CSCI 53700-FALL 2016

Report for Assignment 4
-simulation of clock consistency in distributed systems using Java RMI

Feng Li

12/02/16

1. Problem Description

In sequential systems, the order of different events can be easily determined if using real, physical clocks. However this problem becomes more complex in distributed systems since its usually hard to maintain a ideally synchronization clocks among all the nodes.

A usual way to define a clock is just assigning a number to an event, where the number is thought of as the time at which the event occurred and it increases throughout all the events in one process object.

Java Remote Method Invocation(Java RMI) is a Java API that performs remote method invocation. A server program creates some remote objects, make references to these objects to be accessible, and clients can invoke methods on these objects. In clock-consistency problem, each Process Object is both server and client.

Section 2 will address the system models and a brief summary of the implementation is introduced in Section 3. In Section 4, there will be an analysis about the effects of event probabilities and other experiment settings.

2. System Models

Interaction Model

Since each process in this distributed system has its own internal clock, we will have different time stamps of events in different processes. There is no 'perfect reference clock ', so the clocks of different processes can vary quite significantly if without any measures taken.

Lamport[1] gives two implementation rules to guarantee that the system of clocks satisfies the Clock Condition.

IR1: Each process P_i increments its logic clock between any two successive events

IR2: (a)If event a is the sending of a message m by process P_i, then the message m contains a time stamp T_m, which is the logic clock value in P_i.(b)Upon receiving a message m, process P_j sets its clock greater than or equal to its present value and greater than T_m.

More specifically, Each process will have a logic clock. For each process, every event will be associated with the current logic clock value as the time stamp.

Events can be classified into three types: internal event, sending event, receiving event.

Each PO will maintain a message queue. When a process i wants to send a message(sending message event of process i) to process j, it just attaches that message in process j’s message queue. Then when process j wants to receive a message, it will get the message in the head of message queue. In this way, the proposed system is asynchronous, since we don’t know how much time the delivering(from sending of process i all the way to receiving of process j) takes.

Failure Model

There will be Byzantine failures during receiving message. In the current settings, the Byzantine behavior can either be ‘do not update the logic clock’ or ‘advance the logic clock by 500’ with different probabilities. For each receiving event, we cannot know for sure whether it will proceed successfully or meet some kinds of failures.

The message queue mentioned in Interaction Model will also result in a channel omission failure. Suppose one process sends messages very frequently, but the target process doesn’t have enough receiving circles to digest all those messages, the message buffer can become full and future coming messages may get lost.

3. Implementation

Java RMI is used in the implementation of the clock correction system. Four Process Object will run in four different machines and all of them will run for a large number of iterations(eg. 10000). In each iteration(after advance the logic clocks), each Process Object will randomly start with a internal/ sending/ receiving event.

In my implementation, for all types of events, the increasing of clocks happens before the event actually happens. This simulates the real situation – if a event has a time stamp of 6, it tells us before this event starts, that Process Object already has already advanced its logic clocks to 6.

For receiving events, the ‘time correction’ only happens when the time stamp in received message is greater or equal to the logic clock in the receiving PO. Once this happens, the receiving PO will increase its logic clock by 1; otherwise, there will be no further change to the logic clocks in receiving PO.

Two types of Byzantine failures are configured in this system, each with probability of 1/50. Note that Byzantine only happens in the process which is conducting a receiving event. The process can either:(a)do not increase the clock or (b) increase the clock by a very large number(500).

A log file records the behavior of all the events and all the logic clock information will be printed into this log file in real time. This can help understanding the iteration model and justify the correctness of the system. Each process will dump its logic clock values into a text file in each iteration and a bash script is used to combine results of all the four processes when they finish all the iterations. The results will show the difference of time clocks after each event in different processes.

Though RMI is a server-client model, it is still practical to use it to implement p2p application like the clock consistency problem. Only one remote invocation need to be defined- ‘try to update remote PO’s timestep’. That is to say, when a PO has a ‘send’ event, it will use its own logic counter as a parameter and the operation of ‘remote call’ is “pushing this logic counter into receiver PO’s message queue”.

4. Results

For the default settings, we can suppose that all the three events(internal, send, receive) happens with the same probability, say 1/3. And the results of system without synchronization(do not advance logic clocks even after receiving a message containing a larger time stamp) is shown in figure 1. From the graph, we can find that time drifts are obvious during nearly the whole lifetime of the 10000 iterations. Then after applying the proposed synchronization method, the result is shown in figure 2, where time drifts become much smaller.

