Report : Loggy, an Erlang implementation of logical time

Avneesh Vyas

avneesh@kth.se

September 28, 2016

1 Introduction

The report describes the various steps and procedures followed during the implementation of a logger process which orders and logs parallel events coming from multiple inter-communicating processes. In distributed systems comprising multiple inter-communicating parallel processes, events and messages are frequently shared among various entities. These messages over network are susceptible to delays. And so it is extremely important for the system to somehow identify delayed or dropped messages and act accordingly. The problem becomes harder because of the lack of a perfectly synchronized system clock. This is where the concept of logical time offers elegant way to overcome the problem. Logical time, unlike absolute time is an abstract notion of time based upon the chronology of events. And thus it imparts capability to order events using 'happened-before' relation among events.

2 Main problems

Four worker processes, each at random time interval sends and receives messages to/from each other. And whenever they send or receive message, they log message through a logger process. As the system is based on erlang, the logger process may receive log messages in random order. For instance, logger may receive a reply message before the corresponding request trigger. As the system mimics a distributed system without any perfectly synchronized clock, the problem is to enhance the logger to print log messages in perfect order.

3 Solution

Lamport algorithm : As the first step, worker processes are enhanced to use Lamport algorithm to maintain an incrementing counter as an internal clock. Each worker process increments its internal clock before

sending a message and then marks each message with its internal clock (logical timestamp). On receive, it updates its internal clock to the following value: max(internal_clock , receive_clock) + 1.

With above enhancements, logger receives messages with logical timestamp. The next challenge is to order messages as per the message timestamp. As the messages are continuously coming, logger puts them into a queue ordered by their logical timestamp. It also stores the latest time stamp value from the messages for each processes. Note that the stored timestamp value for particular worker is updated only if the previous value is smaller than the new value. This is to ensure that logger keeps track of latest logical timestamp seen by each worker process.
As the final step, logger starts with the oldest message (with smallest value of logical timestamp) and compares its timestamp with the minimum timestamp seen across all workers. If the value from the message is smaller than oldest timestamp seen across all workers, then logger prints the message and removes that message from the queue.
With the last comparison, logger ensures that it prints only those messages which are older than messages seen by all processes.

4 Verification

During development of the above solution, the output was verified to spot 'out of sequence' log messages. For example, log messages like these:

log: 1 john {sending,{hello,57}}
log: 2 john {received,{hello,13}}
log: 1 paul {sending,{hello,68}}
log: 1 ringo {sending,{hello,13}}
log: 2 ringo {received,{hello,57}}

Here, John's receiving of the message was logged even before the sending of that message by ringo.

Also, it was verified that all messages get logged. During the development, it was detected that some later messages were not logged. With little debugging it was found that in Time:update function, the clock was wrongly updated every time without ensuring that clock should only get updated if the new Time is greater than the previous clock.