2011 10th International Symposium on Parallel and Distributed Computing

Performance Evaluation of Incremental Vector Clocks

Sangyoon Lee, Ajay D. Kshemkalyani and Min Shen
Dept. of Computer Science, University of Illinois at Chicago,
Chicago, IL 60607-7053, USA
Email: {slee14, ajay, mshen6}@uic.edu

***Abstract*—The vector clock is an important mechanism to track logical time and causality in distributed systems. Vector clocks incur an overhead of n integers on each message, where n is the number of processes in the system. The incremental vector clock technique attaches only the changed components of the vector clock to a message. This technique to reduce the size of the message overhead is popularly used. We evaluate the performance of the incremental vector clock technique via extensive simulations under a wide range of network loads and communication patterns. Our simulations confirm the intuition that this technique shows marked gains when application processes communicate with locality patterns. In addition, the simulations revealed the following behaviour: (i) the message overhead is not much dependent on the number of processes; (ii) a higher multicast frequency, as opposed to unicasting, lowers the message overhead; and (iii) a slower network speed relative to the inter-message generation time lowers the message overhead.**

***Keywords*-vector clock; incremental vector clock; simulation; performance evaluation; causality**

I. Introduction

A distributed system (N, L) can be viewed as a network consisting of N, a set of processes that communicate by asynchronous message-passing over L, a set of links. An execution of the system generates events at each process, where an event may be a *send* event, a *receive* event, or an *internal* event. The *causality relation* $\longrightarrow$ on the set of events H in a system execution was defined by Lamport [9] as follows: event $e \longrightarrow e'$ if and only if (i) events e and e' occur at the same process and e occurs before e', (ii) event e is the event at which a message is sent and event e' is the event at which that message is received by a different process, or (iii) there exists some event e'' such that $e \longrightarrow e''$ and $e'' \longrightarrow e'$. The set of events H in an execution forms a partial order $(H, \longrightarrow)$.

Logical time and causality are important tools in the design of distributed applications as elegantly expressed by Schwarz and Mattern [17] and by Raynal and Singhal [15]; see also [8]. Each process i maintains a local clock Clk_i to track logical time. Mattern [10] and Fidge [5] independently formalized vector clocks which have the property that event $e \longrightarrow f$ if and only if $Clk(e) < Clk(f)$. Some example areas in distributed systems that use vector clocks are checkpointing, garbage collection, causal memory, maintaining consistency of replicated files, taking efficient snapshots of a system, global time approximation, termination detection, bounded multiwriter construction of shared variables, mutual exclusion, debugging, and defining concurrency measures. These are well documented in the literature. Emerging and recent areas that use vector clocks include building reliable massive-scale ecommerce systems [4], building software transactional memory [16], studying information spread in social communication networks [7], maintaining data consistency in collaborative peer-to-peer editing [12], dynamic race detection in multithreaded programs [6], and designing massive multiplayer online games [21].

A vector clock consists of $n = |N|$ integers, with the $<$ operator on vectors defined as follows: $V_1 \leq V_2$ if and only if, for all $k \in N$, $V_1[k] \leq V_2[k]$; $V_1 < V_2$ if and only if $V_1 \leq V_2$ and $V_1 \neq V_2$. The following rules are used by process i to maintain its clock using the *vector clock protocol*.

VC1. Before process i executes an internal event, it does the following.
$Clk_i[i] = Clk_i[i] + d \quad (d > 0)$

VC2. Before process i executes a send event, it does the following: $Clk_i[i] = Clk_i[i] + d \quad (d > 0)$.
Send message timestamped by Clk_i.

VC3. When process j receives a message with timestamp T from process i, it does the following.
$(k \in N)\ Clk_j[k] = \max(Clk_j[k], T[k])$;
$Clk_j[j] = Clk_j[j] + d \quad (d > 0)$;
deliver the message.

The importance of vector clocks resulted in several approaches to reduce the size of vector clocks and the timestamp information piggybacked on messages. Charron-Bost showed that the minimum size of vector clocks to satisfy the property $e \longrightarrow f$ if and only if $Clk(e) < Clk(f)$ is the dimension of the partial order $(H, \longrightarrow)$, and in the worst-case, the dimension of the partial order is n [2]. Despite this bound, several techniques have been proposed to reduce the size of vector clock overhead on messages [11], [13], [18], [19], [20]. The incremental vector clock technique sends on each message, only those components of the vector clock that have changed since the last message was sent to this destination [18]. Elegant local data structures of size $2n$ integers were proposed to implement this technique.

The incremental vector clock technique can be used in conjunction with vector clocks, and additionally, it can

978-0-7695-4540-0/11 $26.00 © 2011 IEEE
DOI 10.1109/ISPDC.2011.26

IEEE computer society

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

be superimposed on the other techniques to reduce the timestamp information piggybacked on messages. Incremental vector clocks will work best when *group locality* is established. This means that when the entire system can be partitioned into several groups and a large proportion of the messages are intra-group messages, the message overhead reduction will be maximized. Some application domains have this group locality property. In mobile computing, causal ordering using vector clock [1], [14] can benefit from incremental vector clocks. In [1], the authors proposed a centralized framework where group locality is established. File synchronization using vector clocks [3] is another application. Consider the situation for a global banking system where synchronization within local branches can be done frequently, while synchronization between distant branches can only be done periodically. Group locality is also established in this application setting.

