Stabilizing Causal Deterministic Merge

Sandeep S. Kulkarni Ravikant
Department of Computer Science and Engineering
Michigan State University
East Lansing MI 48824 USA

Abstract

In this paper, we focus on causal deterministic merge —that combines causal delivery and uniform total order— in semi-synchronous publish-subscribe systems that provide simple guarantees related to clock values and message delays. We consider two properties of the timestamps used to obtain causal deterministic merge: (1) they should be bounded, i.e., for a given system, the maximum size of the timestamp should be independent of the *length* of the computation, and (2) they should be scalar, i.e., the time required to compare/update timestamps should be independent of the size of the system. By making certain assumptions about how the timestamps are compared, we show that it is impossible to obtain a solution that uses scalar and bounded timestamps. Hence, we focus on solutions that achieve one of these properties. We present two solutions where the size of the timestamps is bounded; the size of the timestamps in the first solution is logarithmic in the number of processes whereas in the second solution, it is linear in the number of processes. We also present a solution where the timestamp consists of $O(1)$ integers that can grow unbounded; we further show that the timestamps in this solution can be bounded if the application provides a simple guarantee about event creation. Each of these solutions is stabilizing fault-tolerant in that even if the system is perturbed by faults that improperly initialize processes, lose/corrupt messages and temporarily violate system guarantees, the system recovers to states from where subsequent computation satisfies the requirements of causal deterministic merge.

Our solutions improve previously known solutions in several ways: In previous solutions, the timestamps consist of $O(n^2)$ unbounded integers where n is the number of processes. By contrast, one of our solutions uses $O(1)$ unbounded integers and the remaining three solutions use bounded integers.

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

[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
A preliminary version of this paper [1] has appeared in Workshop on Self-stabilization 2001
This work is partially sponsored by NSF CAREER 0092724, DARPA Grant OSURS01-C-1901, ONR grant N00014-01-1-0744, and a grant from Michigan State University.

1 Introduction

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. In addition to issuing subscriptions consisting of predicates on published messages, the subscribers specify constraints that should be met while messages are delivered to them. One common constraint on message delivery is the *uniform total order* that requires that if two subscribers deliver two messages, then they deliver them in the same order. Uniform total order is useful if it is necessary that the action taken by the subscribers that deliver those messages must be the same. For example, consider a replication based system where two (or more) replicas act as subscribers and operate on the messages delivered from publishers; if uniform total order is imposed on messages delivered by those replicas, the synchronization requirement between the replicas will be reduced.

Another practical constraint on message delivery is *causal delivery*. The causal delivery requires that if a subscriber receives messages m_1 and m_2 such that m_2 causally depends on m_1 then that subscriber *deliver* m_1 before delivering m_2. Causal dependency may occur through published messages or through internal communication among publishers. Causal delivery is also desirable in several applications. For example, consider a scenario where a publisher *publishes* a question and another publisher *publishes* its answer. It is desirable that the subscriber delivers the question before delivering the answer.

In this work, we are interested in systems where subscribers expect uniform total order that conforms to the causal order. A uniform total order that conforms to the causal order can be obtained by using a centralized solution where a sequence number is associated with each message in such a way that if message m_2 causally depends on m_1 then the sequence number associated with m_2 is larger than that of m_1. However, such centralized solution lacks scalability and reliability. We are therefore interested in a decentralized solution where each subscriber buffers the messages until it can be certain that they can be delivered while obtaining an uniform total order that conforms to the causal order. We call this the problem of *causal deterministic merge.*

We focus on solutions to the problem of causal deterministic merge in real-time publisher-subscriber systems where the subscribers need to deliver the data they subscribe to within a bound; messages delivered late are useless. Examples of such applications are found in multimedia applications where messages must be delivered within a bound. Other examples occur in control-based applications that consist of sensors and controllers that use those sensor values (see Section 7 for an example of such an application). In these control-based applications, the sensors act as producers, and controllers subscribe to one or more of these sensor values and perform the appropriate control function based on the values delivered. Moreover, in these applications, the controllers must act within required deadlines and, hence, it is important that the sensor values be delivered in a timely manner.

Need for system guarantees. Clearly, it is not possible to solve the problem of causal deterministic merge while satisfying its real-time constraints if one begins with a pure asynchronous system, where the process speeds, process clocks and message delays are arbitrary. By contrast, if one begins with a pure synchronous system —where the clocks of all processes are identical and the message delivery is instantaneous, it is straightforward to obtain a solution that provides causal deterministic merge while satisfying the real-time constraints. However, such synchrony is difficult —if not impossible— to achieve in distributed systems.

With the above motivation, we assume that the underlying system is semi-synchronous. In the semi-synchronous model, we assume that the underlying system makes two guarantees: (1) clock synchronization and (2) timely arrival of messages. Clock synchronization requires that there exists a bound, ϵ, such that the physical clock values of two processes differ at most by ϵ. And, timely arrival means that there exist bounds δ_{min} and δ_{max} such that a message is received within time $[\delta_{min}..\delta_{max}]$ or the message is lost. When $\epsilon = 0$ and $\delta_{min} = \delta_{max} = 0$, the system becomes synchronous (cf. Figure 1).

To make the solution for causal deterministic merge more widely applicable, we permit temporary violation of the system guarantees. To deal with such temporary violations of system guarantees, we require that the solution for causal deterministic merge be stabilizing fault-tolerant; i.e., from an arbitrary state —which may be reached if the system guarantees are temporarily violated— the program should recover to a state from where causal deterministic merge will be provided for subsequent messages. It follows that after the system guarantees are (re) satisfied, eventually the requirements of causal deterministic merge will also be (re) satisfied. By design, a stabilizing solution for causal deterministic merge can also deal with the case where processes are improperly initialized, channels contain garbage messages, messages are lost/corrupted, or the

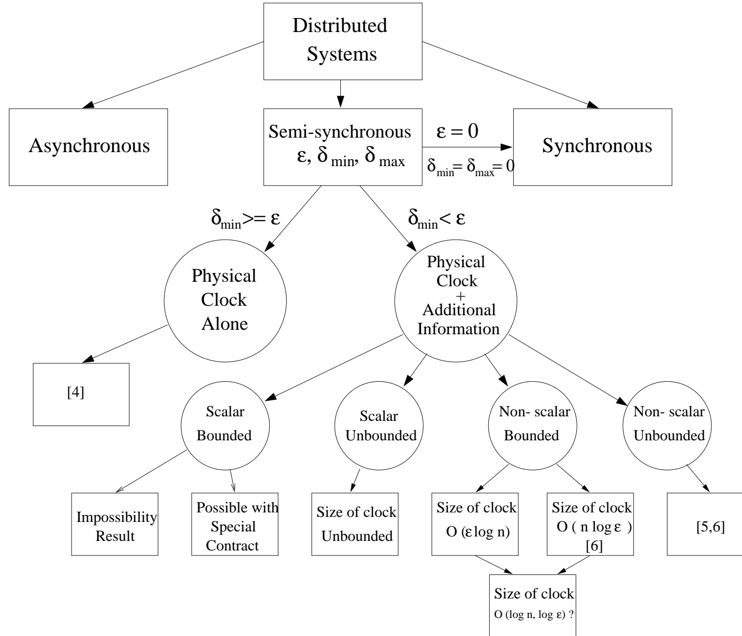

Figure 1: Classification of Solutions for Causal Deterministic Merge

state of one or more processes is arbitrarily corrupted.

Issues in causal deterministic merge. To solve the problem of causal deterministic merge in semi-synchronous systems, we associate a timestamp with each message. This timestamp contains information about the physical clock of the sender and is used to determine the causal relation among messages. It is desirable that for a given system, the value of the timestamp be bounded; if the timestamp is bounded then sufficient space could be allocated to it upfront without worrying about overflows. Another rationale for boundedness is that it is known [2, 3] to be an important —although difficult— problem in the context of stabilizing fault-tolerance.

It is also desirable that the timestamp be a scalar, i.e., the time required to compare two timestamps or the time required to compute the new timestamp be independent of the size of the system. A scalar timestamp will reduce the overhead in computing and comparing timestamps. Finally, it is also desirable that the size of the timestamps, the delay incurred while obtaining causal deterministic merge and the buffering required for obtaining causal deterministic merge be proportional to the system guarantees.

With this motivation, in this paper, we investigate the relation between ϵ, δ_{min}, δ_{max}, the size of the logical timestamps, the number of processes in the system, the buffering requirement to obtain causal deterministic merge, and the delay incurred while obtaining causal deterministic merge. We proceed as follows: First, we consider the simple case where $\delta_{min} > \epsilon$. In this case, we can use the value of physical clock alone to implement causal deterministic merge (cf. Figure 1). Further, if we assume that event creation and clock increment occur simultaneously, physical clock alone is sufficient even for the case where $\delta_{min} = \epsilon$. Lamport had made the observation about the case where $\delta_{min} \geq \epsilon$ in [4].

Next, we consider the case where $\delta_{min} < \epsilon$. If $\delta_{min} < \epsilon$, it is possible that the time (at the sender) when a message is sent is larger than the time (at the receiver) when it is received. Hence, it is not possible to obtain causal delivery using physical clocks alone. It follows that we need to maintain additional information in the timestamp to obtain causal deterministic merge.

Contributions of the paper. As mentioned above, it is desirable that the timestamps be bounded and scalar. Hence, we consider four possible possibilities for timestamps: (1) scalar and bounded, (2) scalar and unbounded (3) non-scalar and bounded and (4) non-scalar and unbounded (cf. Figure 1). Ideally, we prefer to have a solution where the timestamp is scalar and bounded. By making certain assumptions about how the timestamps are compared, we show, in Section 6.1, that it is impossible to obtain causal deterministic merge

using timestamps that are both scalar and bounded. Subsequently, in Section 6.2, we use that impossibility proof to develop a scalar and unbounded solution for logical timestamps. Then, in Section 6.3, we extend that solution to obtain a scalar and bounded program whose correctness depends on a simple guarantee from the application.

Under the non-scalar and bounded category, we present two solutions. In the first solution, the size of the timestamps is proportional to the clock drift ϵ. In this solution, an array of size $O(\epsilon)$ is maintained at each process to count the number of events in the system. Each element in the array uses $O(log\ n)$ space where n is the number of processes in the system. In the second solution, the size of timestamps is proportional to the number of processes in the system, n. In this solution, each process maintains an array of size n to capture its knowledge about the physical clock at all the processes in the system. Each element in this array uses $O(log(\epsilon+\delta))$ space.

We omit the discussion about the last category, non-scalar and unbounded. We simply note that most solutions for causal delivery (e.g., [5,6]) can be modified to obtain a solution for causal deterministic merge. The various parameters for the above-mentioned cases are summarized below:

Section	Size of timestamp	delivery time	Buffer time	Comparison/ update time	Remark
4	$O(\epsilon logn + log\delta)$	$\delta+3\epsilon$	$\delta+3\epsilon$	$O(\epsilon)$	comparison of bounded integers
5	$O(nlog(\epsilon+\delta))$	$\delta+3\epsilon$	$\delta+3\epsilon$	$O(n)$	comparison of bounded integers
6.2	unbounded	unbounded	unbounded	$O(1)$	comparison of unbounded integers
6.3	bounded, depends on the application guarantee			$O(1)$	comparison of bounded integers

Organization of the paper. This paper is organized as follows. In Section 2, we formally specify our system model and guarantees expected from the underlying system. Then, in Section 3, we specify the problem of causal deterministic merge. We present the two non-scalar bounded solutions in Section 4 and 5. In Section 6, we present issues related to the scalar solution. In Section 7, we describe an application where the programs in this paper are applicable. We discuss the role of our model and related work in Section 8. Finally, we make concluding remarks and point out future directions in Section 9.

