CEL72, DSL, Monsoon 2020
Lab 2: Implementation of Clock Synchronization (Logical/Physical)
Batch A, Student ID:

Objective

Implementation of Clock Synchronization (logical/physical).

Clock inaccuracies cause serious **problems** and troublesome in **distributed systems**. The **clocks** of different processors need to be **synchronized** to limit errors. This is to have an efficient communication or resource sharing. Hence the **clocks** need to be monitored and adjusted continuously**.**

https://www.ciscopress.com/articles/article.asp?p=2013217&seqNum=2

Logical clocks

To understand why logical clocks are needed, it is important to understand what a distributed system is. A **distributed system** is a system whose components (here called **processes**) are located on different networked computers, which then coordinate their actions by passing **messages** to one other.

One of the main properties of a distributed system is that it **lacks a global clock**. All the processes have their own local clock, but due to clock skew and clock drift they have no direct way to know if their clock is in check with the local clocks of the other processes in the system, this problem is sometimes referred to as the **problem of clock synchronization**.

Solutions to this problem consist of using a central time server (Cristian's Algorithm) or a mechanism called a **logical clock**. The problem with a central time server is that its error depends on the round-trip time of the message from process to time server and back.

Logical clocks are based on capturing chronological and causal relationships of processes and ordering events based on these relationships. The first implementation, the Lamport timestamps, was proposed by **Leslie Lamport** in 1978 and still forms the foundation of almost all logical clocks.

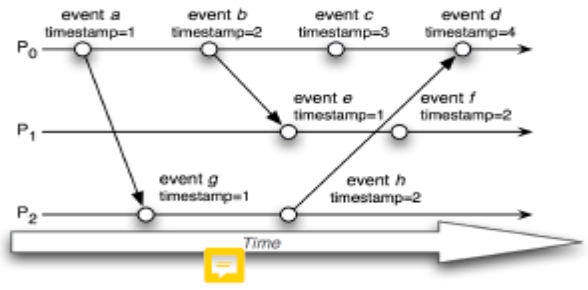

Figure 1

Consider the sequence of events depicted in Figure 1 taking place between three processes.Each event is assigned a timestamp by its respective process. The process simply maintains a global counter that is incremented before each event gets a timestamp.

If we examine the timestamps from our global perspective, we can observe a number of peculiarities. Event *g*, the event representing the receipt of the message sent by event *a*, has the exact same timestamp as event *a* when it clearly had to take place *after* event *a*. Event *e* has an earlier time stamp (1) than the event that sent the message (*b*, with a timestamp of 2).

Lamport's algorithm remedies the situation by forcing a resequencing of timestamps to ensure that the *happens before* relationship is properly depicted forevents related to sending and receiving messages. It works as follows:

- Each process has a clock, which can be a simple counter that is incremented for each event.
- The sending of a message is an event and each message carries with it a timestamp obtained from the current value of the clock at that process (sequence number).
- The arrival of a message at a process is also an event will also receive a timestamp – by the receiving process, of course. The process' clock is incremented prior to timestamping the event, as it would be for any other event. If the clock value is *less* than the timestamp in the received message, the system's clock is adjusted to the (message's timestamp + 1). Otherwise nothing is done. The event is now timestamped.

If we apply this algorithm to the same sequence of messages, we can see that proper message ordering among causally related events is now preserved in following figure. Note that between every two events, the clock must tick at least once.

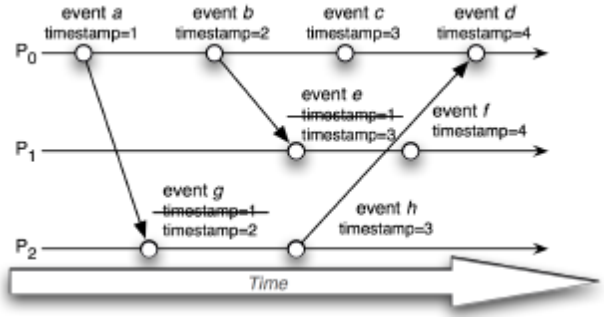

Figure 2

Lamport's algorithm allows us to maintain proper time ordering among causally related events. In summary, Lamport's algorithm requires a monotonically increasing software counter for a "clock" that has to be incremented at least when events that need to be timestamped take place. These events will have the clock value, or "Lamport timestamp," associated with them. For any two events, where $a \rightarrow b$, $L(a) < L(b)$ where $L(x)$ represents the Lamport timestamp for event x.

Lamport timestamps assure us that if there is a causal relationship between two events, then the earlier event will have a smaller time-stamp than the later event. Causality is achieved by successive events on one process or by the sending and receipt of messages on different processes. As defined by the *happened-before* relationship, causality is transitive. For instance, events *a* and *f* are causally related in figure 2(through the sequence *a, b, e, f*).