However, there has been no performance study of this incremental vector clock technique. This paper gives the results of a comprehensive simulation-based empirical evaluation of the message overhead of the incremental vector clock technique. We chose this approach because of its wide applicability, rather than evaluating the clock overhead for some "standard" applications, or for some popular communication patterns such as those in "client-server" configurations.

Section II outlines the Singhal-Kshemkalyani incremental clock algorithm, henceforth referred to as SK. Section III presents the model of the message passing distributed system in which the SK algorithm is simulated. Section IV shows the simulation results of the SK algorithm. Section V concludes.

II. Overview of the SK Algorithm

In the incremental vector clock technique of Singhal-Kshemkalyani (SK), on each message, only those components of the vector clock that have changed since the last message was sent to this destination are sent [18]. Rather than keeping track of the last vector clock values sent to each destination, resulting in $O(n^2)$ overhead, the SK technique uses only two vectors of size n per process.

- $LU_i[1 \ldots n]$: This is vector "Last Upate" at process P_i. $LU_i[j]$ indicates the value of $Clk_i[i]$ when P_i last updated $Clk_i[j]$.
- $LS_i[1 \ldots n]$: This is vector "Last Sent" at process P_i. $LS_i[j]$ indicates the value of $Clk_i[i]$ when P_i last sent a message to P_j.

Since the last communication from P_i to P_j, only those elements $Clk_i[k]$ have changed for which $LS_i[j] < LU_i[k]$. Hence when P_i sends a message to P_j, it only needs to send those entries $Clk_i[k]$ to P_j for which $LS_i[j] < LU_i[k]$. Thus, when P_i sends a message to P_j, it sends the set of tuples:

$$\{\langle x, Clk_i[x]\rangle \mid LS_i[j] < LU_i[x]\}$$

instead of the vector clock of n elements. This optimization is used whenever the size of the set of tuples is less than n. This can substantially reduce the size of the message overhead because only a fraction of the entries of Clk_i are likely to be modified between two successive transfers to P_j.

We define the following terms:

- c: the number of entries in Clk_i that qualify for transmisison in a message using the SK technique. $c \leq n$.
- t: this is defined as the maximum value possible in any component of the vector clock.

The traditional vector clocks require $n \log t$ bits of information piggybacked whereas the SK technique requires $c(\log t + \log n)$ bits of information piggybacked. The message overhead is formally defined as follows.

$$Msg\,Overhead = \frac{c(\log t + \log n)}{n \log t} \cdot 100\% \qquad (1)$$

It is advantageous to use the SK technique on each message for which the message overhead is less than 100%. This is the case whenever:

$$c < \frac{n \log t}{\log t + \log n} \qquad (2)$$

Although the SK technique has been used extensively, there has been no quantitative study of the savings in message overhead by using the SK technique. This paper fills in this gap by performing a statistical analysis of the message overhead of the SK technique. Some inferences are attempted based on the range of messaging dynamics that are exercised using a parameterized synthetic benchmark.

III. Simulation System Model

A distributed system consists of asynchronous processes running on processors which are typically distributed over a wide area and are connected by a network. It can be assumed without any loss of generality that each processor runs a single process. Each process can access the communication network to communicate with any other process in the system using asynchronous message passing. The communication network is reliable and delivers messages in FIFO order between any pair of processes.

A. Process Model

A process is composed of two subsystems viz., the *application subsystem* and the *communication subsystem*. The application subsystem is responsible for the functionality of the process and the communication subsystem is responsible for providing it with a messaging service. The communication subsystem implements the SK technique in the simulation. The application subsystem generates message patterns that exercise the SK technique. The communication subsystem maintains a floating point clock, that is different from any

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

clock in the vector clock algorithm. This clock is initialized to zero and tracks the elapsed run time of the process. Every process has a priority queue called the *in_queue* that holds incoming messages. This queue is always kept sorted in increasing order of the arrival times of messages in it.

B. Simulation Parameters

The system parameters that are likely to affect the performance of the SK technique are discussed next.

Number of processes (n): It is necessary to simulate the incremental vector clock technique over a wide range of the number of processes to examine scalability. The number of processes in the system is limited only by the memory size and processor speed of the machine running the simulation. On an Intel Dual Core 2 2.6 GHz CPU, 4GB memory, and Linux SUSE 11.1 OS, we simulated up to 100 processes.

Mean inter-message time (MIMT): The mean inter-message time is the average period of time between two message send events at any process. It determines the frequency at which processes generate messages. The inter-message time is modeled as an exponential distribution about this parameter.

