KTH H16P01 Distributed Systems Basic Course: Loggy

Thorsteinn Thorri Sigurdsson (ttsi@kth.se)

September 29, 2016

1 Introduction

This assignment is about logical time. A logger is implemented that is able to receive log messages from a distributed system and log them while preserving happened-before order (causal order of events). Two approaches were taken to achieve this, using Lamport clocks and using vector clocks.

The system that was used during this assignment consists of a logger process along with 4 worker processes; john, paul, ringo and george. The workers send messages between each other randomly. For each message a worker sends it sends a log message to the logger and for each message a worker receives it also sends a log message to the logger. It is then the logger's job to log these messages so that causal order is preserved.

2 Basic Loggy (without logical time)

To begin with a basic logger was tested, with no logical time support. As expected, the logged messages did not respect causal order.

```
log: na john {received,{hello,78}}
log: na ringo {sending,{hello,79}}
log: na paul {sending,{hello,24}}
log: na george {sending,{hello,78}}
```

Each message is tagged with a (hopefully) unique random number, which allows us to figure out which two messages form a send-receive pair. We can see here that george has sent the message `hello, 78` to john, and then sent a log message to the logger. When john receives the message, he also sends a message to the logger. As we can see from the output the log message from john reaches the logger before the log message from george, and therefore the receive event gets logged before the send event, violating the happened-before relationship (causal order).

3 Lamport Loggy

Next, logical time support was added to the system, using Lamport clocks. Using lamport clocks we are able to guarantee that if event `b` causally depends on `a`, then the timestamp for `a` will be smaller than the timestamp of `b`:

$$a \rightarrow b \Rightarrow C(a) < C(b)$$

Each worker has a Lamport clock, which is just a counter. The clock gets incremented when one of the following happens:

1. On every local event that happens within the worker (never happens in our system, we are only sending messages)
2. When the worker sends a message. The clock is incremented and the new clock value is sent along with the message.
3. When the worker receives a message. The local clock is adjusted to max(local clock, received clock from message) + 1.

Every time we receive a message or send a message, we send a log request to the logger with the new value of the local clock. The logger is able to use the time values from the messages it receives to maintain causal order when it prints out the log messages.

```
log: 1 john {sending,{hello,57}}
log: 1 paul {sending,{hello,68}}
log: 1 george {sending,{hello,58}}
log: 2 ringo {received,{hello,57}}
log: 3 ringo {sending,{hello,77}}
log: 4 john {received,{hello,77}}
log: 4 ringo {received,{hello,68}}
log: 5 ringo {received,{hello,58}}
```

We can see that the logical time of each message that is logged is increasing. We can also see that a send log message gets printed before the corresponding receive log message.

However, note that this is just a partial ordering. We are only guaranteed that a send-receive pair is logged in order, i.e. the send message is logged before the receive message. We do not know which of the following messages was actually received by the logger first:

```
log: 4 john {received,{hello,77}}
log: 4 ringo {received,{hello,68}}
```

A message may be stored in the hold-back queue for quite a while before it is safe to log it. So the output from the logger does not show the real order of events as they happened, it only preserves causal order.

3.1 The time module

The time module contains the following methods: zero/0, inc/2, merge/2, leq/2, clock/1, update/3, safe/2.

zero/0 gives us the initial clock value (0), inc/2 increments a clock value, merge/2 merges two clock values (i.e. returns the max of them) and leq/2 returns true if the first argument is less than or equal to the second argument.

The rest of the methods are more interesting.

clock/1 returns a logical clock that is used by the logger. It contains an entry for each of the nodes in the system, and is used to keep track of the latest time stamp that the logger has received from each node.

update/3 updates the clock with a new time stamp from a given node.

safe/2 tells the logger if it is safe to log a message that happened at a given time. We decide that it is safe to log a message if the time stamp of the message is less than or equal to all time stamps that we have received, from all other processes (those that are kept in the logger's clock). The logger does not know where a message was sent, so it can only be certain that it is safe to log a send-message with e.g. time 10 if the latest time stamp from all other processes is also 10 or greater. This means that it is guaranteed to have received the receive-message from the recipient of the message.