2 System Model

A system consists of a finite set of processes that communicate via message passing. 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.

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 the message delay is at least δ_{min} and at most δ_{max} unless m_j is lost. In other words,

$$\forall m :: ((\ (st_m+\delta_{min})\ \leq\ rd_m\ \leq\ (st_m+\delta_{max})\ )\ \vee\ rd_m=\infty)$$

Remark. We assume that time is discrete and the values of ϵ, δ_{min} and δ_{max} are integers. If the physical clock is a continuous variable, we can obtain the corresponding discrete version it by using the knowledge about the minimum time between events. We discuss this issue in Section 8. Also, observe that if a program for logical timestamp is correct in a system where the message delay is in the interval $[0..\delta_{max}]$, it will continue to work if the message delay is in the interval $[\delta_{min}..\delta_{max}]$ and δ_{min} is a positive number. Hence, for simplicity, while presenting the programs in this paper, we assume that δ_{min} equals 0. If δ_{min} is positive, the size of the timestamps, the delivery time and the buffer requirements can be reduced. We discuss these improvements at the end of each section where the programs are presented. Finally, although we have defined the constraint on message delivery with respect to the sender's clock, the results in this paper are valid (with minor changes in

buffering/delay) even if we define the constraints for message delivery with respect to receiver's clock or with respect to a third clock, such as 'real'-time.

Notation. We use $ds(\epsilon, \delta)$ to denote a distributed system where the clocks differ by at most ϵ, the minimum message delay is 0 and the maximum message delay (for a message that is not lost) is δ.

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. We further assume that the event, say e_j, at $rt.j = x$ is created when $rt.j$ is advanced from $x-1$ to x. Thus, when e_j is created, the maximum clock value in the system is $x-1+\epsilon$. (Note that the assumption about one event per clock tick 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 d, e, f and g 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 [7] fault-tolerant in the presence of these faults, where stabilizing fault-tolerance is defined as follows:

Definition. A program is *stabilizing fault-tolerant* if starting from arbitrary states, it eventually recovers to states from where its specification is satisfied. □

3 Problem Statement

To determine the causal relation among events, we associate logical timestamps with messages. We recall the requirements on these logical timestamps in Section 3.1. Subsequently, in Section 3.2, we define the specification of causal deterministic merge.

3.1 Logical Timestamps

Towards defining the problem of logical timestamps, we first recall the causal relation [4], $\longrightarrow$, among events. Then, we introduce a definition that will be used in the problem statement.

Happened-before. The happened before relation [4] 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 relation that takes two arguments of type ts and returns a boolean.

Definition. Let ts be a type. A relation $less(ts, ts)$ 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:

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

Examples of logical clock implementations. 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 [8] and Mattern [9] 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 necessarily 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.

In systems that satisfy $G1$ and $G2$, we let ts consist of a representation of the physical clock value and some additional information. For programs where the timestamps are bounded, we bound both the representation of the physical clock and the additional information.

3.2 Causal Deterministic delivery

The problem of causal deterministic merge requires that causal delivery be satisfied and that messages be 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 be 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 in a distributed system $ds(\epsilon, \delta)$, the above problem statement requires causal deterministic merge of messages that reach the destination within δ time. Based on the application requirements and its ability to tolerate message loss, it is possible (cf. Section 8) 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 First Non-scalar and Bounded Solution

In this section, we present our first bounded, stabilizing and non-scalar solution for causal deterministic merge. In this solution, the size of the timestamp is linear in the clock drift, ϵ, and logarithmic in the number of processes, n, and message delay, δ. Towards this end, we first provide a program for the logical timestamps in Section 4.1. This program guarantees that the size of the timestamp is linear in ϵ and logarithmic in n and δ. Then, we show how that program can be made stabilizing in Section 4.2. Finally, we use these timestamps, in Section 4.3, to present a solution for causal deterministic merge.

4.1 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 $r.e_j$ and the maximum clock value in the system that j was aware of when e_j was created. The variable $kn.e_j$ is an array of size $2\epsilon - 1$. The variable $kn.e_j[t]$, $-\epsilon < t < \epsilon$, 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 $rt.j + \epsilon$ or more (cf. Section 2).

Program for logical timestamps. 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 $rt.j$, $r.j$ and $c.j$ 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 program can only read $rt.j$.

When a new event is created, 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 $rt.j$. Just before e_j is created, $r.j+c.j$ 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 for logical timestamps is as follows. (While presenting the program, for simplicity of presentation, we assume that $kn.j[t]$ equals 0 if $t \leq -\epsilon$ or $t \geq \epsilon$.)

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

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

Figure 2: First Non-scalar Bounded Logical Timestamp Program

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 $less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$. Since $r.e_j+c.e_j$ captures the maximum clock value that j was aware of when e_j was created, 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 if $rt.e_j+c.e_j < rt.f_k+c.f_k$. If $rt.e_j+c.e_j = rt.f_k+c.f_k$ then both e_j and f_k are aware the same maximum time, say t. Now, we consider the question: How many events that occurred at time t were j and k were aware of when they created e_j? Clearly, $kn.f_k[c.f_k]$ (respectively, $kn.e_j[c.e_j]$) denotes the number of events that occurred at time t and that k (respectively, j) was aware of when it created f_k (respectively, f_k). Clearly, if $e_j \longrightarrow f_k$ then the number of events that occurred at time t and that k was aware of when it created f_k must be at least equal to the corresponding number of events that j was aware of when it created e_j. Hence, to determine the truth value of $less$ $(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$, we compare $kn.e_j[c.e_j]$ with $kn.f_k[c.f_k]$. 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 $kn.e_j[c.e_j]$ equals $kn.f_k[c.f_k]$ then we compare $kn.e_j[(c.e_j)-1]$ and $kn.f_k[(c.f_k)-1]$, and so on. We continue this comparison until we have compared ϵ elements. (See Theorem 4.4 where we show that if $e_j \longrightarrow f_k$ then the truth value of $less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$ is determined within ϵ comparisons.) Finally, we use the process ID to break the tie. Thus, we define $less$ as follows:

$less(\langle r.e_j, c.e_j, kn.e_j\rangle, \langle r.f_k, c.f_k, kn.f_k\rangle)$

iff

$$\begin{array}{ll} & \langle r.e_j + c.e_j, kn.e_j[c.e_j], kn.e_j[c.e_j - 1], ..., kn.e_j[c.e_j - \epsilon + 1], j\rangle \\ < & \quad // \text{ lexicographic comparison} \\ & \langle r.f_k + c.f_k, kn.f_k[c.f_k], kn.f_k[c.f_k - 1], ..., kn.f_k[c.f_k - \epsilon + 1], k\rangle \end{array} \qquad \text{(Equation 4.1)}$$

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 $r.e+r.f+t-r.e(=r.f+t)$. Thus, comparison of kn values, allows us to determine if f_k was aware of more events than e_j.

Observation 4.2 The lexicographic comparison in Equation 4.1 is necessary for correctness; using pairwise comparison as in vector clocks leads to an incorrect implementation. □

4.1.1 Proof of Correctness and Boundedness

We now show that the program in Figure 2 and the *less* relation that we identified in Equation 4.1 satisfy the specification of logical timestamps. Then, we show that each element in the timestamp is bounded. More specifically, we show that the c value is bounded by ϵ, each element in kn is bounded by n and the information maintained for the physical clock is $O(\epsilon+\delta)$.

Clearly, the lexicographic relation described in Equation 4.1 is irreflexive, asymmetric and transitive. Thus, we observe:

Observation 4.3 The *less* relation given in Equation 4.1 is transitive and well-formed. □

Theorem 4.4 $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$

Proof. The proof is presented in Appendix A2. □

Bound on c and kn values. We present Lemmas 4.5-4.8 to show that c and each element in kn are bounded. More specifically, Lemma 4.6 shows the relation between c and kn values. Lemmas 4.5 and 4.8 show the bound on the c value and each element in kn. These lemmas also show why $kn.j[t]$ need not be maintained for $t \geq \epsilon$.

Lemma 4.5 $\forall e :: c.e < \epsilon$.

Proof. This follows from the fact that $r.e_j+c.e_j$ is the maximum clock value that j was aware of when e_j was created. When process j receives message m, it can learn about the maximum clock value in the system based upon the value of $r.m+c.m$. Clearly, if $c.e_j$ is non-zero, before creating e_j, j must have received a message whose $r.m$ value equals $r.e_j+c.e_j$. Since m must be in transit before e_j was created, m must have been sent before $rt.j$ is advanced from $rt.e_j-1$ to $rt.e_j$. From $G1$, the maximum clock value in the system when $rt.j=rt.e_j-1$ is $(rt.e_j-1)+\epsilon$. Thus, the maximum clock value that j can be aware of when e_j is created is $(rt.e_j-1)+\epsilon$. It follows that $c.e_j < \epsilon$. □

Lemma 4.6

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

Proof. The proof is presented in Appendix A2. □

Lemma 4.7 $\forall e, t : -\epsilon < t < \epsilon : kn.e[t] \leq |\{e\} \cup \{f : f \longrightarrow e \ \wedge \ r.f = r.e+t\}|$. □

Combining Lemma 4.7 with the assumption that at most one event can be created for a given physical clock value, we have

Corollary 4.8 Given a system with n processes, $\forall e, t :: -\epsilon < t < \epsilon : kn.e[t] \leq n$ □

Bound on the representation of the physical clock. From Lemmas 4.5 and 4.8, it follows that the c and kn values are bounded. Also, rt values can be bounded by letting j maintain only $brt.j = (rt.j \ mod \ Rt_{bound})$

where Rt_{bound} is a large enough constant. More specifically, Rt_{bound} should be large enough so that when j receives a message m that carries $brt.m$ (instead of $rt.m$), j is 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 $[rt.j-\epsilon-\delta-\epsilon+1..rt.j+\epsilon+\epsilon-1]$. It follows that if Rt_{bound} 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 $brt.j = rt.j \ mod \ Rt_{bound}$ where $Rt_{bound} \geq 4\epsilon+\delta+1$.

Reducing the size of the timestamp when δ_{min} is positive. So far, we 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 $-(\epsilon-\delta_{min}) < t < (\epsilon-\delta_{min})$. In this case, the number of elements in the array $kn.j$ are reduced to $2(\epsilon - \delta_{min}) - 1$. Finally, when $\delta_{min} \geq \epsilon$, there is no need to maintain the array as the physical clock alone is sufficient (cf. Figure 1).

4.2 Stabilizing Logical Timestamps

We show how the program in Figure 2 is modified so that it recovers from an arbitrary state –that may be reached if the logical timestamps are perturbed by faults such as failure and repair of processes, corrupted timestamps, temporary violation of system guarantees– to a state from where its subsequent computation provides causal deterministic merge. We follow the convergence stair [10] approach. Our proof of stabilization consists of four steps. In each step, we show that the convergence achieved by earlier steps is not disturbed. The proof is as follows.