Multicast frequency (M/T): The behavior of the SK technique may be sensitive to the number of multicasts. The multicast frequency M/T is the ratio of the number of destinations of multicasts to n. This is the parameter on the basis of which the multicast sensitivity of the SK algorithm can be determined. Processes like distributed database updators have M/T $= 100\%$ and a collection of FTP clients have M/T $= 0$. We simulate the SK technique with M/T varying from 0 to 100%. M/T $= 0$ means unicast and M/T $= 1$ means all messages are broadcast to $n-1$ destinations. The destinations of a multicast are randomly chosen from N using a uniform distribution when $0 < M/T < 1$.

Mean transmission time (MTT): The transmission time of a message here implicitly refers to the *msg. size/bandwidth + propagation delay*. We model this time as an exponential distribution about the mean, MTT. For the purpose of enforcing this mean, multicasts are treated as multiple unicasts and transmission time is independently determined for each unicast. When a process needs to send a message, it determines the transmission time according to the formula $Transmission\_time = -MTT * ln(R)$, where R is a perfect random number in the range [0,1]. This formulation of the transmission time can violate FIFO order. As the incremental vector clock technique assumes FIFO ordering, it is implemented explicitly in our system. Every process maintains an array LM of size n to track the arrival time of the last message sent to each other process. $LM[i]$ is the time at which the last message from the current process to process P_i will reach P_i. Should the transmission time determined be such that the arrival time for the next message at P_i is less than $LM[i]$, then the arrival time is fixed at $(LM[i] + 1)ms$. $LM[i]$ is updated after every message send to P_i. MTT is a measure of the speed of the network, with fast networks having small MTTs. We have varied MTT from $50ms$ to $500ms$ in these simulations so as to model a wide range of networks.

Locality factor for communication (L): In typical distributed applications, processes communicate in clusters over a very significant portion of their execution. Occasionally, they may need to communicate outside their cluster. This behavior makes the incremental vector clock technique very attractive. To study the performance under such locality considerations, we define a (spatial) locality factor on the communication. The processes are partitioned into *zones*, and the locality factor L gives the fraction of message send events at which messages are sent to in-zone destination processes. Typical values of L are between 0.9 and 1.0 for applications that demonstrate significant spatial locality of communication.

An important parameter that is dependent on the above is the ratio of MTT to MIMT. This parameter abstracts away the absolute values of MTT and MIMT. A smaller value of this ratio indicates less traffic; a larger value indicates more traffic. As this parameter is derivable from MTT and MIMT, we do not model it explicitly.

The performance of the SK technique is measured by the *message overhead* metric, defined in Equation (1). In our simulation, we assumed that t was represented by an integer, as the $process\_id$ was also represented as an integer. Thus, we assumed $\log t = \log n$, and thus the message overhead simplified to: $\frac{2c}{n} \cdot 100\%$. It is advantageous to use the SK technique on each message whenever: $c < \frac{n}{2}$.

C. Process Execution

All the processes in the system are symmetric and generate messages according to the same MIMT and M/T. The processes in a distributed system execute concurrently. But simulating each process as an independent process/thread involves inter-process/thread communication and the involved delays are not easy to control. Instead, a global linearization scheme was used to simulate the concurrent processes. The notion of global linearization in the simulation model relies on a logical global *time_stamp*. It maintains the order of execution of all the processes to simulate concurrent execution in a serial fashion. This *time_stamp* is also used to sort received messages in the in_queue of a process to simulate FIFO order.

The simulation populates all the send events for every process at the beginning of the execution according to the MIMT and sorts them by their global *time_stamp*. Each send event is identified by $\langle$ sender id, a set of destination processes, transmission time for each destination $\rangle$. As the simulation extracts a send event from the system message queue, the corresponding event owner process gains control.

When a process gets control, it first invokes the communication subsystem. The communication subsystem looks

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

at the head of its in_queue to determine if there are any messages whose $time\_stamp$ is less than or equal to the current value of the process clock. Such messages are the ones that must have already arrived and hence should have been processed before. All such messages are extracted from the queue and are delivered immediately to the application subsystem. Then, the process handles the send event pulled from the system message queue. The sender updates *time_stamp* of a message by adding the current global *time_stamp* with transmission time of this message, and inserts the message into the in_queue of the destination process. Messages in the in_queue stay sorted by *time_stamp*. The system pops the next send message to progress. The simulation stops when all system-wide send events are processed.

IV. Simulation Results

The framework and the algorithm were implemented in C++ with boost library for random number generation. The performance metrics used are the following.

- The average number of integers sent per message under various combinations of the system parameters, viz., n, MTT, MIMT, M/T and locality L. This metric is expressed as a fraction of n, the size of the vector clock.

