Stabilizing Causal Deterministic Merge

Sandeep S. Kulkarni Ravikant

Software Engineering and Networks Laboratory
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

We present a causal deterministic merge program for publish-subscribe systems. Our program ensures that if two subscribers receive two messages then they receive them in the same order. Also, it guarantees that the order in which a subscriber receives messages is a linearization of the causal order among those messages. To develop our program, we expect two guarantees from the underlying system: the first guarantee deals with the difference between physical clocks and the second guarantee deals with message delays. While $O(n^2)$ space is required for causal delivery of unicast messages in asynchronous systems, our program only uses $O(log\ n)$ space. We also show how our program can be made stabilizing while using only a bounded space. And, the recovery time for our program is proportional to the guarantees made by the underlying system.

Keywords : logical timestamps, causal delivery, deterministic merge, clock synchronization, publish-subscribe systems

1 Introduction

We focus our attention on the problem of causal deterministic merge in a publish-subscribe network. A publish-subscribe network consists of a set of *publishers* that publish messages and a set of *subscribers* that subscribe to a subset of those messages. We assume that the subscribers wish to receive messages in an uniform total order that conforms to the causal order. The uniform total order requires that if two subscribers receive two messages, then they receive them in the same order. The causal order requires that if a subscriber receives messages m_1 and m_2 such that m_2 causally depends on m_1 then that subscriber delivers m_1 before delivering m_2. The causal dependency may occur through published messages or through internal communication among publishers. Such causal order delivery is desirable in several applications. For example, consider a scenario where a

[1] Email: {sandeep, ravikant}@cse.msu.edu
Web: `http://www.cse.msu.edu/~{sandeep,ravikant}`
Tel: +1-517-355-2387, Fax: 1-517-432-1061

This work is partially sponsored by NSF CAREER 0092724, ONR grant N00014-01-1-0744, and a grant from Michigan State University.

publisher *publishes* a question and another publisher *publishes* its answer. It is desirable that the subscriber receives the question before receiving the answer.

An uniform total order that conforms to the causal order can be obtained by using a centralized solution. However, a centralized solution lacks scalability and reliability. To redress this, we use replicated mergers (as proposed in [1]). Each merger is associated with a set of subscribers. All published messages are sent to the (relevant) mergers. The mergers reorder the messages so as to obtain a causal deterministic order. The deterministic order guarantees that if two mergers receive overlapping messages, they order those overlapping messages consistently. Moreover, the causal order guarantees that if m_2 causally depends on m_1 then m_1 will be ordered before m_2. The reordered messages are then sent to the subscribers.

We build these mergers by extracting two simple guarantees expected of the underlying distributed system on which the publish-subscribe network is built. The first guarantee is related to the clocks: we require that the system provide a bound, say ϵ, such that the clock drift between different processes (that implement the publishers, subscribers and mergers) in the system is bounded by ϵ. This guarantee can be met by using GPS clocks, network time protocol, atomic clocks or clock synchronization programs. The second guarantee relates to the message delays: we require that messages that reach their destination do so within some bound, say δ. This guarantee can be met by using protocols that characterize messages as being timely or late.

Our solution uses bounded space and provides stabilizing fault-tolerance in the presence of faults. It uses $O(\epsilon log\ n\ +\ log\ \delta)$ space and recovers in $O(\epsilon+\delta)$ time from an arbitrary state (that could be reached in the presence of faults such as message corruption, transients, temporary violation of system guarantees). However, for a given system, the space used by our program is bounded in that it does not grow as the computation progresses.

We decompose the problem of causal deterministic merge into two parts: (1) how to design logical timestamps that capture causality among messages, and (2) how to use the logical timestamps to obtain an uniform total order that conforms to the causal order. Specifically, in the first part, we develop bounded-space and stabilizing logical timestamps. In the second part, we use the logical timestamps to order messages in a uniform total order that conforms to the causal order.

To develop our solution for logical timestamps, we begin with the following observation: *Had the underlying system provided a global clock ($\epsilon=0$), the physical clock alone would have been sufficient to implement the logical timestamps.* We, therefore, develop logical timestamps that consist of the physical clock value and some additional information that is proportional to ϵ and δ. Also, the guarantees given by the distributed system are used to add stabilizing tolerance; the resulting implementation, thus, ensures that even if the logical timestamp values are perturbed, eventually they are restored so that the causality is tracked correctly.

Contributions of the paper. The contributions of this paper are as follows: (1) We present a bounded and stabilizing solution to logical timestamps. The space cost for our program is $O(\epsilon log\ n\ +\ log\ \delta)$ and the recovery time for

our program is $O(\epsilon+\delta)$. (2) Using the bounded and stabilizing logical timestamps, we present a stabilizing implementation of causal deterministic merge. If we concentrate only on one merger, the causal deterministic merge program also serves as a causal delivery program. The state space for our program grows logarithmically in the number of processes. By contrast, the space cost for previous programs (e.g., [2,3]) is quadratic in nature. Our solution guarantees that if a message arrives at its destination within δ time, it will be delivered within $\delta+3\epsilon$. Thus, the resulting system is one where the clocks differ by ϵ and the messages arrive within time $\delta+3\epsilon$ or are lost. Note that these assumptions are similar to the assumptions of the underlying system (with δ replaced by $\delta+3\epsilon$). We find that this observation simplifies the task of designing causal deterministic merge in hierarchical systems. (3) We present extensions of our program where we identify simple conditions to reduce the space complexity further. With these conditions, the space requirement is independent of the number of processes.

Organization of the paper. In Section 2, we formally specify our system model, guarantees expected from the underlying system, and the types of faults. Then, in Section 3, we present our solution to logical timestamps. We use the solution in Section 3 to develop our causal deterministic merge program in Section 4. In Section 5, we discuss extensions of our program, and identify a simple condition under which the message size could be made independent of the number of processes. In Section 6, we present the role of our model in each of these solutions. Finally, we make concluding remarks and point out future directions in Section 7. For reasons of space, we refer the reader to [4] for proofs.

