Effectiveness of Delaying Timestamp Computation

Sandeep S. Kulkarni
Michigan State University
Department of Computer Science and Engineering
428 S. Shaw Lane
East Lansing, MI 48824
sandeep@cse.msu.edu

Nitin H. Vaidya
University of Illinois at Urbana-Champaign
Department of Electrical and Computer Engineering
1308 W Main St
Urbana, IL 61801
nhv@illinois.edu

ABSTRACT

Practical algorithms for determining causality by assigning timestamps to events have focused on *online* algorithms, where a permanent timestamp is assigned to an event as soon as it is created. We address the problem of reducing size of the timestamp by utilizing the underlying topology (which is often not fully connected since not all processes talk to each other) and deferring the assignment of a timestamp to an event for a suitably chosen period of time after the event occurs. Specifically, we focus on inline timestamps, which are a generalization of offline timestamps that are assigned after the computation terminates. We show that for a graph with vertex cover $\mathcal{VC}$, it is possible to assign inline timestamps which contains only $2|\mathcal{VC}|+2$ elements. In particular, for a system with n processes and K events per process, the size of a timestamp for any event is at most $\log_2 n + (2|\mathcal{VC}|+1)\log_2(K+1))$ bits. By contrast, if online timestamps are desired, then even for a star network, vector timestamp of length n (for the case of integer elements) or $n-1$ (for the case of real-valued elements) is required. Moreover, in addition to being efficient, the inline timestamps developed can be used to solve typical problems such as predicate detection, replay, recovery that are solved with vector clocks.

1 INTRODUCTION

The paper considers an asynchronous system consisting of n processes that communicate over message-passing channels. The message-passing channels form a communication network, which may or may not be completely connected. One of the important problems in such distributed systems is detection of causality among events, as defined by the happened-before relation $\rightarrow$ [14]. (As defined in [14], event A happened-before event B iff either (1) A and B are events on the same process and A occurred before B, (2) A is a send event and B is a corresponding receive event, or (3) there is an event C such that A happened-before C and C happened before B.) Such causality detection is important for debugging, monitoring, and other applications. Vector clocks are often used to track causality. An important challenge in detecting causality using vector clocks is that for a system with n processes, without any additional assumptions, the minimum vector length necessary is n.

Much of the existing work on practical algorithms for assigning timestamps, which capture the causality relation (e.g., [7, 16]), inherently requires that timestamp of an event is assigned as soon as that event is created. We denote these algorithms as *online* algorithms. In particular, in vector clock algorithms, each event e is assigned a timestamp vc_e which is a vector consisting of integer elements. Furthermore, to determine causality, these timestamps are compared by pairwise element comparison, such that (i) $vc_e \leq vc_f$ iff for each k, $vc_e[k] \leq vc_f[k]$, and (ii) $vc_e < vc_f$ iff $vc_e \leq vc_f \wedge vc_e \neq vc_f$. We refer to this $<$ operator as the *standard vector clock comparison*.

The goal of this paper is to identify an efficient timestamp design by relaxing the assumptions that (1) a permanent timestamp is assigned as soon as an event is created, and (2) timestamps are compared using the standard vector clock comparison. Relaxing the first assumption, we let the timestamp of an event to be either $\perp$, or a permanent value that will not change subsequently. When an event occurs, it may possibly be assigned value $\perp$, which is eventually changed to its permanent value. We refer to such timestamps as *inline timestamps*. Given two events e and f with timestamps $timestamp_e \neq \perp$ and $timestamp_f \neq \perp$, respectively, we will specify a suitable $<$ operator on the timestamps such that $e \rightarrow f$ iff $timestamp_e < timestamp_f$. Observe that *Online* and *Offline* timestamps can be viewed as two extreme instantiations of the inline timestamps. In particular, for *offline timestamps*, each event's timestamp remains $\perp$ until the execution terminates; only after the execution terminates, each event is assigned a timestamp.

Compared with offline timestamps, inline timestamps provide a significant advantage provided the the inline timestamp is made permanent *quickly*. For example, in an application such as rollback recovery, we can use inline timestamps by simply ignoring events whose timestamps is not finalized. This would cause the *recovery line* to be somewhat earlier than that achievable by online timestamps. However, as long as the timestamps become finalized quickly, this change would be negligible. By contrast, offline timestamps cannot be used for such recovery as they are not available until the system terminates.

In this work, we focus on reducing the size of timestamps by exploiting the knowledge of the underlying communication topology. Specifically, we make the following contributions.

(1) We show that online vector timestamps do not benefit from the knowledge of the communication topology. That is, if we require that the *standard vector clock comparison* be used to compare timestamps, then efficient timestamps cannot be obtained even for specific known topology. In

PODC'17, July 25-27, 2017, Washington, DC, USA

978-1-4503-4992-5/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3087801.3087818

particular, we show that for a *star* network, any online algorithm that uses *standard vector clock comparison* must use clocks of length n if the vector elements are restricted to be integers. The vector length can be reduced slightly to $n-1$ if we permit vector elements to be real numbers. On the other hand, the inline timstamps only require 4 integer elements in the star graph, but use a new $<$ operator. Thus, our results show that requiring online timestamps with the standard vector clock comparison operator is too restrictive, and prevents us from exploiting the underlying communication topology.

(2) The above results for vectors clocks motivate the use of alternative timestamp algorithms. It turns out that there indeed exists a simple timestamp scheme that can reduce the timestamp size by exploiting the knowledge of the communication graph. In particular, for a communication graph with a vertex cover $\mathcal{VC}$, we present an inline algorithm that assigns timestamps containing $2|\mathcal{VC}|+2$ integer elements. In a system of n processes, provided that at most K events occur at each of these processes, $2|\mathcal{VC}|+1$ of the timestamp elements are of size at most $\log_2(K+1)$ bits, and the remaining element is of size $\log_2 n$ bits. Thus, if $|\mathcal{VC}| < \frac{n}{2} - 1$, the above inline timestamp would have a smaller size that the standard vector clock.

Organization of the paper. In Section 2, we identify lower bounds associated with online timestamps. In Section 3, we discuss our algorithm for inline timestamps in star network. Section 4 generalizes this solution to arbitrary graphs. Section 5 discusses related work. Finally, we discuss applications of our work in Section 6 and conclude in Section 7. Feasibility of using inline timestamps to solve problems such as predicate detection and replay/recovery is discussed in [23].

2 ONLINE VECTOR TIMESTAMPS

In this section, we consider the benefits of knowing the communication network on the minimum size necessary for vector timestamps (i.e., using the standard vector clock comparison). It is known that vector clocks of length n suffice in any network. The results presented here are negative in that in most networks, no significant reduction can be achieved even with the knowledge of the communication topology. This motivates the work presented later on *inline algorithms*, which can be significantly more efficient.

Lemma 2.1. *Suppose that an* online *algorithm for the star graph assigns distinct* **real-valued** *vector timestamps to distinct events such that, for any two events e and f, $e \rightarrow f$ if and only if $timestamp_e < timestamp_f$, where $<$ is* standard vector clock comparison. *Then the vector length must be at least $n-1$.*

Proof. The proof is trivial for $n \leq 2$. Now assume that $n \geq 3$. The proof is by contradiction. Suppose that a given online algorithm assigns vector timestamps of length $s \leq n-2$.