For each simulation run, data was collected for 10,000,000 messages and the first 10% and the last 10% system-wide messages were discarded to eliminate the effects of startup and shut-down. Every process P_i in the system accumulates the sum of the number of integers I_i that it sends out on outgoing messages. It also tracks m_i^s, the number of messages sent during its lifetime. Once P_i has sent out $m_i^s = 10,000,000/n$ number of messages, it flags its status as complete and computes its mean message overhead $MMV_i = I_i/m_i^s$. These results are then sent to process P_0 which computes the systemwide average message overhead $\sum MMV_i/n$. All the overheads are reported as a percentage of their corresponding deterministic overhead n of the naive vector clock algorithm without the SK technique.

A. Impact of Locality of Communication

In the first experiment, we try to confirm the intuition that when application processes demonstrate significant spatial locality with communication confined to small groups most of the time, the message overhead of the incremental vector clock technique drops. The intuition is guided by the logic that most of the updates will be coming from processes only within the group, and hence only those updates need to be communicated as the message overhead.

The locality model of simulation is based on the logical partition, *zone*, among processes. The number of processes is fixed, n = 100 for all configurations. Two sets of zoning are tested, one for 10 zones and another for 20 zones. We vary spatial locality L from 0.9 to 1.0 with three configurations of the other parameters, (MTT, MIMT, M/T). The results are shown in Figure 1. As locality L increases, all configurations exhibit rapid decrease of message overhead. When we use 10 zones, result graph illustrates a drastic decrease rate after L = 0.96. For 20 zones, the result graph shows close to linear decrease over L. As we expected, 100% in-zone communication pattern (L =1.0) shows very low overhead. For the 10 zone case, it is 10% overhead, and for 20 zones, it is 5% overhead.

B. Scalability with Increasing n

The second experiment on the optimized vector clock is to vary the number of processes in the system (n = 10 to 100). A total 10 configurations of the parameters (MTT, MIMT, M/T) were run in this experiment. We assumed a default L = 0.0 and a single zone. The simulation results are presented in Figure 2. In most of the configurations, the message overhead is stable with slight increase as n increases.

First consider the unicast cases, where M/T = 0. Let us focus on a specific process P_i. As n increases by a factor of δ, the time duration between two sends by P_i to a fixed other process P_j increases by a factor of δ. Thus, $LS_i[j]$ is that much older, and this potentially allows more clock components k with $LU_i[k] > LS_i[j]$ to be sent as message overhead by P_i to P_j. Note however, that within this increased time duration, the probability of P_i receiving a message from a fixed other process P_k remains the same because P_k is also now unicasting to a larger number of processes. Hence, P_i is likely to receive the same proportion of updated clock components in the increased time duration δ.

In the multicast case, M/T > 0. As n increases by a factor of δ, the time duration between two sends by P_i to a fixed other process P_j remains the same. Thus, $LS_i[j]$ is not likely to change. In this unchanged time duration, the probability of P_i receiving a message from a fixed other process P_k remains the same because P_k is also now multicasting to a larger number of processes. Hence, P_i is likely to receive the same proportion of updated clock components in the same time duration.

This explains why the message overhead does not change much as a function of n.

C. Impact of Increasing Multicast Frequency

As we discussed in Section III-B, M/T is used to determine the number of destinations at each send event. It applies to every send event evenly in our simulation model. A total of 6 configurations of the parameters (MTT, MIMT, n) were tested. We assumed a default L = 0.0 and a single zone. Figure 3 shows the results of the experiments. In general, all configurations show a decrease of message overhead over increasing M/T from 0.0 to 1.0. Note that M/T = 0.0 means that all send events send a message to one

