Loggy - a logical time logger

Mattias Cederlund, mcede@kth.se

September 28, 2014

1 Introduction

The task was to implement a logging service that receives events from multiple workers. All events are tagged with a Lamport timestamp and the loggers job is to print the events in the correct order.

The objective of the assignment was to gain knowledge in how to achieve a correct ordering of events in a distributed system.

2 The initial implementation

Initially the logger is a simple process that receives messages from the workers and then print them. This implementation will print the messages in FIFO order from when they are received, but that might not be correct if we have multiple workers.

The worker is a process that waits for a message from another worker or sends a message to another worker after a random time period. When a message is sent, the worker sends a log message to the logger. The worker also sends a log message to the logger when receiving messages. To make the syncronization problem visible the worker also introduces a delay in between sending a message and reporting it to the logger and all messages includes a "unique" message identifier.

When testing the initial implementation of the logger and workers using the provided test it was clear that log messages sometimes were received in the wrong order. The wrongly ordered messages were identified by looking at

the message identifier. If the receive was printed before the send of a message with the same identifier, it was printed in the wrong order. Decreasing the Jitter (That is deciding the size of the random time workers wait between sending a message to another worker and logging it) reduced the number of messages logged in the wrong order, and setting it to 0 basically eliminated them.

3 Lamport time

The main task was to introduce logical time to the worker process. It should include a counter that is passed along with every message. When receiving a message the worker should update its counter to the maximum of its own counter and the message counter and then imcrement the counter by one.

This functionality was added to the worker process implementation and new tests were performed. Of course messages were still printed in the wrong order sometimes but we know that the log from sending a message will always have a lower counter value than values from receiving messages.

Now, the logger needed to be modified to make sure messages are printed in the correct order. In order to do so I needed to work out when it's safe to print a message. The solution I found was to have the logger keep track of all workers and what counter value was last sent by a specific worker in a list of the format [{workername, counter}, ...]. Every time a message is received, the counter is updated and the message is put in a history list. All messages in the history with a lower counter value than the lowest worker are safe to print. Those messages are also sorted in ascending order, to make sure they are in the correct order if many messages are printed at the same time.

When testing the new logger implementation the messages are now ordered by its Lamport timestamps and I can't identify any event that is printed in the wrong order. This implementation does not guarantee that events are printed in the order they actually happned, but that it doesn't violate the happned before order. Any events that are related to each other are presented in the correct order, but we don't know anything about the order of unrelated events. Another weakness of this implementation is that it's vulnerable to starvation. If a worker never send any events to the logger, then the logger wouldn't be able to print anything.

When testing the message history maximum length I managed to get a

maximum of 56 entries, but of course this depends on test settings and what seed is used. The work the logger does to find out what messages are safe to print depends on the message history length. The more messages in the history, the worse performance.

4 Vector time

The vector time implementation is quite similar to the previous, but the workers send a vector instead of a counter. The interesting part is the logger, therefore im not commenting on the worker implementation. The logger keeps track of the counters and only updates the counter for the worker that sent the message. Any messages with a lower or equal vector to the loggers counter are safe to print. It would also need to order the safe messages before printing them, but I didnt implement it. Although with the current worker implementation (and seeds) I did not find any messages printed in the wrong order. One advantage of this implementation compared to the Lamport time implementation is that it's not vulnerable to starvation.

5 Conclusions

I think that implementing Lamport timestamps was a good assignment to extend the understanding of one of the key concepts in this course. Implementing both algorithms made their advantages and disadvantages clear. Lamport timestamps has a simple implementation but is vulnerable to starvation and with a high number of workers the message history might grow quite large with the risk of poor performance. Vector timestamps has a more advanced implementation but is not vulnerable to starvation because it can detect if events are related or not. Therefore the message history also wont grow as large in the vector timestamp implementation.