Let e_q^j denote the q-th event at process p_j. Consider an execution that includes a send event e_1^i at radial process p_i, $1 \leq i \leq n-1$, where the radial process p_i sends a message to the central process p_0. These $n-1$ send events are concurrent with each other. At process p_0, there are $n-1$ *receive* events corresponding to the above *send* events at the other processes. The execution contains no other events.

Note that $timestamp_{e_q^j}$ denotes the vector timestamp of length s assigned to event e_q^j by the online algorithm. Create a set S of processes as follows: for each l, $0 \leq l < s$, add to S any one radial process p_j such that $timestamp_{e_1^j}[l] = \max_{1 \leq i < n} timestamp_{e_1^i}[l]$. Note that $timestamp_{e_1^i}[l]$ is the l-th element of vector $timestamp_{e_1^i}$. Clearly, $|S| \leq s \leq n-2$. Consider a radial process $p_k \notin S$ (note that $p_k \neq p_0$). Such a process p_k must exist since $|S| \leq n-2$, and there are $n-1$ radial processes.

Suppose that the message sent by process p_k at event e_1^k reaches process p_0 after all the other messages, including messages from all the processes in S, reach process p_0. That is, e_{n-1}^0 is the receive event for the message sent by process p_k. By the time event e_{n-1}^0 occurs, (online) timestamps must have been assigned to all the other events in this execution. This scenario is possible because the message delays can be arbitrary, and an online algorithm assigns timestamps to the events when they occur.

Now consider event e_{n-2}^0. By event e_{n-2}^0, except for the message sent by process p_k, all the other messages, including messages sent by all the processes in S, are received by process p_0.

Define vector E such that $E[l] = \max_{1 \leq i < n} timestamp_{e_1^i}[l]$, $0 \leq l < s$. By definition of S, we also have that $E[l] = \max_{p_i \in S} timestamp_{e_1^i}[l]$, $0 \leq l < s$. The above assumption about the order of message delivery implies that $E \leq timestamp_{e_{n-2}^0}$. Also, since $p_k \notin S$, we have that $timestamp_{e_1^k} \leq E$. The above two inequalities together imply that $timestamp_{e_1^k} \leq timestamp_{e_{n-2}^0}$.

Since $e_1^k \neq e_{n-2}^0$, their timestamps must be distinct too. Therefore, $timestamp_{e_1^k} < timestamp_{e_{n-2}^0}$, which, in turn, implies that $e_1^k \rightarrow e_{n-2}^0$. However, e_1^k and e_{n-2}^0 are concurrent, leading to a contradiction. □

Lemma 2.2. *Suppose that an* online *algorithm for the star graph $\mathcal{G}$ assigns distinct* **integer-valued** *vector timestamps to distinct events such that, for any two events e and f, $e \rightarrow f$ if and only if $timestamp_e < timestamp_f$, where $<$ is* standard vector clock comparison. *Then the vector length must be at least n.*

Proof. Without loss of generality, let us assume that all vector elements of a vector timestamp must be non-negative integers. The proof of the lower bound is trivial for $n = 1$. Now assume that $n \geq 2$. The proof is by contradiction. Suppose that the vector length is $s \leq n-1$.

Consider an execution that includes a send event e_1^i at process p_i, $1 \leq i \leq n-1$, where the radial process p_i sends a message to the central process p_0. Let M be the largest value of any of the s elements of the timestamps of any of these $n-1$ send events. Suppose that process p_0 initially performs P computation events. Assume that there are no other events; thus, process p_0 does not send any messages. Thus, the timestamps of the send events at the radial processes cannot depend on P, the number of computation events at p_0. Thus, we can assume that $P = (M+2)n$. Since these $(M+2)n$ computation events occur at p_0 before it receives any messages, the timestamps for these events are computed before p_0 learns timestamps of any send events at the other processes.

Since the timestamp elements are constrained to be non-negative integers, one of the elements of the timestamp of the last of these computation events at p_0, namely e_P^0, must be $> M$. Recall that $P = (M+2)n$.

Consider set W that contains event e_P^0 and e_1^i, $0 < i < n$. Thus, W contains n events, with one event at each of the n processes. Create a set S of processes as follows: for each l, $0 \leq l < s$, add to S any one process p_j such that the l-th element of the timestamp of its event in W is the largest among the l-th elements of the timestamps of all the events in W. Clearly, $p_0 \in S$ and $|S| \leq s \leq n-1$. Consider a radial process $p_k \notin S$ (note that $p_k \neq p_0$). Such a process p_k must exist since $|S| \leq n-1$, $p_0 \in S$, and there are $n-1$ radial processes.

Suppose that the message sent by process p_k at event e_1^k reaches process p_0 after all the other messages, including messages from the *radial* processes in S, reach process p_0. Let $e = e_{P+n-2}^0$. By event e at p_0, except for the message sent by process p_k, all the other messages, including messages sent by all the radial processes in S, are received by process p_0.

Rest of the proof of this lemma is similar to the proof of Lemma 2.1. In particular, define vector E such that $E[l] = \max_{0 \leq i < n} timestamp_{e_1^i}[l]$, $0 \leq l < s$. By definition of W, we also have that $E[l] = \max_{p_i \in W} timestamp_{e_1^i}[l]$, $0 \leq l < s$.

The above assumption about the order of message delivery implies that

$$E \leq timestamp_{e_{P+n-2}^0}.$$

Also, since $p_k \notin W$, we have that $timestamp_{e_1^k} \leq E$. This implies that $timestamp_{e_1^k} \leq timestamp_{e_{P+n-2}^0}$.

Since $e_1^k \neq e_{P+n-2}^0$, their timestamps must be distinct too. This implies that $timestamp_{e_1^k} < timestamp_{e_{P+n-2}^0}$, which, in turn, implies that $e_1^k \rightarrow e_{P+n-2}^0$. However, e_1^k and e_{P+n-2}^0 are concurrent events, leading to a contradiction.

□

Lemma 2.3. *Suppose that the communication $\mathcal{G}$ graph has vertex connectivity ≥ 2. For this graph, an* online *algorithm assigns distinct vector timestamps to distinct events such that, for any two events e and f, $e \rightarrow f$ if and only if $timestamp_e < timestamp_f$. Then the vector length must be $\geq n$.*

Proof. Recall that $\mathcal{G}$ is the communication graph formed by the n processes.

This proof is analogous to the proof of Lemma 2.1. The proof is trivial for $n \leq 2$.

Now assume that $n \geq 3$. The proof is by contradiction. Suppose that the vector length is $s \leq n-1$.

Consider an execution in which, initially, each process p_i, $0 \leq i < n$, sends a message to each of its neighbors in the communication graph. Subsequently, whenever a message is received from any neighbor, a process forwards the message to all its other neighbors. Thus, essentially, the messages are being flooded throughout the network (the execution is infinite, although we will only focus on a finite subset of the events).

Create a set S of processes as follows: for each l, $0 \leq l < s$, add to S any one process p_j such that $timestamp_{e_1^j}[l] = \max_{0 \leq i < n} timestamp_{e_1^i}[l]$. Clearly, $|S| \leq s \leq n-1$. Consider a process $p_k \notin S$. Such a process p_k must exist since $|S| \leq n-1$.