The logic clocks in figure 2 has larger magnitude: when ‘time correction’ happens, the logic clock of receiving process is always increased. That is to say, faster processes are trying to ‘pull’ processes with slower logic clocks when there is a communication.

figure1: even event probability, withtout clock synchronization

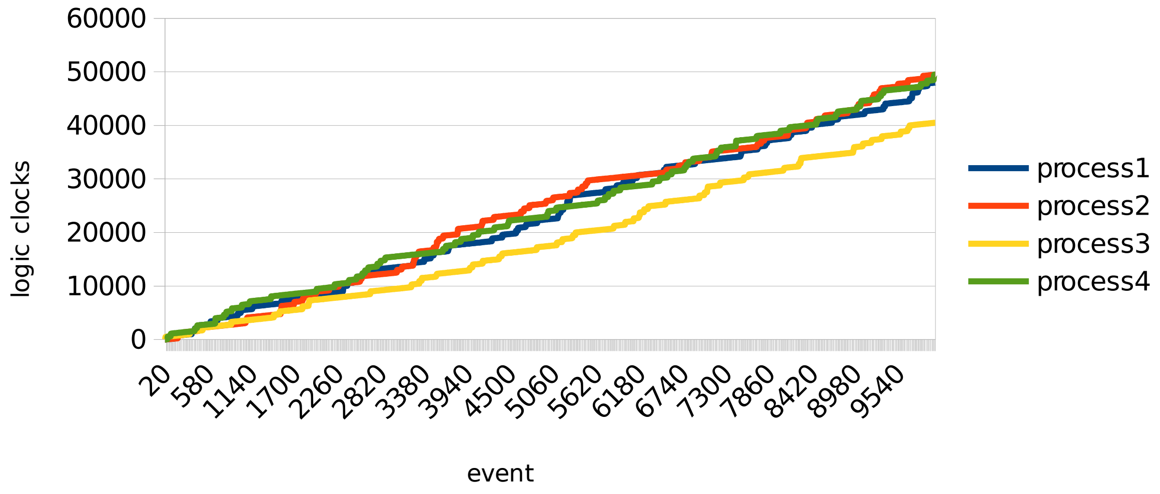
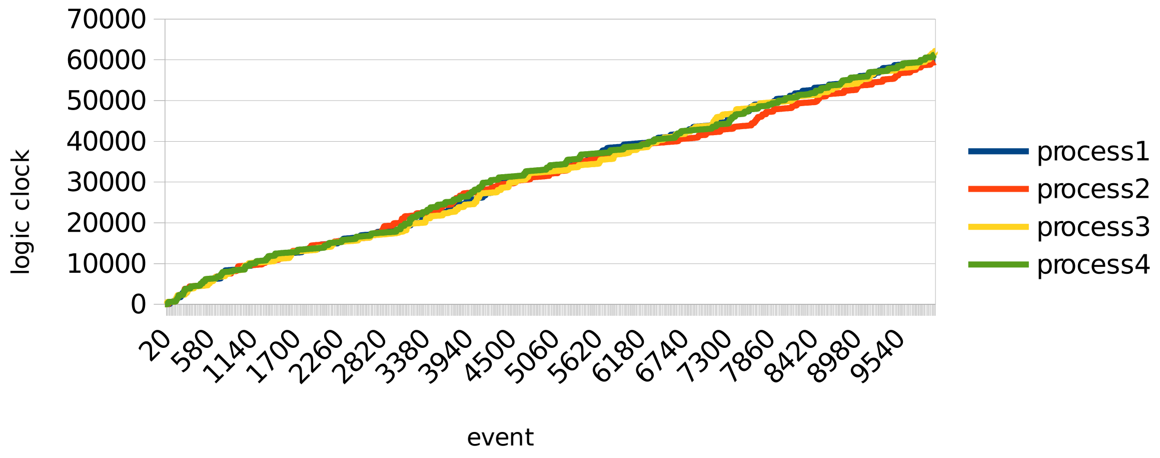
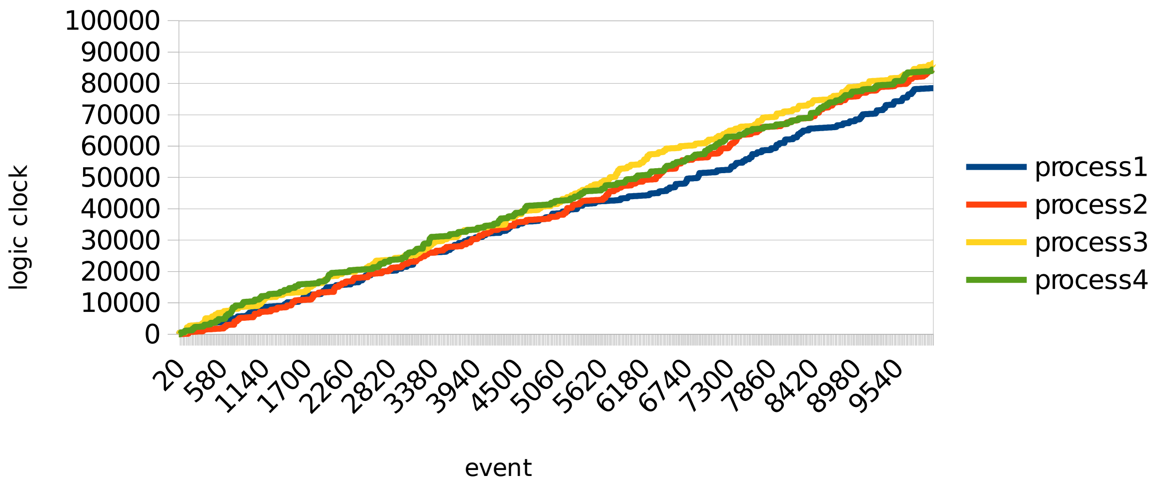


