Loggy: a logical time logger

Vasigaran Senthilkumar

September 30, 2021

1 Introduction

In this report the use of logical time in a distributed system is discussed. In a distributed system, it is not possible in practice to synchronize time across processes within the system. Hence, we can use the concept of a logical clock based on the events through which they communicate. Majority of the Distributed algorithms depend on some method of ordering of events to function. So as part of the exercise first Lamport clock was introduced to determine order of events and then a vector clock was introduced.

2 Main problems and solutions

There are two modules. The worker Module and the logger module. In the worker module, a definite Number of worker processes are created and then messages are sent to each other. We also also add a jitter sleep before after a message is sent. Now Each event of sending a message and receiving a message is logged by sending the message and a unique identifier to the logger module. When we first run the test with a jitter of 100. The sent and received messages for events are not in order. But when the jitter is made zero , the received message comes after the sent message. So the logger module need to have a functionality to order messages before logging.

In order to avoid this we introduce logical time to the worker process. The worker keep track of its own counter and pass this along with any message that it sends to other workers. When receiving a message the worker must update its own timer according to Lamport theory, to the greater of its internal clock and the timestamp of the message before increasing its clock by 1. Hence a new 'time' module had to be implemented for achieving lamport clock.

pseudocode for the algorithm is as follows:

```
  %while sending a message
time = time + 1;
send(message, time);
```

```
%while receiving a message
(message, time_stamp) = receive();
time = max(time_stamp, time) + 1;
```

After implementing this, the messages are still not in order but we can identify that in the event of a message being sent , the sent and received message logging order was appropriate.

To get the order of events between the processes right, the message, time and Name was held in a hold back queue until it was safe to print the message. It was difficult to keep track of the hold back queue and print only the messages that are safe. We maintain a clock list, which contain information of the Nodes and the latest message timestamp. So every time a new message was received, the Clock list was updated. Then the hold back queue is ran in a loop looped and was compared with the updated clock and then decided which messages are safe to print. If the messages are not safe to print, they are held back in the hold back queue.

The time module was also supposed to be used as an API. Containing the lamport clock functionality to time module without changing the logger was a challenge.

3 Evaluation

Several evaluation tests where done to understand the functioning and improvise the code. With jitter 100, the messages are not in order. But when the jitter is made zero , the messages are in order. Then logical time was implemented according to the lamport theory and the messages are not in order but received message is logged correctly after the sent message.

Then a hold back queue was implemented then the messages were received in order with the logical time sorted. The test was done with a jitter value of 100. The sent and the received message were also in order. The screenshot of the result is available in figure 1.

Then a vector clock was implemented which is an improvisation of the lamport clock. In a vector clock we replace the clock value from one dimensional integer to a two dimensional vector consisting of Nodes and their counter. When sending a message , the process increments its own logical clock in vector by 1 and then sends the vector. Upon receiving a message , the process increments its own logical clock by 1 in the vector and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message. The screenshot of the result is available in figure 2.

```
[3> test:run(1500,100
[3>
[3> ).
 log: 1 john {sending,{hello,50}}
 log: 1 george {sending,{hello,85}}
 log: 1 paul {sending,{hello,76}}
 log: 1 ringo {received,{hello,76}}
 log: 1 paul {received,{hello,50}}
 log: 2 george {sending,{hello,50}}
 log: 2 ringo {received,{hello,85}}
 log: 3 paul {sending,{hello,68}}
 log: 4 paul {sending,{hello,80}}
 log: 4 ringo {sending,{hello,20}}
 log: 4 john {received,{hello,20}}
 log: 4 ringo {received,{hello,68}}
 log: 5 john {received,{hello,50}}
 log: 6 ringo {sending,{hello,6}}
 log: 7 ringo {sending,{hello,6}}
 log: 7 john {sending,{hello,10}}
 log: 7 george {received,{hello,10}}
 log: 7 john {received,{hello,6}}
 log: 7 paul {received,{hello,6}}
 log: 8 george {received,{hello,80}}
 log: 9 john {sending,{hello,96}}
 log: 9 paul {received,{hello,96}}
 log: 10 george {sending,{hello,89}}
 log: 11 george {sending,{hello,29}}
```