2 System Model

A distributed system consists of a finite set of processes which communicate via message passing. A process is used to implement a publisher, subscriber or a merger. Each process j has access to a physical clock $rt.j$. We do not assume any relation between $rt.j$ and the auxiliary global time, i.e., we do not assume that any process is aware of the 'real' time.

We assume that in the absence of faults, the distributed system satisfies the following two guarantees, and in the presence of faults, these guarantees may be violated only temporarily. (We treat the above guarantees as assumptions that system users can make. Hence, depending upon the context, we use the word assumption instead of guarantee.)

Guarantees of the distributed system.

*G*1. The value of $rt.j$ is non-decreasing, and at any time, the difference between the clock values of any two processes is bounded by ϵ. In other words,

$$\forall j,k :: |rt.j - rt.k| \leq \epsilon$$

*G*2. Let m_j be a message sent by process j to k. Also, let st_m denote the clock value of j when j sent m_j, and let rd_m denote the clock value of j when k received m_j. We require that k should receive m_j within time δ unless m_j is lost. In other words,

$$\forall m :: ((rd_m \leq (st_m+\delta)) \quad \vee \quad rd_m = \infty)$$

Notation. A distributed system instantiated with parameters ϵ and δ is denoted as $ds(\epsilon, \delta)$.
Remark. We assume that time is discrete and the value of ϵ is an integer; ϵ equals max clock drift/clock precision, where max clock drift is the real value denoting the clock difference and the clock precision equals the minimum time difference between events. The issue of fine-tuning ϵ, δ is discussed in Section 6. Execution of a process consists of a sequence of events; an event can be a local event, a send event, or a receive event. In a local event, the process neither receives a message nor sends a message. In a send event, the process sends one or more messages, and in a receive event, the process receives one or more messages. For simplicity, we assume that, for one clock tick of j, at most one event is created at process j. (Note that this assumption can be easily weakened so that at most K events are generated for each clock tick, where K is any constant.)
Notation. In this paper, we use i, j, k, and l to denote processes. We use e and f to denote events. Where needed, events are subscripted with the process at which they occur, thus, e_j is an event at j. We use m to denote messages. Messages are subscripted by the sender process, thus, m_j is a message sent by j.
Fault Model. In our fault-model, messages can be corrupted, lost, or duplicated. Moreover, processes could be improperly initialized, and channels may contain garbage messages in the initial state. The state of processes could be transiently (and arbitrarily) corrupted at any time. Also, the guarantees made by the system ($G1$ and $G2$) may be temporarily violated. We assume that the number of fault occurrences is finite. Our solutions are stabilizing fault-tolerant in the presence of these faults, where
Definition. A program is *stabilizing fault-tolerant* iff starting from an arbitrary state, it eventually recovers to a state from where its specification is satisfied.

3 Logical Timestamps

In this section, we present our bounded-state, stabilizing solution for logical timestamps. Towards this end, we first precisely define the problem of logical timestamps in Section 3.1. Then, we provide an implementation of the logical timestamps in Section 3.2. Finally, we show how that implementation can be made stabilizing in Section 3.3.

3.1 Problem Statement

Towards defining the problem of logical timestamps, we first define the causal relation [5], $\longrightarrow$, among events. Then we introduce a definition which will be used in the problem statement.
Happened-before. The happened before relation [5] is the smallest transitive relation that satisfies, for any two events e, f, $e \longrightarrow f$ if (1) e and f are events on the same process and e occurred before f, or (2) e is a send event in one process and f is the corresponding receive event in another process. □
Notation. Let ts be a type, and let $ts.e$ and $ts.f$ be values of type ts. Then, $less(ts, ts)$ is a function that takes two arguments of type ts and returns a boolean.

Definition. Let ts be a type. A function $less(ts, ts)$ is well-formed iff it is irreflexive and asymmetric.
The problem of logical timestamps can now be defined as follows.

Specification of logical timestamps. Identify a type ts, a well-formed relation $less(ts, ts)$, and assign a timestamp of type ts to each event in the given program computation such that for any two events e and f with timestamps $ts.e$ and $ts.f$ the following condition is true:

- $e \longrightarrow f \quad \Rightarrow \quad less(ts.e, ts.f)$ □

Remark. In Lamport's scalar clock implementation, ts is instantiated to be $integer$, and $less(ts.e, ts.f)$ iff $ts.e < ts.f$. The vector clock implementation by Fidge [6] and Mattern [7] can also be viewed as solving the logical timestamp problem provided we instantiate ts to be an array of integers, and define $less(ts.e, ts.f)$ to be true iff $((\forall k :: ts.e[k] \leq ts.f[k]) \wedge (\exists k :: ts.e[k] < ts.f[k]))$. Note that $less$ is not a total relation in that for events e and f, it is possible that both $less(ts.e, ts.f)$ and $less(ts.f, ts.e)$ are false. However, both of them cannot be true.

