DSC204A: Scaleable Data Systems Winter 2025

23: Introduction to Distributed Computing (cont.)

Instructor: Umesh Bellur *Scribe: Jean Hsu, Tino Trangia, Yash Vishe*

1 Logical Timing

1.1 Wall clocks do not work for distributed systems

The use of physical clocks for each machine in a distributed system is problematic because it is virtually impossible for every clock to be perfectly synchronized (i.e. skew). This means that the ordering of processes may not be reliable (for example, a message may be received at an earlier timestamp than when it was sent).

In order to avoid clock synchronization issues, distributed systems rely on the concept of **logical time**. This abstraction maps events to a set of numbers that allows them to be reliably ordered. Rather than a physical clock, the passage of time is governed by the execution of program operations. Thus, the semantics of the program determine whether an event precedes another. When a process state changes (e.g. a message is received), timers are incremented for each process.

1.2 Integer clocks

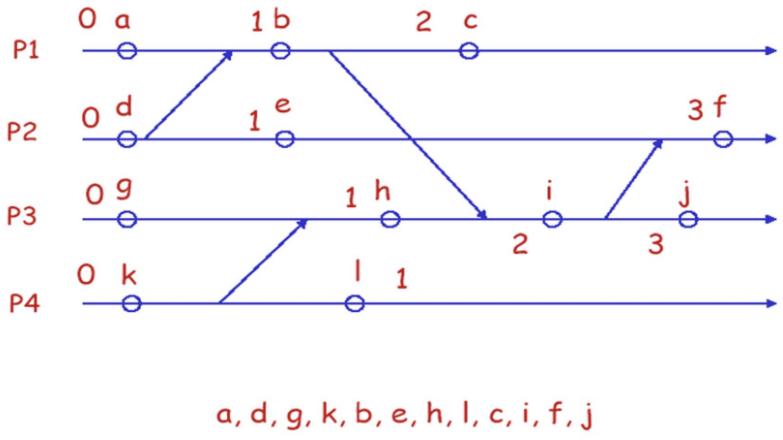

Figure 1: Integer clocks between 4 processes

Figure 1 represents a system with 4 processes. Each process begins with logical time 0. When an event occurs (denoted by letters), each process increments its own integer clock. For example, when P1 receives a message from P2 (event b), it increases the timer from 0 to 1. An event precedes another if and only if it's counter is strictly less than that of the other event.

When a process receives a message, it will take the maximum of the sender's timer and its own counter before incrementing. For example, P3 sends a message to process P2 (event i) when its clock value is 2. P2 receives the message (event f) when its clock value is 1. Therefore it increments its own clock value to 3. If the maximum was not taken, then it will appear as if event f and event i occurred concurrently.

1.2.1 The shortcoming of integer clocks

If an event e happens before event c, this implies that the clock value of e is strictly less than that of c. However, the converse is not necessarily true: if the clock value of e is less than c, we may not actually know if e preceded c. This is issue can be solved by using vector-valued clocks.

