CMPSCI 677 Operating Systems **Spring 2025**

Lecture 12: March 25

Lecturer: Prashant Shenoy *Scribe:* ***Ritika Hooda (2025), Tammy Sukprasert (2024)***

12.1 Overview

The topic of the lecture is ”Vector clocks and distributed snapshots”. The following areas are covered:

- Vector clocks
- Distributed Snapshots
- Termination Detection
- Leader Election

12.2 Logical and Vector Clocks

12.2.1 Recap from last lecture

For 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. We can use the concept of a logical time, which depends on message exchanges 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.

Lamport’s logical clock is a partial ordering of events. Specifically, the events that occur after the send cannot be ordered w.r.t. another process unless there are more messages exchanged. Similarly, events that occur before the 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. In Lamport’s clock if events are concurrent we cannot assign any ordering to them as there is no happen before relationship. Therefore, we only get a partial order using logical clocks.

12.2.2 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. 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. All the Lamport's clock properties still hold. 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.

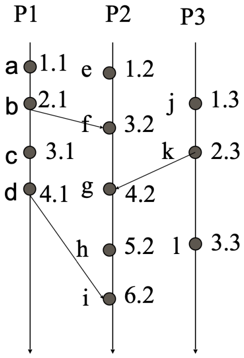

Figure 12.1: Creating total order by appending process id

12.2.3 Example: Totally-Ordered Multicasting

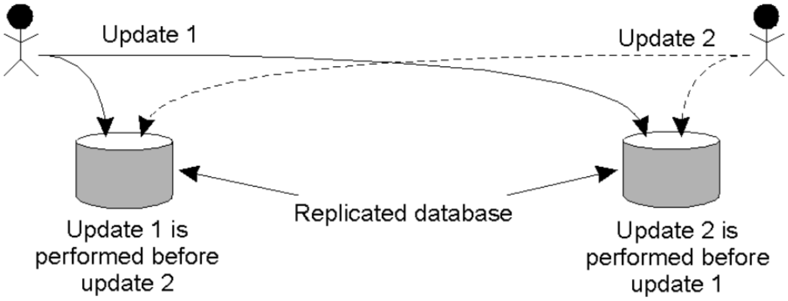
Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

Figure 12.2: Example: Totally-Ordered Multicasting

Here we use the example to show the idea of total orders. There are two replicated databases. Because they are replicated a query goes to both copies and both will update. With a partial ordering, the problem arises when there are queries that are sent in parallel. It may happen that when a user (User 1) sends a query to both replicas and is closer geographically to one of them (thus causing the query to be recieved sooner in the closer replica), and another user (User 2) sends another query at the same time and is closer to the other replica, the queries are executed out of order, thus producing inconsistent results.

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. When a set of transactions comes in to any replica, we use a logical value to clock them similarly with the above idea of total order.

In this example, 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 database replicas as shown in Figure 12.2. Like the total order example, the logical time in each machine can be appended with "." and its machine id, allowing us to 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 the totally-ordered multicasting approach can ensure each replica receives all the transactions and executes them in a consistent order.

12.2.4 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. Nothing can be said about events by comparing timestamps. In some cases, we may need to maintain causality, i.e., if A happened before B, A is causally related to B. *Causal delivery* means that if the send of a message happens before another message, the receive should also be in this order. We need a time stamping mechanism such that if T(A) < T(B), A should have causally preceded (happened before) B.

12.2.5 Vector Clocks

Causality can be captured by means of **vector clocks**. Instead of just one integer, every process maintains a vector as long as the number of processes, so each process i has a vector V_i. $V_i[i]$ is the number of event that have occured at i, which is effectively its Lamport's clock. $V_i[j]$ is the number of events i knows have occured at process j. Vector clocks are updated as follows in each of the below situations:

- Local event: increment $V_i[I]$
- Send a message: piggyback entire vector V
- Receipt of a message: $V_j[k] = \max(V_j[k], V_i[k])$
- Receiver is told about how many events the sender knows occurred at another process k
- Also $V_j[j] = V_j[j] + 1$

The idea of how vector clocks works is shown in Figure 12.3. Assume we have processes P_1, P_2 and P_3. 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 process P_1, P_2, and P_3 respectively. Every time an event occurs in process `i` (`i` can be P_1, P_2 or P_3), the `i-th` element of process `i`'s vector gets incremented. E.g., process P_1 get its logical clocks in the first element of its vector clock incremented in the first 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.