Figure 1: Result of lamport clock with hold back queue

```
[4> test2:run(1500,100).
log: [{george,0},{ringo,0},{paul,1},{john,0}] paul {sending,{hello,91}}
log: [{george,0},{ringo,1},{paul,1},{john,0}] ringo {received,{hello,91}}
log: [{george,1},{ringo,0},{paul,0},{john,0}] george {sending,{hello,60}}
log: [{george,1},{ringo,0},{paul,0},{john,1}] john {received,{hello,60}}
log: [{george,2},{ringo,0},{paul,0},{john,0}] george {sending,{hello,49}}
log: [{george,0},{ringo,2},{paul,1},{john,0}] ringo {sending,{hello,22}}
log: [{george,1},{ringo,2},{paul,1},{john,2}] john {received,{hello,22}}
log: [{george,2},{ringo,3},{paul,1},{john,0}] ringo {received,{hello,49}}
log: [{george,1},{ringo,2},{paul,1},{john,3}] john {sending,{hello,83}}
log: [{george,3},{ringo,2},{paul,1},{john,3}] george {received,{hello,83}}
log: [{george,4},{ringo,2},{paul,1},{john,3}] george {sending,{hello,21}}
log: [{george,4},{ringo,4},{paul,1},{john,3}] ringo {received,{hello,21}}
log: [{george,4},{ringo,5},{paul,1},{john,3}] ringo {sending,{hello,82}}
log: [{george,5},{ringo,5},{paul,1},{john,3}] george {received,{hello,82}}
log: [{george,0},{ringo,0},{paul,2},{john,0}] paul {sending,{hello,50}}
log: [{george,1},{ringo,2},{paul,2},{john,4}] john {received,{hello,50}}
log: [{george,0},{ringo,0},{paul,3},{john,0}] paul {sending,{hello,85}}
log: [{george,6},{ringo,5},{paul,3},{john,3}] george {received,{hello,85}}
log: [{george,7},{ringo,5},{paul,3},{john,3}] george {sending,{hello,30}}
log: [{george,7},{ringo,6},{paul,3},{john,3}] ringo {received,{hello,30}}
log: [{george,8},{ringo,5},{paul,3},{john,3}] george {sending,{hello,81}}
log: [{george,8},{ringo,5},{paul,4},{john,3}] paul {received,{hello,81}}
log: [{george,8},{ringo,5},{paul,5},{john,3}] paul {sending,{hello,70}}
log: [{george,9},{ringo,5},{paul,5},{john,3}] george {received,{hello,70}}
log: [{george,8},{ringo,5},{paul,6},{john,3}] paul {sending,{hello,15}}
log: [{george,8},{ringo,7},{paul,6},{john,3}] ringo {received,{hello,15}}
log: [{george,1},{ringo,2},{paul,2},{john,5}] john {sending,{hello,20}}
log: [{george,10},{ringo,5},{paul,5},{john,5}] george {received,{hello,20}}
log: [{george,1},{ringo,2},{paul,2},{john,6}] john {sending,{hello,80}}
```

Figure 2: Result of vector clock

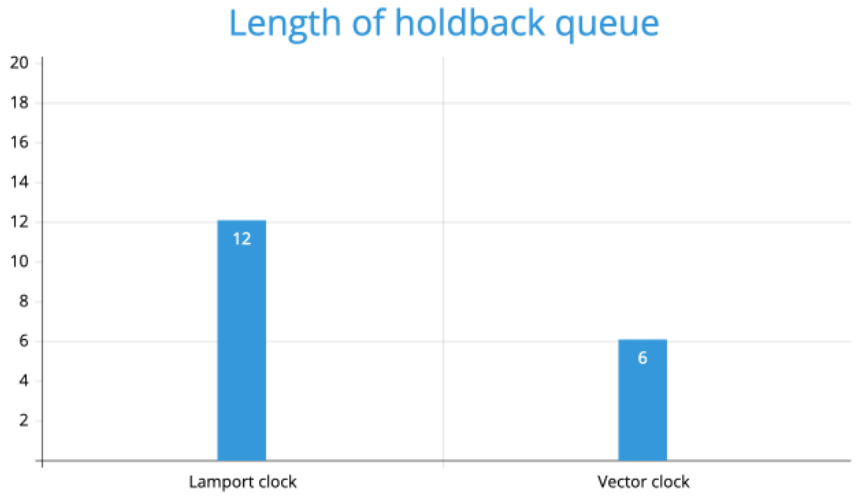

Figure 3: Length of hold back queue

4 Conclusions

After the exercise, the implementation of logical clock using Lamport algorithm was easily understandable. By Improvising to vector clock, I was able to distinguish whether two operations are concurrent or dependent on each other. For the same value of jitter and sleep, the length of the holdback queue for the lamport clock and the vector clock is different. I had experimented with the sleep of 1500ms and jitter 100ms. In case of the lamport clock the the length is 12 and in case of the vector clock the length was 6. This was because the vector clock is now two dimensional , and comparison based on each node instead of a single node in case of lamport theory. Since we are comparing two vectors we are able to identify the safe messages between concurrent events and don't get added to the hold back queue. Thus the hold back queue gets reduced for the same value of jitter and sleep for vector clocks. And the necessity of bringing logical time instead of physical time for distributed systems using a hold back queue was realised through this assignment.