Homework 3 Report

Rafat Khan

September 29, 2021

1 Introduction

The purpose of this homework is to learn how to use logical time to determine the order of events in a distributed computer system. In this scenario, A logging procedure receives messages from workers processes where all the messages are tagged with a Lamport timestamp, which is a simple logical clock algorithm. The goal of this experimentation is to display messages on the logger side with ordering.

2 Main problems and solutions

That problem was a little tricky because we had to create the architecture which is able to order the received messages.To create an ordering system, we used a queue that can contain the messages of the workers and wait with the ordering until all messages had been received.

Another problem was the part of dealing with one of the most fundamental concept in distributed system, to manage the order of events – knowing one event happening before the another. To deal with this problem, logical clock is introduced. I have worked with both Lamport and Vector clock algorithms to create ordering between processes.

3 Evaluation

3.1 The first test

If I execute the program given in the assignment document, messages displayed are not ordered. Even some received message are logged before sending message.

```
log: na worker4 {received,{hello! this is a test message,23}} // Wrong Order
log: na worker1 {sending,{hello! this is a test message,23}}
log: na worker4 {received,{hello! this is a test message,35}} // Wrong Order
log: na worker2 {sending,{hello! this is a test message,35}}
```

There are two reasons behind this problem :

1. The logger process doesn't have a procedure to order these messages. When the logger receives a message, it logs it immediately.
2. Because of the delay that we have introduced, the sending message arrived after the receiving message.

3.2 Second Test: Lamport Time

In this test, I have added logical timestamp into the messages sent by the worker process. The timestamp used Lamport's algorithm which uses a simple counter to order the processes by finding the max value between the local counter and the counter of the received message, and increase the value by 1. Each worker has a list of messages which contains the logical timestamp and the contents of the message.

When a message is received, it is saved into the list associated to the worker that sent the message. Then I look inside all the lists of workers. If all the list contain message, message that have the smaller timestamp is displayed and this process keeps iterating, until one of the worker has no messages waiting in the queue.

The console result is shown below:

```
Log: (1) worker4 {sending,{"hello! this is a test message.",83}}
Log: (1) worker1 {sending,{"hello! this is a test message.",22}}
Log: (2) worker1 {sending,{"hello! this is a test message.",78}}
Log: (2) worker4 {sending,{"hello! this is a test message.",70}}
Log: (2) worker3 {received,{"hello! this is a test message.",83}}
Log: (2) worker2 {received,{"hello! this is a test message.",22}}
Log: (3) worker3 {sending,{"hello! this is a test message.",86}}
Log: (3) worker2 {received,{"hello! this is a test message.",70}}
Log: (4) worker2 {received,{"hello! this is a test message.",78}}
Log: (5) worker2 {sending,{"hello! this is a test message.",78}}
```

3.3 Third Test: Vector Clock

In this test, I have added vector clock instead of logical timestamp into the messages sent by the worker process. Vector clock works in a little different manner compared to Lamport clock. Instead of working as a counter, vector clock works as a list. The vector clock is independent from how many nodes we have in the system. For each entry in Time, there is an entry in Clock and if the entry in Time is less than or equal to the entry in the Clock - then it is safe to log the message.

The console result is shown below:

```
Log: worker1, {sending,{"hello! this is a test message.",22}}
Log: worker2, {received,{"hello! this is a test message.",22}}
Log: worker4, {sending,{"hello! this is a test message.",83}}
Log: worker3, {received,{"hello! this is a test message.",83}}
Log: worker4, {sending,{"hello! this is a test message.",70}}
Log: worker2, {received,{"hello! this is a test message.",70}}
Log: worker1, {sending,{"hello! this is a test message.",78}}
Log: worker2, {received,{"hello! this is a test message.",78}}
Log: worker2, {sending,{"hello! this is a test message.",78}}
Log: worker4, {received,{"hello! this is a test message.",78}}
```

4 Conclusions

This excercise let us work with one of the most fundamental concept in distributed system, to manage the order of events. Though concurrency has a lot of advantages, it brings some problems too, compared to a single process system. Logical time is a very good answer to this problem, because it let us know the order of messages across time.