CMPU4021
Distributed Systems

Clock synchronization in distributed systems

Synchronization and coordination

- Closely related

- Synchronization
  - Make sure that one process waits for another to complete its operation
  - The problem is to ensure that two sets of *data* are the same.

- Coordination
  - The goal is to manage the interactions and dependencies between *activities* in a distributed system
  - Encapsulates synchronization

Clock synchronization

- In a centralized system, time is unambiguous.
- When a process wants to know the time, it simply makes a call to the operating system.
- If process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got.
  - It will not be lower.
- In a distributed system, achieving agreement on time is not trivial.

Clock synchronization

- Strongly related to communication between processes is the issue of how processes in distributed systems synchronize.

- Synchronization is all about doing the right thing at the right time.

- A problem in distributed systems, and computer networks in general, is that there is no notion of a globally shared clock
  - Processes on different machines have their own idea of what time it is.

Clock synchronization

- Physical clocks
  - Keep time of day
  - Consistent across systems, but
    - tend not to be in perfect agreement
    - clock drift, which means that they count time at different rates, and so diverge.

- Logical clocks
  - Keeps track of event ordering

Physical clocks

- Problem
  - Sometimes we simply need the exact time, not just an ordering
  - E.g. file compilation using `make`
    - If the source file `input.c` has time 2151 and the corresponding object file input.o has time 2150, `make` knows that input.c has been changed since `input.o` was created, and thus `input.c` must be recompiled.
    - If `output.c` has time 2144 and `output.o` has time 2145, no compilation is needed

- Solution
  - Universal Coordinated Time (UTC)
    - Based on the number of transitions per second of the cesium 133 atom
    - At present, the real time is taken as the average of some 50 cesium clocks around the world.

- UTC is broadcast through short-wave radio and satellite.
  - Satellites can give an accuracy of about +/- 0:5 ms
  - Satellite sources include the Global Positioning System (GPS).

Logical clocks

- What usually matters is not that all processes agree on exactly what time it is, but
  - that they agree on the order in which events occur
- Requires a notion of ordering
- A logical clock measures the passing of time in terms of logical operations, not the physical time

Logical Clocks

- Assign sequence numbers to messages
  – All cooperating processes agree on order of events
- Assumptions
  – No central time source
    - Each system maintains its own local clock
    - No total ordering of events
  – Multiple processes, each one of them:
    - Has unique IDs
    - Has its own incrementing counter

Logical Clocks

- Main types
  - Lamport's logical clocks
  - Vector clocks

Lamport's logical clocks

- Lamport – in his seminal paper in 1978
  - Showed that although clock synchronization is possible, it need not be absolute.
  - If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.
  - What usually matters is not that all processes agree on exactly what time it is, but rather that they agree on the order in which events occur.
- E.g. In the compilation example, what counts is whether `input.c` is older or newer than `input.o`, not their respective absolute creation times.

Logical clocks – Lamport logical clocks

- Lamport clocks
  - Allow processes to assign sequence numbers, so called *Lamport timestamps*, to messages and other events
    - all cooperating processes can agree on the order of related events.

- A monotonically increasing software counter
  - Whose value does not have particular relationship to any physical clock.

Lamport logical clocks

- To synchronize logical clocks, Lamport defined a relation called ***happens-before*****.**

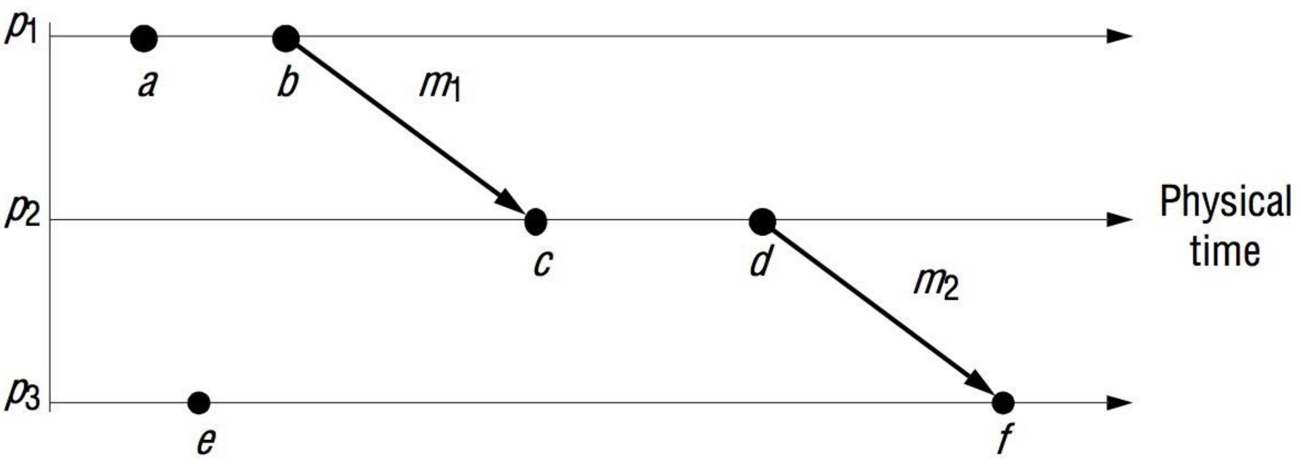
- The ***happened-before*** **relation** (denoted by **->**)
  1. If `a` and `b` are two events in the same process, and `a` comes before `b`, then `a->b`.
  2. If `a` is the sending of a message, and `b` is the receipt of that message, then `a->b`
  3. If a->b and `b->c`, then `a->c`

- This introduces a **partial ordering** of events in a system with concurrently operating processes.

