Report 3: Loggy

Nikita Gureev

September 20, 2016

1 Introduction

In this homework a logging procedure that receives log events froma set of workers. The events are tagged with the Lamport time stamp or vector clock timestamp of the worker and the events are written to output as soon as they are deemed safe.

2 Main problems and solutions

The main problems that were encountered during this task were the implementation of time module for vector clocks, maintaining an ordered queue of messages for vector clocks and determining if a message is safe to print. The first problem stemmed from increased complexity from Lamport time, as all of the operations had to handle vectors of name and time pairs instead of integers. It was solved through extensive modification of worker state and time functions. The second problem was solved through the correct usage of erlang sort with overriden comaprison of two time vectors in accordance to vector clock definition. The last problem was the most difficult, as it included the reasoning out of message safeness. It will be discussed in Evaluation section in detail.

3 Evaluation

The first run of the program provides an output that does not preserve the casual order of messages:

```
test:run(50, 5000).
log: na ringo {received,{hello,57}}
log: na john {sending,{hello,57}}
```

As can be seen, the log about message receiving is printed out before the log of message sending, and that violates the casual order of these two messages. It was caused by the jitter before the logger process received the messages, more specifically, the jitter in "ringo" was less than the jitter in "john",

which caused the wrong ordering.
This problem can be remedied by removing the jitter from the call to test:

```
1> test:run(50, 0).
log: na john {sending,{hello,57}}
log: na ringo {received,{hello,57}}
```

With this call of test function it can be seen that the messages are in correct order, however, the jitter does happen in real life scenarios and a solution should be found to this problem.
The first implementation was of Lamport timestamps, in which all of the worker processes had their own timestamps, which were modified upon sending and receiving of messages. On message sending the counter was incremented and the timestamp sent, on message receiving the local timestamp was compared to received timestamp and the higher of them was incremented. Lamport timestamps are easy in implementing, but there is a problem with them. They provide only one possible ordering of events that has correct casual ordering. It is not possible to derive other possible ordering drom integer timestamps.
A safe message in Lamport timestamps is simple to determine. The logger process maintains a clock that keeps track of the last received timestamps from all of the worker processes. A message is safe to print if it's timestamp is less or equal to the minimal timestamp received from the processes.
The vector clocks are different from Lamport timestamps, as they represent the state of the system from the worker process point of view. Every process keeps track of all the timestamps that it has received from all other processes and attaches this timestamp to all of the messages it sends. The counter increment and merge on receiving is similar to Lamport timestamps. A major difference is that we can determine ordering of any to messages unlike in Lamport timestamps.
In my implementation of vector clocks the logger process keeps track of counters received from all of the processes and updates only the timestamp from the processes that send the message to logger. A message is safe if its vector clock timestamp is less or equal to the timestamp maintained by the logger. A distinct advantage over the Lamport timestamps is that different correct orders of messages can be printed out, as all of the received messages can be comapred to each other. However, it is not used in my system.

4 Conclusions

In conclusion, the concept of logical time is vitally import in distributed systems, as the time in real world nodes can be dependent on many factors. Maintaining the ordering of messages can be extremely important and is used, for instance, in replication.