In the first step, if the system guarantees are violated then they are restored. Several approaches may suffice for this purpose, including stabilizing clock synchronization programs (e.g., [11,12]), GPS clocks, network time protocol, probabilistic clock synchronization protocols [13] 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 4.5, 4.6 and 4.8. 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 4.5, 4.6 and 4.8 are satisfied for that message before processing that message. Since this step does not affect the rt values, the convergence achieved by the first step is not affected.

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 received or lost (from $G2$). Since reception of m does not affect rt values, $G1$ continues to be true. Also, if r, c or kn values are changed in a way that local invariants are violated, step 2 resets 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 $[x-\epsilon-\delta-\epsilon+1..x+\epsilon+\epsilon-1]$. 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 $[x+2\epsilon+1..x+4\epsilon-1]$ 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 $O(\epsilon+\delta)$.

To bound the value of $r.j$, once again, we maintain $brt.j = (rt.j \ mod \ Rt_{bound})$ where Rt_{bound} is a large enough constant. To achieve stabilization, we need to increase the value of Rt_{bound} so that j can distinguish between the different rt values that may occur during the recovery, i.e., in 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 $[x-2\epsilon-\delta+1..x+4\epsilon-1]$. Hence, we maintain $brt.j = rt.j \ mod \ Rt_{bound}$ where $Rt_{bound} \geq 6\epsilon+\delta+1$. Thus, we have:

Theorem 4.9 If the timestamping program in Figure 2 is modified so that the conditions specified in Lemmas 4.5, 4.6 and 4.8 are satisfied (as given in the second step above) and a variable $brt.j = rt.j \ mod \ Rt_{bound}$ is maintained to capture $rt.j$ then the resulting program uses bounded space and is stabilizing fault-tolerant provided $Rt_{bound} \geq 6\epsilon+\delta+1$. □

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 4.10 After adding stabilization to the program in Figure 2, the space used by it continues to be $O(\epsilon \ log \ n \ + \ log \ \delta)$. □

4.3 Causal Deterministic Merge

We use the logical timestamps in Section 4.1 to obtain a solution for causal deterministic merge. In this solution, the messages are timestamped according to the program in Figure 2 and stabilization of timestamps is achieved by the modification in Section 4.2. The problem of causal deterministic merge, thus, requires that if a message is received at the destination then it should be buffered until its place in causal deterministic merge is identified.

We first identify requirements for causal ordering, i.e., we consider the case where two messages m_1 and m_2 are sent to j, $m_1 \longrightarrow m_2$, m_2 is received at j and m_1 is in transit. We show 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 $c.m_1 \geq 0$, 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 $r.m_2+c.m_2+\delta$. And, from $G1$, the clock value of j will be at most $r.m_2+c.m_2+\delta+\epsilon$. It follows that 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 order, 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

$$delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon) \qquad \text{(Equation 4.11)}$$

Program for causal deterministic merge. In our 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 in Equation 4.1: 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. Since $less$ relation in Equation 4.1 is a total relation, there is a unique way to deliver these messages.

We show, in Theorems 4.12 and 4.13, that this program satisfies the specification of causal deterministic merge. The proof of causal delivery follows from the derivation of the delivery condition. And, we provide a proof for deterministic merge in Appendix A2.

Theorem 4.12 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. □

Theorem 4.13 If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered is same on both processes. □

Buffer requirements and delay guarantees in causal deterministic merge. From Lemma 4.5, the value of $c.m$ is less than 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 4.14 In the causal deterministic merge program given above, if j sends message m when its physical clock value was x then it is delivered before the physical clock value of j reaches $x+\delta+3\epsilon$. □

Lemma 4.15 A process buffers a message for at most $\delta+3\epsilon$ time. □

Theorem 4.16 Given a system $ds(\epsilon,\delta)$ if our causal deterministic program (in Figure 2) is used to deliver messages then the resulting system will be $ds(\epsilon,\delta+3\epsilon)$ (cf. Figure 3). □

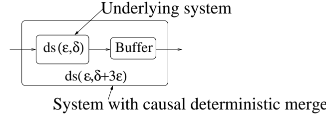

Figure 3: Guarantees of Our Causal Deterministic Merge Program

Stabilization for causal deterministic merge. 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 deterministic merge program never deadlocks since every message is eventually considered for delivery and when multiple messages are considered for delivery simultaneously, the *less* relation determines the order in which they are delivered. Once the logical timestamps are restored to their legitimate states, the subsequent computation will satisfy the requirements of causal deterministic merge. Moreover, the state space used by the causal deterministic merge program is $O(\epsilon log\ n + log\delta)$. Thus, we have:

Theorem 4.17 The causal deterministic merge program given above is stabilizing fault-tolerant and uses timestamps that require uses $O(\epsilon log\ n + log\delta)$ bounded space. □

5 Second Non-scalar Bounded Solution

In this section, we present our second bounded, stabilizing and non-scalar solution for causal deterministic merge. In this solution, the size of the timestamp is linear in the number of processes, n, and logarithmic in the clock drift, ϵ and message delay, δ. The structure of this section is similar to that of Section 4; we first provide a program for the logical timestamps in Section 5.1. This program guarantees that the size of the timestamp is linear in n and logarithmic in ϵ and δ. Then, we show how that program can be made stabilizing in Section 5.2. Finally, we use these timestamps, in Section 5.3, to present a stabilizing solution for causal deterministic merge.

5.1 Solution to Logical Timestamps

In this subsection, we present our second bounded and stabilizing solution for logical timestamps in $ds(\epsilon,\delta)$. We propose that the logical timestamp of any event e_j at process j consist of the physical time $rt.e_j$ and an array $vc.e_j$ of size n where n is the number of processes in the system. When e_j is created, the entry $vc.e_j[k]$ is used to capture $j's$ knowledge about $k's$ clock. The domain of $vc.e_j[k]$ is $[0..B-1]$ where B is a constant whose value is proportional to ϵ and δ and will be determined later. Thus, each element in $vc.e_j$ requires $O(log(\epsilon+\delta))$ space. Similar to the program in Section 4, we bound the physical clock value so that the space required for it is $O(log(\epsilon+\delta))$. Thus, the total space required to represent the timestamp of the form $\langle rt.e_j, vc.e_j\rangle$ is $O(n(log\epsilon+\delta))$.

To simplify the program presentation, we present the following definition. The definition captures what it means for a variable w to be in the interval $[x..y]$ when dealing with bounded variables.

Definition. Let w be a variable whose domain is $[0\ ..\ B-1]$. Given two integers x and y such that $(0\le x, y < B)$, we define (see Figure 4) $w\in[x\ ..\ y]$ to be true iff

$$((x\le y)\wedge(x\le w\ \wedge\ w\le y))\quad\vee\quad((x>y)\wedge(x\le w\ \vee\ w\le y))$$

Notation. We extend the above definition for any two arbitrary (positive or negative) integers x and y. If the domain of w is $[0\ ..\ B-1]$, then we define $w\in[x\ ..\ y]$ to be true iff $w\in[(x\ mod\ B)\ ..\ (y\ mod\ B)]$. We also say that the bound on w is B, and we use the notation $c := x$ to denote $c := (x\ mod\ B)$.

Figure 4: Interval [x .. y] for bounded variables

Program for logical timestamps. We now show how the events in a distributed system can be timestamped to solve the problem of logical timestamps (cf. Section 3.1) by using a bound B where $B \geq 4\epsilon+\delta+1$. For simplicity, we assume that j creates *exactly* one event for each of its clock tick.

In this program, each process j maintains its physical clock value $rt.j$ and a local array $vc.j$ consisting of an entry $vc.j[k]$ for each process k. At each clock tick, j ensures that $vc.j[j]$ equals $rt.j \ mod \ B$, and that $vc.j[k]$ is in the interval $[vc.j[j] - \epsilon \ .. \ vc.j[j] + \epsilon]$. Subsequently, j determines the timestamp, $vc.e_j$, of the current event as follows: (1) if e_j is a local event, $vc.e_j$ is the same as $vc.j$, (2) if e_j corresponds to sending of message m_j, $vc.e_j$ and $vc.m_j$ are both assigned the value $vc.j$, and (3) if e_j corresponds to receiving of message m_l then $vc.e_j[k]$ is assigned $vc.m_l[k]$ iff $vc.m_l[k]$ is in the interval $[vc.j[k] \ .. \ vc.j[j]+\epsilon]$. Thus, the program for timestamping the events is as shown in Figure 5. (If multiple messages are sent/received in one event, the same program is repeated for each message.)

Initially:
    $\forall j, k :: vc.j[k] := 0$

For each clock tick at j :
    $vc.j[j] := rt.j \ mod \ B$
    For each k : $k \neq j$: if $(vc.j[k] \notin [vc.j[j]-\epsilon \ .. \ vc.j[j]+\epsilon])$ $\quad vc.j[k] := vc.j[j]-\epsilon$

    if j wants to create local event, e_j, at $rt.j$
        $rt.e_j, vc.e_j := rt.j, vc.j$

    if j wants to create event e_j that corresponds that to j is sending message m_j
        $rt.e_j, vc.e_j, vc.m_j := rt.j, vc.j, vc.j$

    if j wants to create event e_j that corresponds to j is receiving message m_l
        $rt.e_j, vc.e_j := rt.j, vc.j$
        For each k : $k \neq j$:
            if $(vc.m_l[k] \in [vc.j[k] \ .. \ vc.j[j]+\epsilon])$ $\quad vc.j[k], vc.e_j[k] := vc.m_l[k], vc.m_l[k]$

Figure 5: Second Non-scalar Bounded Logical Timestamp Program

Comparing timestamps. We need to define the relation $less$ such that $less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$ is true if $e_j \longrightarrow f_k$. We consider three cases depending on the values of $rt.e_j$ and $rt.f_k$:

- If $rt.e_j+\epsilon < rt.f_k$, we define $less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$ to be true.
- If $rt.f_k+\epsilon < rt.e_j$, we define $less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$ to be false.
- If $|rt.e_j-rt.f_k| \leq \epsilon$, we use the vc values to determine the truth value of $less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$. Towards this end, similar to vector clock comparison, we compare $vc.e_j[l]$ with $vc.f_k[l]$ for each process l; the vector clock comparison is tailored to deal with the fact that $vc.e_j[l]$ and $vc.f_k[l]$ are bounded. As we show in Theorem 5.5, we can define $vc.e_j[l] < vc.f_k[l]$ if $vc.f_k[l] \in [vc.e_j[l] \ .. \ vc.e_j[l]+3\epsilon]$. Thus, we define the $less$ relation for the program in Figure 5 as follows:

$$less(vc.e_j, vc.f_k) \quad = \quad (\forall l :: vc.f_k[l] \in [vc.e_j[l] \ .. \ vc.e_j[l]+3\epsilon]) \ \wedge \ (vc.e_j \neq vc.f_k)$$

$$less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$$
$$= \quad (rt.e_j+\epsilon < rt.f_k) \ \vee \ ((|rt.e_j-rt.f_k| \leq \epsilon) \ \wedge \ less(vc.e_j, vc.f_k)) \qquad \text{(Equation 5.1)}$$

5.1.1 Proof of Correctness and Boundedness

We first show that if the domain of $vc.e_j[k]$ is $[0..B-1]$ where $B \geq max(6\epsilon, (4\epsilon+\delta)) + 1$ then $vc.e_j[k]$ correctly captures the knowledge j had about the clock of k when e_j was created. We also show that for this value of B, the $less$ relation in Equation 5.1 is well-formed.