3.2 Solution to Logical Timestamps

We propose that the timestamp of event e_j be of the form $\langle r.e_j, c.e_j, kn.e_j \rangle$ where $r.e_j$ denotes the physical clock value of j when e_j was created. The variable $c.e_j$ is used to capture the knowledge that j had about the maximum clock value in the system when e_j was created. Specifically, $c.e_j$ equals the difference between the maximum clock value in the system that j is aware of when e_j was created and $r.e_j$. The variable $kn.e_j$ is an array of size 2ϵ. The variable $kn.e_j[t]$, $-\epsilon \leq t < \epsilon$, is used to capture the knowledge about the number of events f such that $r.f = r.e_j + t$ and $f \longrightarrow e_j$. (We maintain $kn.j[t]$ only for $t < \epsilon$ because j cannot learn of events whose timestamp is at least $rt.j + \epsilon$. To see this, observe that when $rt.j = x$, the maximum clock value in the system is $x + \epsilon$. However, any message sent with timestamp $x + \epsilon$ can be received at j only when $rt.j$ is incremented.)
Our logical timestamp program is as follows: Each process j maintains $rt.j$, $r.j$, $c.j$ and the array $kn.j$. The variable $rt.j$ is the physical time at j, and $\langle r.j, c.j, kn.j \rangle$ is the timestamp of the last event on j. We assume that the first event is created on each process when its physical clock value is 0. Hence, we initialize $rt.j$, $r.j$, $c.j$ to be 0. Also, we initialize $kn.j[0]$ to be equal to 1 and all other elements in $kn.j$ to be 0. The variable $rt.j$ is updated by the underlying system that ensures that $G1$ is satisfied. The logical timestamp protocol can only read $rt.j$.
We ensure that $r.j + c.j$ equals the knowledge that j has about the maximum clock value in the system. Consider the case where j creates a local event e_j at time $rt.j$. Note that $r.j + c.j$ equals the maximum clock value that j was aware of when it created the last event. If $r.j + c.j \geq rt.j$ then it implies that $r.j + c.j$ is still the maximum clock value that j is aware of. In that case, we set $c.j$ to be equal to $r.j + c.j - rt.j$. If $r.j + c.j \leq rt.j$ then it implies that the maximum clock

value that j is aware of is the same as $rt.j$. Hence, we set $c.j$ to 0. We update kn based on the previous value of $kn.j$. Then, we increase $kn.j[0]$ to capture the fact that j is aware of one extra event at time $rt.j+0$. In the send event, $r.e_j$, $c.e_j$ and $kn.e_j$ are updated in the same way and the message carries the timestamp $\langle r.e_j, c.e_j, kn.e_j \rangle$. When j receives message(s), it updates r, c and kn in the same way except that it does the update based on the previous event at j and the event(s) corresponding to the sending of (those) message(s). Thus, the program is as shown in Figure 1. (While presenting the program, for simplicity of presentation, we assume that $kn.j[t]$ equals 0 if $t < -\epsilon$ or $t \geq \epsilon$.)

Initially:
$rt.j, r.j, c.j = 0, \forall t : t \neq 0 : kn.j[t] = 0, kn.j[0] = 1$

Local event e_j/Send event e_j (message being sent is m_j)
$c.j := max(0, r.j + c.j - rt.j)$
$\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := kn.j[t + rt.j - r.j]$
$kn.j[0] := kn.j[0] + 1$
$r.j := rt.j$
$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$
if e_j is a send event then $r.m_j, c.m_j, kn.m_j := r.j, c.j, kn.j$

Receive event e_j (message m received with timestamp $\langle r.m, c.m, kn.m \rangle$)
$c.j := max(0, r.j + c.j - rt.j, r.m + c.m - rt.j)$
$\forall t : -\epsilon \leq t < \epsilon : kn.j[t] := max(0, kn.j[t + rt.j - r.j], kn.m[t + rt.j - r.m])$
$kn.j[0] := kn.j[0] + 1$
$r.j := rt.j$
$r.e_j, c.e_j, kn.e_j := r.j, c.j, kn.j$

Fig. 1. Logical Timestamp Program

For the program in Figure 1, the following lemmas are true in the absence of faults.

Lemma 3.1
$\forall e :: kn.e[c.e] > 0$
$\forall m :: kn.m[c.m] > 0$
$\forall e, t : c.e < t < \epsilon : kn.e[t] = 0$
$\forall m, t : c.m < t < \epsilon : kn.m[t] = 0$ □

Lemma 3.2 $\forall e :: c.e < \epsilon$. □

Lemma 3.3 $\forall e, t : -\epsilon \leq t < \epsilon : kn.e[t] \leq |\{e\} \cup \{f : f \longrightarrow e \wedge r.f = r.e + t\}|$. □

Lemma 3.4 Given a system with n processes, $\forall e, t : : -\epsilon \leq t < \epsilon : kn.e[t] \leq n$ □

Remark. We have assumed that the minimum delay in message transmission is 0 and, hence, messages could be received instantaneously. However, if the underlying system ensures that there is a minimum delay δ_{min} in transmission of messages, we need to maintain $kn.j[t]$ only for $-(\epsilon - \delta_{min}) \leq t < (\epsilon - \delta_{min})$. In this case, the number of elements in the array $kn.j$ are reduced to $2(\epsilon - \delta_{min})$. Finally, when $\delta_{min} \geq \epsilon$, there is no need to maintain the array. (We would like to

point out that Lamport had made the observation for the case where $\delta_{min} \geq \epsilon$ in [5].)

Comparing timestamps. Given two events e_j and f_k with timestamps $\langle r.e_j, c.e_j, kn.e_j \rangle$ and $\langle r.f_k, c.f_k, kn.f_k \rangle$, we first use the r and c values to determine the truth value of $less(\langle r.e_j, c.e_j, kn.e_j \rangle, \langle r.f_k, c.f_k, kn.f_k \rangle)$. As mentioned above, $r.e_j + c.e_j$ captures the maximum clock value that j was aware of when e_j was created. Thus, if $rt.e_j + c.e_j < rt.f_k + c.f_k$, we can safely decide $less(\langle r.e_j, c.e_j, kn.e_j \rangle, \langle r.f_k, c.f_k, kn.f_k \rangle)$ to be true. Finally, if $rt.e_j + c.e_j = rt.f_k + c.f_k$ then we use the array kn to determine the truth value of $less$ $(\langle r.e_j, c.e_j, kn.e_j \rangle, \langle r.f_k, c.f_k, kn.f_k \rangle)$. Towards this end, we use a lexicographical comparison. First, we compare $kn.e[c.e]$ with $kn.f[c.f]$. If these two values are unequal then they determine the truth value of $less(\langle r.e_j, c.e_j, kn.e_j \rangle$, $\langle r.f_k, c.f_k, kn.f_k \rangle)$. If $kn.e[c.e]$ equals $kn.f[c.f]$ then we compare $kn.e[(c.e)-1]$ and $kn.f[(c.f)-1]$, and so on. Moreover, as shown in Theorem 3.6, comparing only ϵ elements in kn is sufficient. Thus, we define relation $less$ as follows:

