e-Learning: Vector Clocks

In this e-Learning session, the goal is to deepen the understanding of **Lamport** and **vector clocks**, as seen in the theory part.

Introduction

As we have seen, logical clocks are used to induce a partial ordering of events in a distributed system. The main idea is to use either scalar or vector counters in order to distinguish between causally related events, in case the counters can be compared, and concurrent events otherwise.

In this session, we will use Python's `multiprocessing` package to emulate a concurrent situation by means of process-based parallelism. The API of this module is similar to that of the `multithreading` package, but this time Python spawns different processes at the OS level to achieve parallel execution.

Multiprocessing

In the `multiprocessing` package, the class `Process` represents a single process at the OS level. Communication between processes is implemented either by **queues** or **pipes**.

- A **queue** is used to implement one-way communication: One process writes to the queue and another one reads from it. The corresponding methods are `Queue.put` and `Queue.get`. **Hint:** It is possible to iterate over the elements of a queue by adding a `None` element and iterating over `iter(queue.get, None)`.
- On the other hand, a **pipe** is used to implement a duplex communication channel, where processes can use the pipe both for sending and receiving data.

Exercises

We will focus on the following situation: We have three processes A, B and C.

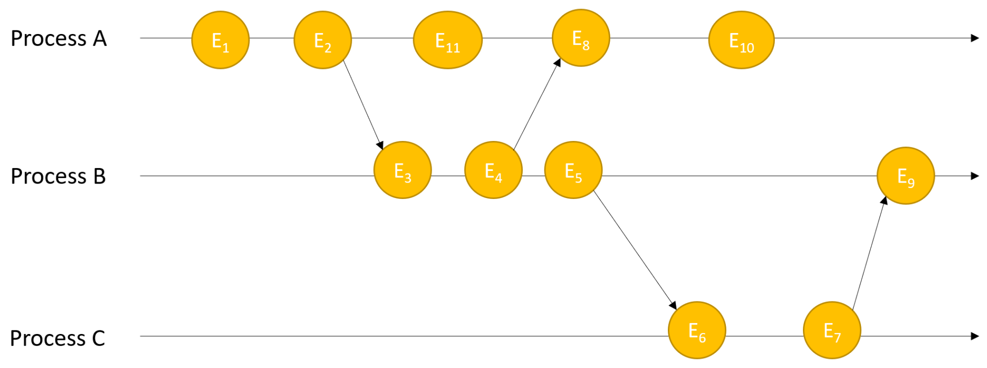

Figure 1: Example process communication.

Now take a look at the `DS_Examples` GitHub repo.

1. Go to the `sync` directory and open the file `clocks_example.py`. This file implements the situation depicted in Figure 1. If you try to run this file, you will get an error, since the program needs the classes `LamportClock` and `VectorClock`, which are not yet defined.

2. Now read the code carefully. The code uses pipes to implement the communication between the three processes. There are two types of events: internal events represent something that happened inside a processes that triggered an increment on its internal clock. Send a receive events use the corresponding pipe to send a message between processes.

3. Your task now is to implement these two classes in the file `clocks_exercise.py`. Both classes inherit from the given class `Clock` that defines four pure virtual methods:

   - `increment_internal`: Increments the value of the clock when an internal event happened.
   - `increment_received`: Increments the value of the clock after receiving a message.
   - `to_string`: Provides a string implementation of the clock (useful for debugging purposes).
   - `get_internal_count`: Returns the current value of the clock.

For instance, for implementing the class `LamportClock` you will need to define a variable to implement the internal counter and implement the aforementioned four methods to deal with it. Same for `VectorClock`, but this time using a vector representation instead.

Finally, **modify** the example so that it performs all pairwise comparisons of events that happened and prints "causally precedes" if $C(a) < C(b)$ or "concurrent" otherwise (**Hint:** each process writes events in a shared queue. After each process terminates, the queue is read by the "master" process)

We will take a look at the solutions in the next session!