To prove these results, we introduce an auxiliary unbounded variable $hrt.e_j[k]$ to denote the knowledge that j had about the clock of k when e_j was created. We first make an observation about the hrt values and then show, in Theorem 5.3, that if $B \geq 4\epsilon+\delta+1$ then $vc.e_j[k]$ correctly captures the knowledge that j had about the clock of k when e_j was created. Then, we show, in Lemma 5.4, that the $less$ relation in Equation 5.1 is well-formed.

Observation 5.2 If $e \longrightarrow f$ then $\forall l :: hrt.e.l \leq hrt.f.l$ and $\exists l :: hrt.e.l < hrt.f.l$.

Proof. This proof is similar to the correctness proof of vector clocks. □

Theorem 5.3 If $B \geq 4\epsilon+\delta+1$, the program in Figure 5 ensures that $vc.j[k] = hrt.j[k] \ mod \ B$.

Proof. The proof is presented in Appendix A2. □

The less relation in Equation 5.1 is clearly irreflexive. Also, if $B \geq 6\epsilon+1$ and $vc.f_k[l]$ is in the range $[vc.e_j[l]..vc.e_j[l]{+}3\epsilon]$ then $vc.e_j[l]$ cannot be in the range $[vc.f_k[l]..vc.f_k[l]{+}3\epsilon]$. Hence, the $less$ relation is also asymmetric. Thus, we have:

Lemma 5.4 $less$ relation defined in Equation 5.1 is well-formed if $B \geq 6\epsilon+1$. □

Theorem 5.5 Given a system $ds(\epsilon, \delta)$, the timestamping program in Figure 5 ensures that for any two events e_j and f_k with timestamps $\langle rt.e_j, vc.e_j\rangle$ and $\langle rt.f_k, vc.f_k\rangle$ respectively the following condition is true provided $B \geq (max(6\epsilon, 4\epsilon+\delta)+1)$:

- $e \longrightarrow f \quad \Rightarrow \quad less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$

Proof. The proof is presented in Appendix A2. □

Bounding the vc value and the physical clock. Observe that in our program, each element in vc is less than $max(6\epsilon, (4\epsilon+\delta))+1$. Thus, the space required for the vc value is $O(nlog(\epsilon+\delta))$.

Regarding the value of the physical clock, j maintains $brt.j = rt.j \ mod \ Rt_{bound}$ where Rt_{bound} is a large enough constant. The value of Rt_{bound} should be such that when brt value rolls over to 0, the vc value should be consistent. Towards this end, it suffices that Rt_{bound} be a multiple of B. This ensures that when brt value rolls over to 0 so does $vc.j[j]$ and, hence, the relation $vc.j[j] = rt.j \ mod \ B$ continues to be true.

Note that the messages do not carry the rt values; if the domain of vc values is $max(6\epsilon, (4\epsilon+\delta))+1$ or more then, Theorem 5.5 shows that a process can update its timestamp appropriately when it receives a message. It follows that as long as Rt_{bound} is a multiple of B, bounding the physical clock of one process does not affect other processes.

5.2 Stabilizing Logical Timestamps

In this subsection, we show how the logical timestamps can be made stabilizing fault-tolerant so that even if they are perturbed by faults such as message loss, message reorder and transients, the system will recover to states from where causality is correctly tracked. More specifically, we show that if the bound B (on $vc.j[k]$) is increased to $6\epsilon+\delta+1$, then starting from an arbitrary state, the program in Figure 5 recovers to states from where causality is correctly tracked. Thus, we have

Theorem 5.6 If the bound on vc values is at least $6\epsilon+\delta+1$ then the program in Figure 5 is stabilizing fault-tolerant.

Proof sketch. This proof also consists of four steps, and in each step, we show that the convergence achieved by earlier steps is not disturbed. In the first step, system guarantees are restored. In the second step, the clock values that a process maintains on behalf of other processes become consistent. In the third step, the clock values of messages become consistent with respect the timestamp of its sender (although the timestamp of the sender itself may be incorrect). Finally, in the fourth step, messages with such incorrect timestamps are delivered without affecting the timestamps of the receiver. The detailed proof is included in Appendix A2. □

Theorem 5.7 After adding stabilization to the program in Figure 5, the space used by it continues to be $O(n \log(\epsilon+\delta))$. □

5.3 Solution to Causal Deterministic Merge

We use the logical timestamps in Section 5.1 to obtain the second bounded and non-scalar stabilizing solution for causal deterministic merge where the space requirement grows linear in the number of processes, n, and logarithmic in the clock drift, ϵ, and message delay, δ. In this solution, the messages are timestamped according to the program in Figure 5 and stabilization of timestamps is achieved as described in Section 5.2.

Our approach is similar to that in Section 4.3. We first identify requirements for causal ordering, i.e., we consider the case where two messages m_1 and m_2 are sent to j, $m_1 \longrightarrow m_2$, m_2 is received at j and m_1 is in transit. We show how long m_2 should wait at j before being delivered so that m_1 is received and delivered before that.

Without loss of generality, let the sender of m_2 be process k and let the sender of m_1 be l. Recalling the definition of *hrt* from the previous subsection, let $hrt.m_2[l]$ denote the knowledge that process k had about the physical clock of l when m_2 was sent by k. Clearly, if $m_1 \longrightarrow m_2$ then $hrt.m_1[l] \le hrt.m_2[l]$. Since $hrt.l[l]{=}rt.l$, m_1 must have been sent when $rt.l$ was less than $hrt.m_2[l]$. From $G2$, m_1 must arrive at j before $rt.l$ reaches $hrt.m_1[l]{+}\delta$. And, from $G1$, m_1 must reach j before $rt.j$ reaches $hrt.m_1[l]{+}\delta{+}\epsilon$. Thus, if j waits until $rt.j$ reaches $hrt.m_2[l]{+}\delta{+}\epsilon$, j can ensure that the causal order will not be violated due to a message sent by l. Thus, to ensure causal delivery, it suffices that j wait until $rt.j$ exceeds $hrt.m_2[l]{+}\delta{+}\epsilon$ for each l.

Now, we identify how j can determine if $rt.j$ exceeds $hrt.m_2[l]$ by using only the *vc* values. If the bound B on the *vc* values is large enough then $vc.j[j]$ equals $rt.j \ mod \ B$ and $vc.m_2[l]$ equals $hrt.m_2[l] \ mod \ B$. Hence, to ensure causal delivery, it suffices if j waits until $vc.j[j]$ exceeds (in circular sense) $vc.m_2[l]{+}\delta{+}\epsilon$ for each l. By $G1$, for any two processes l_1 and l_2, $vc.m_2[l_1]$ and $vc.m_2[l_2]$ can differ by at most 2ϵ. Thus, if $vc.j[j]$ has exceeded $vc.m_2[l_1]{+}\delta{+}\epsilon$ for some value of l_1 then after 2ϵ steps, it would exceed $vc.m_2[l_2]{+}\delta{+}\epsilon$ for any value of l_2. It follows that j can ensure causal delivery if j waits until the following delivery condition is satisfied:

$$delcond(m_k, j) = \forall l : vc.j[j] \in [vc.m_k[l]{+}\delta{+}\epsilon \ .. \ vc.m_k[l]{+}\delta{+}3\epsilon] \qquad \text{(Equation 5.8)}$$

Program for causal deterministic merge. Whenever a message, say m, is received at a process, j, we first check if the timestamp of m is consistent with respect to the system guarantees. More specifically, by $G1$, for any two values l_1 and l_2, the circular difference between $vc.m[l_1]$ and $vc.m[l_2]$ is at most 2ϵ. If this condition is not satisfied, we drop the message. Otherwise, we buffer m 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, we use the *vc* values to uniquely determine the *hrt* values associated with messages and use those *hrt* values to decide the delivery order. More specifically, we deliver the messages based on the sum of the *hrt* values associated with those messages; the message with lower sum is delivered first. Finally, if the sum of the *hrt* values is also the same then we use the process ID of the sender to break the tie. We show, in Theorems 5.9 and 5.10, that this program satisfies the specification of causal deterministic merge. (For messages that are considered for delivery simultaneously, we cannot simply use the *less* relation in Equation 5.1. See Appendix A2 for proof.)

Theorem 5.9 If two messages m_1 and m_2 such that $m_1 \longrightarrow m_2$ arrive at any process j then m_1 is delivered before m_2. □

Theorem 5.10 If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered is same on both processes.

Proof. The proof of Theorems 5.9 and 5.10 is presented in Appendix A2. □

Observation 5.11. If the *less* relation in Equation 5.1 is used to deliver messages that are considered for delivery simultaneously, then the resulting implementation is incorrect. □

Buffer requirements and delay guarantees in causal deterministic merge. The buffering requirement and delay guarantees for this program are the same as those in Section 4. We show this in Theorem 5.12 and 5.13. The proofs are presented in Appendix A2.

Theorem 5.12 In the causal deterministic merge program given above, if j sends message m when its physical clock value was x then it is delivered before the physical clock value of j reaches $x{+}\delta{+}3\epsilon$. □

Lemma 5.13 In the above program, a process buffers a message for at most $\delta{+}3\epsilon$ time. □

Theorem 5.14 Given a system $ds(\epsilon, \delta)$ if the above program is used to deliver messages then the resulting system will be $ds(\epsilon, \delta+3\epsilon)$ (cf. Figure 3). □

Stabilizing causal deterministic merge. Once again, the stabilization of the causal deterministic merge is similar to the stabilization of the logical timestamps discussed in Section 5.2. As discussed above, messages with corrupt timestamp are either dropped or eventually delivered. After the logical timestamps are restored, causal deterministic merge will be provided for subsequent messages. Moreover, the space used by the timestamps is $O(nlog(\epsilon+\delta))$. Thus, we have:

Theorem 5.15 The causal deterministic merge program given above uses $O(nlog(\epsilon+\delta))$ bounded space and is stabilizing fault-tolerant. □

6 Scalar Solutions

In this section, we discuss scalar solutions for the problem of causal deterministic merge. Using a slight variation of the *less* relation from Section 4, in Section 6.1, we show that it is not possible to have a scalar and bounded solution for logical timestamps. In Section 6.1, we also point out the impossibility result for the *less* relation used in Section 5. Then, in Section 6.2, we present our scalar and unbounded solution for causal deterministic merge. Finally, we extend that solution to obtain a scalar and bounded solution for causal deterministic merge; the correctness of this solution depends on a simple guarantee that the application needs to provide.

