Distributed Systems - Loggy

Francis Gniady

September 20, 2022

1 Introduction

In this assignment we will learn about the issues that arise when we want to know the orders of events generated by multiple distributed systems working together, specifically in this assignment we will be talking about logs, however this can apply to any other time-ordered series such as events for APIs, etc.
We will then solve these issues using two different times of counting time, lamport time, and vector time. See the differences between these two methods, and where they may be better or less suited.

2 Main problems and solutions

In test:run, setting jitter to 0 pretty much eliminates the issue as there is barely any delay between sending and receiving, meaning the race-condition never occurs, the higher the jitter the more likely the race-condition is to occur where we send the acknowledgement of receiving a message before the sender sends their message.

We cannot always be sure that messages are printed in the wrong order, but it is very unlikely that the random number in each message gets generated twice in a short time span, and we could always raise the random max from 100 to something higher if we encounter issues.

Issues arise when we try and log messages that have a lamport time greater than one of the workers, thus we don't want to log messages until their lamport time is less than or equal to each workers time. Issues could arise if one of the workers dies and stops sending messages, blocking the queue since its timer wont increment, but we will be ignoring this issue in this assignment.

Lamport time is rather simple, and we implement it like the following; whenever we send a message from a worker we first increment our time by one, and attach it to the message, when we receive a message from another worker we take the greater of our time and the attached time, and then increment it by one.

As for the logger, we have a ordered backlog of messages we have received, we also keep track of the time we have received from each worker, when we receive a message we update the clock with the new time from the worker we received the message from.
We then add the message to our ordered backlog sorted by the lamport time, lowest to highest. Then we will iterate over the queue printing any message that is safe according to `time:safe/2`, the rest of the queue is saved and the process is repeated the next time we receive a message.

3 Time Module

3.1 Lamport Time

Lamport time is represented by a single number that is incremented by all workers together.

3.1.1 zero/0

For representing lamport time we simply use a single number. Thus this method simply returns a zero.

3.1.2 inc/2

Since we represent time by a single number all we have to do is return `Time + 1`, ignoring `Name`.

3.1.3 merge/2

We return the greater of the two numbers.

3.1.4 leq/2

We just check if the first number is less than or equal to the second number.

3.1.5 clock/1

Returns a list of tuples with the first element being the node name, and the second element being the nodes lamport time.

3.1.6 update/3

We either find the find the nodes time and merge it with the new time, or if the node does not have an entry then we create one.

3.1.7 safe/2

We return true only if the provided time is `leq/2` to all of the entries times

3.2 Vector Time

Vector time is represented by a list of tuples, each with the name of a worker and its time.

3.2.1 zero/0

A missing entry for a node indicates it has time zero, hence an empty list means all nodes have time zero.

3.2.2 inc/2

Incrementing a nodes clock comes down to finding its entry in the list, and incrementing it by one, or if there is no entry then we insert a new one with time zero + 1.

3.2.3 merge/2

Merging A into B means going over each node in A, if it exists in B then we compare their times and save the larger one, if a node only exists in one list it is added as is.

3.2.4 leq/2

Less than or equal is similar in a way, we go over each node in A and check if its value is less than or equal to in B, if its not we return false, otherwise we check the next node until we have checked all nodes in A. If a node in A does not exist in B then it means it is zero, thus we return false as we do not support negative times.

3.2.5 clock/1

Since we assume that a node not present in the timestamp is automatically zero, we can simply return an empty list. Meaning any possible node is zero.

3.2.6 update/3

To update the clock we find the entry for the node that sent the message in the first timestamp. The node already increments its own time before it sends a message so we do not have to increment it ourselves. We then find the entry for the node in our clock and save the greater one of those two. If we do not find an entry for the node in our clock, we add it as is from the timestamp provided in the message.

3.2.7 safe/2

Checking if a timestamp is safe to print is very easy with all our previous work, we simply have to call the `leq/2` function with the timestamp as the first argument and our clock as the second one. A timestamp will only be safe to print if each entry in it is less than or equal to the entry for said node in our clock.

4 Conclusions

This ended up being a very nice assignment for introduction to the issues of messages being out of sync, and different ways of solving the problem.

Lamport being the simpler solution turned out to have some unforeseen problems, for one, it did not entirely guarantee the order of events in the queue, two events can be fired with the same lamport time as long as the two nodes are not currently communicating. Since they are not communicating, the events are not related to each, even still they could appear out of order when considering real time. Vector time on the other hand is much better for this, and can give us more guarantees of the actual time of when events happened in relation to each other.

As for results in our testing, lamport time worked for solving the issue of `receive` messages arriving before their `sending` counterparts did, however messages would spend much more time in the queue before a message with a greater lamport time arrived forcing a bunch of messages to be printed at once. Often we would see the sending and receiving message not be next to each other in our printout log. With vector time this was much less of a problem, now most of our `receive` messages get printed immediately after their corresponding `sending` message, on top of that messages spend much less time in the queue and are more prone to be printed immediately. Hence the logs flow more naturally instead of pausing for a while and then printing a whole bunch of log messages at once.

All things considered i would say that vector clocks are a much better solution to the problem that we have, especially for logs where we want to know the actual time something happened where vector time can guarantee us more causality between events. However vector clocks also take more space and processing power, each message has to have a list of all processes times instead of a single number, thus I could see a need for lamport time in applications where we might be sending a lot of smaller messages between our systems and aren't as concerned with ordering.