Suppose that all the communication channels between p_k and its neighbors are very slow, but each of the remaining communication channels has a delay upper bounded by some constant $\delta > 0$. For convenience of discussion, let us ignore local computation delay between the receipt of a message at a process and its forwarding to the neighbors. Let D be defined as the maximum over the diameters of all the subgraphs of $\mathcal{G}$ containing $n-1$ vertices. Let the delay on all communication channels of p_k be $> 2\delta D$. Because the network's vertex connectivity is ≥ 2, within duration δD, $n-1$ processes, except p_k, will have received messages initiated by those $n-1$ processes (i.e., all messages except the message initiated by p_k).

Define vector E such that $E[l] = \max_{0 \leq i < n} timestamp_{e_1^i}[l]$, $0 \leq l < s$. By definition of S, we also have that $E[l] = \max_{p_i \in S} timestamp_{e_1^i}[l]$, $0 \leq l < s$.

Consider any process $p_i \neq p_k$. Let e be the earliest receive event at p_i such that by event e (i.e., including event e), p_i has received the messages initiated by all processes except p_k. Due to the definition of D and δ, event e occurs at p_i by time δD. Since by event e, p_i has received the messages initiated by all other processes except p_k, and $p_k \notin S$, we have

$$E \leq timestamp_e.$$

Also, since $p_k \notin S$, we have

$$timestamp_{e_1^k} \leq E.$$

The above two inequalities together imply that $timestamp_{e_1^k} \leq timestamp_e$.

Since e_1^k and e occur on different processes, $e_1^k \neq e$, and their timestamps must be distinct too. Thus, $timestamp_{e_1^k} < timestamp_e$, which, in turn, implies that $e_1^k \rightarrow e$. However, e_1^k and e are concurrent events, because e_1^k is the first event at p_k, there are no messages received by p_k before $2\delta D$, and similarly, no process receives messages from p_k during $2\delta D$. This results in a contradiction.

□

For any graph, upper bound of n is obtained by using the standard vector clock algorithm for n processes [8, 15]. Thus, the bound n is tight for communication graphs with vertex connectivity ≥ 2.

Lemma 2.4. *Suppose that the communication graph has vertex connectivity = 1. Define X to be the set of processes such that no process in set X by itself forms a vertex cut of size 1. For this graph, an* online *algorithm assigns distinct vector timestamps to distinct events such that, for any two events e and f, $e \rightarrow f$ if and only if $timestamp_e < timestamp_f$. Then the vector length must be at least $|X|$.*

Proof. Recall that $\mathcal{G}$ is the communication graph formed by the n processes.

This proof is analogous to the proof of Lemma 2.3. The proof is trivial for $|X| = 1$.

Now assume that $|X| \geq 2$. The proof is by contradiction. Suppose that the vector length is $s \leq |X| - 1$.

Consider an execution in which, initially, each process $p_i \in X$ sends a message to each of its neighbors in the communication

graph. Subsequently, whenever a message is received from any neighbor, a process forwards the message to all its other neighbors. Thus, essentially, the messages initiated by processes in X are being flooded throughout the network (the execution is infinite, although we will only focus on a finite subset of the events).

Create a set S of processes as follows: for each l, $0 \leq l < s$, add to S any one process $p_j \in X$ such that $timestamp_{e_1^j}[l] = \max_{p_i \in X} timestamp_{e_1^i}[l]$. Clearly, $|S| \leq s \leq |X| - 1$. Consider a process $p_k \in X$ such that $p_k \notin S$. Such a process p_k must exist since $|S| \leq |X| - 1$.

Suppose that all the communication channels between p_k and its neighbors are very slow, but each of the remaining communication channels has a delay upper bounded by some constant $\delta > 0$. For convenience of discussion, let us ignore local computation delay between the receipt of a message at a process and its forwarding to the neighbors. Let D be defined as the maximum over the diameters of all the subgraphs of $\mathcal{G}$ containing all vertices except any one vertex in X (there are $|X|$ such subgraphs). By definition of X, removing any one process in X from the graph $\mathcal{G}$ will not partition the subgraph. Let the delay on all communication channels of p_k be $> 2\delta D$. Within duration δD, all $n-1$ processes, except p_k, will have received the messages initiated by the $|X| - 1$ processes in $X - \{p_k\}$.

Define vector E such that $E[l] = \max_{p_i \in X} timestamp_{e_1^i}[l]$, $0 \leq l < s$. By definition of S, we also have that $E[l] = \max_{p_i \in S} timestamp_{e_1^i}[l]$, $0 \leq l < s$.

Consider any process $p_i \in X$ such that $p_i \neq p_k$. Let e be the earliest receive event at p_i such that by event e (i.e., including event e), p_i has received the messages initiated by all processes except p_k. Due to the definition of D and δ, event e occurs at p_i by time δD. Since by event e, p_i has received the messages initiated by all other processes except p_k, and $p_k \notin S$, we have

$$E \leq timestamp_e.$$

Also, since $p_k \notin S$, we have

$$timestamp_{e_1^k} \leq E.$$

The above two inequalities together imply that $timestamp_{e_1^k} \leq timestamp_e$.

Since e_1^k and e occur on different processes, $e_1^k \neq e$, and their timestamps must be distinct too. Therefore, $timestamp_{e_1^k} < timestamp_e$, which, in turn, implies that $e_1^k \rightarrow e$. However, e_1^k and e are concurrent events, because e_1^k is the first event at p_k, there are no messages received by p_k before $2\delta D$, and similarly, no process receives messages from p_k during $2\delta D$. This results in a contradiction.

Observe that for star graph, vertex connectivity is 1, and X consists of all the radial processes. Thus, $|X| = n - 1$.

For a communication graph with vertex connectivity 1, an upper bound of $n-1$ (not necessarily tight) is obtained by assigning the role of p_0 in the star graph to any one process that forms a cut of the communication graph, and then using the vector timestamping algorithm presented below for the star graph. In general, there is a gap between the above upper bound of $n-1$ and lower bound of $|X|$. It is presently unknown whether $|X|$ is a tight bound for online algorithms that assign vector timestamps.

□

3 ILLUSTRATING INLINE ALGORITHM: CASE STUDY WITH STAR NETWORK

Before presenting the inline algorithm for general network topologies, we illustrate it in the special case of a *star* network. The generalized algorithm is presented in Section 4. A star network consists of one *central* process, say C, and $n-1$ *radial* processes. Each radial process can communicate directly only with process C. The key idea behind the inline algorithm is that for any two events e and f at *different* radial processes such that $e \rightarrow f$, there must exist an event g at C such that $e \rightarrow g \rightarrow f$. This observation is used to reduce the timestamps for events at radial processes, essentially by using appropriate events at C as proxies for the events at radial processes. The generalized algorithm in Section 4 extends this observation to a vertex cover of the communication graph (observe that the center process by itself is a vertex cover of the star graph).