6.1 Impossibility Result

If the timestamp consists of a physical clock and a bounded scalar then it will be of the form $\langle r.j, c.j\rangle$ where $r.j$ captures information about the physical clock of the last event at j and $c.j$ is a scalar that keeps track of the messages whose timestamp is larger than the value of $rt.j$ when j receives it. We use the following *less* relation to show our impossibility result.

$$\begin{aligned}
& less(\langle r.e_j, c.e_j\rangle, \langle r.f_k, c.f_k\rangle) \\
\text{iff}\quad & (r.e_j+\epsilon < r.f_k) \quad \vee \qquad\qquad \text{(Equation 6.1)} \\
& ((|r.e_j - r.f_k| \leq \epsilon) \ \wedge \ ((r.e_j+c.e_j < r.f_k+c.f_k) \ \vee \ ((r.e_j+c.e_j = r.f_k+c.f_k) \wedge r.e_j < r.f_k))) \ \vee \\
& (r.e_j = r.f_k \wedge c.e_j = c.f_k \wedge j < k)
\end{aligned}$$

To compare the timestamps $\langle r.e_j, c.e_j\rangle$ and $\langle r.f_k, c.f_k\rangle$, similar to the *less* relation in Section 4, we first compare $r.e_j+c.e_j$ with $r.f_k+c.f_k$. If $r.e_j+c.e_j$ equals $r.f_k+c.f_k$ then we compare $r.e_j$ and $r.f_k$ to determine the truth value of $less(\langle r.e_j, c.e_j\rangle, \langle r.f_k, c.f_k\rangle)$. The motivation for this comparison is based on the observation that if e and f are on the same process and $e \longrightarrow f$ then $r.e_j$ is less than $r.f_k$. Finally, we use the process ID to break the tie. Now, we show that this *less* relation requires that the value of $c.j$ is unbounded even in a system with only 4 processes, say i, j, k and l. Wlog, let $i > j > k > l$.

We begin in a state where $rt.i = \epsilon$, $rt.j = 0$, $rt.k = 0$, $rt.l = 0$ and the c values for all the processes are 0. Now, let i send a message m_{i1} to j. This send event, say d_{i1}, has the timestamp $\langle \epsilon, 0\rangle$. Let this message be received when $rt.j = 1$ and let the corresponding receive event be e_{j1}. To ensure that the causal relation between d_{i1} and e_{j1} is captured correctly, $c.e_{j1}$ must be at greater than or equal to ϵ. Wlog, let $c.e_{j1} = \epsilon$. Now, let j send a message m_{j2} to k when $rt.k$ equals 2. To capture the causality between e_{j1} and e_{j2}, $c.e_{j2}$ must be greater than or equal to $\epsilon-1$. Once again, wlog, let $c.e_{j2}$ be equal to $\epsilon-1$. We continue this scenario until k receives m_{j2} when $rt.k = 1$, k sends a message to l when $rt.k = 2$, l receives this message when $rt.l = 1$, and l sends a message to i when $rt.l = 2$. As shown in Figure 6, the timestamp of the message sent by l is $\langle 2, \epsilon+1\rangle$. Let this message be received at i when $rt.i = \epsilon+1$. To ensure that the timestamp of d_{i2} is *larger* than the events created so far, $c.d_{i2}$ must be greater than or equal to 2. Now, we repeat the scenario with i sending a message m_{i2} with timestamp $\langle \epsilon+2, 1\rangle$ as shown in Figure 6. It is straightforward to verify that the c value is unbounded.

Impossibility results for other approaches of comparing timestamps. Even with different approaches for comparing timestamps, it is not possible to obtain a bounded and scalar implementation for logical timestamps. We would like to point out that if we generalize the comparison relation from Section 5, it is still impossible to obtain bounded and scalar implementation of logical timestamps. In Section 5, we considered three cases: (1) $r.e+\epsilon < r.f$, (2) $r.f+\epsilon < r.e$, and (3) $|r.e-r.f| \leq \epsilon$. In the first two cases, the truth

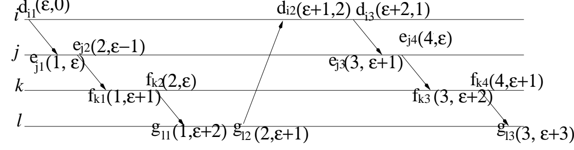

Figure 6: Impossibility Example

value is determined only by the r values whereas in the third case, the truth value is determined only by the vc values.

Based on this observation, we consider the case where $less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ is of the form $r.e+\epsilon < r.f \lor (|r.e-r.f| \leq \epsilon \land less(c.e, c.f))$. In Appendix A1, we sketch a scenario to show that for $less$ relation of this form, scalar and bounded timestamps cannot be designed.

6.2 Scalar and Unbounded Causal Deterministic Merge

The example used to demonstrate the impossibility of scalar and bounded timestamps can be easily converted into an unbounded scalar implementation of logical timestamps. For this solution, we use the $less$ relation from Section 6.1. In this solution, the values of $r.j$ and $c.j$ are updated as shown in Figure 7.

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 $(r.m + 2\epsilon < rt.j \ \land \ r.j + 2\epsilon < rt.j)$ then $c.j = 0$
  else $c.j := max(0, r.j+c.j-rt.j, r.m+c.m-rt.j+1)$
  $r.j := rt.j$
  $r.e_j, c.e_j := r.j, c.j$

Figure 7: Scalar Logical Timestamp Program

Theorem 6.2 $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ □

Program for Causal Deterministic Merge. We now use the program in Figure 7 to obtain a solution for causal deterministic merge. This solution is also scalar and unbounded. Towards this end, 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. From the $less$ relation, we have $m_1 \longrightarrow m_2 \Rightarrow (r.m_1+\epsilon < r.m_2 \ \lor \ r.m_1 + c.m_1 \leq r.m_2 + c.m_2)$. Moreover, since $c.m_1 \geq 0$, 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, the delivery condition for message m is $delcond(m, j)$ where

$$delcond(m, j) = (rt.j = r.m + c.m + \delta + \epsilon) \qquad \text{(Equation 6.3)}$$

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 defined in Section 6.1: Given two messages m_k and m_l if $less(\langle r.m_k, c.m_k\rangle, \langle r.m_l, c.m_l\rangle)$ is true we deliver m_k before m_l.

Based on the derivation of the delivery condition, it follows that the above program satisfies the specification of causal delivery. Also, this program satisfies the specification of deterministic merge; this proof is similar to

that of the program in Section 4. Similar to the proof in Section 4.3, the buffering requirement and delay for any message is at most $(c.m + \delta + 2\epsilon)$. However, in the program in Figure 7, the value of $c.m$ is not bounded. Hence, for the program in Figure 7, buffering time and the delay in delivery are unbounded.

Stabilizing causal deterministic merge. Even if the program is perturbed to an arbitrary state, the program in Figure 7 ensures that $less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ is true if e and f are successive events on the same process. Also, if e is the event corresponding to the send of m, f is the corresponding receive event and the timestamp of m is not corrupted in transit, the program in Figure 7 ensures that $less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ is true. Thus, after messages whose timestamps are corrupted during transit are delivered, the program in Figure 7 will track the causality correctly. In other words, the program in Figure 7 is already stabilizing fault-tolerant.

6.3 Bounded and Scalar Solution using an Application Guarantee

In the logical timestamp program in Figure 7, 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.

With such a guarantee, it is straightforward to see that the c values in the program in Figure 7 will be reset to 0 every time the application provides a window where no events are created. Hence, the c value will be bounded. Moreover, 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.

7 Application of Causal Deterministic Merge

In this section, we briefly sketch an application where each of the programs discussed in Sections 4, 5, 6.2 and 6.3 is used. Towards this end, we first describe the application and show how causal deterministic merge is used in it. Subsequently, we show how the system guarantees are satisfied in this context. Finally, we show the applicability of our programs for this application.

We use an application that is a variation of the beam experiment used in [14]. This beam experiment consisted of a simply supported elastic beam with a number of nodes. Each node contained co-located a sensor and actuator pair. The beam was perturbed by an external vibration. The goal of the experiment was to limit the vibration at each node by using the sensors and actuators associated with the nodes. Towards this end, the sensors provided the velocity measurement of the beam vibration while the actuator provided the point force. The control algorithm collected *suitable* sensor inputs and determined the *suitable* actuator outputs. We consider two of the algorithms used for control, *centralized* and *distributed*. In the centralized control, the sensor values of all nodes were communicated to a central node that computed the actuator outputs for all nodes. In the distributed control, each node communicated with nodes in its proximity to determine its actuator output. Due to the nature of the computation involved, the distributed algorithm is suitable for the case where quick response is desired whereas the centralized algorithm is desirable when a thorough analysis of sensor values is to be performed —although at a slower pace.

We focus on the 2-D version of this problem (cf. Figure 8) where both the centralized and the distributed control are used. More specifically, each node communicates its sensor values to its neighbors in the distributed control algorithm. It also communicates those sensor values to a central node which performs the global algorithm. For reliability considerations, the central node is replicated and, hence, each node sends its sensor values to all replicas. These replicas then compute the actuator responses and send them to individual nodes. The output of a node is a combination of the values received from neighbors and the feedback received from the central node.

(The goal of this section is to illustrate a simple application where causal deterministic merge is used. Hence, we show the need for causal deterministic merge in this application and point out how the underlying system provides the required guarantees. The details of the application, e.g., how the control algorithms work, how the distributed and centralized algorithms are combined, are outside the scope of this paper.)

In this application, causal delivery to central node is important. Towards this end, consider the scenario where node j communicates its sensor value to the central node and to its neighbor, say k, k uses this value to decide

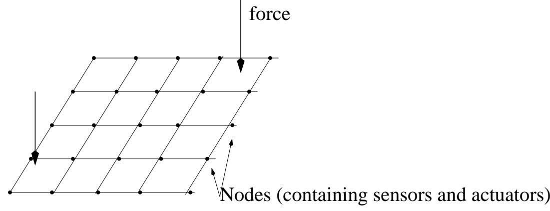

Figure 8: Application Scenario

the actuator output in the distributed algorithm, and then k sends its sensor value to the central node. In this case, the causality relation between j' response and $k's$ response should be considered so that the central algorithm can perform the computation appropriately.

Also, with replicated central node, deterministic delivery is important; to avoid synchronization among replicas, each replica should act independently. If the replicas see the sensor inputs in the same order, they can act independently without synchronization.

From the above discussion, it follows that in this application the individual nodes act as publishers and the (replicas of) the central nodes act as subscribers. The publishers communicate among themselves as well as publish messages. The replicas of the central node subscribe to these messages and each of them requires causal deterministic merge.

To apply the causal deterministic merge programs presented in this paper, we now address how the system guarantees can be met in this example. Regarding $G1$, many clock synchronization algorithms can be used and the exact algorithm used is outside the scope of the paper.

Regarding $G2$, we focus on identifying value of δ. In the absence of faults, the value of δ can be determined by computing the maximum distance between a node and the replicas of the central nodes with which it communicates. In case of failure of a node, one can route messages around the failed node (cf. Figure 9). (An approach based on such rerouting is presented elsewhere [15]) Thus, the value of δ is determined by the maximum distance of a node from the central node and the maximum number of such reroutes that are performed before dropping a message.

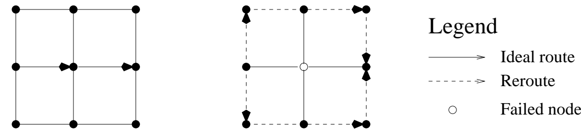

Figure 9: Rerouting in the Presence of Failures

Once the values of ϵ and δ are determined, all four programs discussed in this paper can be used to obtain stabilizing causal deterministic merge. The choice of the algorithm depends upon the algorithm used for clock synchronization and the number of nodes. For example, if clocks could be synchronized more closely then the algorithm in Section 4 is desirable whereas for small systems, the algorithm in Section 5 is desirable.