Two examples are shown in Figure 12.3 when a message is passed from P_1 to P_2 and V_2 is updated to `(1,2,0)` and then when a message is passed from P_2 to P_3 and V_3 is updated to `(1,3,4)`. Through sequence of messages each vector clock knows something about the other processes either directly or indirectly.

To check if there is a happen before relationship maintains lets take the send (S) from P_1 and the recieve at P_3 (R). Clearly, S has happened before R. We need to have `(1,3,4)` > `(1,0,0)`. We define greater between vectors as V1¿V2 if every element of V1 is $\geq$ every element of V2 and there is at least one element that is strictly greater. This is true in the case of S and M. We cannot compare `(1,4,0)` and `(1,3,4)`. That means they are concurrent events. Vector clocks claims the causality property.

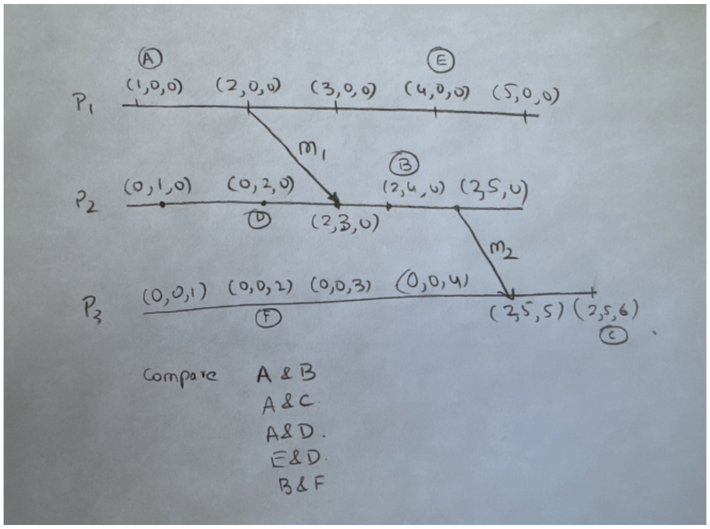

Figure 12.3: Vector Clocks

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: Are the clock distributed among each process?
A: The clock is not distributed, it is a vector that each process maintains (i.e., each process maintains its local array)

12.2.6 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 `P0`, `P1`, and `P2` as shown in Figure 12.4. 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 `m*`, which arrives at `P2` sooner than `m`. At that point, the delivery of `m*` is delayed by `P2` until `m` has been received and delivered to `P2`'s application layer.

12.3 Global States and Distributed Snapshots

12.3.1 Global State

Problem definition: We want to run a distributed application where 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.

In a distributed system, when there are n processes communicating with each other, taking a checkpoint is harder. We need to take the snapshot in a consistent manner.

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

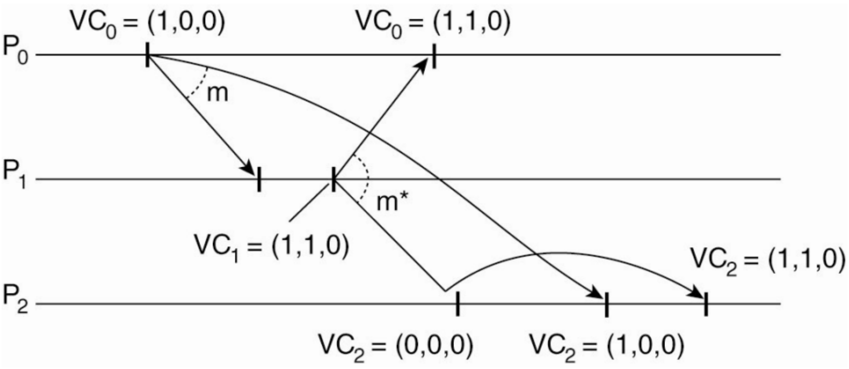

Figure 12.4: Enforcing causal communication

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. We will eliminate the notion of a clock and derive a technique independent of clock synchronization.