1.3 Vector clocks

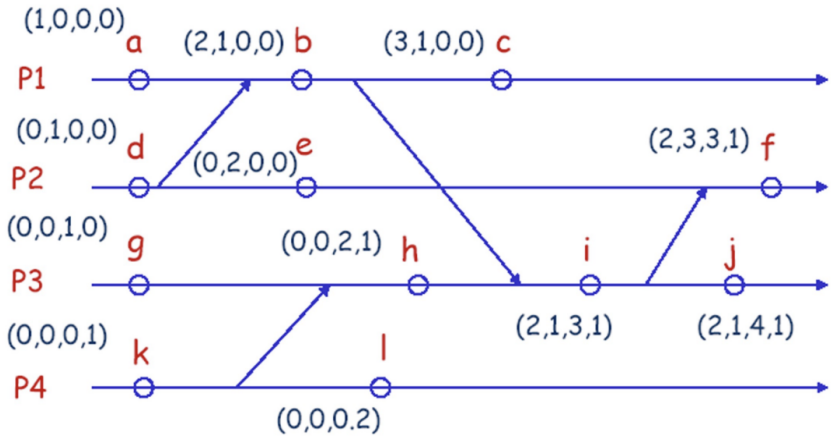

Figure 2: Vector clocks for partial ordering between processes

Rather than maintaining a single integer, each process now maintains a vector of length n, where n is the number of processes. For internal events, a process will increment the element corresponding to its process index. For external events, a process will replace each element in its vector with the maximum corresponding element, in addition to incrementing its own element by 1. In Figure 2, P1 with vector $(1, 0, 0, 0)$ receives a message from P2 with vector $(0, 1, 0, 0)$. Thus, it replaces the second element with the maximum value (from P2) while incrementing the first element (its own index) by 1, resulting in $(2, 1, 0, 0)$.

When comparing vector clocks, we take a component-wise less than. An event is only considered to precede another event if and only if at least one element of its vector is strictly less than the corresponding element of the other vector (all other elements can be less than or equal to its respective element). Otherwise, the events are considered unrelated (i.e. concurrent).

1.3.1 Why vector clocks are a stronger solution

Vector clocks are more useful than simple integer clocks in that the relation between clock values and ordering is symmetric: if the vector for event e is less than the vector for event c, than that *does* imply that event e preceded event c. The proof of this guarantee is out of scope, but stems from the fact that the vectors provide a *partial* ordering of events, while the set of natural numbers can only be used for *total* ordering.

2 Mutual Exclusion

2.1 Centralized approach

The centralized approach to mutual exclusion involves a single controller that manages access to a critical section. Processes request access from this controller, which then grants a token allowing entry. Although this approach ensures safety—only one process is allowed in the critical section at any given time—it and liveliness- for a process that has made access to the critical section will eventually be granted access to the seals, it has significant drawbacks. One key issue is fairness: due to network congestion or delays, requests may arrive at the controller out of chronological order, causing a process that made an earlier request according to logical clock to receive access after a later requester. Another critical disadvantage is that the centralized controller represents a single point of failure. If this controller fails, the entire mutual exclusion mechanism collapses, halting all access to the critical section and severely impacting system reliability. So, is there a distributed algorithm for this? Yes, one among them is Lamport's algorithm.

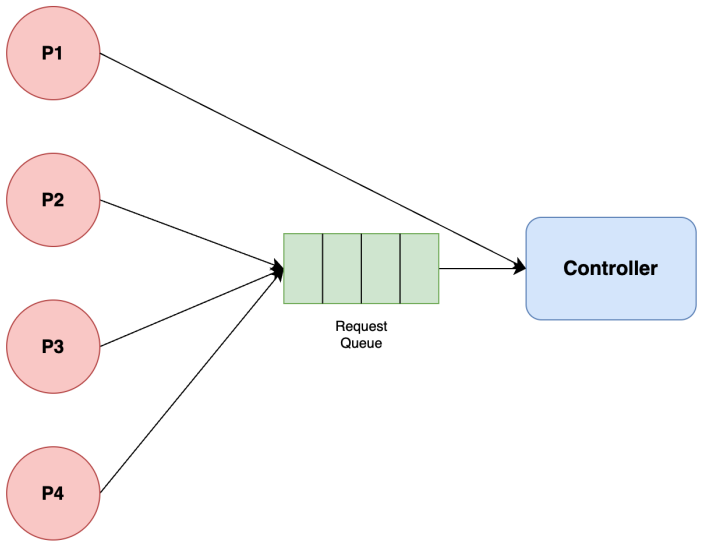

Figure 3: Centralized Approach

2.2 Lamport's algorithm

Each process maintains a logical clock and a request queue.

Algorithm:

- To request the Critical Section (CS), a process sends a (logical) time-stamped message to all other processes and adds the request to its own queue.
- On receiving a request, the request message is stored and an acknowledgement is sent back.
- To release the CS, a process sends a release message to all other processes.
- On receiving a release message, the corresponding request is deleted from the queue.

