we are not guarantee the real-world ordering. If you want real-world ordering, you need to do synchronization but there is no perfect clocks.

13.1.4 Causality

In Lamport's algorithm we couldn't say that if the clock value of A is less than B, then A has happened before B. We want this relationship to also be true for the clock values. So we have to determine a causal relationship between events in a distributed system. If the clock time of A is before B then A should have also preceded B, so the time stamps are comparable and meaningful.

13.1.5 Vector Clocks

Causality can be captured by means of **vector clocks**.

The idea of how vector clocks works is shown in Figure 13.2. Assume we have processes `A`, `B`, and `C`. Each process keeps an integer logical clock and the logical clock will be ticked whenever a local event happens. However, rather than keeping a single clock value like before, each process maintains a vector of clock value, one for each process in the system. Essentially, all clocks get initialized to `(0,0,0)`. The three elements are for the logical clocks in process `A`, `B`, and `C` respectively. Every time an event occurs in process `i` (`i` can be `A`, `B` or `C`), the `i-th` element of process `i`'s vector gets incremented. E.g., process `A` get its logical clocks in the first element of its vector clocks incremented in a series of events. When a message is sent by process `i`, the vector clock is piggybacked onto that message. When a process `j` received the message, the `i-th` element will be the maximum of `i-th` element of vector clock `i` and local vector clock `j`, and the `j-th` element will get incremented by one. Vector clock `(3,0,1)` and `(2,2,0)` are two instances.

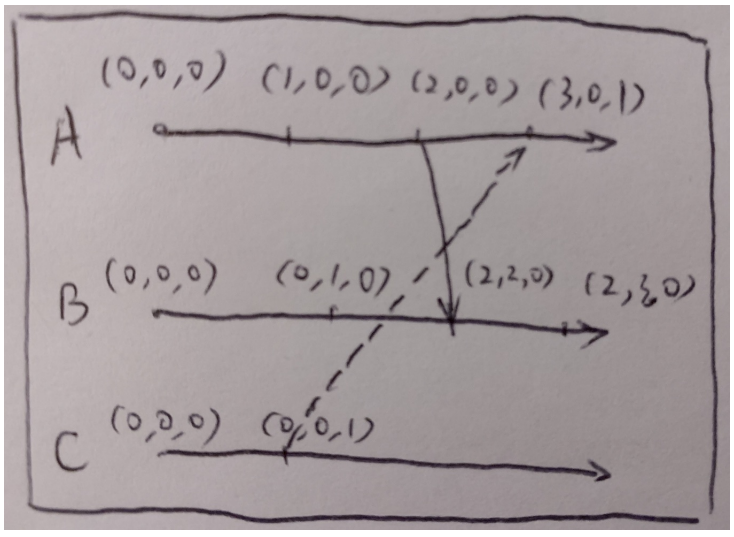

Figure 13.2: Vector Clocks

Vector clocks claims the causality property. If two events precede one another, the vector clocks are ordered. If two events are not ordered by exchanging messages, their order are not defined. Specifically, we define that a vector clock `A` is greater than a vector clock `B` iff every element of `A` is greater than or equal to every element of `B` and there is at least one element in A which is strictly greater than `B`. Otherwise, we say the greater than relationship is undefined in two vector clocks. It is worth noting that, unlike the logical clocks which are all comparable integers, vector clocks give undefined order relationship to two concurrent or independent events, which respects the reality and causality.

Q: A message is sent by one process but does not get received, so the message will be resent after timeout. Before timeout, another event occurs on that process with a new clock value. What clock value will the resent message use?
A: It should get the old clock value because it is just trying to send the same message.
Q: Is there any way to do total ordering?
A: Total ordering only gives an arbitrary order. Two events can always be arbitrary ordered by some tiebreaking rules. If you really want total ordering, you should just do Lamport's clock. However, technically, two events cannot be ordered. So you can have total order by using a tiebreaking rule but that does not mean that was the order in which events actually occurs.
Q: What is the overhead of the vector clocks?
A: If there is n processes in the systems that are exchanging messages. The clock has n integers in it as a vector. When the message has exchanged, you need to piggyback the clock value with the message. So there is small overhead in the message exchanging. Beyond that, between Lamport's clock and vector clocks, there is not a lot of differences.
Q: If you actually incremented the receiver's clock at the sender and send it, could you not compared that?
A: Our goal is not to impose a total ordering. You only derive an ordering when there is a message exchange that allows us to see the order. If two processes are never exchanging messages, the event order comparison will never work between them.