3.1 Timestamping Algorithm

Instead of describing the algorithm for the star graph operationally (i.e., what occurs when events take place), we first describe it in a declarative manner. The algorithm assigns timestamp of event e on process j as $timestamp_e = \langle id_e, ctr_e, pre_e, post_e \rangle$, defined as follows:

- $id_e = j$, the process where event e occurred.
- ctr_e (denoting 'counter'): If event e is x^{th} event on process j then $ctr_e = x$. For the first event at any process, ctr is 1.
- $pre_e = \max\,(ctr_f \mid f$ is an event at C such that $e = f$ or $f \rightarrow e)$).

  For convenience, we adopt the convention that maximum of an empty set is 0.

Finally, we define $post_e$ for event e. $post$ is defined only for events at radial processes.

- $post_e = \min\,(ctr_f \mid f$ is an event at C such that $e \rightarrow f)$).

  For convenience, we adopt the convention that minimum of an empty set is ∞ (this may seem counter-intuitive, but the benefit of the convention should become clearer later).

Observe that for an event e at C, $pre_e = ctr_e$ (and $post_e$ is not defined for events at C). When $post_e$ in the timestamp of event e is ∞, we sometimes write $timestamp_e = \perp$ for brevity. Since $post_e$ is not defined for events on C, if e is an event on C, $timestamp_e$ will never be equal to $\perp$. ($\perp$ should be interpreted here as "unknown").

3.2 Operational Description

Since the above algorithm is given in a declarative fashion as opposed to operational manner, next, we identify how and when these timestamps can be determined. Please refer Figure 1.

At each process j, ctr_j and pre_j are initialized to 0. In our pseudocode in Figure 1, $a, b, c = p, q, r$ is equivalent to $a = p;\ b = q; c = r$.

Rules for assigning ctr_e: For any event e, ctr_e is just the index of the event at process j. The index of first event at

If e is a local or send event at radial process j
$ctr_j = ctr_j + 1$ (ctr_j initialized to 0)
$ctr_e, pre_e, post_e = ctr_j, pre_j, \infty$
If e corresponds to sending of message m,
include $\langle ctr_e, pre_e \rangle$ with message m.

If e corresponds to receiving $m\langle ctr_m, pre_m \rangle$ at radial process j
$ctr_j = ctr_j + 1$
$pre_j = max(pre_j, pre_m)$

If e is a local or send event at C
$ctr_C = ctr_C + 1$
$ctr_e, pre_e = ctr_C, ctr_C$
If e corresponds to sending of message m,
include $\langle ctr_e, pre_e \rangle$ with message m.

If e corresponds to receiving $m\langle ctr_m, pre_m \rangle$ at C
$ctr_C = ctr_C + 1$
$ctr_e, pre_e = ctr_C, ctr_C$
Send control message with $\langle ctr_m, ctr_C \rangle$ to j
// This message can be piggybacked with a future message to j

Upon Receiving control message $\langle a, b \rangle$ at radial process j
For each event e on process j such that $ctr_e \leq a$
$post_e = min(post_e, b)$

Figure 1: Algorithm for Star Network

j is 1. ctr_j is used as a counter at process j to determine ctr_e.

Rules for assigning pre_e: For an event e on the central process, $pre_e = ctr_e$. This follows from the definition of pre.

Each radial process j maintains a variable pre_j that keeps track of the maximum value of pre_e for any event e created on process j. Initially, $pre_j = 0$. Local events and send events at process j leave pre_j unchanged. For a new send/local event e, $pre_e = pre_j$. When sending a message m, the tuple $\langle ctr_e, pre_e \rangle$ is included with m.

Now consider a receive event e at radial process j. Let the message received at e be m. Let the tuple piggybacked on m be $\langle ctr_m, pre_m \rangle$. When m is received, process j sets pre_e and pre_j to equal $max(pre_j, ctr_m)$. Observe that ctr_m equals the index of the send event corresponding to m – the send event must necessarily have occurred at C, since the message is received at a radial process in the star graph.

Rules for assigning $post_e$: For an event e at C, $post_e$ is not defined. For event e on a radial process j, $post_e$ is defined when a message is sent by j at or after e and is received by C. Until this point, $timestamp_e$ is equal to $\perp$. Recall that we treat $timestamp_e$ as $\perp$ while $post_e$ remains ∞.

There are several approaches for defining $post_e$. One approach is as follows: Upon receiving a message m from a radial process j with timestamp $\langle ctr_m, pre_m \rangle$, C sends a *control message* that contains $\langle ctr_m, ctr_f \rangle$, where f is the corresponding receive event. If this *control channel* is FIFO then j can update *post* entry for any event that corresponds to sending of m and any previous events for which the *post* field is still ∞. Specifically, let e be any event on process j such that $post_e$ is not yet defined and either $e = send(m)$ or f happened before sending of $send(m)$. Upon receiving the control message for message m, process j marks $post_e$ to be ctr_f.

We note that there are several possible approaches for computing the *post* field in events on radial process. The above discussion relies on *FIFO control channels* so that the radial process learns about receive events on C in order. The application messages still could be delivered in any order. In an implementation, instead of separate control messages, this information could be piggybacked to any future messages as well. However, the piggybacking approach may potentially delay assignment of permanent timestamp assignment to events on radial processes. It is easy to simulate a FIFO channel for the control messages that are piggybacked on the application messages, even if the application messages themselves are not necessarily delivered to the processes in a FIFO order.

Figure 1 summarizes how the above rules can be implemented in an operational manner. Each process j maintains ctr_j and pre_j, initialized to 0. The *id* in the timestamp corresponds to the process where the event occurred. The pseudo-code assumes that the control messages are sent on separate control channels that are FIFO.

3.3 Correctness Property

If $timestamp_e \neq \perp$ and $timestamp_f \neq \perp$ (i.e., $post_e$ and $post_f$ are finalized) then they can be used to deduce causality information, as specified by Theorem 3.1.

Theorem 3.1. *For any two events, e and f, if $timestamp_e \neq \perp$ and $timestamp_f \neq \perp$ then*
e happened before f iff $timestamp_e < timestamp_f$, where

$$timestamp_e < timestamp_f$$

iff

$$\begin{cases} pre_e < pre_f & \text{if } id_e = C \text{ and } id_f = C \\ pre_e \leq pre_f & \text{if } id_e = C \text{ and } id_f \neq C \\ post_e \leq pre_f & \text{if } id_e \neq C \text{ and } id_f \neq id_e \\ ctr_e < ctr_f & \text{if } id_e \neq C \text{ and } id_f = id_e \end{cases}$$

Proof. We sketch the proof from the four cases in the comparison relation above.

- $id_e = id_f = C$**:** In this case $pre_e = ctr_e$ captures the sequence number for event e. Hence, e happened before f iff $pre_e < pre_f$.
- $id_e = C$ **and** $id_f \neq id_e$**:** By definition of pre_f, event x on C happened before f iff $ctr_x \leq pre_f$. Also, for event x on C, $ctr_x = pre_x$. Hence, if e is on C and f is on a radial process then e happened before f iff $pre_e \leq pre_f$.
- $id_e \neq C$ **and** $id_f \neq id_e$**:** First consider the case where f is on C. In this case, $ctr_f = pre_f$. And, by definition of $post$, if x is an event on C then $post_e \leq pre_x$. Hence, e happened before f iff $post_e \leq pre_f$.
  Next, we consider the case where f is outside C. Since the graph is a star network and e and f are outside C, e happened before f is true iff there exists an event x on C such that e happened before x and x happened before e. If x is on C then from above discussion e happened before x iff $post_e \leq pre_x$. Also, from the second case, x happened before f iff $pre_x \leq pre_f$. Thus, e happened before f iff $post_e \leq pre_f$.
- $id_e = C$ **and** $id_f = id_e$**:** In this case, by definition of ctr, it follows that e happened before f iff $ctr_e < ctr_f$.

