A Performance Evaluation and Critique of Singhal Kshemkalyanis Implementation of Vector Clocks

Manas Hardas
Department of Computer Science
Kent State University
Kent, Ohio 44240
Email: mhardas@cs.kent.edu

***Abstract*—In distributed systems clocks are used for synchronization, ordering etc. Physical clocks which use materials gradually get out of sync over long periods of time called "clock drift". Therefore logical clocks are used instead. Lamports scalar clocks offer a very easy implementation of logical clocks however are not strongly consistent. Vector clocks are strongly consistent, but their implementations have a very big overhead in message transmission. Singhal Kshekalyani offer an implementation of vector clocks where in the message overhead can be decreased by transmitting only the differentials instead of the whole vector. The objective of this work is to test their approach.**

I. Introduction

In distributed systems processes often need to communicate with each other. This is done through message passing. All the actions of sending, receiving and processing a message can be cumulatively labeled as an event on a process. These events occur in some order, which gives a notion of time or chronology of events. The notion of time in state is engraved in interleaving semantics itself. The fact that only one process can execute at a time in distributed system also signifies an order. Thus in distributed systems a notion of time is very important for tasks like synchronization of clocks and ordering of events.

Typically a physical clock for each process seems to be a solution but that method suffers from pitfalls like clock drift. Therefore logical clocks are used. Logical clocks offer a means to capture the causality or happened before relation for two different events. The causality relation is given by $\propto$ symbol and give a partial ordering of events. It is called as partial order because it allow loops in the causality (i.e. c$\propto$d$\propto$e$\propto$c). We will see how logical clocks can be implemented using Lamport's scalar clocks and Vector implementation of Singhal-Kshemkalyani (henceforth referred to as "SK").

II. Logical clocks

Every process in the distributed system is expected to maintian a variable which stores the "logical" time for that process. Using this time other process can synchroize themselves and ordering of events is possible. Ordering of events is important to understand while comparing computations. Two computations are said to be equivalent if they only differ by the order of their concurrent events.

A. Lamports scalar clocks

Lamport suggested a method to maintain logical time by keeping a clock variable and then increment it everytime an event occurs depending upon the event. The clocks are designed such that if 'a' happened before 'b' then 'a's' timestamp will be less than 'b' in case of causally related events. This is called consistencty. Strong consistency is when the reverse, i.e. if the timestamp of a event 'a' is more than event 'b' then 'a' happens after 'b' is also true.

The main problem with scalar clocks is that they do not impose total order on the events. Concurrent events cannot be ordered and therefore strong consistency can be violated. It is not possible to determine from timestamps where one event occured before another, i.e whether the two events were causally related or concurrent.

B. SK's implementation of vector clocks

To remove this inconsistency in the timestamps, vectors clocks are introduced. In this implementaion all the processes maintain a vector of values signifying the clocks of all the processes. Thus now a process knows from its vector what the exact current clock value of another process is by just looking at it own vector. Vector clocks are strongly consistent. However, because vector clock implementations need to exchage the whole vector of information along with any process it wants to communicate, it became an overhead cost. Sending large messages containing clock values of every clock in the system is costly as well as worthless. In the simple implementation of vector clocks, at every state of the computation a process sends a message with maximum size n*n. This is a very costly method to maintain clocks.

In distributed systems tasks are often logically divided into a number of processors with processes working on some common task are grouped together in clusters. Most of the communicaion occurs between these processes. Therefore its is wasteful to send message containing whole n by n vectors of clock values when only a few timestamps are changing. SK though of this and designed an algorithm which only transmits the differential when ever a process sends a new message to another. Two new vectors are maintained at each processes which will store the last updated and the last sent values for clocks along with original vector clock. However, now instead of sending the whole n by n vector, only the

updates were being sent and the remaining processes were updatig accordingly. This saves a lot of overhead in message sizes. SK also give the effeciency of their system in terms of how "localized" the communication is. Meaning, if the communication is more localized i.e. only within a select few processes then SK technique is very efficient. The percent effenciency is given by,

$$\frac{(1-\log_2 N+b).n}{b.N}.100 \quad (1)$$

where, n=number of localized processes; N=total number of processes; b=number of bits in a sequence $\log_2$N=number of bits needed to encode N Therefore, SK technique is only efficient when,

$$n < \frac{N.b}{\log_2 N+b} \quad (2)$$

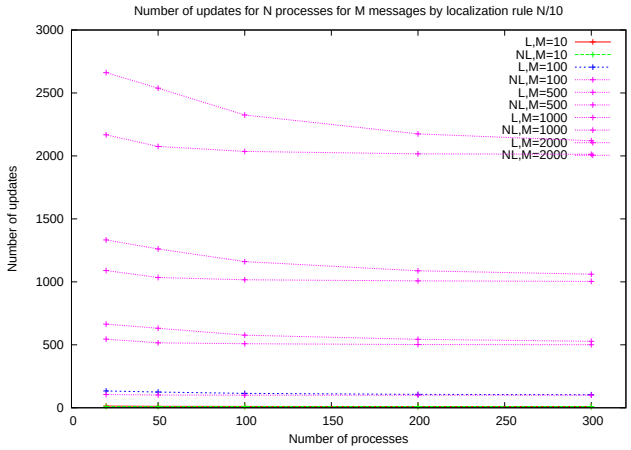

Fig. 1. "Number of updates for N processes for M messages by localization rule N/10"