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 12.5, 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. the send should be to the left of the cut and the receive to the right.

Q: Can we instead send messages with unique IDs, so that we can track which message is sent/received?
A: You could, but the consistent cut eliminates the need for the unique IDs.

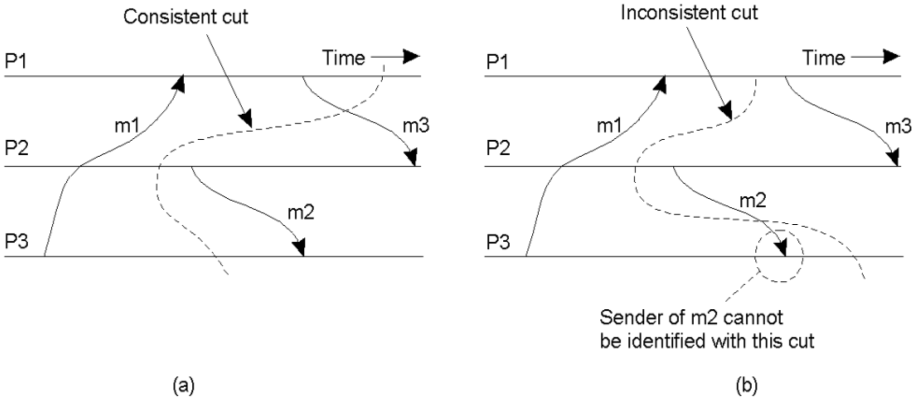

Figure 12.5: Consistent and Inconsistent cuts

12.3.2 Distributed Snapshot

A simple technique to capture 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: Imagine a photographer taking a picture of the sky with birds flying around in it. They cannot capture entire sky in one picture, so instead they stitch pictures of the left, middle, and right parts of sky. If they take a picture of the left, and then take a picture of the middle, but a bird has flown from left to the middle, they 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 of the birds that were in the sky exactly once.

Photographs are like snapshots (in fact, this is where the name comes from). 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 you do not want a message lost or double counted.

12.3.3 Distributed Snapshot Algorithm

Assume each process communicates with another process using undirectional point-to-point channels (e.g., TCP connections). Any process can initiate the snapshot algorithm. When a process initiates 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.

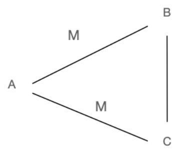

Figure 12.6: Distributed Snapshot Example

For example, in Figure 12.6, process 1, 2 and 3 communicates with each other through the duplex TCP connections. When process 1 initiate a snapshot, it will first save its memory contents (state) into the disk, and then send a marker (in blue) to 2 and 3. It will also start to save the incoming messages(in red) (TPC buffers) from 2 and 3. When 2 receives a marker from 1, it will start to checkpoint state, send a marker out to 1 and 3, and start saving messages from 3. Assume 3 sees the marker sent from 1 first, it will checkpoint its state, sends out a marker to 1 and 2, and start saving messages from 2. 1 will stop saving messages for 2 or 3 until a marker from 2 or 3 arrives. 2 will stop saving messages for 3 until a marker from 2 arrives. And P3 will stop saving messages for 2 until a marker from 3 arrives. Each process will finish this snapshot when it sees a marker from every incoming channel. Thus, a distributed snapshot captured.