$$less(\langle r.e, c.e, kn.e \rangle, \langle r.f, c.f, kn.f \rangle)$$

iff

$$(r.e + c.e < r.f + c.f) \quad \vee$$
$$((r.e + c.e = r.f + c.f) \wedge lexgraph(kn.e, c.e, kn.f, c.f, \epsilon))$$

where

$$\begin{array}{lllll} lexgraph(kn1, c1, kn2, c2, n) = & \text{if} & (n = 0) & \text{then} & false \\ & \text{elseif} & kn1[c1] < kn2[c2] & \text{then} & true \\ & \text{elseif} & kn1[c1] > kn2[c2] & \text{then} & false \\ & \text{else} & lexgraph(kn1, c1-1, kn2, c2-1, n-1) & & \end{array}$$

If for two events e_j and f_k, $\neg less(\langle r.e_j, c.e_j, kn.e_j \rangle, \langle r.f_k, c.f_k, kn.f_k \rangle)$ and $\neg less(\langle r.f_k, c.f_k, kn.f_k \rangle, \langle r.e_j, c.e_j, kn.e_j \rangle)$, then the events are ordered based on their process ID.

Note that kn values are compared only when $r.e + c.e$ equals $r.f + c.f$. Thus, $kn.f[c.f]$ is compared with $kn.e[r.f + c.f - r.e]$ $(= kn.e[c.e])$. More generally, $kn.f[t]$ is compared with $kn.e[r.f + t - r.e]$. This is due to the fact that $kn.f[t]$ denotes the knowledge about events at $r.f + t$. Likewise, $kn.e[r.f + t - r.e]$ denotes the knowledge about events at $r.e + r.f + t - r.e (= r.f + t)$.

For the above relation, we first make the following observation and then prove that the program in Figure 1 satisfies the specification of logical timestamps.

Observation 3.5 The $less$ relation given above is transitive and well-formed. □

Theorem 3.6 $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e, kn.e \rangle, \langle r.f, c.f, kn.f \rangle)$

Proof. We prove this by induction. Initially, for any two events, e, f, $e \longrightarrow f$ is false. Hence, the theorem is trivially true. Now, consider the case where a new event, say f, is created at some process, say j.

1. f is a send/local event:
   Let e be the event that occurred at j just before f. From the assignment to $c.f$, we have: $r.e + c.e \leq r.f + c.f$. If $r.e + c.e < r.f + c.f$ then it follows that

$less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is true. Hence, we consider the case where (a) $r.e{+}c.e = r.f{+}c.f$. In this case, based on how the program updates $kn.f$, we have (b) $\forall t : -\epsilon \leq t < \epsilon \wedge \ -\epsilon \leq t{+}r.f{-}r.e < \epsilon : \ kn.e[t{+}r.f{-}r.e] \leq kn.f[t]$. Also, since $kn.f[0]$ is increased by the program, we have (c) $kn.e[r.f - r.e] < kn.f[0]$.

In evaluating $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$, we first compare $kn.f[c.f]$ with $kn.e[c.e]$. Then, we compare $kn.f[c.f-1]$ with $kn.e[c.e-1]$, and so on. When we compare the $c.f+1$ elements, the truth value of $less$ will be determined since $kn.f[c.f{-}c.f]$ is greater than $kn.e[c.e{-}c.f]$ $(= kn.e[r.f{-}r.e])$. If the truth value of $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is determined before we compare $kn.f[c.f{-}c.f]$ with $kn.e[c.e{-}c.f]$, from (b), $less(\langle r.e, c.e, kn.e\rangle$, $\langle r.f, c.f, kn.f\rangle)$ must be true. Moreover, since $c.f$ is less than ϵ at most ϵ elements in kn are compared. Finally, from (a), (b) and (c), it follows that $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ is true.

Now for any event a, $a \neq e$, $a \longrightarrow e$ iff $a \longrightarrow f$. Moreover, if $a \longrightarrow e$ then $less(\langle r.a, c.a, kn.a\rangle, \langle r.e, c.e, kn.e\rangle)$ is true by induction. Hence by transitivity, $less(\langle r.a, c.a, kn.a\rangle, \langle r.f, c.f, kn.f\rangle)$ is also true.

2. f is a receive event.
   This proof is similar to case 1 except that we need to consider two events e_1, the event on j just before f, and e_2, the corresponding send event. As in the previous, we show that $less(\langle r.e_1, c.e_1, kn.e_1\rangle, \langle r.f, c.f, kn.f\rangle)$ and $less(\langle r.e_2, c.e_2, kn.e_2\rangle, \langle r.f, c.f, kn.f\rangle)$ are true.
   Once again, for any event a, $(a \neq e_1 \wedge \ a \neq e_2), a \longrightarrow f$ iff $(a \longrightarrow e_1 \vee a \longrightarrow e_2)$. Hence, by transitivity, $less(\langle r.a, c.a, kn.a\rangle, \langle r.f, c.f, kn.f\rangle)$ □

Bounding the state space of logical timestamps. From Lemmas 3.2 and 3.4, it follows that the c and kn values are bounded. Also, rt values can be bounded by letting j maintain only $brt.j = (rt.j \ mod \ B)$ where B is a large enough constant. More specifically, B should be large enough so that when j receives a message m that carries $brt.m$ (instead of $rt.m$), j should be able to update its logical timestamp appropriately. When j receives message m_l from l, from $G1$, $rt.l$ is in the range $[rt.j - \epsilon..rt.j + \epsilon]$. From $G2$, when m was sent, $rt.l$ was in the range $[rt.j{-}\epsilon{-}\delta..rt.j{+}\epsilon]$. Hence, the vector $kn.m_l$ carries information about events that occurred at time $[rt.j{-}\epsilon{-}\delta{-}\epsilon..rt.j{+}\epsilon{+}\epsilon{-}1]$. It follows that if B is greater than or equal to $4\epsilon{+}\delta{+}1$, j would be able to update the logical timestamp appropriately. Hence, to bound the space used by j, we maintain $brt.j = rt.j \ mod \ B$ where $B \geq 4\epsilon{+}\delta{+}1$.

