Report 3: Loggy

Yanghang Zeng

September 28, 2022

1 Introduction

This assignment is about implementing a logical clock for the events that happened among a set of workers. The main task is to implement a Lamport timestamp of the worker and the events, in order to log the events' information in the correct order. My main job is complete the Lamport timestamp mechanism and then apply it to the logger module. The topic related to distributed systems is the time and clock synchronization among distributed nodes. It is important because we want to know the correct order of happened events, while it's difficult to achieve a perfectly synchronized system. The Lamport timestamp algorithm is used to get the partial ordering of events, and it's also a starting point of the advanced vector clock algorithm.

2 Main problems and solutions

The first problem is to add a counter in each worker so that it knows its own timestamp when an event happens. Then the worker is able to send the timestamp along with messages.

In the time module, there are four functions to achieve this. The zero() is for initializing the clock, it doesn't matter what the initial value is, it could be 0, 1, 2... The inc(Name, T) returns the increment of time T. The merge(Ti,Tj) function is used to take the greater timestamp when a node receives a message. The leq(Ti, Tj) is called by merge(Ti,Tj) to compare two timestamps. In the worker module, each worker maintains an internal clock. When a worker is sending a message, its clock increments by one, and then it sends the new timestamp within the message. When a worker is receiving a message, it merges its internal timestamp with the received timestamp, i.e., it takes the greater value for its new clock. After that, its clock increments by 1 and send the 'receive' event log to the logger. In this way, for every message, the timestamp of receiving will always be larger than the timestamp of sending. Therefore, in the logger, we will be able to identify the partial order of the sending and receiving of the messages.

The second problem is to log the events in the correct order. To achieve this, the logger doesn't print the event immediately when received. Instead, the logger maintains a clock that keeps track of the timestamps of the last messages seen from each worker. The logger also has a queue to hold messages that are not printed yet. Whenever the logger receive a message, it will update its clock to the message timestamp, add the message to the queue and then go through the queue to log the messages that are safe to print. Here, the definition of safe is that, if the timestamp of a message in the queue is smaller than the minimal timestamp in the clock, then it's safe to print. The reason behind this is, if a message has a smaller timestamp than the last seen message from this node, it is most likely that there is no earlier messages than it, so it is safe to print.

The queue is sorted according to the timestamp. The queue will also empty itself when the logger is killed.

3 Evaluation

To verify my implementation, I ran tests under three different cases. All the tests are run with Sleep value of 3000ms and Jitter value of 2000ms, i.e., test:run(3000,2000).

The first case is the original code, without any timestamp mechanism. The result of a random test is shown in Figures 1. From the result, we can see most of the messages are printed in the wrong order. For example, the receiving of {hello,57} is printed before the sending of it. This wrong order also applies to messages 77, 20 and 58. The reason is the system has no knowledge about the time when an event happens.

After solving the first problem mentioned above, a clock is added to each worker. The result of a test is shown in Figures 2. From the result, we can notice that messages are still not in correct order. For example, the receiving of message 57 is logged before the sending log. The order of the message was not improved much. But what is always true is the timestamp of receiving is always larger than the sending of the same message. For example, the receiving of message 57 has timestamp 2 and the sending has timestamp 1, which means the receiving happens before sending. We are also able to identify the order of events for each node. For example, for a specific node, if event A has a larger timestamp than event B, then B must happened before A.

```
(r@17R4-Rick.mshome.net)2> test:run(3000,2000).
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na paul {sending,{hello,68}}
log: na john {received,{hello,20}}
log: na ringo {sending,{hello,77}}
log: na ringo {received,{hello,68}}
log: na ringo {received,{hello,58}}
log: na paul {sending,{hello,20}}
log: na john {sending,{hello,84}}
log: na george {sending,{hello,58}}
log: na george {received,{hello,84}}
log: na george {received,{hello,16}}
stop
```

Figure 1: Result without timestamp

```
Eshell V12.3.2.5
(r@17R4-Rick.mshome.net)1> test:run(3000,2000).
log: 2 ringo {received,{hello,57}}
log: 1 john {sending,{hello,57}}
log: 4 john {received,{hello,77}}
log: 1 paul {sending,{hello,68}}
log: 5 john {received,{hello,20}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 5 ringo {received,{hello,58}}
log: 2 paul {sending,{hello,20}}
log: 6 john {sending,{hello,84}}
log: 1 george {sending,{hello,58}}
log: 7 george {received,{hello,84}}
log: 8 george {received,{hello,16}}
stop
```

Figure 2: Result after adding a clock

After solving the second problem, a clock and a hold-back queue are added to the logger. The result of a test is shown in Figures 3. From the result we can see the correct order of events are logged. For massage 57, the sending happened at time 1 is logged first, then receiving happened at time 2 is logged later than that. For message 68, sending at 1 is logged before receiving at 4. For message 58, sending at 1 is logged before receiving at 5. Therefore, we can tell the order of event presented by the log.

```
Eshell V12.3.2.5
(r@17R4-Rick.mshome.net)1> test:run(3000, 2000).
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 1 george {sending,{hello,58}}
log: 2 ringo {received,{hello,57}}
log: 2 paul {sending,{hello,20}}
log: 3 ringo {sending,{hello,77}}
log: 4 john {received,{hello,77}}
stop
log: 4 ringo {received,{hello,68}}
(r@17R4-Rick.mshome.net)2> log: 5 john {received,{hello,20}}
log: 5 ringo {received,{hello,58}}
log: 6 john {sending,{hello,84}}
log: 7 george {received,{hello,84}}
log: 8 george {received,{hello,16}}
```

Figure 3: Result after full implementation

4 Conclusions

After this assignment, I have a brief understanding of the Lamport logical clock algorithm, as well as its implementation in Erlang. The code implementation works well according to the test verification. The Lamport timestamp is useful in distributed systems. However, the drawback of the Lamport timestamp is that it cannot be used to infer the causality between events or tell the concurrency of events. To improve this, a more advanced version, the vector clock can be used.