So a process can enter a critical section if and only if following conditions are met:

- It has a request in the queue with timestamp t
- t is less than all other requests in the queue
- It has received a message from every other process with timestamp greater than t.

Except for the acknowledgement, all other messages are multicast messages.

Analysis of Lamport's Algorithm:

- **Message complexity:** To send a request, the process has to multicast N - 1 messages. Acknowledgement has to be send via the process and to release the critical section, that's another N - 1 messages. So the complexity is order of N.
- The algorithm is **fair**
- The algorithm is **safe**
- The algorithm is **live**

3 Multicast Ordering

To fix the issue that, when multiple collective communications occur simultaneously in a distributed system, different delivery orders across processes can lead to inconsistent results. For example, if there are two broadcast B1 and B2 happen at the same time, processes may receive them in different orders. Therefore, we need multicast ordering to ensure that all nodes deliver messages to the application in the same order.

3.1 FIFO ordering

FIFO ordering ensures that messages sent from the same sender are received by all recipients in the order they were sent. For example, we can see in Fig 4 and assume that we have a distributed system with four processes. Suppose process P1 sends two multicast messages: M1:1 and M1:2, and process P3 sends a multicast M3:1. Since M1:1 and M1:2 are sent by the same process (P1), there is an inherent order between them: **M1:1 must be delivered before M1:2** at every recipient. If any process receives M1:2 before M1:1, the messaging substrate will reorder them to respect FIFO semantics (similar to how TCP ensures ordered delivery).
However, there is no ordering constraint between M1:1 and M3:1, since they come from different processes, and the system will still provide FIFO order semantics as the guarantee to the application.
Only multiple sends within the same process need to be strictly ordered in the order in which they are sent to all processes, and sends from different processes don't matter.

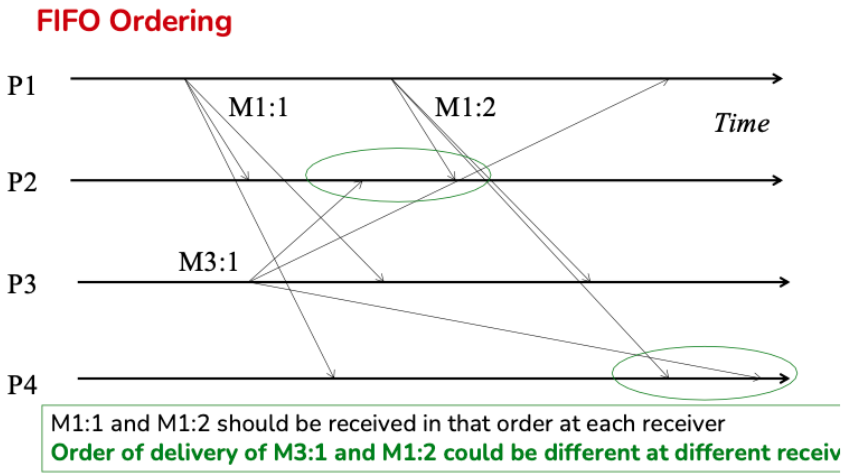

Figure 4: FIFO ordering

3.2 Causal ordering

Causal ordering is **stricter** than FIFO ordering. It not only requires the correct order of messages from the same process, but also ensures the correct order among different processes based on causal dependencies. For example, in Fig 5, just like FIFO ordering, every process should see M3:1 before M3:2. However, M1:1 actually happens before M3:1, since M3:1 was sent after M1:1 was received. As a result, every process must see M1:1 before M3:1.
If there is a strict causal order governed by the happens-before relationship, every process must respect it. However, messages that are concurrent (i.e., not causally related) can be delivered in any order.

