Loggy - a logical time logger

Abdullah Abdullah

September 30, 2021

1 Introduction

The seminar consisted of implementing logical clocks in a practical example. The example was a logging service that receives log events from different set of workers. The assignment consisted of tagging all events with the Lamport timestamp of the specific worker that is sending or receiving the event and logging all the events in correct order known as happened before.

The goal of the assignment was to gain knowledge about the importance of logical clocks in a distributed system and how logical clocks can contribute to a correct ordering of events.

2 Main Problems and Solution

2.1 Problem with the Simple Try

So initially code for a simple logger, worker and test was provided.

The logger is simply a process that receives an event from the worker and prints it. The event is in format of {From, Time, Msg} where From reprsents the worker, Time represents the timestamp but initially for now set to na and Msg represents the msg sent or received by the worker. The implementation of logger prints the msgs from multiple workers.

The worker is a process which waits for a message from another worker or sends a message to another worker after a sleep which can be specified

from terminal by us. The worker sends a log to logger each time when it either sends a message to another worker or receives a message from another worker. There is a small jitter value introduced between a worker sending message to another worker and sending log to logger so the ordering problem can be visualized.

The problem with the implementation was visible when testing was done with test code provided. The order of log messages printed was wrong. The wrong order was identified by looking at the unique identifier of each message. For each message identifier if the receive message was printed before the send message then it was identified as in wrong order and this was the case with many pairs. When passing different values for jitter the printing of wrong orders were reducing and passing jitter value 0 eliminited the wrong order but in real world systems this is not the case and jitter value is always there so the solution to the problem was introducing Lamport Clocks to the implementation.

2.2 Lamport Clocks as Solution

So logical time (Lamport Clock) was introduced to the worker process. The lamport clock states that

- Each worker have a counter.
- Each worker increments its counter when sending a message to other worker.
- When sending message each worker tags the msg with its counter and sends it along.
- Each worker when receives a message with the counter, it will take maximum value of its own counter and the receiving counter and after that increment it by one.

2.2.1 Time Module

After implementing the methods zero, inc, merge, leq and adding the lamport clock to worker processes the order of printing message was still wrong but now each message has its lamport counter/timestamp. The wrong order of messages was identified by seeing lamport stamp value for receiving and sending message and if receive message has occured before the send message

then the order is wrong for that pair. The lamport clock counters now gave us a lead to actually correct the ordering of print. For that the logger needed some changes which are discussed in next section.

2.2.2 The Clock

The first step to resolve the problem was to have a central clock that can track all worker nodes and update the timestamp to latest value for each worker node when receiving a log.

2.2.3 Hold Back Queue

After updating clock when receiving a message th next step was to not directly print the message but to add it in hold back queue of messages. So for that I added a hold back queue in the logger and each time when a logger received message instead of printing I add it to the hold back queue. The format to add message = {From, Time, Msg}. Also after adding message the hold back queue is sorted by time value so that the oldest message is at the start having least counter value.

2.2.4 Log Safe Message, Return UnSafe Queue

The last step was to now check the messages in hold back queue and compare it with the time value received by the logger and if the least value in queue is less then the time value it means that the message in the queue is now safe and we can log it else the messages are still unsafe and return back the unsafe queue to loop.

After implementing these methods the ordering of messages was correct and for each message pair the "happened-before" order was followed in my case. The messages were printed out in correct order like sending before receiving regardless of how the messages arrived at logger. The max hold back queue length depends on the different values of sleep and jitter and it can sometime give no issue with how large the queue becomes.

3 Bonus : Vector Clocks

The limit to Lamport Clocks was that we are not able to know that an event happens-before another event just by comparing their lamport timestamps and we just assume that the events are occuring concurrent.

Vector Clocks were introduced to solve the problem of event happens-before for adjacent events.

In Vector Clocks now each worker have a vector of lamport clocks and each worker can only increment its own clock value in the vector when sending a message. When receiving a message the worker will merge its own vector with incoming vector and take the max value of it and then increment only its own worker clock value in the vector.

Another advantage of Vector Clocks is that now we can now have different set of workers and the logger still works and events occur in happen-before order.

4 Evaluation and Results

Different tests were done for lamport clock logger and vector clock logger. From test results it was concluded that when the jitter value is lower the performance of ordering events is good and events are printed in correct manner for almost all pairs. To compare the length of hold back queue for lamport and vector clocks test results are given below
Some results for Lamport Clock Logger are

```
Test 1
  Sleep = 1000,
  jitter = 100,
  Total Messages sent = 63,
  Max Hold Back Queue Length = 16

Test 2
  Sleep = 500,
  jitter = 50,
  Total Messages sent = 134,
  Max Hold Back Queue Length = 18
```

Some results for Vector Clock Logger are

```
Test 1
  Sleep = 1000,
  jitter = 100,
  Total Messages sent = 63,
  Max Hold Back Queue Length = 5

Test 2
  Sleep = 500,
  jitter = 50,
  Total Messages sent = 134,
  Max Hold Back Queue Length = 5
```

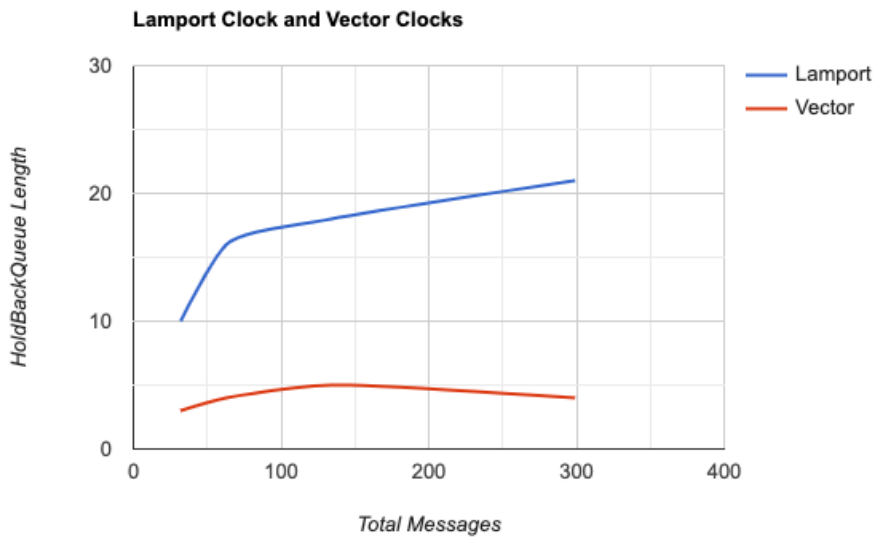

Figure 1: Comparing Lamport and Vector Clocks

From the graph and tests given above we can see that the Max HoldBack-Queue Length for Vector Clock is very less then Lamport Clock because of the reason that Vector Clock achieves the "happens-before" order truly between events. The queue length for Lamport Clocks are greater because there is limitation to lamport clock implementation that we cannot know about an event that it happens-before another event and we consider it as concurrent percesses which results in increasing the size of queue.

Now how vector clocks manages the "happens-before" for different events. It does that when we are logging safe messages. In Vector Clocks method we are now actually checking if all the values in vector of queue message are less then all the values of the clock vector and this ensures that an event has actually happened before another event.

5 Conclusion

From this assignment I got a glimpse of how logical time works in distributed systems and how it can be used to log the correct ordering of events occuring. I learned about the difference between lamport clock and vector clocks and how vector clocks can solve the issue of happens-before between events that was occuring in lamport clock implementation.