Lamport Timestamps

- Each process has its own clock, i.e. `sequence #`
  - Lamport timestamps need a monotonically increasing software counter
- Clock is incremented before each event
- Each message carries a timestamp of the sender’s clock
- When a message arrives:
  - if receiver’s clock ≤ `message_timestamp`
    - set system clock to (`message_timestamp + 1`)
  - set event timestamp to the system's clock
- Partial ordering
  - Lamport timestamps allow to maintain time ordering among related events

Happened-before: Events occurring at three processes

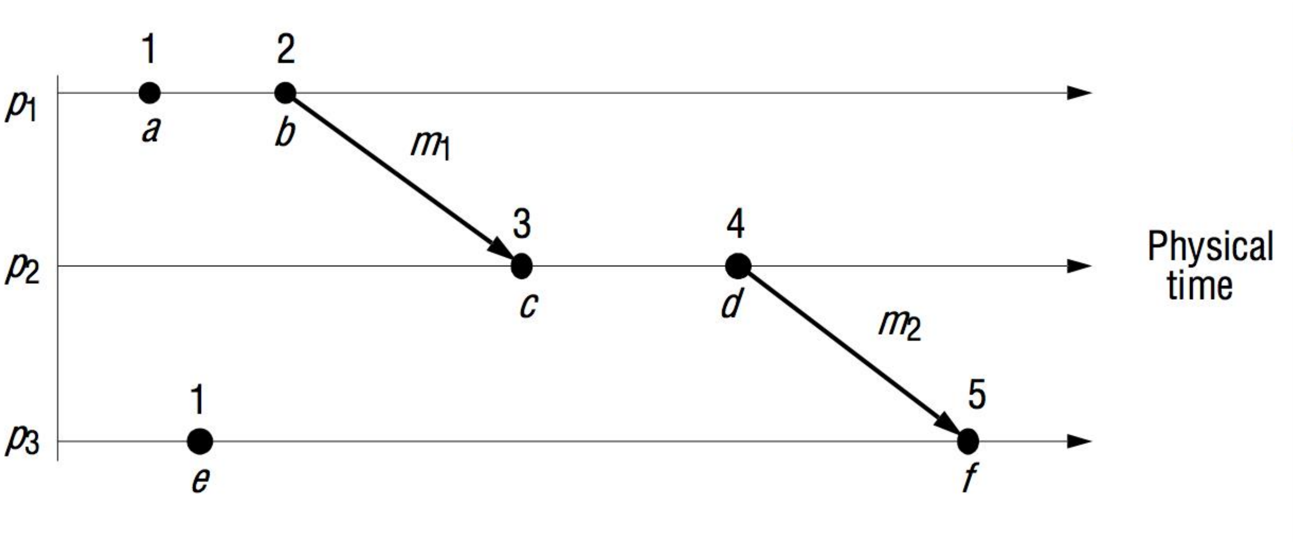


- `a->b`, since the events occur in this order at process `p1 (a->b)`, and similarly `c->d`.
- `b->c`, since these events are the sending and reception of message `m1` , and similarly `d->f`.
- Combining these relations, we may say that, for example, `a->f`.

- Happened-before
  - relation captures a flow of data intervening between two events.

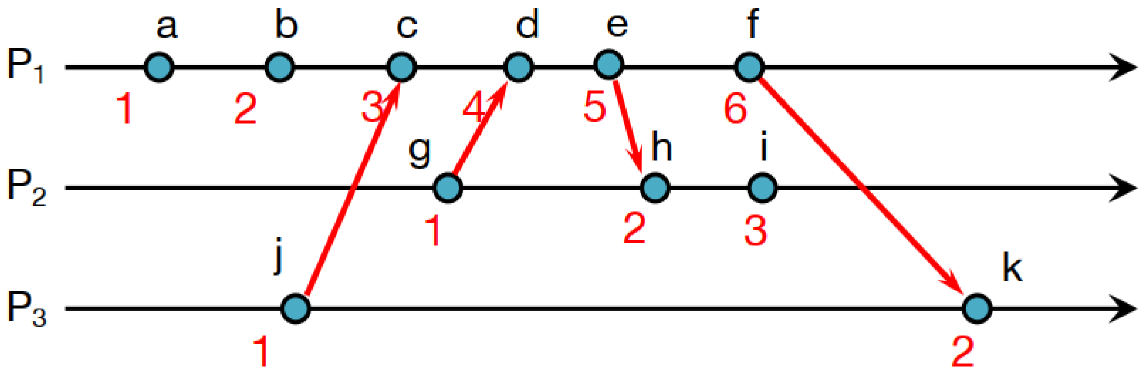
- Not all events are related by the relation, e.g. `a` and `e` , since they occur at different processes, and there is no chain of messages intervening between them. We say that events such as `a` and `e` that are not ordered by `->` , are concurrent and write this as `a||e`.