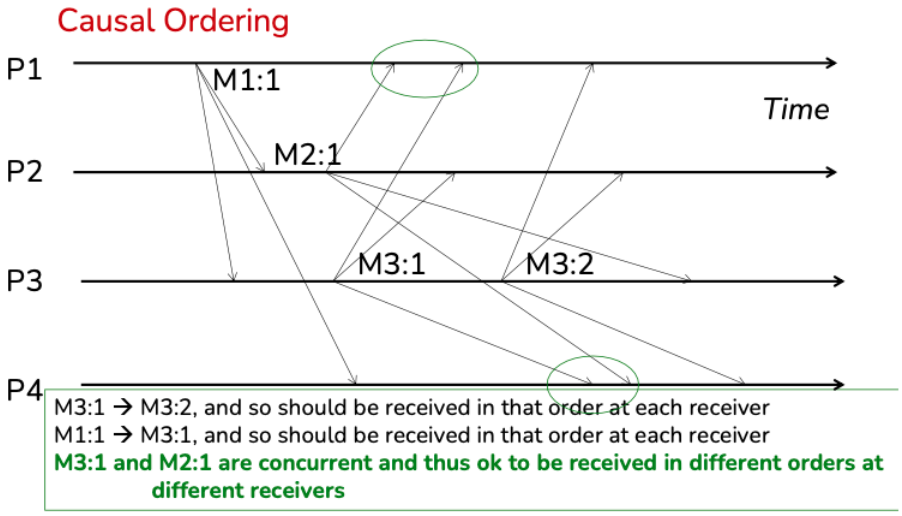

Figure 5: Causal ordering

3.2.1 Why use Causal ordering

Causal ordering is important in many real-world applications. For example, we want to email a bunch of people, and somebody replies to one of the emails: everybody should see the reply after the original email message. However, in practice, Gmail does not guarantee causal delivery although they try their best. The same logic applies in social media: everyone should see the post first, then the responses.
This ordering is easy to enforce in a centralized system, but maintaining it in a fully distributed system is much more difficult.
How we implement causal ordering is what we need to think about; for example, using timestamps captures the causal relationships between events.

3.3 Total ordering

For two concurrent sends, we cannot guarantee which one should come first, but **total order mandates** that **the same order is seen by all processes**. Whatever order we choose between two concurrent sends, it must be consistently seen by every process.
This is also known as atomic broadcast, which means that if a process decides to do a broadcast, then either every process receives the message or none do. A broadcast by itself is not inherently atomic, so this property must be enforced.
Under total ordering, the ordering of messages at every process is identical, and the original send order of multicasts doesn't matter. Only the order in which messages are delivered is guaranteed.
FIFO ordering is implied, but causal ordering is not implied. That means the process's internal message order is respected, but causal relationships between messages are not necessarily preserved.
For example, Fig.6 shows that every process has the same delivery order: M1:1 → M2:1 → M3:1 → M3:2

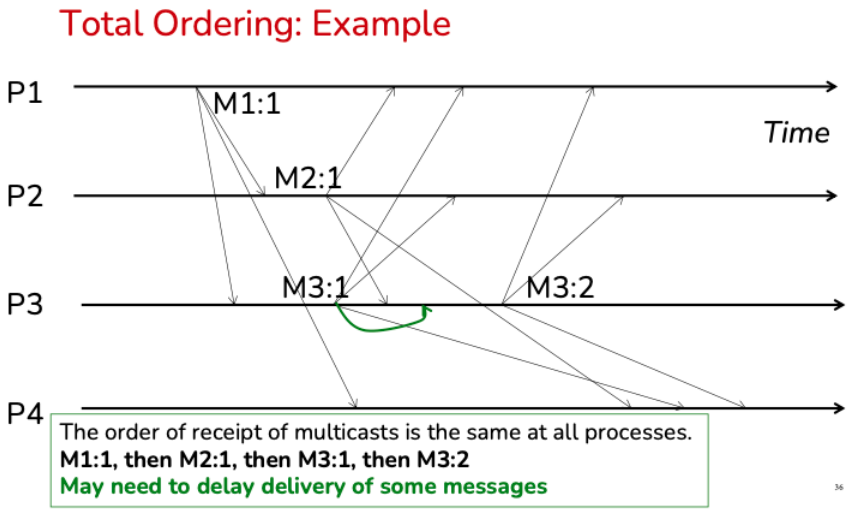

Figure 6: Total ordering