□

4 GENERALIZATION TO ALL COMMUNICATION GRAPHS

While the discussion in Section 3 focuses on the star graphs, we can extend it to other (static) graphs by identifying the role played by the central process. The key property of interest of the central process was that every message was either sent to it or from it. We use this observation to extend the algorithm in Section 3 for arbitrary topology.

Instead of having just one central process C, we use the vertex cover associated with the given communication graph. Let $\mathcal{VC}$ be the vertex cover associated with the given communication graph. In other words, every message is either sent from some process in $\mathcal{VC}$ or to some process in $\mathcal{VC}$ (or possibly both).

Before we explain our algorithm, we identify the two extremes: When $\mathcal{VC}$ consists of just one process; this corresponds to a star graph, and the algorithm in Section 3 can be applied here. On the other extreme, when $\mathcal{VC}$ contains all the processes, one can use the traditional vector clocks.

Our algorithm combines both these extremes. Specifically, processes in $\mathcal{VC}$ maintain a vector clock among themselves. These entries play the same role as pre in the algorithm in Section 3. Nodes outside $\mathcal{VC}$ maintain entries similar to that of pre and $post$. However, instead of just maintaining it for one process, they have to maintain it for every process in $\mathcal{VC}$. In other words, pre and $post$ become vectors with one entry per process in $\mathcal{VC}$. We use $mpre$ and $mpost$ to denote these vectors. Specifically, let e be an event on process j (either in $\mathcal{VC}$ or outside $\mathcal{VC}$) and c be a process in $\mathcal{VC}$. We define

- $id_e = j$, i.e., the process where j occurred.
- If e is x^{th} event on process j then $mctr_e = x$.
- $mpre_e[c] = max(\{mctr_f | f$ is an event on process c such that $e = f$ or $f \rightarrow e\})$

Recall our convention that *maximum* of an empty set is 0, and *minimum* of an empty set is ∞. Finally, we define $mpost$ only for events e that occur on processes outside $\mathcal{VC}$, where

- $mpost_e[c] = min(\{\ mctr_f \mid f$ is an event at process c such that there exists a message m such that
  m is sent from j to c,
  $(e = send(m)$ or $e \rightarrow send(m))$ and
  $f = receive(m)$ or $receive(m) \rightarrow f)$
  $\}\ )$

Remark. Let e be an event at process $j \notin \mathcal{VC}$. Observe that if there is no channel between $c \in \mathcal{VC}$ and j then $timestamp_e[c]$ is always ∞. Only when there is a channel between c and j, but for an event e at j, we have $post_e[c] = \infty$, then we treat the $timestamp_e$ as being $\perp$ (here $\perp$ should be interpreted as "unknown").

4.1 Illustration of our Timestamping Algorithm

Figure 2 provides an illustration of our algorithm for the case where the vertex cover is p_0 and p_1, process p_2 is connected to p_1 and process p_3 is connected to p_0. For brevity, we omit the id from the timestamps. In this figure, processes p_0 and p_1 are in $\mathcal{VC}$. Hence, the timestamp at these processes show the values of $mctr$ and $mpre$. For instance, for event e at p_0, $mctr_e = 4$ and $mpre = (4, 1)$. For events at processes outside $\mathcal{VC}$, $mctr$, $mpre$ and $mpost$ are shown. For instance, for event h, $mctr_h = 1$, $mpre_h = (0, 1)$ and $mpost_h = (4, \infty)$.

Computation of $mctr$ is straightforward since it simply is incremented by 1 every time. For sake of illustration, consider event g. Computation of $mpre_g$ is based on events on p_0 and p_1 that happened before g. There is no such event on p_0 and there is only one event on p_1 (with $mctr$ value 1). Hence, $mpre_g$ is set to $(0, 1)$. Next, consider computation of $mpost_g$: $mpost_g[0]$ is set when a message is sent by p_3 is received at p_0. When such a message is received, the corresponding event has $mctr$ set to 4. Hence, $mpost_g[0]$ is set to 4. Since there is no link between p_3 and p_1, $mpost_g[1]$ will always be ∞.

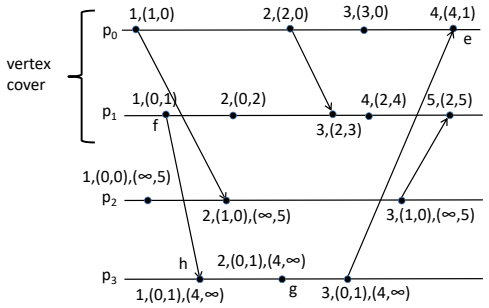

Figure 2: Illustration of Our Algorithm

4.2 Proof of Correctness

To show that the definitions of *mpre* and *mpost* suffice to capture causality between two events e and f, we need to identify how to compare their timestamps. Since these inline timestamps combine the ideas from vector clocks (for processes in $\mathcal{VC}$) and the ideas from the clock for star network (for processes outside $\mathcal{VC}$), the comparison also focuses on whether events are from $\mathcal{VC}$ or outsides $\mathcal{VC}$.

Recall that the standard vector clock comparison to compare two vectors is to compare the corresponding elements. Time. Also, $mpre_e < mpre_f$ iff $mpre_e \leq mpre_f \wedge mpre_e \neq mpre_f$. Using this comparison, we formalize our correctness requirement in Theorem 4.1:

Theorem 4.1. *For any two events, e and f, if $timestamp_e \neq \perp$ and $timestamp_f \neq \perp$ then $e \rightarrow f$ iff $timestamp_e < timestamp_f$, where*
$timestamp_e < timestamp_f$ iff

$$\begin{cases} mpre_e < mpre_f & \text{if } id_e \in \mathcal{VC} \text{ and } id_f \in \mathcal{VC} \\ mpre_e \leq mpre_f & \text{if } id_e \in \mathcal{VC} \text{ and } id_f \notin \mathcal{VC} \\ \exists c \in \mathcal{VC} :: mpost_e[c] \leq mpre_f[c] & \text{if } id_e \notin \mathcal{VC} \text{ and } id_f \neq id_e \\ mctr_e < mctr_f & \text{if } id_e \notin \mathcal{VC} \text{ and } id_f = id_e \end{cases}$$

Proof. We sketch the proof based on the four cases in comparison of timestamps. In this discussion, let c be a process in $\mathcal{VC}$.