Implementation:
Lamport's **logical clock** runs on top of some message-passing protocol, adding additional state at each process and additional content to the messages (which is invisible to the underlying protocol). Every process maintains a local logical clock. When a process sends a message or executes an internal step, it sets clock ← clock + 1 and assigns the resulting value as the clock value of the event. If it sends a message, it piggybacks the resulting clock value on the message. When a process receives a message, it sets clock ← max(clock, message timestamp)+1; the resulting clock value is taken as the time of receipt of the message. (To make life easier, we assume messages are received one at a time.)

There are many different implementations of logical clocks that are consistent with Lamport's clock conditions. He gives one:

- Each process Pi maintains a local counter Ci
- IR1:oEach process Pi increments Ci between any two successive events
- IR2:oEach process piggybacks timestamp Tm on a message it sents, where Tm is Ci at the time of sending m
    - If a = send(m) by Pi, then m contains Tm = Ci(a)
    - On receiving m, Pj sets Cj to max(Cj, Tm+1)
    - The receipt of m is a separate event that then separately advances Cj
- Properties of this implementation?
    - Respects causality
        - If a à->b, then C(a) < C(b)
    - But, converse is not true
        - If C(a) < C(b), don't know that a a->b
        - Why? Both cases are possible
            - Could be concurrent
            - Could be causally preceding
- Global ordering
    - Use logical clock to set order
    - If tie, use process IDs as tie breaker
    - i.e., global order is (Logical timestamp) . (process ID)

Conceptual architecture of the distributed system you have considered

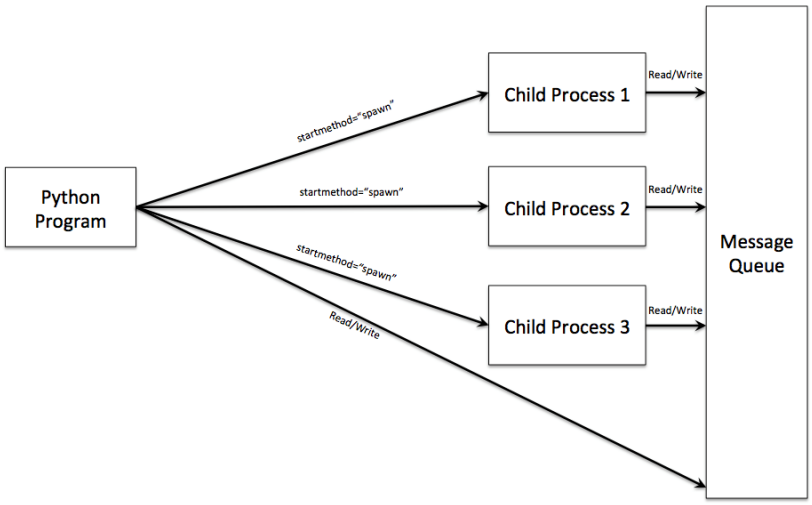

Fig.1 Multiprocessing in python

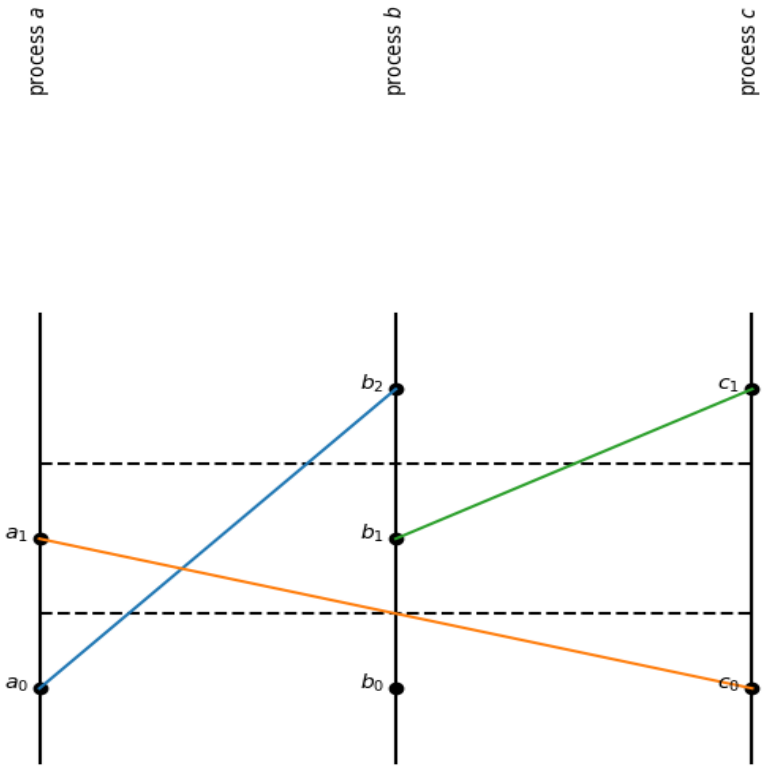

Fig2. process **a**, process **b** and process **c** communicating with each other

System view:

The system consists of three parallel processes using python Multiprocessing library which spawns new process using an API similar to multi-threading API's.