figure2: even event probability, with proposed synchronization



Remind that if receiving message event has much lower probability than sending message event, the messages obtained by receiving process from the message queue can be ‘out of date’(imagine the case when you check your inbox only once a week). The message in the head of the queue is sent long time before, so it has no influences in the receiving process logic clock. That is to say, we have less probability to “correct the time”.

figure3: higher receiving probability, without clock synchronization



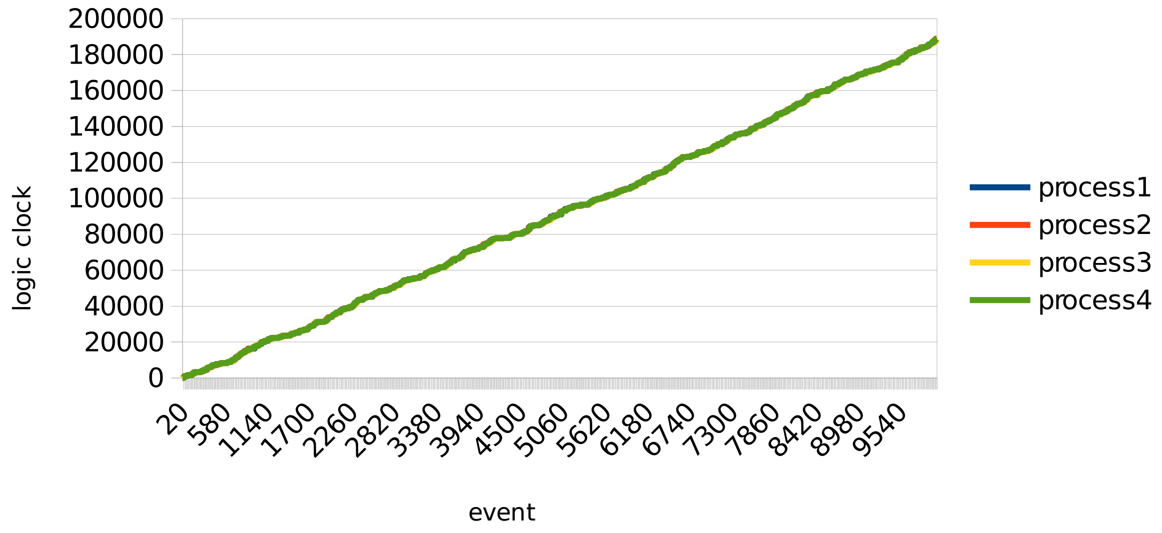
Then a different experiment is done when receiving message has higher probability than sending and internal messages.(receive:send:internal = 4:1:1). This will bring two major changes to the results:

1. time drifts will be smaller, since there will be more chances for time correction

2. The scale of logic lock will increase, since having more receiving events means higher probability of Byzantine error and some of Byzantine errors will advance the clocks significantly.

To keep the consistency, this experiment is also done both with synchronization and without synchronization. The result is shown in figure 3 and figure 4. The time drifting in figure 3 is more obvious than figure 1, since it has higher probability of Byzantine failures. But after applying the proposed synchronization method, the four process nearly have totally identical logic clocks, as shown in figure 4. This also justifies our assumptions.

figure4: higher receiving probability, with proposed synchonization



5. Conclusion

Time in distributed system can be an abstract of the ordering of different events. Although real, physical clocks can be used in sequential systems, it’s not practical to maintain a consistent reference clock for distributed systems. However, ‘happened before’ can be defined using local counters in each process instead. ‘Advance local clock when receiving a message with larger time stamp’ is a measure which can helps synchronize the timing(logic clocks) among the process objects in the system.

Different probability of receiving event may affect the time drifts. For a system without any synchronization method, the drifts become larger since there is higher probability of Byzantine failures. However after applied proposed method, time drifts can be significantly reduced because there is higher chance that a receiving event can correct the logic clock of receiving process.

Java RMI has a typical server-client pattern, but can still be used in other distributed applications as the clock consistency problem in this assignment. In that case, each PO play the role of both client and server.

Bibliography

1: Lamport, L, Time, clocks, and the ordering of events in a distributed system, 1978
2. https://docs.oracle.com/javase/tutorial/rmi/overview.html

3.Using java-rmi in p2p applications
http://isp.vsi.ru/library/Java/JExpSol/ch16.htm#CreatingPeertoPeerRMIApplications.
4. some contents are from the assignment 1(Clock consistency using java threads).