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

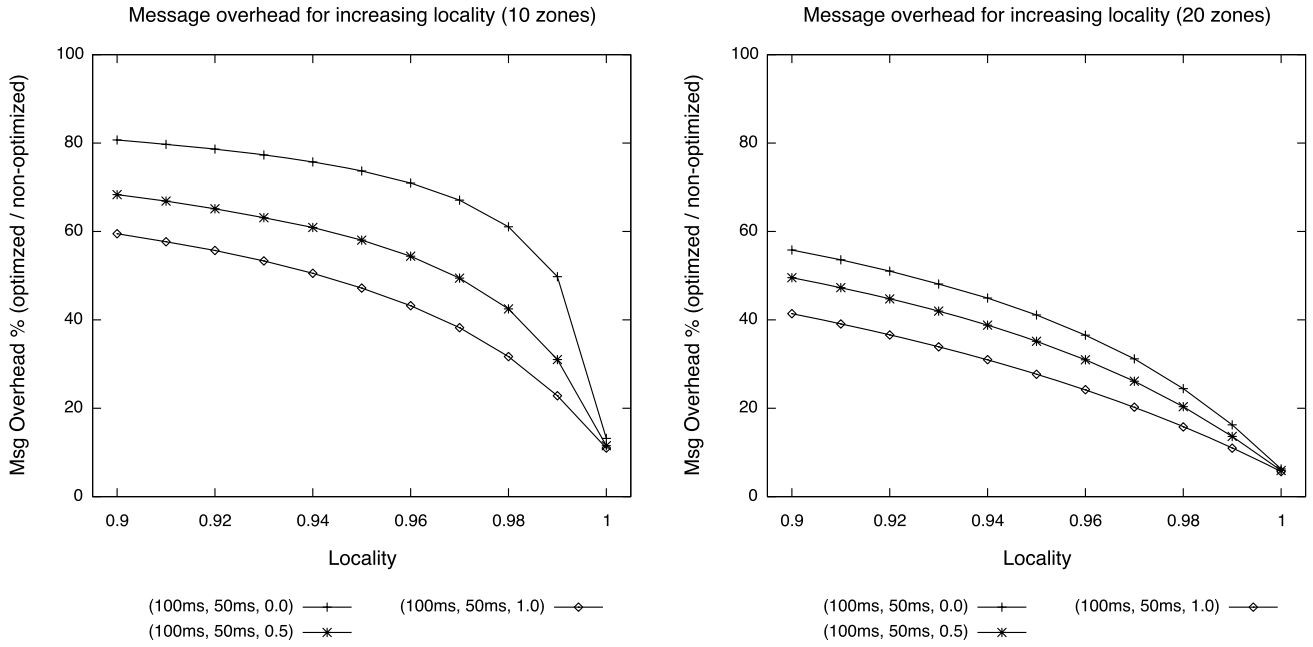

Figure 1: Average message overhead as a function of L

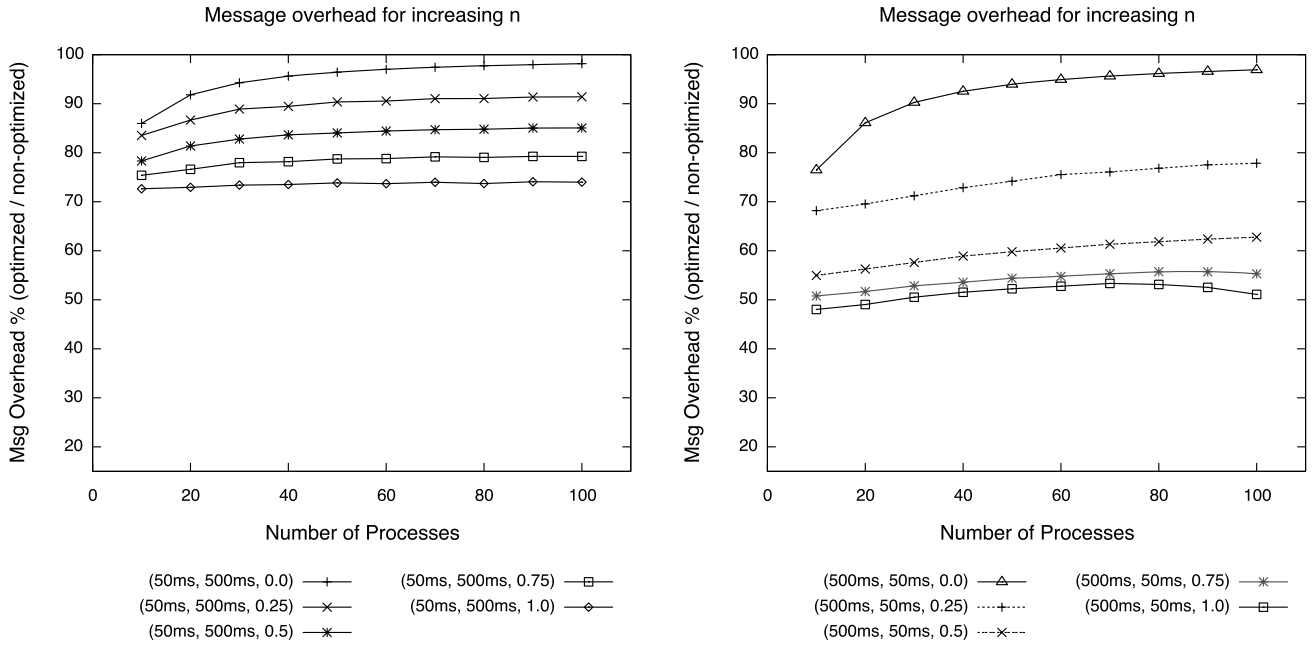

Figure 2: Average message overhead as a function of n

destination and 1.0 means all send events send a message to $n-1$ destinations.

When the multicast ratio M/T > 0 increases by a factor of δ, the time duration between two sends by P_i to a fixed other process P_j decreases by a factor of δ. Thus, $LS_i[j]$ is that much newer, and this potentially allows fewer clock

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

components k with $LU_i[k] > LS_i[j]$ to be sent as message overhead by P_i to P_j. Note now, that within this decreased time duration, the probability of P_i receiving a message from a fixed other process P_k increases by a factor of δ because P_k is also now multicasting to a larger number of processes. Despite this increased probability, if multiple messages are received from P_k, they will result in a single tuple (corresponding to P_k's local clock value in the causally *latest* such message) for which $LU_i[k] > LS_i[j]$. Thus, this increased probability of receiving messages from P_k has a relatively smaller impact in increasing the message overhead. Whereas, the decreased value of the time interval since the last message sent to P_j has a relatively larger impact in reducing the message overhead.

