Timestamping Messages and Events in a Distributed System using Synchronous Communication*

Vijay K. Garg‡
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA
garg@ece.utexas.edu

Chakarat Skawratananond§
eServer Solutions
IBM Austin, Inc.
Austin, TX 78758 USA
chakarat@us.ibm.com

Neeraj Mittal
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083, USA
neerajm@utdallas.edu

Abstract

Determining order relationship between events of a distributed computation is a fundamental problem in distributed systems which has applications in many areas including debugging, visualization, checkpointing and recovery. Fidge/Mattern's vector-clock mechanism captures the order relationship using a vector of size N in a system consisting of N processes. As a result, it incurs message and space overhead of N integers. Many distributed applications use *synchronous messages* for communication. It is therefore natural to ask whether it is possible to reduce the timestamping overhead for such applications.

In this paper, we present a new approach for timestamping messages and events of a *synchronously ordered computation*, that is, when processes communicate using synchronous messages. Our approach depends on decomposing edges in the communication topology into mutually disjoint *edge groups* such that each edge group either forms a star or a triangle. We

*An earlier version of this paper appeared in 2002 Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS).

†Supported in part by the NSF Grants ECS-9907213, CCR-9988225, an Engineering Foundation Fellowship.

‡Corresponding author.

§This work was done while the author was a Ph.D. student at the University of Texas at Austin.

show that, to accurately capture the order relationship between synchronous messages, it is sufficient to use one component per edge group in the vector instead of one component per process. Timestamps for events are only slightly bigger than timestamps for messages.

Many common communication topologies such as ring, grid and hypercube can be decomposed into $\lceil N/2 \rceil$ edge groups, resulting in almost 50% improvement in both space and communication overheads. We prove that the problem of computing an optimal edge decomposition of a communication topology is NP-complete in general. We also present a heuristic algorithm for computing an edge decomposition whose size is within a factor of two of the optimal.

We prove that, in the worst case, it is not possible to timestamp messages of a synchronously ordered computation using a vector containing fewer than $2\lfloor N/6 \rfloor$ components when $N \geq 2$. Finally, we show that messages in a synchronously ordered computation can always be timestamped in an offline manner using a vector of size at most $\lfloor N/2 \rfloor$.

Key words

synchronous communication, timestamping messages and events, vector clocks, edge decomposition, vertex cover

1 Introduction

A fundamental problem in distributed systems is to determine the order relationship between events of a distributed computation as defined by Lamport's *happened-before relation* [22]. The problem arises in many areas including debugging and visualization of distributed programs and fault-tolerance of distributed systems. It arises in visualization of a computation when debugging distributed programs (*e.g.*, POET [21], XPVM [20], and Object-Level Trace [5]). It also arises when evaluating a global property in a distributed computation [10, 16, 2]. An important problem in rollback recovery is to determine whether a message has become *orphan* and rollback its receiver to undo the effect of the message [29, 6].

Vector clocks, which were introduced independently by Fidge [9, 10, 11] and Mattern [24], and their variants [23] are widely used to capture the causality between events in distributed systems. To capture the causality, each event is timestamped with the current value of the local vector clock at the time the event is generated. The order relationship between two events can then be determined by comparing their timestamps. A vector clock contains one component for every process in the system. This results in message and space overhead of N integers in a distributed system consisting of N processes.

Charron-Bost [4] shows that, for every $N \geq 2$, there exists a distributed computation involving N processes such that any algorithm has to use a vector containing at least N components to faithfully capture the happened-before relation between events in the computation. We prove in [15] that Fidge/Mattern's (FM's) vector clock is equivalent to a *string realizer* of the poset corresponding to the distributed computation. Further, a vector of size equal to the string dimension of the poset [8, 15] is necessary and sufficient for timestamping events. In general, timestamps computed using dimension theory cannot be used in an online manner because the knowledge of the entire poset is typically required to compute a realizer. Further, the problem of determining the size of a smallest realizer is NP-complete in general [35]. Although these results indicate that, in the worst case, an N-dimensional vector clock is required to timestamp events, they do not exclude timestamps which use fewer than N components for interesting subclasses of computations on N processes. From a practical point of view, a natural question to ask is whether there exists an efficient timestamping algorithm for a class of applications in which a timestamp contains fewer than N integers.