III. Experiments

In this section we set out to invistigate SK's claim that the message communication overhead is greatly reduced by their method in case of localized communication. The message overhead is in terms of the updates mostly which are dependent on three factors. 1. The size of the computation M, i.e. the number of messages in the computation. The more the size of the computation more will be the total updates generated by a single run of SK's algorithm. 2. The number of processes N in the system. As the number of processes would increase so would the communication and consequently the total number of updates. 3. Thirdly the most important factor which should affect the effeciency is the *localization rule.*

In our experiments, we stongly control the number of processes which are allowed to communicated between each other by a rule called as the localization rule. Generally it is a function of N, so that a subset of localized processes can be generated. We expirement with two localization rules, n=N/10 and n=N/2. In the runs for the algorithm, we measure the average number of updates generated for that run for particular values of M and N. We then vary the values for N and M to observe the change in the behavior of the graph measuring the average number of updates. These measure for both rules is compared with measuements for non-localized communication. By non-localized communication we mean that all the processes are allowed to exachange message with any other process.

The design of the computation is completely random where in the sender and receiver of each message is randomly generate using a pseudo random number generator using a uniform distribution and seeded with the system clock time. We increase the messages in the queue to see what effect it has on the number of updates.

A. Localization rules

According to SK's calculation for effeciency of technique, if N=20; b=4 and $\log_2 N = 5$ then by eq.2 n <20.4/(5+4)≈8. This means that if there are 20 processes in the system and only 8 or less are exchanging messages then their technique will be effective.

Therefore we choose the rules such that we get two values of n by rule 1 and 2. By rule 1 we get n=2 and by rule 2 we get n=10. Thus we know that for rule 1, SK technique should be more efficient than for rule 2.

We also know that, since the maximum number of updates any message can contain is N (in case of non-localized communication), the increase or decrease in the number of updates will be linear in nature.

IV. Observation and Inference

A. Algorithm performance for rule1 n=N/10

In figure 1 we plot the number of updates versus the number of processors in the system. The number of processors involved in localized communication are calculated by the rule n=N/10 and are denoted by L. For non localized communication (NL) any processes is allowed to communicate with any other process. We compare the average number of updates generated for L vs. NL. Each data point in the graph is a result of average of 50 runs. The runs could be varied to get a finer average value. The number of messages in the queue is also kept on increasing with the total number of processes in the system.

From the graph we observe that for every line in the graph, as N goes on increasing the number of updates "U" go on decreasing. The line starts to flatten out in the end indicating less decrease in U for higher values of N. However this phenomenon is strinking in contrast to an intuitive understanding of SK's technique. According to SK until "n" is kept less than a certain amount the algorithm does perform effeciently. However as N increases, so does n because of the rule 1. As n increases, more processes part take in the localized communication theoritically increasing the number of updates. However this is not the observed case. It can be seen from the graph behaviour that for L as well as NL, the number of updates go on decreasing as N increases.

A possible explanation for this phenomenon is that, because the computations are random, as the number of processes N increases, so does the set L, thereby decreasing the probability of same processes being picked for communication even in set L. Say N=100, so we have n=100/10=10 by the localization rule. Thus we have [L]=10 processes taking part in the communication. The maximum number of updates which can be contained in a message in this case is 10. *However this happens only when a single process receives a message from all the rest of the process after their clock values have changed.* To understand this we need to understand what happens at a process for different events,

1) Receive: a process when receives a message, goes through the updates and then if there is any update from the sending process to it only then does it update the clock of the sending process in its own vector
2) Send: a process while sending updates, compares its current vector with the previous vector, if there is a change in the values then adds the updated value along with the process id to an "updates" array. So if its own value has changed, the updates will contain the updated value. Each process tell other process only about its own clock value.
3) Local: a process simply increases its clock count by *d*.

So now we know that, to have 10 updates in the message, a process should first receive updated values from the other 9 processes, then update its own clock value and then send the message out to any other process with the 10 updates. However the probability of this happening decreases as N, and consequently n, goes on increasing! Thus as N increases, the probability of a computation containing states which computation in the desired result goes on decreasing too. The number of updates decrease as a consequence as shown in Fig. 1.

Another observation is that for localized L or non localized NL, as the number of messages increases, so do the updates, keeping the number of processes N constant, which is according to expectation. It is also observed for constant M, there is a greater difference between L and NL. However this gap narrows down as N is increased. This happens because after a certain value of N, the probability of a computation resulting in the increase of average updates is so less that the average of updates generated are more or less the same! We experimented with 300 processes, but it wouldn't be surprising to see the same behaviour for an even larger number of processes.