Observe also from Figure 3 that the rate of decrease in the message overhead over M/T is affected by the ratio of MTT over MIMT. The configurations in the left graph have smaller ratio. The ratio MTT/MIMT for these three configurations is 0.5, 1.5, and 2.5. The overhead percentage is decreased from 96.9% to 73.9% in the (100ms, 200ms, 60) case. In the higher ratio case, (500ms, 50ms, 60), the overhead percentage drops from 94.9% to 52.8%. In the right side graph, we also notice that the (500ms, 25ms, 60) case reports more rapid decrease than the other configurations. The ratio MTT/MIMT for these three configurations is 20.0, 10.0, and 6.67. We explore this behavior more in the next experiment.

D. Impact of the Ratio of Mean Transmission Time to Mean Inter-Message Time

During our preliminary simulation runs, we found that MTT and MIMT had a close correlation to the efficiency of the SK algorithm. In this section, we further nail down this factor by varying the ratio of MTT to MIMT instead of examining each factor separately. For the simulation parameters, we fixed n at 80 processes and varied the ratio from 0.1 to 50 for M/T = (0.0, 0.5, 1.0). We assumed a default L = 0.0 and a single zone. Results are shown in Figure 4. Curves are shown for (n, M/T). In the left graph, we fix MIMT=10ms and MTT ranges from 1ms to 500ms to obtain the variation of ratio. In the right graph, we use MTT=250ms and vary MIMT from 5ms to 800ms. This configuration is designed to test if there is any isolated effect from MTT or MIMT itself. The result illustrates that there is no significant difference between those two graphs (left vs. right). The overhead decreases in both cases as the ratio increases, especially for the higher M/T parameters (0.5 and 1.0).

The higher ratio of mean transmission time (MTT) to mean inter-message time (MIMT) achieves lower message overhead. This higher ratio implies a slower network speed relative to the inter-message generation time. There are two implications of this: (i) at any given time, there are more messages in transit to the destinations, and (ii) as the inter-message generation time is the same, the number of messages that arrive at P_i in a fixed time duration is the same at different ratios of MTT to MIMT. Together, (i) and (ii) further imply that the messages that arrive at P_i in a fixed time duration are "older" messages at higher values of the MTT/MIMT ratio. From the above observations, we could not explain why the message overhead decreases as the MTT/MIMT ratio increases. One precaution we took in this experiment is to determine that the simulation is stable enough to measure the overhead with such a large amount of messages in transit. We verified this with a system stability test with various numbers of system-wide messages and all configurations showed a stabilized result after 10,000,000 messages. Therefore, we do not expect the side-effect from these extreme configurations.

V. CONCLUSIONS

In this study, we carried out a number of experiments to examine the efficiency of the Singhal-Kshemkalyani incremental vector clock [18] with various configurations including spatial locality (L) of communication, number of processes (n), multi-cast frequency (M/T), transmission time (MTT), and inter-message time (MIMT). We summarize our simulation results as follows.

1) Spatial locality (L) greatly affects the overall message overhead. Higher locality shows better efficiency. This confirms the intuition that the incremental technique shows marked gains when application processes communicate with locality patterns.
   Some application domains communicate with this locality property, e.g., imposing causal ordering in mobile computing [1], [14]. Another example is in file synchronization, as used in banking and enterprise organizations [3]. Such applications can directly benefit by using incremental vector clocks.
2) The message overhead of the SK algorithm is not much dependent on the number of processes (n) in the system.
3) A higher multicast frequency (M/T), as opposed to unicasting messages, incurs lower message overhead compared to the naive algorithm.
4) The higher ratio of mean transmission time (MTT) to mean inter-message time (MIMT) achieves lower message overhead. Thus, a slower network speed relative to the inter-message generation time lowers the message overhead.

Result 1 of the simulations confirms for the first time what was intuitively believed to be true. The other results reveal new, interesting properties of the incremental vector clock technique.

We note that in several applications such as social communication networks, transactional memory, or shared memory race detection where vector clocks are used, it is reasonable to think about stencil applications in which