In this paper, we show that timestamping of events can be done more efficiently for a distributed computation that uses *synchronous messages*. Informally, a message is said to be synchronous when the send is blocking, that is, the sender waits for the message to be delivered at the receiver before executing further. We refer to a computation in which all messages are synchronous as *synchronously ordered computation*.

Synchronous communication is widely supported in many programming languages (*e.g.*, Occam and Ada Rendezvous) and programming paradigms (*e.g.*, Synchronous Remote Procedure Calls (RPCs)). While programming using asynchronous communication allows potentially higher degree of parallelism because computation and communication can overlap, programs that use synchronous message-passing are easier to understand and develop [28].

It is well-known that a computation using synchronous communication is logically equivalent to a computation in which all message exchanges are logically instantaneous. In other words, it is always possible to draw the time diagram for a synchronously ordered computation such that arrows for messages appear *vertically* (assuming time progresses from left to right) [4, 25]. If we ignore *internal* events in a synchronously ordered computation, then the problem of timestamping events of the computation reduces to that of timestamping its messages. (Note that, in a distributed system using synchronous communication, timestamping messages is equivalent to timestamping communication events. This is an important problem in itself, especially when communication events are the only *relevant* events in a computation.) Using the Lamport's happened-before

relation, we define a partial order on messages and describe an online algorithm for timestamping messages that accurately capture the partial order. Instead of associating a component in the vector with each process in the system, we exploit the structure of the communication topology to reduce the size of the vector. Specifically, we decompose the edges in the communication topology into mutually disjoint edge groups such that each edge group either forms a star or a triangle. Intuitively, when the communication is synchronous, messages exchanged along the edges of an edge group (star or triangle) are totally ordered and their relationship can be captured using a single integer [31, 17]. Therefore it is sufficient to use one integer in the vector for each edge group in the decomposition. We show how timestamps assigned to messages can be used to timestamp internal events by employing only few additional integers. Further, we demonstrate that, like Fidge/Mattern's timestamps [9, 10, 11, 24], our timestamps can be used to test for precedence between two events in $O(1)$ time.

Note that our technique requires that the decomposition of edges into edge groups be known to all processes. Many common topologies including ring, grid and hypercube can be easily decomposed into at most $\lceil N/2 \rceil$ edge groups. This immediately implies that, with our timestamping approach, space and communication overheads improve by almost 50% for these topologies. For general topologies, however, computing an optimal edge decomposition is an NP-complete problem. We present a heuristic algorithm that can be used for computing an edge decomposition whose size is within a factor of two of the optimal.

We show that, using an offline algorithm, synchronous messages can be timestamped with vectors containing at most $\lfloor N/2 \rfloor$ integers. This result is derived using dimension theory of posets. We also show that, for every $N \geq 2$, there exists a synchronously ordered computation on N processes such that any vector-based timestamping mechanism with component-wise comparison requires at least $2\lfloor N/6 \rfloor$ components to accurately capture the partial order on messages. This holds even when the communication topology is *sparse* in the sense that the number of edges in the topology is within a small constant factor of the number of processes.

To summarize, the paper makes the following contributions:

1. We define a causal relationship between synchronous messages based on the Lamport's happened-before relation on events. We present an online algorithm to timestamp messages using a vector of size less than N. We prove that these vector timestamps accurately capture the order relationship between messages.

2. Using timestamps assigned to messages, we assign timestamps to all events in the computation. Our timestamps for events use only few additional integers than timestamps for messages. We also show that, similar to Fidge/Mattern's timestamps, our timestamps can be used to test for precedence between any two events in $O(1)$ time.

3. We prove that the problem of computing an optimal edge decomposition is NP-complete in general. We present a heuristic algorithm for computing edge decomposition such that the size of the decomposition is at most twice the size of an optimal edge decomposition.

