Report 3: Loggy

Mallu Mallu

September 29,2016

1.Introduction

In this homework I have implemented a logger in erlang which logs the entry for messages sent among some peers in the sorted order of lamport time stamps. Lamport algorithm basically states the happened before relationship so in the log we should not see any entry for a peer to receive a message before it is even sent. This was the key idea behind this homework that log of messages should be in the order of Lamport time stamps on the messages preserving happened before relationship.

2. Main Problems & Solutions

a. Implementing Lamport clock:

For implementing the Lamport clock I have created a “time.erl” file in which I have implemented Lamport time feature through clock, safe, inc, merge, leq functions. In the zero() function it returns the initial time value because in lamport time for occurring event the clock increments by 1 i.e. implemented in inc() function. Also, merge() function merges two time values returning the maximum one of them. Clock function manages time for each node so that it marks the time for the coming node. Update function updates the clock for a node after it has received message from any of the peers. Safe function returns the time entry which is safe to enter in the log. Also, the update, clock functions are used as API in logger to log entries and print the safe ones.

b. Printing values through logger:

After implementing the lamport clock(time.erl) we have to use that in logger to log entries based on the lamport time. In logger the entries in the queue are managed through using the lists keysort function so that every new entry is sorted tin the queue. Also, for updating new clock value for every new entry I have used the update function defined in the time.erl module. After updating clock and queue for the logger we have to choose safe values to get printed and unsafe values to get sorted thus I have recursively called loop function in logger, also for filtering safe values I have defined safe function in time.erl which is being called in the loop function using splitwith function from the lists library so that safe and unsafe values get separated. This was the difficult part I have implemented till now in loggy. Thus the logger prints the safe values in lamport time order.

c. Working with worker:

“Worker” in loggy is used for sending and receiving messages among peers. Choice of peers to send messages is random. Initially the time for every event of send and receive is given as “na” but I have initialized it to zero as well used the inc() function from my time module to increment the counter everytime a new events occur. Also, jitter values are introduced to put some delay in messages sending and receiving. Through the init() function in worker we can start as many peers as we want to.

3. Evaluations:

a) Without Lamport Clock:

For checking that if my logger works I run the test module in which I initiated four peers named: john, paul, ringo, George to communicate with each other and I found that with **jitter value 10 and sleep value 1000 I got 29 log entries** but in unsorted form. Also with increasing jitter value the log entries go down and message passing becomes slow. Also, the results I got are not followed by happened-before relation because I saw some

b) With Lamport Clock:

After implementing the clock, I used to time module in logger to log new and safe entries and then after I ran the test with

test: run (1000,100).

And I got 23 ordered entries logged in logger which are sorted according to the Lamport clock preserving happen-before relationship i.e. I did not see any received messages before sending them. Also as I decreased the values for both sleep and jitter as:

test: run (10,10).

I got 437 sorted entries in the log.

4. Conclusion:

I have learnt how to practically implement lamport clock algorithm preserving happened before relationship. Furthermore, I have learnt how the logger logs the sending and receiving events and what is the key role played by sleep and jitter values in message passing. Also, we can enhance the accuracy by using vector clocks, in which everytime an event occurs the respective process increments its logical clock by value one and when it sends a message to other process it sends the entire vector with the message to be sent and the receiving vector updates its vector by picking up maximum of the vector values from its own vector and received one (for each element).