B. Algorithm performance for rule1 n=N/2

Even for this rule the graph follows similar characteristics as the previous graph. As explained in the previous section, the number of processes in the system, and consequently in localized communication, have no bearing on the number of updates generated, but rather the computation design does. We follow a random approach to computation design and therefore on an average the updates actually go on decreasing becuase of the decreased probability of having a "illustrative" computation with increase in n. In this rule, we select N/2 i.e.

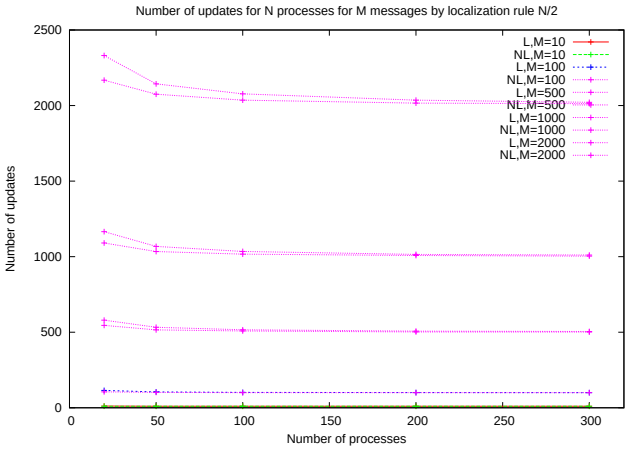

Fig. 2. "Number of updates for N processes for M messages by localization rule N/2"

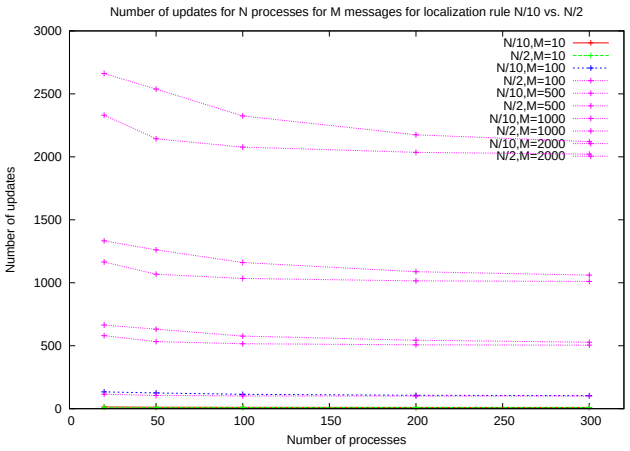

Fig. 3. "Number of updates for N processes for M messages by localization rule N/10 versus N/2"

half of the processes for localized communication. Therefore the average updates generated for this rule start at even lower values than the first rule. Remaining behaviour for this rule is pretty much the same as before.

C. Performance for rule1 vs. rule 2

In this analysis we compare the performance of the two rules tested. Quite expectedly number of updates are more for rule N/10 rather than for rule N/2 because of the above state hypothesis.

From the results presented above it is obvious that SK have a very strict definition of "localized" which is not clearly stated. It seems like even for the localization rules stated above, SK technique is not proved efficient, unless a certain chronology of events is followed, implying a stricter control on the communication. SK technique is definitely better than sending n^2 updates in every single message. However it is not just the localization of processes which works in the techniques favour, there is also an choronology in the events which is needed. This chronology should be defined more

formally.

V. Conclusion and Future work

We tested SK vector clock implementation using a simulated distributed environment with varying number of processes and size of message queues. As a quantitative test parameter we averaged the total number of updates any run of a random computation would result in. Contrary to inuitive expectations we found that the updates went on decreasing as the number of processes and size of queues went on increasing. However we later figured out that becuase the computations were randomly generated, for any localization rule, as N increased so did the *"randomness"* of the computation thus resulting in a computation contrary to one which may have been able to justify SK technique.

To truly test the effeciency of SK implementation of vector clocks, we need to design a very controlled environment, where in the number of process and messages in the computation can be varied however the actual *progression of a computation* should be stricly controlled. Once we know that we can conclusively test SK's hypothesis. As future work I plan to test different types of computations on specific topologies. Star topology seems to be ideal where a single process actively receives updates from other processes, hence providing a test bed for our propositions.

Acknowledgment

I would like to thank Dr.Nesterenko for this wonderful course.

References

[1] Leslie Lamport (1978). "Time, clocks, and the ordering of events in a distributed system". Communications of the ACM 21 (7): 558-565.

[2] Mukesh Singhal, Ajay D. Kshemkalyani: An Efficient Implementation of Vector Clocks. Inf. Process. Lett. 43(1): 47-52 (1992)

[3] "Distributed Computing: Principles, Algorithms and Systems" Kshemkalyani and Singhal.