4. We show that the vector of size $\lfloor N/2 \rfloor$ is sufficient to capture relationship between synchronous messages using an offline algorithm.

5. We show that, for every $N \geq 2$, there exists a synchronously ordered computation on N processes such that any vector-based timestamping mechanism for messages requires at least $2\lfloor N/6 \rfloor$ entries in the vector.

The remainder of this paper is organized as follows. Section 2 provides background for the problem discussed in this paper. An online algorithm for timestamping messages is given in Section 3. We also demonstrate how timestamps for messages can be used to generate timestamps for events using only few additional integers. We show that the problem of edge decomposition is NP-complete in Section 4 and also present an approximation algorithm for solving the problem. Section 5 describes an offline algorithm. Section 6 compares our work with others.

2 Model and Notations

We assume a loosely-coupled message-passing system without any shared memory or a global clock. A distributed program consists of N processes, denoted by $\{P_1, P_2, \ldots, P_N\}$, communicating via messages. In this paper, we assume that all messages are synchronous. A computation that uses only synchronous messages is called a *synchronously ordered computation*. It can be shown that a computation is synchronously ordered if it is possible to timestamp send and receive events with integers in such a way that (1) timestamps increase within each process and (2) the send and the receive events associated with each message have the same timestamp. Therefore, the space-time diagram of the computation can be drawn such that all messages arrows are vertical, assuming that time progresses from left to right [4] (see Figure 1).

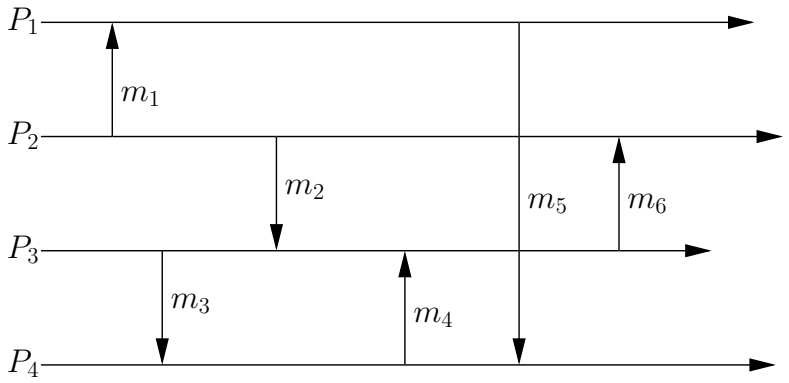

Figure 1: A synchronously ordered computation with 4 processes.

Determining the order of messages is crucial in observing distributed systems. We write $e \prec f$ when event e occurs before f in a process. Here, we define the order among synchronous messages. The set of messages M in a given synchronously ordered computation forms a poset $\mathcal{M} = (M, \mapsto)$, where $\mapsto$ is the transitive closure of $\triangleright$ defined as follows:

$$m_i \triangleright m_j \iff \begin{cases} m_i.send \prec m_j.send & \text{, or} \\ m_i.send \prec m_j.receive & \text{, or} \\ m_i.receive \prec m_j.send & \text{, or} \\ m_i.receive \prec m_j.receive \end{cases}$$

We say that m_i *synchronously precedes* m_j when $m_i \mapsto m_j$. Also, when we have $m_{i_1} \triangleright m_{i_2} \triangleright \ldots \triangleright m_{i_k}$, we say that there is a *synchronous chain* of size k from m_{i_1} to m_{i_k}. Finally, when $m_i \neq m_j$ and neither $m_i \not\mapsto m_j$ nor $m_j \not\mapsto m_i$ holds, we write $m_i \| m_j$.

In the example given in Figure 1, $m_1 \| m_3$, $m_1 \triangleright m_2$, $m_2 \mapsto m_6$, and $m_3 \mapsto m_5$. There is a synchronous chain between m_1 and m_5 of size 4.