3.3 Bounding and Stabilizing Logical Timestamps

If the logical timestamps are perturbed by faults such as failure and repair of process, corrupted timestamps, temporary violation of system guarantees, we ensure that the program reaches a state from where causality is tracked correctly. The stabilization proceeds in four steps. First, if the system guarantees are violated then they are restored. Several approaches may suffice for this purpose, including clock synchronization programs, GPS clocks, network time protocol, etc. The particular approach used is not relevant for our purpose; any approach may be used.

In the second step, we satisfy the local properties in Lemmas 3.1, 3.2 and 3.4. Towards this end, for any process j, if $c.j$ is ever assigned a value that is greater than or equal to ϵ, $kn.j[t]$ is assigned a value that is greater than n, $kn.j[c.j]$ is zero or for some t, $t > c.j$, $kn.j[t]$ is nonzero, we set $c.j$ to 0, $kn.j[0]$ to 1 and all other elements of $kn.j$ to 0. Likewise, whenever a message is received we ensure that Lemmas 3.1, 3.2 and 3.4 are satisfied for that message before processing that message.

In the third step, messages with corrupt timestamps are either delivered or are lost. Consider a message m whose timestamp is corrupted after it was sent. After δ time has passed, m is delivered or lost (from $G2$). Since reception of m does not affect rt values, $G1$ continues to be true. Also, if r, c, kn values are changed in a way that local invariants are violated, step 2 restes them. Thus, the correction achieved by steps 1 and 2 is not affected and the number of messages whose timestamp is corrupted while in transit, is reduced by 1. It follows that eventually for any message m_k in transit, the timestamp of m_k is the same as the timestamp of k when m_k was sent.

Consider a state, say s, obtained after steps 1-3. Let $rt.j = x$ in state s. Thus, from $G1$, we find that for any process k, $rt.k$ is in the range $[x-\epsilon..x+\epsilon]$. Moreover, from $G2$, for any message, say m_k, in transit, m_k was sent when $rt.k$ was in the range $[x-\epsilon-\delta..x+\epsilon]$. Hence, m_k can carry information of about events whose physical time is in the range $[x-\epsilon-\delta-\epsilon..x+\epsilon+\epsilon-1]$. It follows that no process (or message) has knowledge about events that occur at physical time t where $t \geq x+2\epsilon$. In other words, if j acquires knowledge about events that occur at time t, $t \geq 2\epsilon$, it must be due to actual events rather than due to faults such as transients. However, the knowledge about events in the range $[x..x+2\epsilon-1]$ may be incorrect. Now, if $rt.j$ is unbounded and $rt.j$ is advanced to $x+3\epsilon$, the $kn.j$ vector will only maintain information about events whose physical time is in the range $[x+2\epsilon..x+4\epsilon-1]$ and, as discussed above, $kn.j$ will be consistent. After all kn values become consistent, causality will be tracked correctly.

Also, note that the time required for steps 2 and 3 is δ. And, the time required for step 4 is 3ϵ. Thus, the time required for recovery is $O(\epsilon+\delta)$.

Once again, we maintain $brt.j = (rt.j \ mod \ B)$ where B is a large enough constant. For stabilization, we increase the value of B so that j can distinguish between the different rt values that may occur during the computation where $rt.j$ is advanced from x to $x+3\epsilon$. From the above discussion, the rt values encountered in that computation are in the range $[x-2\epsilon-\delta..x+4\epsilon-1]$. Hence, we maintain $brt.j = rt.j \ mod \ B$ where $B \geq 6\epsilon+\delta+1$. Thus, we have:

Theorem 3.7 If the timestamping program in Figure 1 is modified so that the conditions specified in Lemmas 3.1, 3.2 and 3.4 are satisfied (as given in the second step above) and a variable $brt.j = rt.j \ mod \ B$ is maintained to capture $rt.j$ then the resulting program uses bounded space and is stabilizing fault-tolerant provided $B \geq 6\epsilon+\delta+1$. □

Now, we consider the state space required to implement our logical timestamps. As discussed above, the maximum value for c and brt is $O(\epsilon+\delta)$. Thus, the space required for those variables is $O(log(\epsilon+\delta))$. Each element in kn is bounded by n

and, hence, requires $O(log\ n)$ space. It follows that the maximum space required for kn is $O(\epsilon\ log\ n)$. Summing up the space requirement for brt, c and kn, we have

Theorem 3.8 The space used by the stabilizing logical timestamp program is $O(\epsilon\ log\ n\ \ +\ \ log\ \delta)$. □

4 Causal Deterministic Merge

In this section, we use the timestamp introduced in Section 3 to present our program for causal deterministic merge. We first define the problem in Section 4.1. Then, we give the bounded-space stabilizing solution in Section 4.2.

4.1 Problem Statement

The problem of causal deterministic merge requires that causal delivery is satisfied and that the messages are merged deterministically. Thus, whenever a message is received, it should be buffered until the receiving process determines the place where the message should be put while obtaining the causal deterministic merge. Whenever the appropriate place in the causal deterministic merge is determined, we *deliver* that message. Thus, the problem requires that the following two specifications are satisfied.

