Homework 3 Loggy - a logical time logger

1 Introduction

The task was to implement a logging procedure that receives messages from a set of workers which are tagged with a Lamport timestamp to keep logical order among the messages. Each action of the workers are tagged with a timestamp and the loggers job is to make sure that these messages are be ordered before they are written to stdout.

Time in a distributed system is an important factor and synchronizing clocks is not an easy task. In a distributed system there are components that communicate with each other via message exchange. Therefore it is good to keep track of the order of the messages.

2 Main problems and solutions

The logger module accepts messages and prints them to the screen, it has a clock that keeps track of the timestamps of the last messages seen from each of the workers.

The worker module will wait for messages from a worker or send messages to a worker after a random period of time. When a worker sends or receives a message it sends a log message to the logger. The first version of the worker module will not keep track of the logical time, it will simply send the messages to the logger. All the messages have a unique message.

When running the initial implementation of the worker from a benchmark program called test, the messages arrived in the wrong order:

```
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
log: na john {received,{hello,77}}
log: na ringo {sending,{hello,77}}
log: na ringo {received,{hello,68}}
log: na paul {sending,{hello,68}}
```

When a messages with the same unique identifier are received before they are sent it is printed in the wrong order. This is because the logger prints a message immediately after receiving it. By decreasing the jitter which is the time delay between sending the message to the peer and informing the logger, reduced the wrong order logging off the messages.

The task was then to introduce logical time to the worker process. The module includes a counter that is passed along with every message. The algorithm of the Lamport timestamp is the following:

1. A process increments its counter before each event
2. When a process sends a message it includes its counter value with the message
3. When a process receives a message it takes the maximum value of the received message and its current value, and increments the counter by one.

From the printout we receive the following:

```
log: 2 ringo {received,{hello,57}}
log: 1 john {sending,{hello,57}}
log: 4 john {received,{hello,77}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 1 paul {sending,{hello,68}}
```

The messages still arrive in the wrong order but the logical time provides an order within the workers.

In order for the logger to print the messages in the correct order during execution time the module needed to be modified with some changes. It needs a clock that keeps track of the timestamps of the last messages seen from each of the workers. The logger also needs a hold-back queue where it keeps log messages that are still not safe to print because we don't know if we will receive a message with an earlier timestamp. When a new message arrives it should update the clock, add the message to the hold-back queue and then go through the queue to find messages that are now safe to print:

```
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 2 paul {sending,{hello,20}}
log: 2 ringo {received,{hello,57}}
log: 3 paul {sending,{hello,16}}
log: 3 ringo {sending,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 4 john {received,{hello,77}}
```

The messages are now displayed in the right order.

The optional task was to implement a vector clock to maintain order among the processes. Just as with Lamport timestamps, messages contain the state of the sending process's logical clock. A vector clock of a system of N processes is a vector of N logical clocks, one per process.

- All clocks are initially zero
- When a process sends a message it increments its own clock and sends a copy of its own vector.
- When a process receives a message it increments its own logical clock and takes the maximum of the value in its own vector and the received vector, for every element.

The printout after implementing vector clocks was the following:

```
log: [{john,1}] john {sending,{hello,57}}
log: [{ringo,1},{john,1}] ringo {received,{hello,57}}
log: [{paul,1}] paul {sending,{hello,68}}
log: [{ringo,2},{john,1}] ringo {sending,{hello,77}}
log: [{john,2},{ringo,2}] john {received,{hello,77}}
log: [{ringo,3},{john,1},{paul,1}] ringo {received,{hello,68}}
log: [{john,3},{ringo,2}] john {sending,{hello,90}}
log: [{paul,2},{john,3},{ringo,2}] paul {received,{hello,90}}
log: [{paul,3},{john,3},{ringo,2}] paul {sending,{hello,40}}
log: [{john,4},{ringo,2},{paul,3}] john {received,{hello,40}}
```

The messages arrived in the right order but now with a vector instead of a single timestamp.

3 Evaluation

One drawback when using Lamport time is that if a process has a higher timestamp than another we still cannot be sure if that event occurred before or after the other event. However, it works in our case since our program only does send and receive operations between the workers.

The vector clock solves this problem has all the clock values from other processes and not just the sending process. This way we now know that if one vectors values are smaller than all values of another vector the event will have have happened before the other.

4 Conclusions

This experiment taught me about how processes can incorporate logical time when sending and receiving messages. I also learned about the two algorithms and how they differ from each other.