Note that the hold-back queue that the logger uses is ordered by time stamps, so when we go through the queue to log safe messages they will be logged in happened-before (causal) order.

4 Vector Loggy

The logger was upgraded to use vector clocks instead of Lamport clocks. Using vector clocks we are able to guarantee that if event `b` causally depends on `a`, then the timestamp for `a` will be smaller than the timestamp of `b`. Furthermore, if the timestamp for `a` is smaller than the timestamp of `b` we can be sure that `b` causally depends on `a`:

$$a \rightarrow b \Leftrightarrow C(a) < C(b)$$

Each worker has a vector clock where it keeps track of the latest logical time it has seen for itself and all other processes. Similar to the Lamport clocks, the worker increments its own entry in the vector on all local events within the worker. When the worker sends a message it increments its own entry in the vector and sends the new vector along with the message. When the worker receives a message, it merges its own vector with the vector in the message it just received by taking a component by component maximum of the vector that it received and its own vector, and then incrementing its own entry in the new vector.

Again, using the vector clocks from the messages it receives, the logger is able to print out the log while preserving happened-before (causal) order.

```
log: [{ringo,5},{john,10},{paul,5},{george,4}] john {sending,{hello,82}}
log: [{ringo,11},{john,9},{paul,9},{george,6}] george {sending,{hello,24}}
log: [{ringo,11},{john,9},{paul,11},{george,6}] paul {received,{hello,24}}
log: [{ringo,11},{john,10},{paul,12},{george,6}] paul {received,{hello,82}}
log: [{ringo,11},{john,10},{paul,13},{george,6}] paul {sending,{hello,40}}
log: [{ringo,11},{john,11},{paul,13},{george,6}] john {received,{hello,40}}
```

4.1 The time module

The time module (actually named "vect" instead of "time" in my implementation) now deals with vector clocks instead of Lamport clocks.

inc/2 now takes a vector of clock values and increments my value in the vector.

merge/2 now takes the component by component max of two vectors.

leq/2 returns true if one vector is less than or equal to the other. A vector is less than or equal to another vector if each component in it is less than or equal to the corresponding component in the other vector.

The clock that the logger maintains is the same as in the Lamport implementation. It holds the latest timestamps received from each of the nodes in the system.

safe/2 tells us if it is safe to log a certain message. It is safe to do so if the vector time stamp in the message is less than or equal to the logger's vector clock. That means that the logger is up-to-date with the state of the system as it was at the time the message was sent.

5 Comparison

To compare the two clock implementations the maximum size of the hold-back-queue was checked. Tests were done with jitter and sleep both set to 500 (test:run(500,500)) and the average of 10 runs for each clock type was taken. As can be seen in table 1, the average max size of the Lamport hold-back-queue was 37,1 and only 6,2 for the vector clock hold-back-queue.

In the Lamport clock case the logger has to wait for the timestamps of all workers to become equal to the timestamp on the message it is trying to log, even though those messages may be completely causally independent from the message it is trying to log.

In the vector clock case the logger can log a message when the time stamp of the message is less than or equal to that of the logger's clock. It is checking the timestamps of all the processes, like in the Lamport clock case, but the difference is that every worker has a more up to date view of the entire system. If worker A sends a message to worker B, and worker B then sends a message to worker C, worker C will know about the message sent from A to B.

When worker C then tries to log a message, the message from A to B will be included in the time stamp of the message that it is trying to log.

The logger uses this time stamp to figure out if it is safe to log the message or not.

The logger can then say that it is safe to log the message if the clock time for A is for example 10, and the clock time for B is for example 15, but the clock time for C (the worker that is trying to log a message) is 999. The logger can then log the message much sooner than if it had to wait for the time stamps of A and B to also reach 999, like it would do in the Lamport clock case.

Lamport	Vector
34	6
35	6
36	6
33	7
33	7
43	6
46	6
34	6
37	6
40	6
Average	
37,1	6,2

Table 1: Maximum size of the hold back queue for both clock types. Jitter and sleep both set to 500

Another benefit of the vector clock implementation is that it does not need to know beforehand how many nodes are in the system. More workers could be started later on and they would be added to the vector clock and the logger could continue to function correctly.