Also, the bounded and scalar program in Section 6.3 can be used in this system by providing a 'window of silence'. Arora et. al. [16] have showed that faults that transiently corrupt the sensor inputs and actuator outputs minimally affect the control. Hence, it is acceptable if the centralized algorithm is periodically suspended for a small window of 2ϵ. If such a window of silence is provided then the application meets the contract required by the program in Section 6.3 and, hence, stabilizing causal deterministic merge can be implemented using bounded and scalar timestamps.

8 Discussion and Related Work

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

Role of $G1$ and $G2$. The guarantees assumed in this paper, $G1$ and $G2$, were necessary to obtain bounded

stabilizing solutions. In this sense, the guarantees our solutions expect 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 causal deterministic merge. Also, if only guarantee $G2$ were available, a bounded stabilizing solution would not be possible; in the solution in Section 4, $G1$ was used to determine the number of elements in array kn and in the solution in Section 5, $G1$ was used to bound the domain of vc.

Fine-tuning $G1$ and $G2$. The value of ϵ used in this protocol 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. We can reduce the maximum clock drift 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. An architecture based on these ideas has been proposed elsewhere [17]. And, regarding δ, it is easy to see that there is a tradeoff between the value of δ and the percentage of messages lost.

Other interpretations of $G1$ and $G2$. In our model, the value of $rt.j$ may not necessarily 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 [18], $rt.j$ could denote the number of reset operations that j has performed. 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.

8.1 Related Work

There are a number of solutions (e.g., [19, 20]) for the causal delivery of messages in distributed systems. However, these solutions assume that no messages are lost. In these solutions, if a message is lost then other messages that causally depend on it can be delayed indefinitely. Hence, these solutions cannot be used for real-time applications.

Previous causal order solutions that use physical clocks include [5, 6]. Specifically, the solution in [5] assumes the existence of a global clock, and requires $O(n^2)$ unbounded integers. The solution in [6] allows the clock values to differ. However, they also require $O(n^2)$ unbounded space or they can miss some causal dependencies. By contrast, we do not assume the existence of a global clock and use smaller space. Also, our solutions in Sections 4 and 5 use $O(log\ n)$ and $O(n)$ bounded integers. And, our solution in Section 6.2 uses $O(1)$ unbounded integers. Moreover, apart from causal delivery, our solutions also satisfy the requirements of deterministic merge.

Regarding work on deterministic delivery, Aguilera and Strom [21] 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, in their solution, if the producers produce messages at a faster rate than expected then the message delay grows unbounded. Finally, 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.

9 Conclusion and Future Work

In this paper, we focused on the problem of causal deterministic merge where the message delivery satisfies the requirements of causal delivery and deterministic merge. In addition to constraints on message delivery, we also concentrated on bounding the buffering requirements and providing delay guarantees for causal deterministic merge.

In developing our solutions, we expected the underlying system to make two guarantees. The first guarantee dealt with clock drift among clocks used by processes, and can be achieved by several stabilizing clock synchronization algorithms (e.g., [11, 12]). The second guarantee imposed a maximum bound on delivery time for messages that are not lost, and can be obtained by fail-aware datagram service [22]. In our solutions, the buffering requirement and the delay encountered to ensure causal deterministic merge were proportional to the underlying system guarantees. Thus, based on the guarantees required by the application, the underlying system guarantees can be fine-tuned. We showed how this fine-tuning is performed in Section 8.

In developing our solutions for causal deterministic merge, we associated timestamps with messages. We

focused on two properties of timestamps: their size should be bounded, i.e., it should not grow as the computation progresses, and they should be scalar, i.e., the time to compare and update timestamps should be $O(1)$. We considered the following three possible scenarios for timestamps: (1) scalar and bounded, (2) scalar and unbounded, and (3) non-scalar and bounded. In the first category, by making certain assumptions about how timestamps are compared, we showed (cf. Section 6.1) that a bounded and scalar implementation of timestamps (respectively, causal deterministic merge) cannot be achieved. In this category, we also presented (cf. Section 6.3) a solution whose correctness depends on a simple contract that the application needs to satisfy. In the second category, we presented one solution (cf. Section 6.2) and in the third category, we presented two solutions (cf. Sections 4 and 5).

The solution in Section 4 uses timestamps whose size is $O(\epsilon \ logn \ + \ log\delta)$, i.e., it grows linearly in the clock drift and logarithmically in the number of processes. Thus, this solution is ideal for systems where the number of processes is high but high precision clock synchronization algorithm is used to limit the clock drift. The solution in Section 5 uses timestamps whose size is $O(nlog(\epsilon + \delta))$, i.e., it grows logarithmically in the clock drift and linearly in the number of processes. Thus, this solution is ideal for systems where the number of processes is small. Also, in these solutions, for a given system, the size of the timestamp is bounded. Moreover, the solutions are stabilizing fault-tolerant in that even if the system is perturbed by faults such as temporary violation of system guarantees, failure and repair of processes, message loss or corruption, or transients, the system recovers to states from where subsequent computation satisfies the requirements of causal deterministic merge. Moreover, the recovery time is quick; the programs in Sections 4 and 5 recover within $O(\epsilon+\delta)$ time.

Our solutions improve previously known solutions [5,6]. Specifically, the solution in [5] assumes the existence of a global clock, and requires $O(n^2)$ space –where n is the number of processes– that grows unbounded as the computation progresses. The solution in [6] allows the clock values to differ. However, it also requires $O(n^2)$ unbounded space or it can miss some causal dependencies. By contrast, our solution in Section 6.2 uses $O(1)$ unbounded integers and our solutions in Sections 4 and 5 use bounded integers.

Our programs are useful in multimedia real-time applications and group-ware real-time applications that require causal delivery and/or deterministic merge. In these applications, buffering is limited and the data is valuable only if it is delivered within some limited time. Our programs allow pre-computation of buffering requirements and expected delays. Moreover, as discussed in Section 8, 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. We would like to note that in these applications, if only causal delivery is required, the impossibility result for bounded and scalar implementation still applies. However, if only deterministic merge is required then it is possible to obtain a bounded and scalar implementation [23].

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 $G2$. Our solution in Section 4 has the space complexity of $O(\epsilon logn)$ whereas the solution in Section 5 has the complexity of $O(nlog\epsilon)$. It would be interesting to determine if we can reduce the complexity further so that it is logarithmic in both ϵ and n. Another extension includes developing methods to use system level guarantees in designing and verifying distributed programs. This work will leverage upon our previous work [18] that shows how contracts between components and their clients can be systematically exploited. Regarding the system guarantees, future work includes identifying how these guarantees can be used improve existing distributed programs, especially to allow bounded and stabilizing implementation.

References

[1] S. S. Kulkarni and Ravikant. Stabilizing causal deterministic merge. *Fifth International Workshop on Self-Stabilizing Systems, Lecture notes in computer science, Self-stabilizing Systems, volume 2194*, pages 183–199, October 2001.

[2] L. Lamport and N. Lynch. *Handbook of Theoretical Computer Science: Chapter 18, Distributed Computing: Models and Methods.* Elsevier Science Publishers B. V., 1990.

[3] A. Arora and M. G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

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

[5] 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.

[6] F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. *International Conference on Distributed Computing Systems*, pages 36–43, 1995.

[7] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[8] 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.

[9] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[10] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[11] J. M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison. In *ICDCS92 Proceedings of the 12th International Conference on Distributed Computing Systems*, pages 486–493, 1992.

[12] A. Arora, S. Dolev, and M. G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

[13] Flaviu Cristian and Christof Fetzer. Probabilistic internal clock synchronization. In *Symposium on Reliable Distributed Systems*, pages 22–31, 1994.

[14] A. Ledeczi, M. Maroti, and I. Bartok. Simple nest application simulator. Technical report, Institute for Software Integrated Systems, 2001. Also available at http://www.isis.vanderbilt.edu/projects/nest/index.html.

[15] G. Chakrabarti and S. S. Kulkarni. A modified approach to dynamic source routing in mobile ad-hoc networks. *AD-HOC Networks and Wireless (ADHOC-NOW)*, 2002.

[16] A. Arora, M. Gouda, T. Herman, S. Kulkarni, and M. Nesternko. Self-stabilization in networked embedded software technology (NEST). Available at: http://www.darpa.mil/ipto/research/proceedings/nest2002feb/OhioStateU200202.pdf, February 2002.

[17] P. Verissimo and A. Casimiro. The timely computing base model and architecture. *Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8), August 2002. A preliminary version of this document appeared as Technical Report DI/FCUL TR 99-2, Department of Computer Science, University of Lisboa, Apr 1999.

[18] A. Arora, S. Kulkarni, and M. Demirbas. Resettable vector clocks. *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 269–278, 2000.

[19] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 219–232, 1989.

[20] M. Raynal, A. Schiper, and S. Toueg. Causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, pages 343–350, 1991.

[21] 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.

[22] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[23] Ravikant. Stabilizing causal deterministic merge. Master's thesis, Michigan State University, 2002.

!

Appendix A1: Impossibility Results for Scalar and Bounded Solutions

We sketch the scenario to show that for $less$ relation of this form, $less(e,f) iff r.e+\epsilon < r.f \;\vee\; (|r.e-r.f| \leq \epsilon \wedge less(c.e,c.f))$, scalar and bounded implementation of logical timestamps (respectively, causal deterministic merge) is not possible.

For this sketch, we first make an observation about properties that need to be met by $less(c.e,c.f)$. First to prevent the possibility that $c.e$ is a juxtaposition of several scalars, we require that $less(c.e,c.f)$ to be true only if $c.f$ is in the interval $[c.e{+}1..c.e{+}M]$ where M is a predetermined constant. Also, if $less$ is well-formed, the domain of c must have at least $2M+1$ distinct values. With this observation, now we present the scenario to show the required impossibility result.

In our scenario, the system consists of three processes, say j, k and l. Initially, let $rt.j, rt.k, rt.l, c.j, c.k, c.l$ be all equal to 0. Now, let the computation progress as follows: Let j create a send event when $rt.j=1$ by sending a message to k. Clearly, for this event $c.j$ must also be increased so that the causal relation between the initial event $(rt.j=0, c.j=0)$ and the current event $(rt.j=1)$ is correctly tracked. Without loss of generality, let $c.j$ equal 1.

When j sends a message to k, let $rt.k$ be still equal to 0. Let k receive this message when $rt.k = 1$. Now, consider the value of $c.k$ when this event is created. Initially, the value of $c.k$ was 0. Based on the constraint on $less(c.e,c.f)$, the new value of $c.k$ should be in the range $[1..M]$. Moreover, since the c value of the incoming message is 1, it follows that the new value of $c.k$ should be in the range $[2..M{+}1]$. Without loss of generality, let the value of $c.k$ be equal to 2. Observe that $c.k$ equals $rt.k+1$.

Now let k create a local event when $rt.k=2$ and send a message to j when $rt.k=3$. From the same argument as in case of j, the c value in the message will be 4. In the meanwhile, let $rt.j$ be still equal to 2, and let j receive the message from k when $rt.j$ equals 3. Once again, the new value of $c.j$ must be in the range $[3..M{+}2]$ since the initial value of $c.j$ is 2. The new value of $c.j$ must also be in the range $[5..M{+}4]$ since the c value of the incoming message is 4. Without loss of generality, let $c.j$ be equal to 5. Observe that $c.j$ equals $rt.j+2$.