13.1.6 Enforcing Causal Communication

Using vector clocks, it is now possible to ensure that a message is delivered only if all messages that causally precede it have also been received as well. To enable such a scheme, we will assume that messages are multicast within a group of processes. As an example, consider three processes `P1`, `P2`, and `P3` as shown in Figure 13.3. At local time `(1,0,0)`, `P0` sends message `m` to the other two processes. After its receipt by `P1`, the latter decides to send `m2`, which arrives at `P2` sooner than `m`. At that point, the delivery of `m2` is delayed by `P2` until `m` has been received and delivered to `P2`'s application layer.

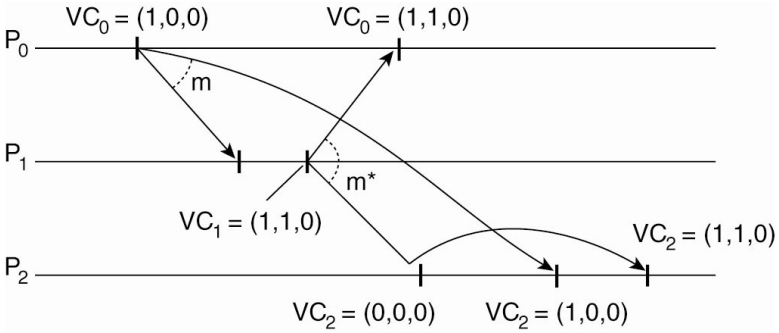

Figure 13.3: Enforcing causal communication

13.2 Global States and Distributed Snapshots

13.2.1 Global State

Problem definition: We want to run a distributed application and one of n processes crashes. Rather than killing all the processes and starting from the beginning, we can periodically take snapshots (or checkpoints) of the distributed application to keep a global state and start from the latest snapshot.

The global state includes the local state of each process and messages that are in transit (like the TCP buffers). The snapshot for a global state should be captured in a consistent fashion even without clock synchronization. A consistent fashion means that whenever restarting the computation from a checkpoint, you should get the same end result as if there was no cache at all.

Specifically, when a message exchanging crosses the snapshot taking, the sending point of the message instead of the receipt should be captured in the state. As shown in Figure 13.4, a consistent cut (a) can achieve the consistent state while a inconsistent cut (b) can not. For (b), if you restart the computation from whenever the dotted line hits each of the processes, m2 received by P3 can be inconsistent – P3 already saw the m2 before the snapshot and it will see m2 again after re-computation from the snapshot and P2 resent it.

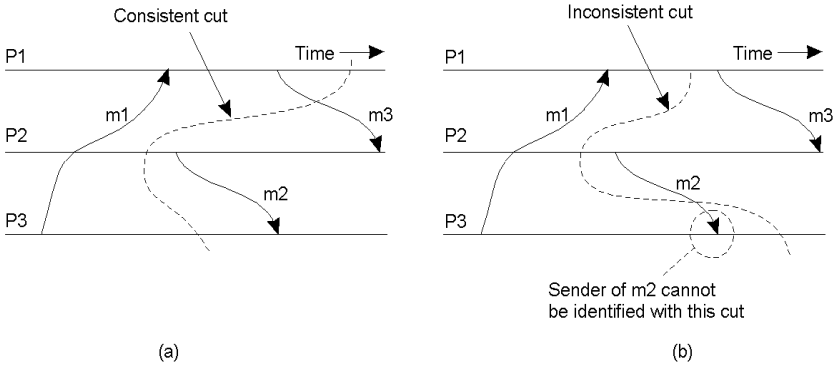

Figure 13.4: Consistent and Inconsistent cuts

Q: By "restart", are you saying all the processes are rollback to the previous state?
A: Yes. By restarting from the previous state, you can still get the same end results without the clock synchronization.
Q: What happens to m3 if we just restart in the computation?
A: Because p1 is starting from a point after it has sent m3, it is not going to resend m1. P2 is going to start from a point without seeing m3. So m3 is actually somewhere in the network and will be received by p2 after restarting. It would either be in the TCP buffer of the sending process p1, or the TCP buffer of the receipt process but not yet delivered to the application. So if you save these buffers and reinstall these buffers, you will recreate the message. Thus, you do not have to send it at the application level. So m3 will get delivered eventually because you are saving the memory state of the network as well in some sense.

13.2.2 Distributed Snapshot

A simple technique to capture that is to assume a global synchronized clock and freeze all machines at the same time and capture whatever that are in all memories at the exactly same time and capture everything

that is in transit. But there is no global clock synchronization. Distributed snapshotting is a mechanism that gives a consistent global state without a global clock synchronization.

The Photograph Example Photographer taking a picture of the sky with birds flying around in it. Can't capture entire sky in one pic. Stitch pics of left, middle, right parts of sky. If you take a picture of the left, and then take a picture of the middle, but a bird has flown from left to the middle, you have taken a picture of that bird twice because of inconsistency. The goal is to get a consistent photo - for it to be consistent, then all three pictures have captured all the birds that were in the sky.