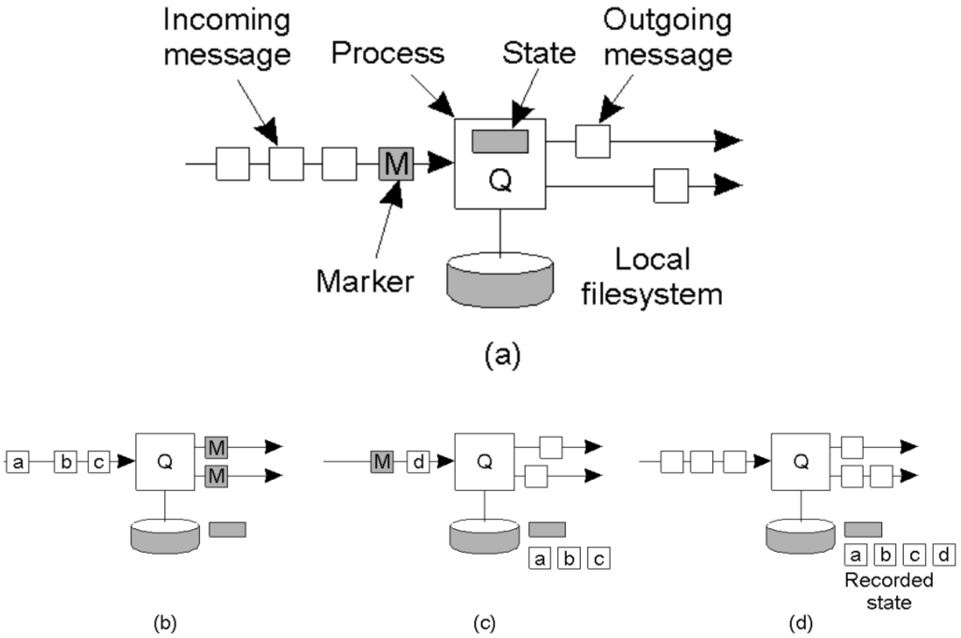

Figure 12.7: Snapshot algorithm example.

12.3.4 Snapshot Algorithm Example

Q decides to take a snapshot. Consider Figure 12.7. b) It receives a marker for the first time and records its local state; c) Q records all incoming message; d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel. First incoming marker says start recording and all other incoming messages say stop recording.

Q: If you have N processes in the system, does that mean each process has to wait for $N-1$ markers before it can stop recording?
A: The number of the expected markers is not dependent on the number of processes in the system. Instead, the number of expected markers of each process depends on the number of socket connections of that process.

12.4 Termination Detection

This involves detecting the end of a distributed computation. We need to detect this since a process cannot just exit after completing its events if another process wants to send it a message. Let sender be the predecessor, and receiver be the successor. There are two types of markers: Done and Continue. After finishing its part of the snapshot, process Q sends a Done or a Continue to its predecessor. Q can send a Done only when:

- All of Q's successors send a Done
- Q has not received any message since it check-pointed its local state and received a marker on all incoming channels
- Else send a Continue

Computation has terminated if the initiator receives Done messages from everyone.

Q: What happens when there is a failure?
A: Terminal detection algorithm does not handle failures. We need to go previous snapshot if there is a crash.

12.5 Election Algorithms

12.5.1 Bully Algorithm

The bully algorithm is a simple algorithm, in which we enumerate all the processes running in the system and pick the one with the highest ID as the coordinator. In this algorithm, each process has a unique ID and every process knows the corresponding ID and IP address of every other process. A process initiates an election if it just recovered from failure or if the coordinator failed. Any process in the system can initiate this algorithm for leader election. Thus, we can have concurrent ongoing elections. There are three types of messages for this algorithm: *election*, *OK* and *I won*. The algorithm is as follows:

1. A process with ID i initiates the election.
2. It sends *election* messages to all process with ID $> i$.
3. Any process upon receiving the election message returns an OK to its predecessor and starts an election of its own by sending *election* to higher ID processes.
4. If it receives no OK messages, it knows it is the highest ID process in the system. It thus sends *I won* messages to all other processes.
5. If it received OK messages, it knows it is no longer in contention and simply drops out and waits for an *I won* message from some other process.
6. Any process that receives *I won* message treats the sender of that message as coordinator.

An example of Bully algorithm is given in Figure 12.9. Communication is assumed to be reliable during leader election. If the communication is unreliable, it may happen that the elected coordinator goes down after it being elected, or a higher ID node comes up after the election process. In the former case, any node might start an election process after gauging that the coordinator isn't responding. In the latter case, the higher ID process asks its neighbors who is the coordinator. It can then either accept the current coordinator as its own coordinator and continue, or it can start a new election (in which case it will probably be elected as the new coordinator). This algorithm runs in $O(n^2)$ time in the worst case when lowest ID process initiates the election. The name bully is given to the algorithm because the higher ID processes are bullying the lower ID processes to drop out of the election.