Specification of causal delivery.
For messages m_1 and m_2 and for process j that satisfy
    $send(m_1) \longrightarrow send(m_2)$,
    Process j is included in the destination set of m_1 and m_2, and
    m_1 and m_2 are received at j
the following two conditions are satisfied:
    m_1 is delivered before m_2, and
    m_1 and m_2 are eventually delivered

Specification of deterministic merge.
For messages m_1 and m_2 and processes j and k that satisfy
    Processes j and k are included in the destination set of m_1 and m_2, and
    Both m_1 and m_2 are received at j and k
the following condition is satisfied:
    The order in which m_1 and m_2 are delivered at j and k is the same, and
    m_1 and m_2 are eventually delivered

Note that the above problem statement requires causal deterministic merge of messages that reach the destination within δ time. In other words, messages lost in transit are ignored. Such delivery pattern is useful for applications, e.g., audio/video streams, where such message loss can be easily tolerated. Moreover, many such applications require that most of the messages be delivered in a timely fashion even though some messages are never delivered. In other words, they require that even if a message is lost, the messages that causally depend on it should not suffer excessive delay. Based on the application requirements and its ability to tolerate message loss, it is possible (cf. Section 5) to fine-tune the

value of δ: increasing the value of δ decreases the number of messages lost but increases the maximum delay that messages may incur and the amount of buffer required to obtain causal deterministic merge.

4.2 Solution

We first present the condition for causal ordering of two messages, m_1 and m_2, received at process j such that $m_1 \longrightarrow m_2$. Let us assume that m_2 is received first at j. We now find how long m_2 should wait at j before being delivered so that m_1 is received and delivered before that. From the timestamp comparison, we have $m_1 \longrightarrow m_2 \Rightarrow r.m_1 + c.m_1 \leq r.m_2 + c.m_2$. Moreover, since $c.j \geq 0$, it follows that $r.m_1 \leq r.m_2 + c.m_2$. In other words, when m_1 was sent the physical clock value of the sender process was at most $r.m_2 + c.m_2$. From $G2$, when m_1 is received at j, the clock value of the sender (of m_1) will be at most $r.m_2 + c.m_2 + \delta$. Moreover, from $G1$, the clock value of j will be at most $r.m_2 + c.m_2 + \delta + \epsilon$. So, m_1 will reach j before $rt.j$ equals $r.m_2 + c.m_2 + \delta + \epsilon$ or m_1 will be lost. Hence, to obtain causal delivery, it suffices that m_2 wait until $rt.j$ equals $r.m_2 + c.m_2 + \delta + \epsilon$. Thus, the delivery condition for message m is $delcond(m, j)$ where

$$\boxed{delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon)}$$

Causal Delivery Program. In our causal delivery program, whenever message m is received at process j, m is buffered until $delcond(m, j)$ is satisfied. As soon as $delcond(m, j)$ is satisfied, message m is delivered. If multiple messages satisfy the delivery condition simultaneously, j determines the causal relation (if any) between them using $less$ relation: given two messages m_k and m_l if $less(\langle r.m_k, c.m_k, kn.m_k \rangle, \langle r.m_l, c.m_l, kn.m_l \rangle)$ is true we deliver m_k before m_l. If both $less(\langle r.m_k, c.m_k, kn.m_k \rangle, \langle r.m_l, c.m_l, kn.m_l \rangle)$ and $less(\langle r.m_l, c.m_l, kn.m_l \rangle, \langle r.m_k, c.m_k, kn.m_k \rangle)$ are false then we deliver them based on their process ID. Observe that there is only one way to deliver these messages.

From Lemma 3.2, the value of $c.m$ is at most ϵ. Thus, message m will be delivered before $rt.j$ reaches $r.m + \delta + 2\epsilon$. From $G1$, the clock value of the sender when m is delivered is at most $r.m + \delta + 3\epsilon$. Also, if j receives m when $rt.j$ is x then $r.m$ is at least $x - \epsilon$. Thus, m can be buffered only for time $\epsilon + \delta + \epsilon + c.m$. It follows that no message is buffered for longer than $\delta + 3\epsilon$. Thus, the following theorems are true.

Lemma 4.1 If j sends message m when its physical clock value was $r.m$ then it would be delivered before the physical clock value of j reaches $r.m + \delta + 3\epsilon$. □

Lemma 4.2 A process buffers a message for at most $\delta + 3\epsilon$ time. □

Theorem 4.3 Given a system $ds(\epsilon, \delta)$ if our causal delivery program is used to deliver messages then the resulting system will be $ds(\epsilon, \delta + 3\epsilon)$ (cf. Figure 2). □

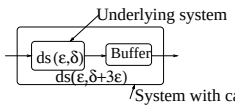

Fig. 2. Guarantees of Our Causal Delivery Program

Now, we show that the causal delivery program given above suffices for ensuring causal deterministic merge in a publish-subscribe system. Formally,

Theorem 4.4 If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered would be same on both processes. □

Theorem 4.5 If two messages m_1 and m_2 such that $send(m_1) \longrightarrow send(m_2)$, arrive at any process j then m_1 is delivered before m_2. □

The stabilization of causal deterministic merge is similar to that of the logical timestamps. Note that even if the logical timestamps of messages are corrupted, the causal delivery program never deadlocks since every message is eventually considered for delivery and when multiple messages are considered for delivery simultaneously the *less* relation (along with the tie-breaker based on the process ID) determines the order in which they should be delivered. Once the logical timestamps are restored to their legitimate states, the subsequent computation will satisfy the requirements of causal deterministic merge. Thus, we have

Theorem 4.6 The causal delivery program given above is stabilizing fault-tolerant. □

5 Extensions

In this section, we discuss extensions to our logical timestamps presented in Section 3.2 such that the array $kn.j$ is eliminated. This is achieved at the cost of maintaining unbounded value of $c.j$. Also, for a standard *less* relation, we show that $c.j$ value cannot be bounded. Subsequently, we show that if the application provides certain guarantees about creation of events in the system, then $c.j$ can also be bounded. In both cases, the timestamp is of the form $\langle r.j, c.j \rangle$. Each of these implementations can be used to solve stabilizing causal deterministic merge.

