Report - Homework 3

Emil Karlsson

28 September 2022

1 Testing - No Lamport

Using the test case presented in the PDF, experiments could be made to evaluate the effect the parameters *Sleep* and *Jitter* had on a the logging system.

A quick way of checking whether messages are out of order, albeit not a proof, was to chose a message of type *sending* and look at its random number. Then, if the *receiving* counterpart was above in the list, the message is printed out of order.

The following test cases were made:

Test case 1

Sleep: 1
Jitter: 0

The test case was chosen to test whether out-of-order messages could be found when sending as quick as possible, without jitter.

```
...
log: na paul {received,{hello,85}}
log: na ringo {sending,{hello,16}}
log: na john {sending,{hello,22}}
log: na george {sending,{hello,18}}
log: na george {received,{hello,16}}
log: na george {received,{hello,22}}
log: na paul {sending,{hello,48}}
log: na john {received,{hello,18}}
log: na ringo {received,{hello,48}}
...
```

With no sleep present, messages are sent as quickly as possible. However, as pointed out in the PDF, with not jitter, hardly any out-of-order messages

occur on the same virtual machine. Therefore, despite no sleep, all messages appears to be in order.

Test case 2

Sleep: 1
Jitter: 1

The test case was chosen to test whether a tiny jitter could cause out-of-order messages when sending as quick as possible.

```
...
log: na george {received,{hello,71}}
log: na ringo {sending,{hello,62}}
log: na ringo {received,{hello,81}} <-- receiving first
log: na paul {sending,{hello,24}}
log: na john {sending,{hello,10}}
log: na john {received,{hello,62}}
log: na john {received,{hello,24}}
log: na george {sending,{hello,81}} <-- sending later
log: na george {received,{hello,10}}
...
```

Introducing puny 1 ms as a range for the jitter randomizer, many messages out of order was found. To verify this was actually the case (not just the same random number being sent earlier), more messages earlier was inspected. In the result above, the message sending 81 was out of order since the receiving message was printed before the sending part.

Test case 3

Sleep: 5
Jitter: 30

The test case was chosen as more realistic case where messages aren't sent as fast as possible and a wider range of jitter is possible.

```
...
log: na george {received,{hello,83}} <-- out of order
log: na john {sending,{hello,53}}
log: na ringo {sending,{hello,83}} <-- out of order
log: na ringo {received,{hello,13}} <-- out of order
log: na paul {sending,{hello,19}}
```

```
log: na paul {received,{hello,74}}
log: na george {sending,{hello,13}} <-- out of order
log: na george {received,{hello,32}} <-- out of order
log: na paul {sending,{hello,32}} <-- out of order
...
```

Virtually all the messages were out of order in this test case. Since this test case was run on a single virtual machine, the jitter tried in this test case could have been too large. However, it proves the importance of synchronizing incoming messages.

2 Testing - With Lamport

Using a basic implementation of Lamport time, noticing out-of-order messages was much simpler. Using the same method as earlier, it could be verified by looking at the integer representing the Lamport time; if the earlier message has a higher number, it must be out of order.

```
...
log: 7 paul {received,{hello,7}}
log: 7 john {sending,{hello,7}}
log: 8 george {received,{hello,97}} <-- out of order
log: 6 ringo {sending,{hello,97}}   <-- out of order
log: 9 george {received,{hello,23}}
...
```

3 Testing - With Safe Log

This test case uses the same setup as *Testing - No Lamport : Test Case 3*, where practically all messages where out of order. This was done to verify that all of the previous issues are fixed.

```
...
log: 974 paul {sending,{hello,93}} (Queue size: 15 | Max: 33)
log: 974 john {received,{hello,78}} (Queue size: 15 | Max: 33)
log: 975 john {received,{hello,72}} (Queue size: 13 | Max: 33)
log: 976 john {received,{hello,93}} (Queue size: 13 | Max: 33)
log: 977 john {received,{hello,69}} (Queue size: 13 | Max: 33)
log: 979 john {received,{hello,13}} (Queue size: 13 | Max: 33)
log: 979 john {sending,{hello,48}} (Queue size: 13 | Max: 33)
log: 979 paul {received,{hello,48}} (Queue size: 13 | Max: 33)
log: 980 paul {received,{hello,74}} (Queue size: 13 | Max: 33)
...
```

The result shows that no messages are out of order, and the added information about the hold back queue shows a small number of messages needed to be hold back. Since the *time:safe* was implemented as described in *4.1 Time Module*, the logs are all in perfect order by Lamport time (even though that is not always necessary would the nodes communicate independently).

4 Discussion

4.1 Time Module

The time module was based on an incremental integer that was synced between processes. The messages was essentially tagged with an integer representing their age, i.e. newer messages had a higher number. Implementing increments and merging was therefore straightforward.

Implementing the clock and determining what a safe message is proved to be more challenging. The clock was just a list of node names paired with their own Lamport time. In order to determine whether a message is safe to log, it was essential that the clock was sorted on the Lamport time. This done using *lists:keysort* that compared the integer (Lamport time). Since if the clock is sorted, it is possible to extract the minimum Lamport time that was previously logged. Then, if the current message is larger than the lowest contestant, it must be stored.

For example:

```
[john, 2}, {paul, 4}]
```

Followed by the message:

```
{paul, 3, 'Hello'}
```

This will result in the incoming message being stored, since an older pending message must be printed before.

4.2 Vector Clock

The addition of Vector Clocks instead of basic integer based clocks was that of independent communications. This made the hold back queue smaller and thus a more efficient communication.