- $id_e \in \mathcal{VC}$ **and** $id_f \in \mathcal{VC}$**:** Observe that *mpre* essentially creates the vector clock among processes in $\mathcal{VC}$. Hence, $e \rightarrow f$ iff $mpre_e < mpre_f$.
- $id_e \in \mathcal{VC}$ **and** $id_f \notin \mathcal{VC}$**:** By definition of $mpre_f$, given an event x on a process in $\mathcal{VC}$, x happened before f iff $mctr_x \leq mpre_f[c]$. Also, for event x on c, $mctr_x = mpre_x[c]$. Hence, if e is on $\mathcal{VC}$ and f on a process outside $\mathcal{VC}$ then e happened before f iff $mpre_e \leq mpre_f$.
- $id_e \notin \mathcal{VC}$ **and** $id_f \neq id_e$**:** Since $\mathcal{VC}$ is a vertex cover, if $id_f \neq id_e$ then e happened before f iff there exists an event x on a process, say c in $\mathcal{VC}$ and a message m from id_e to c such that (1) $e = send(m)$ or e happened before $send(m)$, (2) $x = receive(m)$ or $receive(m)$ happened before x and (3) either $f = x$ or x happened before f. Regarding the first condition, e happened before x iff $mpost_e[c] \leq mpre_x[c] = mctr_x$. Regarding the second condition, ($x = f$ or x happened before f) iff $mpre_x \leq mpre_f$. Thus, e happened before f iff $\exists c$ such that $mpost_e[c] \leq mpre[c]$.
- $id_e \notin \mathcal{VC}$ **and** $id_f = id_e$**:** In this case, by definition of *mctr*, it follows that e happened before f iff $mctr_e < mctr_f$.

□

Theorem 4.2. *For any event e, $|timestamp_e| \leq 2|\mathcal{VC}| + 2$*

Proof. This follows from the fact that *mpre* and *mpost* each contain at most $|\mathcal{VC}|$ elements. Additionally, *id* and *mctr* each require one element. □

Theorem 4.3. *Given a system with n processes where each process has (at most) K events, for any event e, $timestamp_e$ requires at most $(2|\mathcal{VC}| + 1)\log(K + 1) + \log n$ bits.*

Proof. This follows from the fact that *ctr*, each entry in *mpre* and each entry in *mpost* requires $\leq \log(K + 1)$ bits each, and *id* requires $\log n$ bits.

□

4.3 Lower Bounds for Offline Timestamps

In Section 2, we showed that for online timestamps, with standard vector clock comparison, size of the clock was either n or $n - 1$ depending upon whether we permit real-valued timestamps or only integer-valued timestamps. In Section 3, we presented an algorithm for the star graph that had 4 entries in every timestamp. In this section, we consider the minimum size for such timestamps. This lower bound also applies to offline timestamps.

Theorem 4.4. *Given a star graph consisting of 4 processes, there does not exist an offline algorithm that assigns each event e in every execution a vector vc_e of size 2 such that*

$$e \rightarrow f$$

iff

$$vc_e < vc_f, \text{ where } < \text{ denotes the standard vector clock comparison}$$

Proof. The proof is presented in [23]. □

Based on the results in this section, we find that inline timestamps provide a significant reduction compared to the size of online timestamps. Also, for the star graph, the size of the inline vector timestamps is within 1 of feasible lower bound. An open question in this context is whether the size of vector timestamps can be reduced to 3 elements.

5 RELATED WORK AND DISCUSSION

Closest to our work is (i) a timestamp algorithm for *synchronous messages* by Garg et al. [10, 11], (ii) timestamps used in causal shared memory, particularly, *SwiftCloud* [25] and *Lazy Replication* [13], and (iii) a hierarchical cluster timestamping scheme [24]. We will discuss the prior work below.

Timestamps in message-passing systems: The concept of vector clock or vector timestamp was introduced by Mattern [15] and Fidge [8]. Charron-Bost [2] showed that there exist communication patterns that require vector timestamp length equal to the number

of processes. Schwarz and Mattern [19] provided a relationship between the size of the vector timestamps and the dimension of the partial order specified by happened-before. Garg et al. [9] also demonstrated analogous bounds on the size of vector timestamps using the notion of event *chains*. Singhal and Kshemkalyani [21] proposed a strategy for reducing the communication overhead of maintaining vector timestamps. Rodrigues and Veríssimo [18] focus on providing causal delivery in arbitrary network with the help of causal separators. In particular, similar to the vertex cover used in our work, they utilize the common nodes used in the routes of different messages to assign timestamps. However, since they only need to focus on causal delivery of messages, they do not need to timestamp all events in the system. Shen et al. [20] encode of a vector clock of length n using a single integer that has powers of n distinct prime numbers as factors. Torres-Rojas and Ahmad propose constant size logical clocks that trade-off clock size with the accuracy with which happened-before relation is captured [22]. Meldal et al. [17] propose a scheme that helps determine causality between two messages *sent to the same process*. Because their timestamps do not need to capture the happened-before relation between *all events*, the timestamps can be smaller. Some of the algorithms by Meldal et al. [17] exploit information about the paths over which messages may be propagated.

Synchronous messages [10, 11]: For synchronous messages, the sender process, after sending a message, must *wait* until it receives an acknowledgement from the receiver process, as illustrated in Figure 3. This constraint is exploited in [10, 11] to design small timestamps. In particular, if the communication network formed by the processes is decomposed into, say, d components that are either *triangles* or *stars*, then the timestamps contain $d+4$ integer elements. Due to the synchronous nature of communication, messages within each component are totally ordered. The timestamps in [10, 11] exploit this total ordering, such that the j-th element of a vector included in the timestamp for an event represents the number of messages within the j-th component (of the decomposition) that happened before the given event.

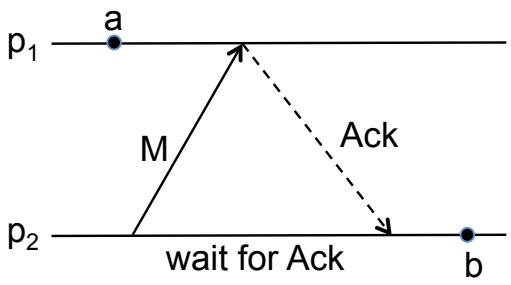

Figure 3: Synchronous message

Our timestamping algorithm does *not* constrain the messages to be synchronous. Our approach has some similarities to [10, 11] and also some key differences. In our case, the timestamp contains $2|\mathcal{VC}|+2$ integer elements, where $\mathcal{VC}$ is a vertex cover of the communication network formed by the processes. Thus, the timestamps contain more elements because we allow the flexibility of using asynchronous messages. A consequence of allowing asynchronous messages is that the *mpost* field becomes permanent only after the process communicates (and hears from) every process in $\mathcal{VC}$ to which it is connected. Given a vertex cover $\mathcal{VC}$, the network can be decomposed into $|\mathcal{VC}|$ stars. However, our algorithm does not utilize the decomposition as such (but instead uses the knowledge of a cover set $\mathcal{VC}$). On the other hand, the algorithm in [10, 11] explicitly uses the decomposition into triangle and stars.

The *mpost* field in our timestamp for events outside $\mathcal{VC}$ includes an index for the receive event of one message sent to each of the processes in the vertex cover. Thus, the *mpost* field includes $|\mathcal{VC}|$ elements. On the other hand, the timestamps in [10, 11] include just 1 index that has functionality analogous to one of the elements in our *mpost* field. This index in the timestamp in [10, 11] counts messages in a component of the edge decomposition, whereas in our case, the index counts number of events at a process. These distinctions are caused by the restriction of synchronous messages in [10, 11], and allowance for asynchronous messages in our scheme.