Three processes are able to communicate with each other using named pipes.

The process can:

1. Execute a local process **(event)**
2. Send a message to other process **(send_message)**
3. Receive a sent message **(recv_message)**

A logical clock (Lamport's Clock) is implemented by all the processes and instead of an absolute clock which may lead to inconsistency and chaos in communication and produce erroneous results.

Source files

```
from multiprocessing import Process, Pipe
from os import getpid
from datetime import datetime

def local_time(counter):
    now = datetime.now()
    ftime = now.strftime("%H:%M:%S")
    return 'LAMPORT={}'.format(counter)

def calc_recv_timestamp(recv_time_stamp, counter):
    return max(recv_time_stamp, counter) + 1

def event(pid, counter):
    counter += 1
    print('Local Event occured at ' + str(pid) + '\t' +
local_time(counter))
    return counter


def send_message(pipe, pid, counter):
    counter += 1
```

```
    pipe.send(('Empty shell', counter))
    print('Message sent from ' + str(pid) +'    ' +  '\t' +
local_time(counter))
    return counter


def recv_message(pipe, pid, counter):
    message, timestamp = pipe.recv()
    counter = calc_recv_timestamp(timestamp, counter)
    print('Message received at ' + str(pid)  + '\t' +
local_time(counter))
    return counter


def process_one(pipe12):
    pid = getpid()
    counter = 0
    counter = event(pid, counter)
    counter = send_message(pipe12, pid, counter)
    counter  = event(pid, counter)
    counter = recv_message(pipe12, pid, counter)
    counter  = event(pid, counter)

def process_two(pipe21, pipe23):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe21, pid, counter)
    counter = send_message(pipe21, pid, counter)
    counter = send_message(pipe23, pid, counter)
    counter = recv_message(pipe23, pid, counter)


def process_three(pipe32):
    pid = getpid()
    counter = 0
    counter = recv_message(pipe32, pid, counter)
    counter = send_message(pipe32, pid, counter)


if __name__ == '__main__':
```

```
    oneandtwo, twoandone = Pipe()
    twoandthree, threeandtwo = Pipe()
    process1 = Process(target=process_one,
                       args=(oneandtwo,))
    process2 = Process(target=process_two,
                       args=(twoandone, twoandthree))
    process3 = Process(target=process_three,
                       args=(threeandtwo,))

    process1.start()
    process2.start()
    process3.start()

    process1.join()
    process2.join()
    process3.join()
```

Graph based program

```
import lamport

def a(clock):
    clock.send(1)
    clock.recv(2)

def b(clock):
    clock.local()
    clock.send(2)
    clock.recv(0)

def c(clock):
    clock.send(0)
    clock.recv(1)


lamport.wind([a,b,c],"clock.png")()
```

Show process/event call steps

The system consists of three processes which are implemented as functions and the processes use named pipes for sending messages. The processes can execute a) local b) send_message
c) recv_message

Example:

1. Suppose a process executes a local event Counter resets by incrementing (Counter += 1).

2. Suppose a process sends a message to another process it increments a counter (Counter +=1) and then sends the message with the counter piggybacked on it.

3. Suppose a process receives a message with the counter piggybacked on it then the counter is first checked with the local counter and then maximum of the both is chosen and then incremented. Counter = max(Counter,Local) +1.

Test Procedure for objective validation:

```
(opensource) D:\College Programs and Writeups\4Year\7Sem\DS\DS2\Lamport>python lamportClock.py
Local Event occured at 18124    LAMPORT=1
Message sent from 18124         LAMPORT=2
Local Event occured at 18124    LAMPORT=3
Message received at 26232       LAMPORT=3
Message sent from 26232         LAMPORT=4
Message received at 18124       LAMPORT=5
Message sent from 26232         LAMPORT=5
Message received at 40068       LAMPORT=6
Local Event occured at 18124    LAMPORT=6
Message sent from 40068         LAMPORT=7
Message received at 26232       LAMPORT=8
```

Fig3. Lamport's Clock implementation.

According to the various events and communications between processes the Clock counter is piggybacked on the messages and the process local clocks can then synchronize themselves accordingly.

Conclusion:

Implemented the Lamport's logical clock for synchronization using python's multiprocessing library where multiple processes were interacting with each other through message passing.

Evaluation

Grading is performed for this assignment.

RPC laboratory work grading details Points 5 + Instructor Signature	**Requirements**
5	• Clock Synchronization with given Algorithm(3M) • Documentation (2M)

Feedback	**Yes/No**
• Contents in this write up has been useful to perform experiment? • Level of understanding DS has improved?	