5.1 Eliminating $kn.j$ at the Cost of Unbounded c Value

In this case, we modify our logical timestamp to be of the form $\langle r.j, c.j \rangle$ where $r.j$ is still the physical clock value of the last event at j. The values of $r.j$ and $c.j$ are updated as shown in Figure 3.

Initially:
$rt.j, r.j, c.j = 0, 0, 0$

Local event e_j/Send event e_j (message being sent is m_j)
if $r.j+2\epsilon < rt.j$ then $c.j = 0$;
else $c.j := max(0, r.j+c.j-rt.j)$
$r.j := rt.j$
$r.e_j, c.e_j, r.m_j, c.m_j := r.j, c.j, r.j, c.j$

Receive event e_j (message m received with timestamp $\langle r.m, c.m \rangle$)
if $(r.m + 2\epsilon < rt.j \ \wedge \ r.j + 2\epsilon < rt.j)$ then $c.j = 0$
else $c.j := max(0, r.j+c.j-rt.j, r.m+c.m-rt.j+1)$
$r.j := rt.j$
$r.e_j, c.e_j := r.j, c.j$

Fig. 3. Revised Logical Timestamp Program

In the program in Figure 3, we first consider the case where the previous event was more than 2ϵ time apart. In that case, we reset $c.j$ to 0. Otherwise, for the send event the program remains unchanged. For the receive event, if $r.m{+}2\epsilon < r.f$ then we ensure that $r.m{+}c.m < r.f{+}c.f$ where f is the event that corresponds to receive of message m. (In the previous program, we had permitted $r.m{+}c.m \leq r.f{+}c.f$; this allowed us to bound c value at the cost of maintaining the array kn.)

For the program in Figure 3, we define the $less(ts, ts)$ relation for our timestamp of the form $\langle r.j, c.j\rangle$ as follows: Given two events e and f,

$less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$

iff

$(r.e{+}\epsilon < r.f) \quad \vee$

$((|r.e - r.f| \leq \epsilon) \wedge ((r.e{+}c.e < r.f{+}c.f) \vee ((r.e{+}c.e = r.f{+}c.f) \wedge r.e < r.f)))$

We note that stabilization can be added to the program in Figure 3 as discussed in Section 3.3.

Proof for unboundedness. Now, we show that the use of the above $less$ relation requires that the value of $c.j$ is unbounded even in a system with only 3 processes, say j, k and l. We begin in a state where $rt.j = \epsilon$, $rt.k = 0$, $rt.l = 0$ and the c values for all the processes are 0. Now, let j send a message to k. This message is sent with the timestamp $\langle \epsilon, 0\rangle$. Let this message be received when $rt.k = 1$. The receive event at k has the timestamp $\langle 1, \epsilon\rangle$. Now, let k send a message to l when $rt.k$ equals 2. This message is sent with the timestamp $\langle 2, \epsilon{-}1\rangle$. Let this message be received at l when $rt.l = 1$. The receive event at l has the timestamp $\langle 1, \epsilon{+}1\rangle$. Now, let l send a message to j when $rt.l$ equals 2. This message is sent with the timestamp $\langle 2, \epsilon\rangle$. Let this message be received at j when $rt.j = \epsilon{+}1$. The receive event at j has the timestamp $\langle \epsilon{+}1, 2\rangle$. Now, we repeat the scenario with j sending a message with timestamp $\langle \epsilon{+}2, 1\rangle$ as shown in Figure 4. It is straightforward to verify that the c value is unbounded.

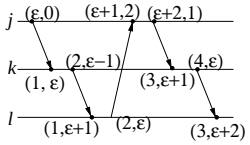

Fig. 4. Eliminating $kn.j$ at the cost of unbounded c value for Program in Figure 3

5.2 Bounding c Value Using Application Guarantee

In the logical timestamp program in Figure 3, the c value is unbounded. However, if an application still requires a bounded solution, it can do so by satisfying a simple guarantee about creation of events. More specifically, if the application periodically provides a window of size 2ϵ such that no events are created on any process in that window then $c.j$ can be bounded. The application can easily satisfy this guarantee if each process ensures that no events are created when the physical clock of that process is in the interval $[\alpha x..\alpha x{+}2\epsilon]$ where x is the period and α is a natural number. In this case, the bound on c will be proportional to the period x.

6 Discussion

In this section, we discuss the role that our assumptions played in our program, their fine-tuning and their generality. We also discuss other interpretations of $G1$ and $G2$.

Role of $G1$ and $G2$. In presenting our programs for logical timestamps and causal deterministic merge, we expected two guarantees from the underlying systems. Both these guarantees were necessary to obtain bounded stabilizing solutions. In this sense, the guarantees our solution expects are minimal. It is obvious that if only guarantee $G1$ were available, we could not derive a bounded stabilizing solution; a message that is delayed for a long enough time, so that a message with a similar timestamp can be generated in the meanwhile, can violate the requirements of logical timestamps (and causal deterministic merge). If only guarantee $G2$ were available, it would not be possible to bound the number of elements in kn; we used $G1$ to determine the number of elements that are maintained in kn.

Fine-tuning $G1$ and $G2$. In a given system, one needs to determine the values of ϵ and δ. It may also be possible to fine-tune these values depending upon other application requirements. The value of ϵ depends upon the closeness of clock values and their precision. The closeness of clock values will depend upon the clock synchronization algorithm used to correct them. If we provide higher priority for the process that corrects the clock on each processor, provide higher priority for messages sent by the clock synchronization algorithm, and reduce the non-determinism involved in the clock synchronization algorithm, we can reduce the value of ϵ (and, hence, buffer requirements, recovery time, etc). The value of ϵ can also be reduced by reducing the precision of the clock. Regarding δ, it is easy to see that there is a tradeoff between the value of δ and the percentage of messages lost. For example, we can use techniques such as forward-error-correction and send parity messages to reduce the message loss. However, in that case, the value of δ will be high as we need to wait for these parity messages to arrive at the destination.