Causal memory [1, 13, 25]: The purpose of the timestamps used in the work on causal memory is to ensure causal consistency. There are close similarities between our timestamps and comparable objects maintained in some causal memory schemes [1, 13, 25].

In *Lazy Replication* [13], a client-server architecture is used to implement causally consistent shared memory. Each server maintains a copy of the shared memory. Each client sends its updates and queries to one of the servers. A server that receives an update from one of the clients then propagates the update to the other servers. Each server maintains a vector clock: the i-th entry of the vector at the j-server essentially counts the number of updates propagated to the j-th server by the i-th server. Each client also maintains a similar vector: the i-th element of the client's vector counts the number of updates propagated by the i-th server on which the client's state depends. Additionally, when a client sends its update to the j-th server, the j-th element of the client's vector is updated to the index of the client's update at the j-th server. The client may potentially send the same update to multiple servers, say, j-th and k-th servers; in this case, the j-th and k-th elements of the client's vector will be updated to the indices of the client's updates at the respective servers. Finally, a server cannot process an update or a query from a client until the server's vector clock is $\geq$ the vector at the client. The $\geq$ operator here performs an element wise comparison of the vector elements: vector $v \geq w$ only if each element of v is $\geq$ the corresponding element of vector w.

The mechanism used in *SwiftCloud* [25] is motivated by *Lazy Replication* [13], and has close similarities to the vectors in [13]. In *SwiftCloud*, if a client sends its update to multiple servers, then the indices returned by the servers are *merged* into the dependency vector maintained by the client (optionally, some of the returned indices may not be merged). Importantly, a server can only respond to future requests from the client provided that the server's vector covers the client's dependency vector. *Swiftcloud* uses objects with elements comparable to *pre* and *post* in our timestamps, but relies on the assumption that the servers are completely connected.

Beyond some small differences in how the timestamp comparison is performed, the other difference between shared memory schemes above and our solution is that the above schemes rely on a set of *servers* through which the processes interact with each other.

Thus, the communication network in their case is equivalent to a clique of servers to which the clients are connected. The size of the timestamps is a function of the number of servers. On the other hand, we allow arbitrary communication networks, with the size of the timestamps being a function of a *vertex cover* for the communication network. The vertex cover is not necessarily completely connected. Secondly, dependencies introduced through *events* happening at the servers (e.g., receipt of an update from a client) in the shared memory systems are not necessarily *true* dependencies. For instance, suppose that process p_0 propagates update to variable x to replica R, then process p_1 propagates update to variable y to replica R, and finally process p_2 reads updated value of y from replica R. In the shared memory dependency tracking schemes above, the update of x by p_0 would be treated as having happened-before the read by p_2. In reality, there is no such causal dependency. But the dependency is introduced artificially as a cost of reducing the timestamp size. On the other hand, in the message-passing context, if the communication network reflects the communication channels used by the processes, then no such artificial dependencies will arise. However, in the message-passing case as well, we can introduce artificial dependencies by disallowing the use of certain communication channels in order to decrease the vertex cover size. This is illustrated below through the example in Figure 4.

In [1], authors maintain a dotted vector (d, v) where v is a version vector and d is a *dot* that has $O(1)$ size, and the value of d is utilized in evaluating causality relation in $O(1)$ time. However, to capture causality, they need to maintain several version vectors to capture possible concurrent operations by multiple clients. In our work, one such vector is maintained by processes in $\mathcal{VC}$, the vertex cover and two vectors by processes outside $\mathcal{VC}$.

The size of the timestamps in above causal memory schemes is a function of the number of servers (that are completely connected to each other). On the other hand, we allow arbitrary communication networks, with the size of the timestamps depending on *vertex cover* size for the communication network. This enable alternative implementations. For instance, consider a client-server architecture, wherein a large number of clients may interact with a large number of servers. Due to the dense communication (or interconnection) pattern in this case, the cover size will be large, resulting in large timestamps. An alternative is illustrated in Figure 4, where the solid edges represent an abstract communication network. A *client* or a *server* may communicate with multiple *sequencers*. By design, the *sequencers* form a cover of this network. When the number of servers is much larger than the number of sequencers, this approach can result in a much smaller vertex cover. In Figure 4, all communication must go through the sequencers, and the *inline* timestamp size is proportional to number of sequencers. However, routing all server communication via sequencers can be expensive, since the sequencers will have to handle a large volume of data. A simple optimization can mitigate this shortcoming. For example, as shown by a dashed arrow in Figure 4, server R_1 may send message contents (data) *directly* to server R_2, but server R_2 will need to wait to receive *metadata*, in the form of timestamp information, via sequencer S_1 (as shown by a dotted arrow in the figure). Thus, while the sequencers must still handle small messages to help determine inline timestamps, bulk of the traffic can travel between the servers directly (or similarly between servers and clients). A similar optimization was suggested previously for *totally-ordered multicast* using a sequencer [4]. This optimization, in conjunction with our scheme, provides a trade-off between timestamp size and the delay incurred in routing metadata through sequencers.

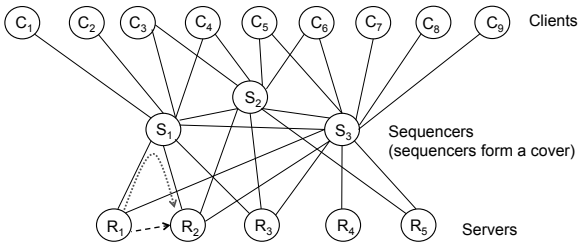

Figure 4: Alternate architecture

Exploiting Physical Time: In [12], authors have focused on a weaker version of causality that only focuses on causal relation introduced in the recent past and assume that events that occurred beyond a window of ϵ, a configurable parameter, are already reflected. In [5, 6], authors utilize physical time to ensure that servers in a data center are aware of updates on other servers on the same replica. In our work, we assume that the system is asynchronous and do not utilize synchrony assumptions associated with physical time. Also, in [5, 6], it is possible that a version is not visible to a client even though no dependency exists.

Cluster timestamps: Ward and Taylor [24] describe an improvement over the strategy previously proposed by Summers, which divides the processes into clusters. They maintain short timestamps (proportional to cluster size) for events that occur *inside* the cluster, and longer timestamps (vectors with length equal to total number of processes) for "cluster-receive" events. In [24], the "cluster-receive" events are assigned long timestamps; such long timestamps are not necessary in our case.

6 APPLICATIONS

Inline timestmaps considered in this paper are useful for solving typical problems in distributed systems for which vector clocks are used. Examples of such applications include predicate detection [3], checkpointing and recovery, detection of concurrent updates, and so on. Unlike offline timestmaps that defer assignment of timestamps until the computation is complete, inline timestamps assign timestamps after the process has completed round trip communication with all processes in $\mathcal{VC}$ to which it is connected.