Continuing with this scenario, we can increase $|c.j{-}rt.j|$ (respectively $|c.k{-}rt.k|$). More specifically, the system will eventually reach a stage where timestamp of j (or k) is $\langle x, x{+}M\rangle$.

In the meanwhile, let l have created one event for each increment of its physical clock and let $rt.l=x$. Hence, the timestamp of the event on l is $\langle x, x\rangle$. Now, let j send a message to l. When l receives this message, the new value of $c.l$ must be in the range $[x{+}1..x{+}M]$ since the initial value of $c.l$ was x. Also, the new value of $c.l$ must be in the range $[x{+}M{+}1..x{+}2M]$. This is impossible since these intervals are disjoint.

The above scenario shows that if $c.e$ is a scalar, process l cannot choose appropriate value for $c.l$. It follows that if $c.e$ is a scalar, the problem of logical clocks is unsolvable.

To create a counterexample for causal deterministic merge, we introduce two additional processes that receive a copy of all messages sent in the above scenario. We leave it to the reader to verify that causal deterministic merge cannot be obtained for messages received by these two processes.

Appendix A2: Proofs

In this appendix, we provide the proofs for Theorems in Sections 4-6. The proofs for theorems in Section 4 are in Appendix A2.1. The proofs of theorems in Section 5 are in Appendix A2.2. And, the proofs of theorems in Section 6 are in Section A2.3.

Appendix A2.1: Proofs for Theorems in Section 4

In this appendix, we prove theorems from Section 4. In these theorems, the logical timestamp program refers to the program in Figure 2, the *less* relation refers to Equation 4.1, the delivery condition refers to Equation 4.11, and the causal delivery program refers to the program in Section 4.3.

Theorem 4.4 $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$

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.

   Now, we consider the case where $r.e{+}c.e = r.f{+}c.f$. In this case, based on how the program updates $kn.f$, we make two observations (1) $\forall t : -\epsilon{<}t{<}\epsilon \ \wedge \ -\epsilon{<}\ t{+}r.f{-}r.e{<}\epsilon : \ kn.e[t{+}r.f{-}r.e] \leq kn.f[t]$, and (2) $kn.e[r.f - r.e] < kn.f[0]$. 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 have compared $c.f{+}1$ 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 (1), $less(\langle r.e, c.e, kn.e\rangle, \langle r.f, c.f, kn.f\rangle)$ must be true. Moreover, as shown in Lemma 4.5, $c.f$ is less than ϵ and, hence, at most ϵ elements in kn are compared.

   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 case, 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, $less(\langle r.a, c.a, kn.a\rangle, \langle r.f, c.f, kn.f\rangle)$ is also true. □

Lemma 4.6

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

Proof. We prove this by induction. This is clearly true in the initial state where the only events are the initial events on each process. Since kn values are never decremented, they are always non-negative. When a new send event, say f, is created, we consider the following two cases:

- $c.f = r.e{+}c.e{-}r.f$ (where e is the previous event on that process). In this case, we have

$$\begin{array}{ll} & \forall t : t > c.f : kn.f[t] = 0 \\ \Leftarrow & \forall t : t > r.e{+}c.e{-}r.f : kn.e[t{+}r.f - r.e] = 0 \end{array}$$

$\Leftarrow\ \forall u : u{+}r.e{-}r.f > r.e{+}c.e{-}r.f : kn.e[u{+}r.e{-}r.f{+}r.f{-}r.e] = 0$
$\Leftarrow\ \forall u : u > c.e : kn.e[u] = 0$
$\Leftarrow$ true

In the above proof, if we replace '$t{>}c.f$' by '$t{=}c.f$' and '$kn.f[t]{=}0$' by '$kn.f[t]{>}0$', we would get that $kn.f[c.f]$ is greater than 0.

- $c.f = 0$

  Using the same proof as above, in the initial assignment to kn, we have $\forall t : t > r.e{+}c.e{-}r.f : kn.f[t] = 0$. Also, if $c.f$ equals 0 then $r.e{+}c.e{-}r.f \leq 0$. Hence, in the initial assignment of kn, we have: $\forall t : t > 0 : kn.f[t] = 0$. Subsequently, only $kn.f[0]$ is incremented. Hence, $\forall t : t > 0 : kn.f[t] = 0$ continues to be true. Moreover, $kn.f[0]$ is positive as it is incremented.

If f is a receive event, a similar proof suffices. In that case, we need to consider three cases based on whether $c.f$ is assigned $r.e{+}c.e{-}r.f$, $r.m{+}c.m{-}r.f$, or 0. □

Lemma 4.7 $\forall e, t : -\epsilon{<}t{<}\epsilon : kn.e[t] \leq |\{e\} \cup \{f : f \longrightarrow e \ \wedge\ r.f{=}r.e{+}t\}|$.

Proof. This proof is similar to that of Lemma 4.6. □

Theorem 4.13 If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered is same on both processes.

Proof. We consider three cases: (1) $r.m_1{+}c.m_1 < r.m_2{+}c.m_2$, (2) $r.m_1{+}c.m_1 > r.m_2{+}c.m_2$ and (3) $r.m_1{+}c.m_1 = r.m_2{+}c.m_2$. From $delcond(m, j)$, in the first case, m_1 will be delivered before m_2 and in the second case m_2 will be delivered before m_1. In the third case, m_1 and m_2 will be considered for delivery simultaneously. Hence, the *less* relation will be used to decide the order in which they should be delivered. Since *less* relation in Equation 4.1 is a total relation, there is a unique way in which messages can be delivered. Thus, the order in which m_1 and m_2 are delivered will be same irrespective of any other messages that are being considered for delivery at that time. □

Appendix A2.2: Proofs for Theorems in Section 5

In this appendix, we prove theorems from Section 5. In these theorems, the logical timestamp program refers to the program in Figure 5, the *less* relation refers to Equation 5.1, the delivery condition refers to Equation 5.8, and the causal delivery program refers to the program in Section 5.3.

Theorem 5.3 If $B \geq 4\epsilon{+}\delta{+}1$, the program in Figure 5 ensures that $vc.j[k] = hrt.j[k]\ mod\ B$.

Proof. For any event e_j, by definition, $hrt.e_j[j] = rt.e_j$. The program in Figure 5 ensures that $vc.j[j] = (hrt.j[j]\ mod\ B)$ by setting $vc.e_j[j]$ to be equal to $(rt.e_j\ mod\ B)$. For the case where $j \neq k$, we prove this by induction by considering how vc values are updated when a new event is created.

- *Local/send event.* Just before the new event, e_j, is created, by induction, $vc.j[k]$ equals $hrt.j[k]\ mod\ B$. Upon creating a local/send event, j needs to change $hrt.j[k]$ only if the previous value was less than $rt.j{-}\epsilon$. If the previous value for $hrt.j[k]$ was less than $rt.j{-}\epsilon$, j sets $hrt.j[k]$ to $rt.j{-}\epsilon$ by setting $vc.j[k]$ to $vc.j[j]{-}\epsilon$; since $vc.j[j]{=}(rt.j\ mod\ B)$.

- *Receive event.* When j receives message m_l from l and generates a new event e_j, it needs to set $hrt.j[k]$ to $hrt.m_l[k]$ iff $hrt.m_l[k]$ is greater than $hrt.j[k]$. Now, we show that j can change $hrt.j[k]$ consistently even though it only has information about $vc.j[k]$ and $vc.m_l[k]$. Towards this end, by induction, we first observe that just before the reception of m_l, $vc.j[k]$ equals $hrt.j[k]\ mod\ B$ and $vc.j[j]$ equals $rt.j\ mod\ B$.

  From $G1$, when j receives m_l, the physical clock of l is in the interval $[rt.e_j - \epsilon\ ..\ rt.e_j + \epsilon]$. From $G2$, when l sent m_l, the physical clock of l was in the interval $[rt.e_j - \epsilon - \delta\ ..\ rt.e_j + \epsilon]$ and, hence, $hrt.m_l[k]$ was in the interval $[rt.e_j - \epsilon - \delta - \epsilon\ ..\ rt.e_j + \epsilon + \epsilon]$.

  Now, if $B \geq 4\epsilon + \delta + 1$, j can use $vc.m_l[k]$ to uniquely determine the value of $hrt.m_l[k]$. Moreover, $hrt.m_l[k] \geq hrt.j[k]$ is true iff $vc.m_l[k]$ is in the range $[vc.j[k]..vc.j[j]{+}2\epsilon]$. We can strengthen it further by observing that in the absence of faults $rt.k$ cannot be more than $rt.j{+}\epsilon$. Hence, to check if $hrt.m_l[k]$ is greater than $hrt.j[k]$, it suffices to check that $vc.m_l[k]$ is in the range $[vc.j[k]..vc.j[j]{+}\epsilon]$.

(Note that from the above discussion it follows that if $vc.m_l[k]$ is in the range $[vc.j[j]+\epsilon+1..vc.j[j]+2\epsilon]$ then the value of $vc.m_l[k]$ has been corrupted by some fault. However, such a corrupted value does not affect $vc.j[k]$; this fact is used while adding stabilizing fault-tolerance to the program in Figure 5.) $\square$

Theorem 5.5 Given a system $ds(\epsilon,\delta)$, the timestamping program in Figure 5 ensures that for any two events e_j and f_k with timestamps $\langle rt.e_j, vc.e_j\rangle$ and $\langle rt.f_k, vc.f_k\rangle$ respectively the following condition is true provided $B \geq (max(6\epsilon, 4\epsilon+\delta)+1)$:

- $e \longrightarrow f \quad \Rightarrow \quad less(\langle rt.e_j, vc.e_j\rangle, \langle rt.f_k, vc.f_k\rangle)$

Proof. We prove this by considering on the values of $r.e_j$ and $r.f_k$.

- If $r.e_j+\epsilon < r.f_k$, we have defined $less(\langle r.e_j, vc.e_j\rangle, \langle t.f_k, vc.f_k\rangle)$ to be true. Hence, for these events, the above theorem is satisfied.
- If $r.f_k+\epsilon < r.e_j$, by $G1$, f_k occurred before e_j. Hence $e_j \longrightarrow f_k$ must be false. Hence for these events, the above theorem is satisfied.
- If $|r.e_j - r.f_k| \leq \epsilon$, we used the vc values to determine the truth value of $less(\langle r.e_j, vc.e_j\rangle, \langle r.f_k, vc.f_k\rangle)$. In this case, by $G1$, for any process l, the following two conditions are true: $hrt.e_j[l] \geq r.e_j - \epsilon$ and $hrt.f_k[l] \leq r.f_k.l$. Combining these two inequalities with $|r.e_j - r.f_k| \leq \epsilon$, we have: $hrt.e_j[l]+3\epsilon \geq hrt.f_k[l]$ for each process l. Moreover, if $e_j \longrightarrow f_k$ then $hrt.e_j[l] \leq hrt.f_k[l]$ for each process l. Thus, $hrt.f_k[l] \in [hrt.e_j[l] \,..\, hrt.e_j[l]+3\epsilon]$. By Theorem 5.3, $vc.e_j[l] = hrt.e_j[l] \ mod \ B$ and $vc.f_k[l] = hrt.f_k[l] \ mod \ B$. Hence, if $e_j \longrightarrow f_k$ then $vc.f_k[l] \in [vc.e_j[l] \,..\, vc.e_j[l]+3\epsilon]$. Thus, for these events the above theorem is satisfied. $\square$

Theorem 5.6 If the bound on vc values is at least $6\epsilon+\delta+1$ then the program in Figure 5 is stabilizing fault-tolerant.