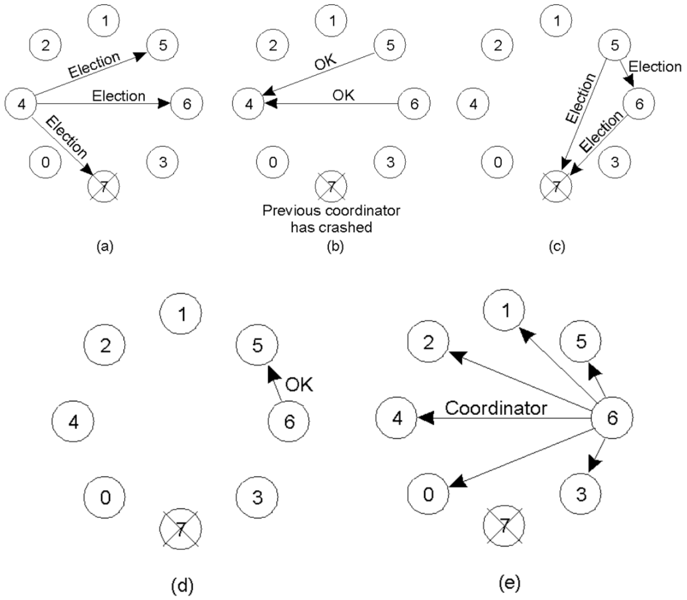

Figure 12.8: Depiction of Bully Algorithm

12.5.2 Ring Algorithm

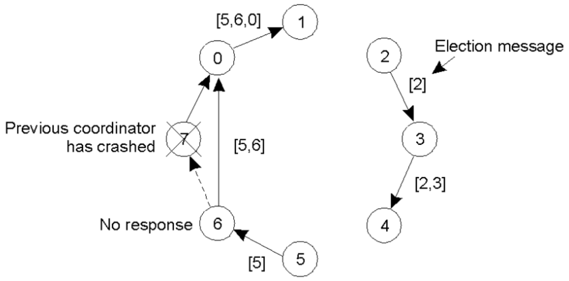

Figure 12.9: Depiction of Ring Algorithm

The Ring Algorithm is a leader election algorithm used in distributed systems where processes are arranged in a logical ring structure. Each process has a unique identifier (ID) and knows its immediate neighbors in the ring. The goal of the algorithm is to select the process with the highest ID as the new coordinator when the previous coordinator fails or when a process recovers from failure. Figure 12.9 depicts ring algorithm. The algorithm is as follows:

1. A process with ID i detects that the coordinator has failed or has just recovered and initiates the election.

2. It sends an *Election* message to its immediate successor in the logical ring.

3. Each process that receives the *Election* message adds its own ID to the message and forwards it to the next active process in the ring.

4. When the *Election* message completes a full cycle and returns to the initiator, the initiator selects the process with the highest ID from the collected IDs.
5. The initiator then sends a *Coordinator* message around the ring to inform all processes of the newly elected leader.
6. Any process that receives the *Coordinator* message recognizes the sender as the new leader and updates its state accordingly.

12.5.3 Comparison of Ring and Bully Algorithm

The Ring Algorithm and the Bully Algorithm are both leader election algorithms used in distributed systems, but they differ in efficiency and message complexity. The Bully Algorithm requires processes to directly communicate with all higher-ID processes, leading to a worst-case message complexity of O(n^2) when the lowest-ID process initiates the election. In the best case, when the highest-ID process initiates the election, it sends (n - 2) messages, resulting in an O(n) message complexity. On the other hand, the Ring Algorithm ensures a more structured election process, where messages circulate sequentially around a logical ring, always requiring 2(n - 1) messages regardless of the initiator. While the Ring Algorithm is simpler and guarantees leader election with fewer messages on average, it may take longer due to its sequential nature. The Bully Algorithm, though more message-intensive, can elect a leader more quickly, making it preferable in systems where faster recovery is essential.