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

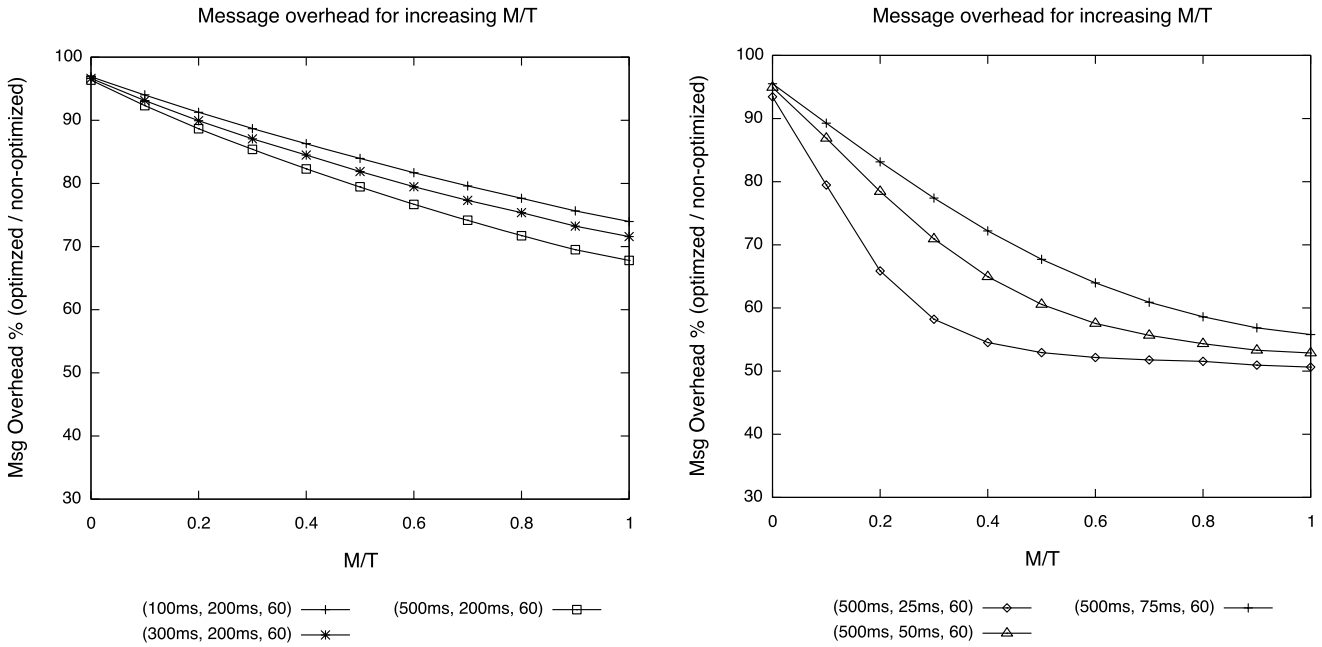

Figure 3: Average message overhead as a function of M/T

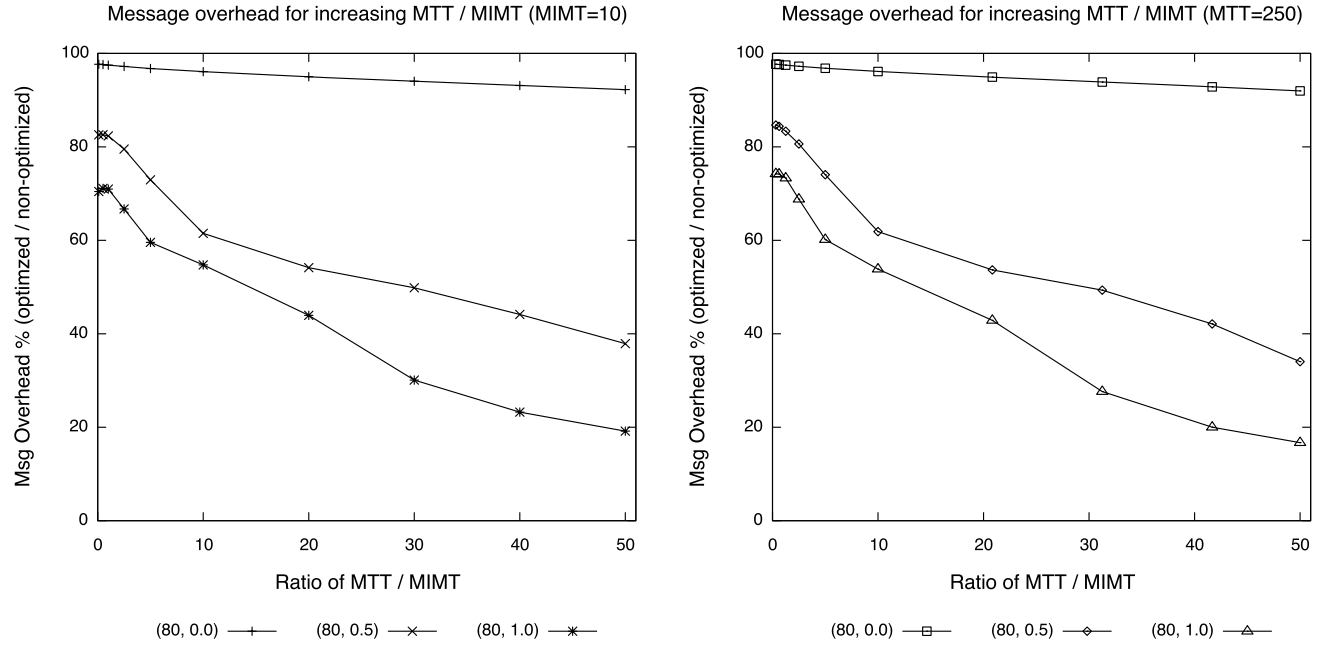

Figure 4: Average message overhead as a function of MTT/MIMT

processes communicate with or share data with only a small number of other processes. However, it may not be possible to partition the processes nicely into disjoint zones. Here, any partitioning into zones may not give a high locality because the communication partners of a process may also be communicating with other processes, such as

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.

their neighbors. If the overall communication graph has a regular structure like a mesh or torus, we have a logical time wave front moving across all the process space. Thus, having a small number of communication partners does not imply high locality. Without high locality, the incremental vector clock technique has low benefits.