Proof. This proof also consists of four steps, and in each step, we show that the convergence achieved by earlier steps is not disturbed. In the first step, if the system guarantees are violated then they are restored. Once again, as discussed in Section 4.2, several approaches may be used for this purpose.

In the second step, the information that j maintains about the clock of k will be consistent with the system guarantees (although it may still be inconsistent with the clock of k). More specifically, after j executes one step, $hrt.j[k]$ will be in the range $[rt.j-\epsilon..rt.j+\epsilon]$. Likewise, in one step, $vc.j[k]$ will be equal to $hrt.j[k] \ mod \ B$ where the domain of $vc.j[k]$ is $[0..B-1]$. Thus, predicate I_1 will be true where

$$I_1 = \forall j,k :: ((hrt.j[k] \in [rt.j-\epsilon..rt.j+\epsilon]) \ \wedge \ (vc.j[k] = (hrt.j[k] \ mod \ B)))$$

Also, from $G1$, $|rt.j - rt.k| \leq \epsilon$. After I_1 becomes true for all processes, the program will reach a state where I_2 is true, where

$$I_2 = \forall j,k :: ((hrt.k[j] \in [rt.j-2\epsilon..rt.j+2\epsilon]) \ \wedge \ (vc.k[j] \in [vc.j[j]-2\epsilon..vc.j[j]+2\epsilon]))$$

Since our timestamping program only reads the rt values, the second step does not affect the convergence achieved by the first step.

In the third step, any message (whose timestamp may have been corrupted) that is in transit when I_1 and I_2 become true will reach its destination or will be lost . Thus, the system will reach a state where for any message m_k in transit, the timestamp of m_k corresponds to the timestamp of k when it sent m_k. Note that even in the presence of messages with corrupted timestamps, I_1 and I_2 will continue to be true (cf. Theorem 5.3). It follows that the convergence achieved by first two steps is not affected.

Finally, in the fourth step, the information that process k maintains about the clock of j is consistent. Note that I_1 and I_2 allow $hrt.k[j]$ to be greater than $rt.j$. We now show that in the fourth step, any such clocks will be consistent in that $hrt.k[j]$ will be less than $rt.j$. In other words, the program will reach a state where I_3 is true, where

$I_3 = \forall j,k :: (hrt.k[j] \leq rt.j)$

For this proof, consider a state s where I_1 and I_2 are true and the messages with corrupted timestamps have been removed from the system. Let $rt.j$ equal x in state s. Now, we identify lower and upper bound for $hrt.m_k[j]$ for any message m_k in transit.

- *Lower bound on* $hrt.m_k[j]$*:* From $G1$, in state s, the value of $rt.k$ is at least $x-\epsilon$. Also, by the assumption of timely arrival, if message m_k is in transit in state s then m_k must have been sent when $rt.k$ was at least $x-\epsilon-\delta$. Finally, from I_1, $hrt.m_k[j]$ must have been at least $x-\epsilon-\delta-\epsilon$.

- *Upper bound on* $hrt.m_k[j]$*:* From I_2, $hrt.k[j]$ and $hrt.m_k[j]$ can be at most $x+2\epsilon$.

Combining the lower and upper bounds, $hrt.m_k[j]$ is in the range $[x-2\epsilon-\delta..x+2\epsilon]$. Moreover, from I_1, $rt.k$ is in the range $[x-\epsilon..x+\epsilon]$. Combining these results with I_1, $vc.m_k[j]$ is in the range $[(x \ mod \ B)-2\epsilon-\delta..(x \ modB)+2\epsilon]$, and $vc.k[j]$ is in the range $[(x \ mod \ B)-2\epsilon..(x \ modB)+2\epsilon]$.

We now show that in any computation that starts from s the value $hrt.m_k[j]$ (respectively $hrt.k[j]$) stays in the range $[x-2\epsilon-\delta..x+2\epsilon]$ until $rt.j$ is advanced to $x+2\epsilon$. Towards this end, we observe that until $rt.j$ is advanced to $x+2\epsilon$, from I_1 and I_2, the value of $rt.k$ is in the range $[x-\epsilon, x+3\epsilon]$. Thus, k will advance $vc.k[j]$ beyond $(x \ mod \ B)+2\epsilon$ only if it receives a message m_l such that $vc.m_l[j]$ is in the range $(x \ modB+2\epsilon+1, x \ mod \ B+4\epsilon]$. As shown above, however, $vc.m_l[j]$ is in the range $[(x \ mod \ B)-2\epsilon-\delta..(x \ modB)+2\epsilon]$. It follows that if the bound on $vc.m_l[j]$ is at least $6\epsilon+\delta+1$, k cannot advance $hrt.k[j]$ beyond $x+2\epsilon$ unless $rt.j$ is advanced beyond $x+2\epsilon$. Thus, I_3 becomes true.

Finally, after I_1, I_2 and I_3 become true, the causality relation is tracked correctly. The recovery time for the second step 1. The recovery time for third step is δ and the time of fourth step is 2ϵ. Thus, after the system guarantees are restored, the the causality relation is correctly tracked in $2\epsilon+\delta+1$ time. □

Theorem 5.9 If two messages m_1 and m_2 such that $m_1 \longrightarrow m_2$ arrive at any process j then m_1 is delivered before m_2.

Proof. Based on the discussion of the delivery condition, if $m_1 \longrightarrow m_2$ then m_1 is considered for delivery before m_2 or m_1 and m_2 will be considered for delivery simultaneously. If m_1 is considered for delivery before m_2, m_1 will be delivered before m_2.

Now, we show that even if m_1 and m_2 are considered for delivery simultaneously, m_1 will be delivered before m_2. To see this, from Lemma 5.2, observe that if $m_1 \longrightarrow m_2$ then $hrt.m_1[l] \leq hrt.m_2[l]$ for all values of l. And, for some value of l, $hrt.m_1[l] < hrt.m_2[l]$. Thus, the sum of the hrt values for m_2 is higher. Hence, m_2 will be delivered after m_1. □

Theorem 5.10 If two messages m_1 and m_2 arrive at processes j and k then the order in which they are delivered is same on both processes.

Proof. Based on the delivery condition, if j considers m_1 for delivery before (respectively, after) m_2 then k considers m_1 for delivery before (respectively, after) m_2. Also, if j considers m_1 and m_2 for delivery simultaneously, k does the same. When messages are considered for delivery simultaneously, their delivery order is based on the sum of the hrt values (with a tie break based on the sender-ID). Hence, the order in which m_1 and m_2 are delivered at j and k is the same. □

Observation 5.11. If the *less* relation in Equation 5.1 is used to deliver messages that are considered for delivery simultaneously, then the resulting implementation is incorrect.

Proof. When multiple messages are considered simultaneously for delivery, we cannot simply use the *less* relation in Equation 5.1 with a tie-break on process ID. To illustrate this point, consider the scenario where a process simultaneously considers three messages, m_j, m_k and m_l where $k < l < j$ for delivery. Let $less(m_j, m_k)$ be true and let $less(m_j, m_l)$, $less(m_l, m_j)$, $less(m_k, m_l)$, $less(m_l, m_k)$ be false. Now, if we use the *less* relation in Equation 5.1 with a tie-break on process ID then we need to satisfy the following constraints: (1) m_j should be delivered before m_k, (2) m_k should be delivered before m_l, and (3) m_l should be delivered before m_j. Observe that these constraints cannot be satisfied simultaneously. For this reason, we used the sum of the hrt values to decide the order in which messages should be delivered. The delivery based on this sum will ensure that m_j will be delivered before m_k while ensuring that the delivery constraints are acyclic. □

Theorem 5.12 In the causal deterministic merge program in Section 5.3, if j sends message m when its physical clock value was x then it is delivered before the physical clock value of j reaches $x+\delta+3\epsilon$.

Proof. Let $rt.j = x$ when m_j is sent. Recalling that in the absence of faults, I_1 and I_2 are true, we note that $vc.m_j[l]$ is in the range $[(x \ mod \ B)-\epsilon \ .. \ (x \ mod \ B)+\epsilon]$. Now, consider the time when $rt.k$ reaches $x+\delta+2\epsilon$. At this time, $vc.k[k]$ equals $(x \ mod \ B)+\delta+2\epsilon$. It follows that for each l, $vc.k[k]$ is in the range $[vc.m_j[l]+\delta+\epsilon..vc.m_j[l]+\delta+3\epsilon]$. In other words, the delivery condition is satisfied before $rt.k$ reaches $x+\delta+2\epsilon$ and, hence, m_j is delivered before $rt.j$ reaches $x+\delta+3\epsilon$. □

Lemma 5.13 In the causal deterministic merge program in Section 5.3, a process buffers a message for at most $\delta+3\epsilon$ time.

Proof. If j receives a message with $r.m = x$ then $rt.j$ is at least $x-\epsilon$. As shown in the proof of Theorem 5.12, m is delivered before $rt.j$ reaches $x+\delta+2\epsilon$. Hence, j buffers m for at most $\delta+3\epsilon$. □

Theorem 5.14 Given a system $ds(\epsilon, \delta)$ if our causal delivery program in Section 5.3 is used to deliver messages then the resulting system will be $ds(\epsilon, \delta+3\epsilon)$ (cf. Figure 3).

Proof. Follows from Theorem 5.12. □

Appendix A2.3: Proofs for Theorems in Section 6

In this appendix, we prove theorem 6.2 from Section 6 where we use the logical timestamp program from Figure 7 and the *less* relation in Equation 6.1.

Theorem 6.2 $\forall e, f :: e \longrightarrow f \Rightarrow less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$

Proof. We prove this theorem also 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.

- *f is a local/send event.* Let e be the previous event on j. If $r.e+2\epsilon < r.f$ then clearly $less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ is true. Moreover, by $G1$, for any event a such that $a \longrightarrow e$, $r.a < r.e+\epsilon$. Thus, for any event a such that $a \longrightarrow e$, $r.a+\epsilon < r.f$. Finally, for any event a, $a \neq e$, $a \longrightarrow e$ iff $a \longrightarrow f$. Hence, for any a, if $a \longrightarrow f$ then $less(\langle r.a, c.a\rangle, \langle r.f, c.f\rangle)$ is true.

  If $r.e+2\epsilon \not< r.f$ then the program in Figure 7 ensures that $r.e+c.e \leq r.f+c.f$ and $r.e < r.f$ are both true. Hence, $less(\langle r.e, c.e\rangle, \langle r.f, c.f\rangle)$ is true. Also, if $a \longrightarrow e$ then either $r.a+\epsilon < r.e$ or $r.a+c.a \leq r.e+c.e$. Combining this with the fact that $r.e$ is less than $r.f$, it is straightforward to see that $less(\langle r.a, c.a\rangle, \langle r.f, c.f\rangle)$ is true.

- *f is a receive event.* This proof is similar to the previous case; we simply need to consider the effect of both the previous event on j and the event corresponding to the sending of the message. □