Lamport timestamps for the events shown in previous figure



- Each of the processes *p*1 , *p*2 and *p*3 has its logical clock initialized to 0.
- The clock values given are those immediately after the event to which they are adjacent.
- Note that, for example, `L(b)>L(e)` but `b||e`.

Event counting example: bad ordering

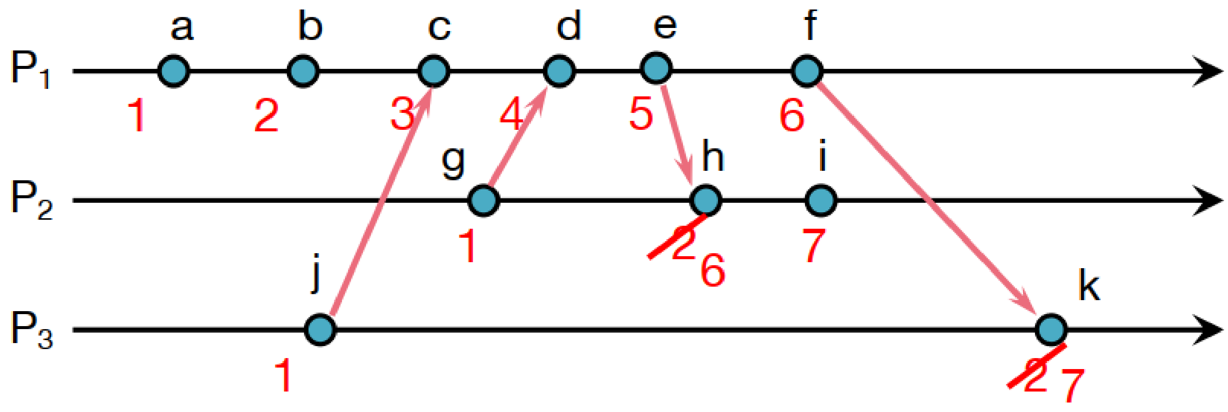


- Bad ordering – for logical clocks

  e->h but 5>=2

  f -> k but 6>=2

Applying Lamport stamps: Event counting example



- Good ordering – for logical clocks

  e->h but 5 < 6

  f -> k but 6 < 7

Lamport's logical clocks

- Problem
  - How do we maintain a global view on the system's behaviour that is consistent with the happened-before relation?

- Attach a `timestamp C(e)` to each event `e`, satisfying the following *properties*:
  - P1: If `a` and `b` are two events in the same process, and `a->b`, then we demand that `C(a) < C(b)`.
  - P2: If `a` corresponds to sending a message `m`, and `b` to the receipt of that message, then also `C(a) < C(b)`.

- Problem
  - How to attach a timestamp to an event when there's no global clock)
    - maintain a consistent set of logical clocks, one per process.

Vector clocks

- Lamport's clocks do not guarantee that if `C(a)< C(b)` that `a` causally preceded `b`.

- Vector clocks
  - Developed to overcome the shortcoming of Lamport's clocks

- A vector clock for a system of N processes is an array of N integers.
  - Each process keeps its own vector clock, V_i , which it uses to timestamp local events.

- Processes piggyback vector timestamps on the messages they send to one another

Vector clocks

- A vector is a logical clock that guarantees that
  - If two operations can be ordered by their logical timestamps, then one must have happened before the other.

- Implemented with an array of counters, one for each process in the system.

Vector Clocks

- A way of identifying which events are causally related
- Guaranteed to get the sequencing correct
- The problem:
  - The size of the vector increases with more actors
    - the entire vector must be stored with the data
  - Comparison takes more time than comparing two numbers
  - If messages are concurrent
    - Application will have to decide how to handle conflicts

Summary

- There are various way to synchronize clocks in a distributed system
- All methods are essentially based on exchanging clock values, while taking
  into account the time it takes to send and receive messages.
- Variations in communication delays and the way those variations are dealt with, largely determine the accuracy of clock synchronization algorithms.
- In many cases, knowing the absolute time is not necessary.
- What counts is that related events at different processes happen in the correct order.

Summary

- The *happened-before* relation is a partial order on events that reflects a flow of information between them
  - Within a process, or via messages between processes.
- Lamport clocks are counters that are updated in accordance with the happened-before relationship between events.
  - Each event `e`, such as sending or receiving a message, is assigned a globally unique logical `timestamp C(e)` such that when event `a` happened before `b`, `C(a) < C(b)`.
- Vector clocks are an improvement/extension on Lamport clocks.
  - If `C(a) < C(b)`, we even know that event `a` *causally* preceded `b`.

References

- Chapter 14: Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 5thEd.
- Chapter 6: Maarten van Steen, Andrew S. TanenbaumDistributed Systems, 3rd edition (2017)
- Understanding Distributed Systems: What every developer should know about large distributed applications by Roberto Vitillo, February 2021
- Paul Krzyzanowski, Logical Clocks, 2021