The basic idea to apply inline timestamps is to consider a cut of the system that removes all events e such that $timestamp_e = \perp$. When we remove event e, we must also remove event f such that e happened before f to ensure that the cut is consistent. Once such events are removed, we have a consistent cut of the system where all timestamps are finalized and, hence, we can utilize them in the same way as we can use online timestamps. The only difference is that the analysis will focus on an earlier cut of the system. For example, predicate detection would only be feasible by considering the events in this cut. However, as timestamps get finalized the cut in which predicates can be detected will grow. Thus, if the

predicate of interest becomes true, it would be detected eventually. Additional details of these applications are available in [23].

7 CONCLUSION AND FUTURE WORK

In this paper, we focused on the following problem: If we delay assigning the timestamp to an event, then is it possible to reduce the size of the timestamps necessary to deduce causality? Another related question is: Is this delay necessary to achieve the desired reduction in the size of timestamps? With this motivation, we introduced the notion of *Inline* timestamps that delay assignment of timestamps to events. They generalize *Online* timestamps that require that this delay be 0 and *Offline* timestamps that allow this delay to be ∞. In our work, we consider the case where we have a vertex cover $\mathcal{VC}$ such that any message is sent either from a process in $\mathcal{VC}$ or to a process in $\mathcal{VC}$. The timestamps at processes in $\mathcal{VC}$ are finalized immediately. Moreover, the timestamps at other processes are finalized after they communicate with all processes in $\mathcal{VC}$ to which they are connected. Thus, introducing a delay in finalizing timestamps can reduce the size of the timestamp substantially. Specifically, we show that it is possible to assign timestamps with $2|\mathcal{VC}| + 2$ entries for *Inline* timestamps. By contrast, timestamps of size n (for integer valued timestamps) or $n - 1$ (for real valued timestamps) are necessary even for a star graph. We demonstrate this by generalizing the result by Charron-Bost [2] for the case of star graphs when *Online* timestamps are required. By contrast, for star graph where $|\mathcal{VC}| = 1$, *Inline* timestamps of size 4 suffice for any number of processes.

Although there is a delay in assigning timestamps, these timestamps can be used in many applications that utilize vector clocks. This is due to the fact that in long running applications, we anticipate that the timestamps for most events would already be finalized; only timestamps of recent events are expected to change. We can focus on solving the problem at hand by only considering events whose timestamps are finalized. In [23], we have shown how the timestamps proposed in this work can be used to perform predicate detection or rollback recovery.

Acknowledgements: Kulkarni is supported in part by NSF CNS 1329807, NSF CNS 1318678, and XPS 1533802. Vaidya is supported in part by NSF 1409416 and Toyota InfoTechnology Center USA. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

The authors thank Vijay Garg, Ajay Kshemkalyani and Jennifer Welch for their feedback. We thank Marek Zawirski for answering questions about SwiftCloud [25].

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. 2014. *Scalable and Accurate Causality Tracking for Eventually Consistent Stores.* Springer Berlin Heidelberg, Berlin, Heidelberg, 67–81. DOI: http://dx.doi.org/10.1007/978-3-662-43352-2_6

[2] Bernadette Charron-Bost. 1991. Concerning the Size of Logical Clocks in Distributed Systems. *Inf. Process. Lett.* 39, 1 (1991), 11–16. DOI: http://dx.doi.org/10.1016/0020-0190(91)90055-M

[3] Himanshu Chauhan, Vijay K. Garg, Aravind Natarajan, and Neeraj Mittal. 2013. A Distributed Abstraction Algorithm for Online Predicate Detection. In *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013.* IEEE Computer Society, 101–110. DOI: http://dx.doi.org/10.1109/SRDS.2013.19

[4] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company, USA.

[5] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13).* ACM, New York, NY, USA, Article 11, 14 pages. DOI: http://dx.doi.org/10.1145/2523616.2523628

[6] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing.* ACM, 1–13.

[7] C. J. Fidge. 1989. Partial Orders for Parallel Debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices* 24, 1 (Jan. 1989), 183–194.

[8] Colin J. Fidge. 1991. Logical Time in Distributed Computing Systems. *IEEE Computer* 24, 8 (1991), 28–33. DOI: http://dx.doi.org/10.1109/2.84874

[9] Vijay K. Garg and Chakarat Skawratananond. 2001. String realizers of posets with applications to distributed computing. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, Newport, Rhode Island, USA, August 26-29, 2001.* 72–80. DOI: http://dx.doi.org/10.1145/383962.383988

[10] Vijay K. Garg and Chakarat Skawratananond. 2002. Timestamping Messages in Synchronous Computations. In *ICDCS.* 552–559. DOI: http://dx.doi.org/10.1109/ICDCS.2002.1022305

[11] Vijay K. Garg, Chakarat Skawratananond, and Neeraj Mittal. 2007. Timestamping messages and events in a distributed system using synchronous communication. *Distributed Computing* 19, 5-6 (2007), 387–402. DOI: http://dx.doi.org/10.1007/s00446-006-0018-5

[12] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *International Conference on Principles of Distributed Systems.* Springer, 17–32.

[13] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.* 10, 4 (1992), 360–391. DOI: http://dx.doi.org/10.1145/138873.138877

[14] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[15] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms.* North-Holland, 215–226.

[16] F. Mattern. 1989. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms.* Elsevier Science Publishers B.V. (North-Holland), 215–226.

[17] Sigurd Meldal, Sriram Sankar, and James Vera. 1991. Exploiting Locality in Maintaining Potential Causality. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991.* 231–239. DOI: http://dx.doi.org/10.1145/112600.112620

[18] Luís E. T. Rodrigues and Paulo Veríssimo. 1995. Causal Separators for Large-Scale Multicast Communication. In *ICDCS.*

[19] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (1994), 149–174. DOI: http://dx.doi.org/10.1007/BF02277859

[20] Min Shen, Ajay D. Kshemkalyani, and Ashfaq A. Khokhar. 2013. Detecting Unstable Conjunctive Locality-Aware Predicates in Large-Scale Systems. In *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013.* 127–134. DOI: http://dx.doi.org/10.1109/ISPDC.2013.25

[21] Mukesh Singhal and Ajay D. Kshemkalyani. 1992. An Efficient Implementation of Vector Clocks. *Inf. Process. Lett.* 43, 1 (1992), 47–52. DOI: http://dx.doi.org/10.1016/0020-0190(92)90028-T

[22] Francisco J. Torres-Rojas and Mustaque Ahamad. 1999. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. *Distrib. Comput.* 12, 4 (Sept. 1999), 179–195. DOI: http://dx.doi.org/10.1007/s004460050065

[23] Nitin H. Vaidya and Sandeep S. Kulkarni. 2016. Efficient Timestamps for Capturing Causality. *CoRR* abs/1606.05962 (2016). http://arxiv.org/abs/1606.05962

[24] Paul A. S. Ward and David J. Taylor. 2001. Self-Organizing Hierarchical Cluster Timestamps. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing (Euro-Par '01).* Springer-Verlag, London, UK, UK, 46–56. http://dl.acm.org/citation.cfm?id=646666.699445

[25] M. Zawirski, N. Preguica, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. 2015. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *ACM Middleware, Vancouver.* https://pages.lip6.fr/Marc.Shapiro/papers/write-fast-read-past-middleware-2015.pdf