Generality of $G1$ and $G2$. The guarantees expected in our model are satisfied by most existing systems. These guarantees are, however, stronger than that in [8]. Specifically, $G2$ can be easily obtained using the fail-aware datagram service. However, to the best of our knowledge, it is not known whether one can implement $G1$ and ensure that in the presence of faults, $G1$ is established in that model.

Other interpretations of $G1$ and $G2$. In our model, the value of $rt.j$ may not be related to the auxiliary global time. We have permitted this explicitly to allow for the case where $rt.j$ denotes some other progress measure for the program. For example, in [9], $rt.j$ could denote the number of reset operations that j has performed. Or, in a checkpointing and recovery program, $rt.j$ could denote the incarnation number of j. (Of course, in this case, we will permit K events between incrementing of $rt.j$ where K is a predetermined constant.) Thus, if a program provides the two guarantees $G1$ and $G2$ with respect to the new interpretation of $rt.j$, our solutions can also be applied.

7 Conclusion and Future Work

In this paper, we presented (in Section 4) a bounded state, stabilizing solution for causal deterministic merge. Our solution ensured that even if faults such as message corruption, improper initialization, temporary violation of system guarantees or transients occur, eventually the causal deterministic merge is provided. In our algorithm, the recovery was quick and proportional to the system guarantees. Our solution used only $O(\epsilon log\ n\ +\ log\ \delta)$ space.

To develop a solution to causal deterministic merge, we presented (in Section 3) a self-stabilizing solution for logical timestamps. In our solution, the space cost to implement logical timestamps and the time required to recover from faults is proportional to the guarantees provided by the system.

We also presented (in Section 5) variations of the above solution where we showed that the space required to implement logical timestamps can be made independent of the number of processes if the application satisfies a simple condition on how events are created. For a particular '*less*' relation, in Section 5, we showed that it is impossible to bound the size of logical timestamps without such a condition.

In developing the above solutions, we expected the underlying system to make two guarantees $G1$ and $G2$. We argued (in Section 6) that these guarantees are reasonable in that existing distributed systems satisfy them. We also pointed out how these guarantees can be fine-tuned in a given system.

We showed that given a system that satisfies $G1$ and $G2$, the system obtained after considering the buffering introduced by our causal delivery program also satisfies $G1$ and $G2$ (with slightly different parameters). We expect this property to be useful while building a hierarchical system. At the top level of this system, we will have subnetworks that consist of producers and mergers (which act as subscribers in this subnetwork). In turn, the mergers act as producers in another subnetwork and so on. In this case, the values of ϵ and δ may be different in different subnetworks.

Our solution also improves the previous Δ causal order solution in [2,3]. Specifically, the solution in [2] assumes the existence of a global clock, and requires $O(n^2)$ space that grows unbounded as the computation progresses. The solution in [3] allows the clock values to differ. However, they also require $O(n^2)$ unbounded space, and can miss some causal dependencies. By contrast, we do not assume the existence of a global clock and use only $O(log\ n)$ bounded space for our timestamps. In [2], a message received within time Δ will be delivered within time Δ. If we assumed the existence of global clocks ($\epsilon = 0$) then our solution will also provide the exact same guarantee (in addition to stabilizing fault-tolerance and bounded logarithmic space).

Regarding work on deterministic merge, Aguilera and Strom [1] have presented a deterministic merge program. In their program, the authors assume that the expected message rate of all producers is known and that the producers send dummy messages if they do not produce the data at the given rate. Also, if the producers produce messages at a faster rate than expected then the message delay grows unbounded. And, the order in which the messages are delivered in

their program is not related to the causal order between them. By contrast, we provide causal delivery, do not assume the knowledge about the rate of production of messages, and limit the amount of time for which a message needs to be buffered. However, unlike the program in [1], our program requires that $G1$ and $G2$ are satisfied.

Our causal delivery algorithm is useful in multimedia real-time applications and group-ware real-time applications. In these applications, buffering is limited and the data is valuable only if it is received within some limited time. Our protocol allows precomputation of buffering requirements and expected delays. Moreover, as discussed in Section 6, we can fine-tune the suitable values for ϵ and δ based upon available buffers and maximum permitted delay. It follows that it would be possible to exploit system level guarantees to further provide guarantees about the flow of the multimedia real-time data.

Our work suggests several directions for future work. Regarding logical timestamps and causal deterministic merge, future work includes identifying the lower bound to solve these problems with $G1$ and $G2$. In [10], we have presented a causal delivery program whose complexity is $O(nlog(\epsilon+\delta))$ whereas the complexity of the program presented in Section 3 is $O(\epsilon\ logn\ +\ log\ \delta)$. It would be interesting to determine if we can reduce the complexity further so that it is logarithmic in both ϵ and n.

References

1. M. Aguilera and R. Strom. Efficient atomic broadcast using deterministic merge. *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, 2000.
2. R. Baldoni, M. Mostefaoui, and M. Raynal. Causal deliveries in unreliable networks with real-time delivery constraints. *Journal of Real-Time Systems*, 10(3):1–18, 1996.
3. F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. *International Conference on Distributed Computing Systems*, pages 36–43, 1995.
4. S. S. Kulkarni and Ravikant. Tracking causality with physical clocks. Technical Report MSU-CSE-01-20, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, July 2001.
5. L. Lamport. Time, clocks, and the ordering of events in a disributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
6. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.
7. F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.
8. F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
9. A. Arora, S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.
10. S. S. Kulkarni. Loosely synchronized timed model. Technical Report MSU-CSE-01-4, Computer Science and Engineering, Michigan State University, East Lansing, Michigan, January 2001.