Report 3: Loggy

Thomas Galliker

September 21, 2011

1 Introduction

In this exercise we learned how to use logical time in the example of a message logger. A logging procedure receives simple log messages from worker processes. All messages are tagged with a Lamport time stamp. The main task was to order the messages before they are printed out to the console.

2 Main problems and solutions

This is in fact a little tricky since we can't just buffer a certain amount of messages, sort them and put them to the console. We can also not wait for the end of the message sequence since we neither know when nor whether the involved processes will end. This was the main problem in this exercise. It was solved by a list that contains the actual clock values of all worker processes. (See below to get further details about the implementation).

One of the most helpful functions used to solve this exercise was lists:splitwith. This function splits a particular list into two new lists based on a given criteria.

```
lists:splitwith(Pred, List) -> {takewhile(Pred, List), dropwhile(Pred, List)}.
```

Another function that I got to know was lists:keystore. This function helps creating new list entries and updating existing ones. Very useful to implement vector clocks.

```
lists:keystore(Key, N, TupleList1, NewTuple) -> TupleList2
```

3 Tests & Evaluation

- **Chapter 3 - The First Test:** If we run the test cases as given in chapter 3 of the exercise paper, the messages are sorted in a weird order. Right after a few seconds it can be observed that messages arrive before they were sent (what of course is not possible).

```
log: george na {sending,{hello,1}}
log: george na {received,{hello,29}}
log: ringo na {sending,{hello,29}}
log: ringo na {received,{hello,44}}
log: paul na {sending,{hello,44}}
log: paul na {received,{hello,64}}
log: john na {sending,{hello,64}}
```

The reason for this is that the logger does not concern about the casual order of messages. It just processes one after the other. The process scheduler may interrupts the "sending" message for a few clock cycles while the "received" message is delivered immediately.

- **Chapter 4 - The Lamport Time:** In this part of the exercise, a logical time stamping mechanism was added to the worker processes. The time stamp is calculated by the Lamport algorithm which takes the maximum value of the remote and local time and increases it by 1:

  ```
  NewTime = (erlang:max(LocalTime, RemoteTime)+1)
  ```

  If we run the test again, we can see that the logger now shows time stamped messages - but still in the wrong order. By having added the Lamport time stamp to messages we have just ensured intra-process ordering. The logger cannot know about the different clocks that may exist on the associated worker processes.

  ```
  log: george 1 {sending,{hello,1}}
  log: george 2 {received,{hello,29}}
  log: ringo 1 {sending,{hello,29}}
  log: john 2 {received,{hello,1}}
  log: george 4 {received,{hello,32}}
  log: john 3 {sending,{hello,32}}
  log: george 5 {received,{hello,2}}
  ```

- **Chapter 5 - The Tricky Part:** The solution of the previously described situation is pretty tricky. In my first approach I compared time stamps of incoming messages on the logger with the time stamp of the last console print. All incoming messages were queued in a message buffer queue. As soon as a message time stamp has had a time difference greater than 1, messages with the same or lower time stamps were printed to the console. This approach was not 100% accurate. (In other words: Unusable). There were (very few) situations where messages were printed in the wrong order. The more the worker process clocks diverged, the more cases of incorrect orderings have appeared. (The dashed lines illustrate the point of time where the message buffer was printed to the console).

  ```
  log: john 4 {sending,{hello,78}}
  log: ringo 5 {received,{hello,78}}
  log: george 5 {received,{hello,2}}
  --------------------
  log: paul 3 {sending,{hello,6}}
  log: paul 4 {sending,{hello,2}}
  log: john 5 {sending,{hello,9}}
  log: george 6 {received,{hello,2}}
  log: ringo 7 {received,{hello,9}}
  --------------------
  log: john 6 {sending,{hello,9}}
  log: john 7 {sending,{hello,75}}
  log: john 8 {sending,{hello,69}}
  ```

  The correction of this behavior was achieved by allowing the logger to maintain a list of workers and their most actual clocks ({workername, clock}). Whenever a message arrives

we first update the worker clock list and then put the message to the message buffer queue. From the worker clock list we pick the lowest value of all known clocks (Tmin). This value is used to split the sorted message buffer list (MsgQ) into two new lists: One that is used for the print out (PrintQ) and one that is passed back to the logger loop (NewMsgQ):

```
{PrintQ, NewMsgQ} = lists:splitwith(fun({_,T,_})-> T<Tmin end, MsgQ)
```

After having implemented these changes, the test results looked good. (Even after ten thousands of simultaneously sent log messages, everything is still in perfect order).

```
log: ringo 2 {received,{hello,44}}
log: george 2 {received,{hello,29}}
--------------------
log: john 3 {sending,{hello,79}}
log: ringo 3 {sending,{hello,2}}
--------------------
log: ringo 4 {sending,{hello,67}}
log: paul 4 {received,{hello,79}}
```

It is hard to predict the maximum size of the message buffer queue since it mainly depends on the time drift between the involved worker processes. That means, if we have a very fast worker (let's say with 100ms timeout) together with a very slow one (with 10s timeout), the faster will have to wait a relatively long time until the slow one comes up again and updates its clock. During this time the fast process keeps on filling the message buffer queue on the logger. I experienced situations where the buffer size contained up to 400 messages...

4 Conclusions

- First (and most important) conclusion: Concurrency is nice but brings up new problems. Things are less predictable than on single tasking systems. This demands a fundamental rethink of how problems like "keeping log messages in a casual order" can be solved. The point of matter in this exercise was to associate different independently running processes to find a common sense of time. ID2201 lectures as well as many Internet articles helped me a lot in getting a better understanding of how logical time is used in practice.
- Secondly: I've improved my Erlang programming skills to a quiet good level within only three weeks. These exercises start to make fun!
- Last but not least: Latex and me are still not friends.

5 Links

http://www.vs.inf.ethz.ch/publ/papers/logtime.pdf
http://www.erlang.org/course/concurrent_programming.html
http://techtricks.co.in/programming/design-implementation-of-totally-ordered-multicast-using-lamport-logical-clockssynchronization
http://www.cs.fsu.edu/ xyuan/cop5611/lecture5.html
http://www.thomasgalliker.ch