1. Introduction

This assignment covers the topic of logical time. A central term within logical time is logical clocks, which consist of logical timestamps that keep track of the numbers of events occured in a distributed system. Logical timestamps are essentially counters that increment every time a process / event takes place in a distributed system, e.g. sending or receiving a message. One has to keep in mind that logical time does not correlate with physical time (seconds, minutes, hours, days, months, years).

There are two different logical clocks being considered in the assignment, namely lamport clocks and vector clocks. The main difference between the clocks is that the processes using a lamport clock only consider one timestamp which is incremented, whereas processes using vector clocks consider vectors (e.g. arrays/lists) of integers which are incremented uniquely for each process.

```
A          B      C
1                 1
2  —> 3
           4 —>   5
```

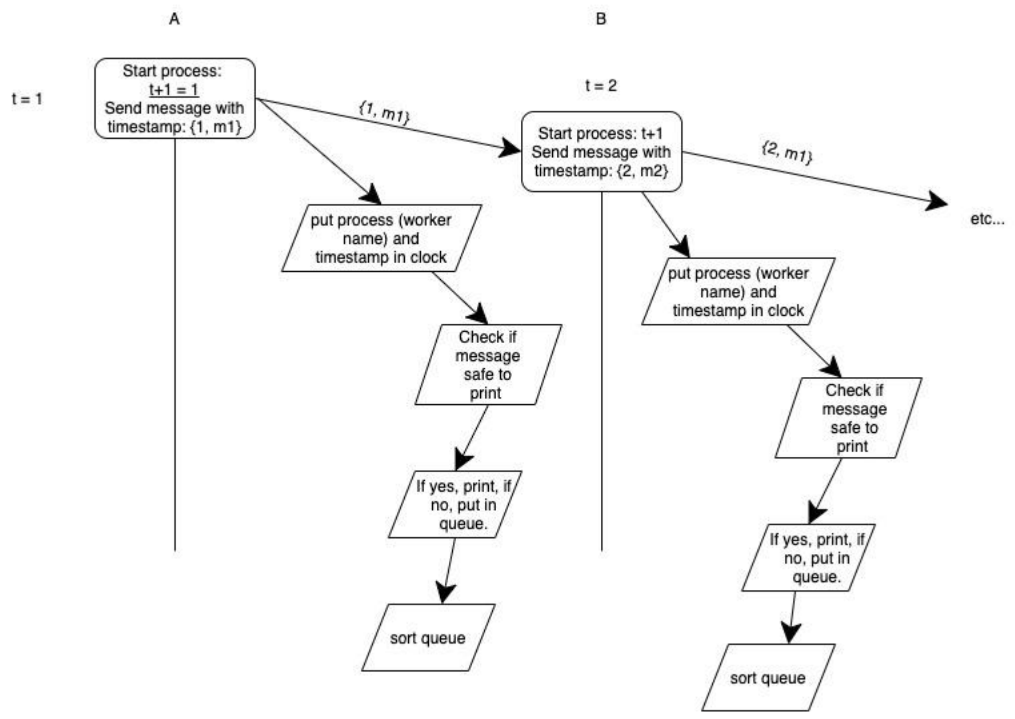
Simple example of a lamport clock.

```
A                                      B                                C
{1,0,0}                                                                 {0,0,1}
{2,0,0}    —> 2({2,0,0},m1) —>         {2,1,0}
{3,0,0}                                {2,2,0}   —> ({2,2,0},m2)—>      {2,2,2}
```

Simple example of a vector clock.

2. Main problems encountered

Like usual, it is challenging to be given many lines of code that represent a quite complex topic and be expected to implement (minor) solutions to it. In other words, although the majority of code is given beforehand, one has to fully interpret the theory and concepts behind the code and also the code itself in order to solve the problem. However, drawing diagrams of the process facilitates the implementation of the code. Below is a first draft of the initial perception of the lamport clock problem and how the holdback-queue and printing should be solved.


A
B
t = 1
Start process: t+1 = 1 Send message with timestamp: {1, m1}
{1, m1}
t = 2
Start process: t+1 Send message with timestamp: {2, m2}
{2, m1}
etc...
put process (worker name) and timestamp in clock
Check if message safe to print
If yes, print, if no, put in queue.
sort queue
put process (worker name) and timestamp in clock
Check if message safe to print
If yes, print, if no, put in queue.
sort queue

2.2 Time module

The time module consists of seven functions:

- zero/0, inc/2, merge/2, leq/2, clock/1, update/3, safe/2

- **Zero/0** is returning a new counter, namely 0.
- **Inc/2** simply increments an entry's counter by one once a new event is identified (for example sending or receiving a message).
- **Merge/2** checks whether an enry's counter value, Ti, is larger than another entry counter value, Tj. This is done in order to get the maximum timestamp value to keep track of current processes.
- **Leq/2** checks whether a timestamp Ti is smaller or equal to another timestamp, Tj. This is necessary to decide whether we can log a message or not, since it is comparing Ti (current message) with Tj, which is held by the clock. We want to log the messages in a chronological order, and we know that a timestamp, e.g. 1, has occurred in a process prior to another timestamp, e.g. 2, and hence is safe to print.
- **Clock/1** initiates a new clock given nodes/process, e.g. outputs [{john, 0}, {paul, 0}] from the input [john, paul].
- **Update/1** updates a clock by replacing a timestamp at index 1, where Node matches the Name (process) in the clock. This is done since we need to update the timestamp in a clock once a message is received.
- **Safe/2** checks whether it is safe to print a message in the console. By sorting the clock on key 2, it means that we are sorting the clock in ascending order with regards to the timestamps. From there, we pop the first entry in the sorted clock, which contains the smallest timestamp value. Afterwards, we check whether Time (argument), which consists of the current message timestamp, is less or equal to the smallest timestamp value in the clock. Returns true or false based on this.

2.3 Vect module

The vect module consists of seven functions:

- zero/1, inc/2, merge/2, leq/2, clock/1, update/3, safe/2

- **Zero/1** is adding 0 to a node (process), making it a tuple (vector clock).
- **Inc/2** finding the node name, e.g. John, increments the timestamp by one and returns it. If not found, we simply add the node to time as a new entry/process.
- **Merge/2** is taking the “Time vector” and merging it with another time vector in order to get the current, updated time vector. This happens upon the message receiving process at an entry, as a new vector is created. If we cannot find the entry in the Time vector, we add it to it.
- **Leq/2** compares the nodes in the hold back queue with the vector clock in order to decide whether it is safe to print the message.
- **Clock/1** creates/initiates a new vector clock, this is done in the logger 3 module.
- **Update/3** checks whether a new event has occurred in the Time vector, which is necessary since a new event means that the timestamp of that process has incremented. If we find it, we replace the new entry containing the new timestamp with the old one in our vector clock. If we don’t find it, we add the new event to the vector clock.
- **Safe/3** checks if it is safe to print the message with help from leq/2.

3. Evaluation / Results

The output of the program shows that each process is printed in a chronological order. This regards both the lamport clock and the vector clock. Demonstrations will be shown in the presentation.

4. Conclusion

Logical clocks are used to capture different event occurrences in a chronological order within a distributed system. The assignment truly facilitated the process of understanding logical clocks. In order to solve the problems, one needs to have a strong theoretical understanding of the concepts. By watching the lecture again and refreshing the memory, it helped a lot. The most tricky part was for sure to connect the dots once the time and vect module was implemented - editing function calls and adding more code in order to get the run.