Loggy - a logical time logger

Xin Ren

September 19, 2017

1 Introduction

In this exercise, logical time is practiced and a logging procedure that receives log events from a set of workers is implemented. The events are tagged with the Lamport time stamp of the worker and the events is ordered before written to stdout.

2 Detailed Implementation

Following modules are implemented in this homework.

Time Part	
logger.erl	Receives log events from workers, put the events to a queue and then prints events from the queue in the right order
worker.erl	Starts processors for workers who receive messages and send messages to each other and send log events to the logger
test.erl	Test module
time.erl	This module implements the lamport time for the workers and a clock for the logger. For the workers, when the processor starts, the time is set to 0 with zero/0. The time increases by 1 before a message is sent (inc/2) and is included in the message. When a message is received, the processor changes the time to the greater value of current local time and the timestamp of the message, and then increases the time by 1, this is implemented by inc/2 and merge/2. For the logger, the clock keeps track of the timestamps of the last messages seen from each of the workers. It is represented by a tuple list, with worker name as first element, the timestamp of the last messages seen from the worker as the 2nd element. The initial value of the timestamps are 0, as implemented by clock/1. Every time a log event is received from a worker, the logger find the tuple related to the worker in the clock and replace the timestamp with the timestamp of the log event and then the message is put into a sorted queue by timestamp, and this is implemented by update/3. And then logger goes through the queue to find the log events whose timestamp is smaller than the timestamps of any workers in the clock, and print these events out. This is implemented by safe/2 and leq/2.

Vector Part	
logger.erl	Receives log events from workers, put the events to a queue and then prints events from the queue in the right order
worker.erl	Starts processors for workers who receive messages and send messages to each other and send log events to the logger
test.erl	Test module

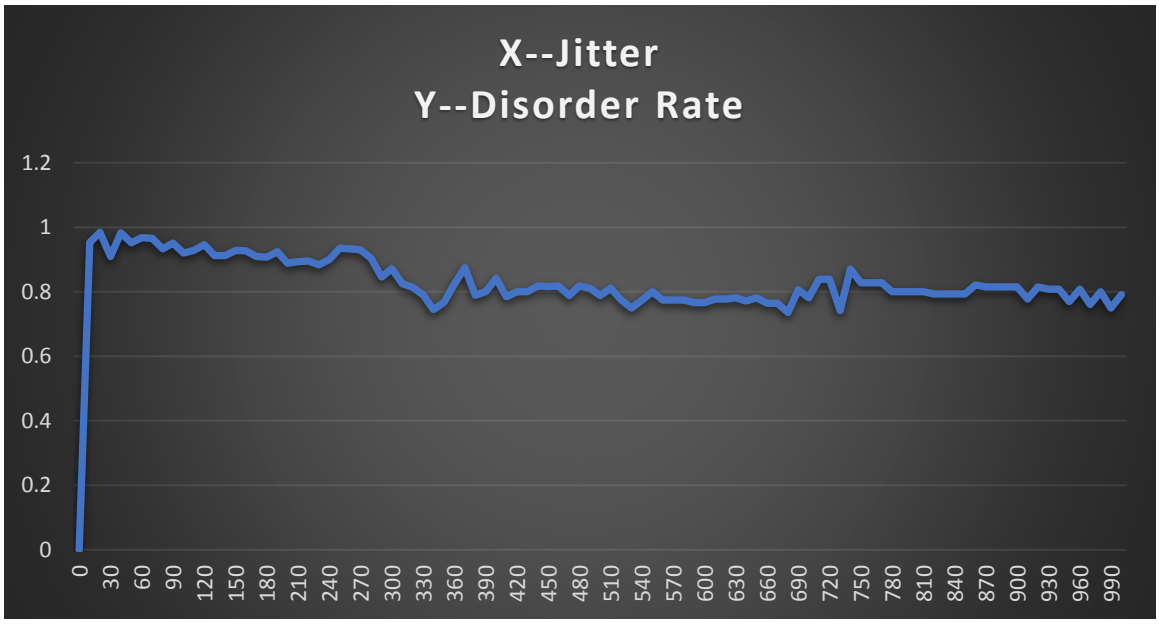
vect.erl	This module implements the vector time for the workers and a vector clock for the logger. Both the time and clock are represented by a tuple list, with a worker's name as the first element of the tuple, and a number as the 2nd element. The initial value of them are empty list as implemented with zero/0 and clock/0. For the workers, the time is updated with the number related to their own name increases by 1 before a message is sent and the time is included in the message. If no tuple found in the time has the worker's name, a new tuple is added to the time with the worker name and number 1 as implemented with inc/2. When a message is received, the worker takes every tuple from the time of the message and compare to the tuple with the same worker name in the local timestamp, update the number to the greater value of the numbers from the two tuples. If no tuple is found with the same worker name, the tuple from the timestamp of the message is added into the local timestamp. After that, the number related to the worker's own name is increased by 1. This is implemented by inc/2 and merge/2. For the logger, every time a log event is received from a worker, the logger find the tuple related to the worker from the message timestamp and replace the tuple with the same worker name in the clock with it, if there is no tuple with the worker name in the clock, the tuple from the message timestamp is added to the clock. Then the message is put into a sorted queue by timestamp, and this is implemented by update/3. After that the logger goes through the queue to find the log events whose timestamp is smaller than the the clock, and print these events out. This is implemented by safe/2 and leq/2. A time is smaller than another time or the clock when any number in the time is smaller than the number related to the same worker time in the other time or the clock. If the tuple with a worker name does not exists in the other time or the clock, it is treated the same as the number is 0.

3 Problems and solutions

1. In my first implementation, to add a log event to the queue in logger, I just appended it to the queue, then I found in the printout, for some messages, the received logs were printed before the sending logs.
   This happens when received event arrives before sending event in the logger and both of them are pending in the queue because another event happens before both of them is not arrived yet. And then the other event arrives in the logger, they are printed in the same order as they are in the queue which is the wrong order.
   To add an event to the front of the queue when it arrives does not work either in the case that sending event arrives before received event and then both of them are pending in the queue for the same reason mentioned above.
   So the solution is to sort the queue in the Time Part by the timestamps and in the Vector Part, to put the new arrived event in the queue after the events whose timestamp is smaller than its timestamp or not comparable.

4 Evaluation

Before time and vector were implemented, sleep and jitter were introduced in the worker to cause random message delay. Give a sleep time, we can tell from following figure, any jitter value can cause similar disorder rate unless jitter is 0 in which case the disorder is also 0.



In the Vector Part, when the logger is stopped, it prints out the max queue size it has had. I ran some tests, and the max size I observed is 10 and the trend is that the bigger the jitter, the bigger the max queue size. The queue occupies system memory so when we are developing a distributed system, we need to consider if it is worth to sacrifice system performance for maintainability for example logging event and try to find a balance.

5 Conclusions

From the work, I have learnt the algorithm of lamport time and vector clock, and used them to implement logger procedure. It helps me understand how different parts in a distributed system sync with each other about the time and how to tell if an event happens before another event.