ACKNOWLEDGMENT

This publication is based on work supported in part by grants to the Electronic Visualization Laboratory (EVL) at the University of Illinois at Chicago form the National Science Foundation (NSF), awards CNS-0703916 (Lifelike), CNS-0821121 (OmegaDesk), CNS-0959053 (NG-CAVE) and OCI-0943559 (SAGE). EVL also receives funding from Adler Planetarium, Argonne National Laboratory, Air Force, National Institute for Nursing Research, the NASA ASTEP Program, Science Museum of Minnesota, State of Illinois, and Pacific Interface on behalf of NTT Network Innovation Laboratories in Japan. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies and companies.

REFERENCES

[1] S. Alagar, S. Venkatesan, Causal ordering in distributed mobile systems, *IEEE Transaction on Computers*, pp 353-361, March 1997.

[2] B. Charron-Bost, Concerning the size of clocks in distributed systems, *Information Processing Letters*, 39: 11-16, 1991.

[3] R. Cox, W. Josephson, File Synchronization with Vector Time Pairs, *Tech. Rep. MIT-CSAIL-TR-2005-014*, 2005.

[4] G. DeCandia, D. Hastorun, M. Jampani, et al. Dynamo: Amazon's highly available key-value store, *ACM SIGOPS Operating Systems,* Volume 41, Issue 6 (December 2007), SOSP '07, 205-220.

[5] C. Fidge, Timestamps in message-passing systems that preserve partial ordering, *Australian Computer Science Communications*, 10(1): 56-66, February 1988.

[6] C. Flanagan, S. Freund, FastTrack: Efficient and precise dynamic race detection, *PLDI 2009, ACM SIGPLAN Notices,* Volume 44 , Issue 6, 121-133, (June 2009).

[7] G. Kossinets, J. Kleinberg, D. Watts, The structure of information pathways in a social communication network, *Proc. 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 2008.

[8] A.D. Kshemkalyani, M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems,* Cambridge University Press, 2008.

[9] L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM*, 21(7): 558-565, July 1978.

[10] F. Mattern, Virtual time and global states of distributed systems, *Parallel and Distributed Algorithms*, North-Holland, pp 215-226, 1989.

[11] S. Meldal, S. Sankar, J. Vera, Exploiting locality in maintaining potential causality, *Proc. ACM Symposium on Principles of Distributed Computing,* 1991.

[12] G. Oster, P. Urso, P. Molli, A. Imine, Data consistency for P2P collaborative editing, *Proc. 20th Anniversary Conference on Computer Supported Cooperative Work,* 259-268, 2006.

[13] R. Prakash, M. Singhal, Dependency sequences and hierarchical clocks: Efficient alternatives to vector clocks for mobile computing systems, *Wireless Networks,* No. 3 (1997), pps 349-360.

[14] R. Prakash, M. Raynal, M. Singhal, An adaptive causal ordering algorithm suited to mobile computing environments, *Journal of Parallel and Distributed Computing*, pp 190-204, March 1997.

[15] M. Raynal, M. Singhal, Logical time: Capturing causality in distributed systems, *IEEE Computer*, 49-56, February 1996.

[16] T. Riegel, C. Fetzer, H. Sturzrehm, P. Felber, From causal to z-linearizable transactional memory, *Proc. 26th Annual ACM Symposium on Principles of Distributed Computing,* 340-341, 2007.

[17] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: In search of the holy grail, *Distributed Computing*, 7:149-174, 1994.

[18] M. Singhal, A. Kshemkalyani, Efficient implementation of vector clocks, *Information Processing Letters*, 43, 47-52, August 1992.

[19] F. J. Torres-Rojas, M. Ahamad, Plausible clocks: Constant size logical clocks for distributed systems, *Distributed Computing,* 12(4): 179-195, 1999.

[20] P. A. S. Ward, An online algorithm for dimension-bound analysis, *Proc. 5th International Euro-Par Conference on Parallel Processing,* p.144-153, 1999.

[21] T. Weis, A. Wacker, S. Schuster, S. Holzapfel, Towards logical clocks in P2P-based MMVEs, *Proc. 1st International Workshop on Concepts of Massively Multiuser Virtual Environments,* CoMMVE09, 2009.

Authorized licensed use limited to: Hochschule HS Hamburg. Downloaded on June 17,2021 at 10:18:16 UTC from IEEE Xplore. Restrictions apply.