Photographs are like snapshots that are taking of processes. Birds are messages that are going from one process to another. You do not want a bird missing or duplicated in your picture just like a message lost or double counted. This is why it is called distributed snapshots.

13.2.3 Distributed Snapshot Algorithm

Assume each process communicates with another process using undirectional point-to-point channels (e.g., TCP connections). Any process can start to a snapshot algorithm. When a process initial the algorithm, it will first checkpoint its local state, and then send a marker on every outgoing channel. When a process receives a marker, it will have different actions depending on whether it has already seen the marker for this snapshot. If the process sees a marker at the first time, it will checkpoint its state, send markers out and start saving messages on all the other channels. If the process sees a marker for the snapshot at the second time, it will stop saving messages for the channel from which the marker comes. A process will finish a snapshot until it receives a marker on each incoming channel and processes them all.

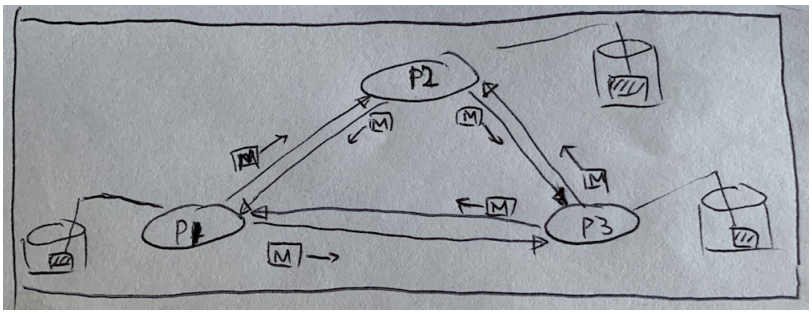

Figure 13.5: Distributed Snapshot Example

For example, in Figure 13.5, process P1, P2 and P3 communicates with each other through the duplex TCP connections. When process P1 initial a snapshot, it will first save its memory context (state) into the disk, and then send a maker to P2 and P3. It will also start to save the incoming messages (TPC buffers) from P2 and P3. When P2 receives a marker from P1, it will start to checkpoint state, send a marker out to P1 and P3, and start saving messages from P3. Assume P3 sees the marker sent from P1 first, it will checkpoint its state, sends out a marker to P1 and P2, and start saving messages from P2. P1 will stop saving messages for P2 or P3 until a marker from P2 or P3 arrives. P2 will stop saving messages for P3 until a marker from P2 arrives. And P3 will stop saving messages for P2 until a marker from P3 arrives. Each process will finish this snapshot when it sees a marker from every incoming channel. Thus, a distributed snapshot captured.

Q: Once a process has received a marker, does it continue executing?
A: As soon as a marker arrives, the process will initial the algorithm, save the state, send out marker messages, and just continue executing. But it still keep recording incoming messages. It is still executing, but not freezing.

Next, let us see why this algorithm gives us a consistent cut and gives us all the processes' state and all the messages in transit. On the one hand, if you send a marker of message on your outgoing channel, you are going to always receive a marker message on the incoming channel. On the other hand, all the messages in transit are saved. Those messages in transit are precisely the ones sent by one sender process but not processed by the receipt process. Therefore, the snapshot can take a consistent cut without synchronizing time. Besides, the network losing is not a problem since we are using the reliable TCP protocol.

Q: What happens if a process fails during a snapshot?
A: If you have failure in a snapshot, it is a partial snapshot, not a complete snapshot. So you got to go back to the previous completed snapshot and start from that point. So you can not take a partial snapshot and restart.
Q: Is the TCP buffer part of the local state?
A: TCP buffer is not part of the memory state. When doing snapshot, you save the memory state and separately take all the messages that are arriving in your TCP buffer and saving them as well. They will be eventually part of the local state that will save on your disk.
Q: Why not just save the memory state and whatever in your memory buffer and discard all the messages that in the router.
A: You could modify the algorithm but then you need to save not only the incoming buffer but also the outgoing buffer.
Q: How does this related to at least once, at most once and exact once?
A: This is going to be essentially the exact once message delivery because if we assume the network is reliable, any message that is in transit has not been delivered will still be delivered. You also not going to be losing your messages. This is using the TCP properties.
Q: Why could you use this as opposed to vector clock?
A: Vector clock is not a snapshot algorithm. It can used to reason about which messages are in transit and save them, and you can still get consistency if you want. But since they still need to keep the state of memory and the state of network, marker algorithm is a much simper way to do this.
Q: What if P1 and P2 decide at the same time to start a snapshot?
A: It is ok because each actually is a distinguish snapshot. What we assume is that whenever you start a snapshot, you will get a unique id for it. You can have multiple snapshot in progress so long as you distinguish marker messages between two snapshots by their id.
Q: Will the new snapshot override the old one?
A: Yes, you can. But you need to make sure the snapshot is fully completed.