To perform precedence-test based on synchronously-precede relation, we devise a timestamping mechanism that assigns a vector to each message m (or, equivalently, to send and receive events of the message). Let $m.v$ denote the vector assigned to message m. Our goal is to assign timestamps that satisfy the following property,

$$m_i \mapsto m_j \iff m_i.v < m_j.v \tag{1}$$

Given any two vectors u and v of size t, we define the less-than relation, denoted by $<$, as

Symbol	Domain	Meaning
$\prec$	events	relation on events executed on the same process
$\rightarrow$	events	Lamport's happened-before relation on events
$\triangleright$	messages	relation on messages involving a common process
$\mapsto$	messages	transitive closure of $\triangleright$
$\underline{\mapsto}$	messages	reflexive closure of $\mapsto$

Table 1: Various relations on messages and events used in this paper.

follows.

$$u < v \iff \begin{cases} \forall k : 1 \leq k \leq t : u[k] \leq v[k] & \wedge \\ \exists l : 1 \leq l \leq t : u[l] < v[l] \end{cases} \tag{2}$$

We call the relation given in Equation (2) *vector order*.

From Equations (1) and (2), one can determine if $m_i \mapsto m_j$ by checking whether $m_i.v < m_j.v$. If $m_i.v$ is not less than $m_j.v$ and $m_j.v$ is not less than $m_i.v$, then we know that $m_i \| m_j$ (assuming $m_i \neq m_j$).

For convenience, Table 1 lists various relations that we use in this paper.

3 An Online Algorithm

In this section, we describe an algorithm for assigning timestamps to messages and events in a synchronously ordered computation to accurately capture their order relationship. Note that, in a distributed system using synchronous communication, timestamping messages is equivalent to timestamping communication events. This is an important problem in itself, especially when communication events are the only *relevant* events in a computation.

As opposed to Fidge/Mattern's approach which is based on using one component for each process, our algorithm uses one component for each *edge group*. We first define the notion of *edge decomposition* and *edge group*.

3.1 Edge Decomposition

The communication topology of a system that consists of N processes, $P_1, \ldots, P_N$, can be viewed as an undirected graph $G = (V, E)$ where $V = \{P_1, \ldots, P_N\}$, and $(P_i, P_j) \in E$ when P_i and P_j can

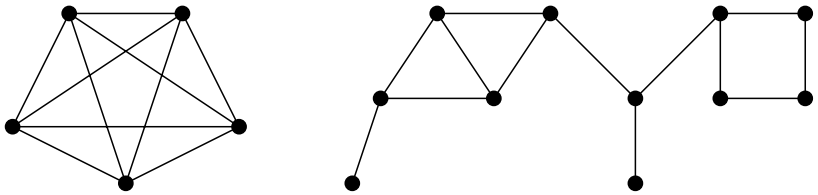

Figure 2: Examples of the communication topologies. (a) A topology where every pair of processes can communicate directly with each other. (b) A topology where not every pair of processes communicate directly with each other.

communicate directly. Figure 2(a) gives the communication topology of a system in which every process can communicate directly with each other. Figure 2(b) gives the communication topology of another system in which not every pair of processes communicates directly with each other.

Some particular topologies that will be useful to us are the *star* and the *triangle* topologies. An undirected graph $G = (V, E)$ is a star if there exists a vertex $x \in V$ such that all edges in E are incident to x. We call such a star as *rooted* at node x. An undirected graph $G = (V, E)$ is a triangle if $|E| = 3$, and these three edges form a triangle. We denote a triangle by a triple such as (x, y, z) denoting its endpoints.

The star and triangle topologies are useful because messages in a synchronously ordered computation with these topologies are always totally ordered. In fact, we have the following:

Lemma 1 *The message sets for all synchronously ordered computations in a system with* $G = (V, E)$ *as the communication topology are totally ordered if and only if* G *is a star or a triangle.*

Proof: Given any two messages in a star topology, there is always one process (the center of the star) which is a participant (a sender or a receiver) in both the messages. Since all message events within a process are totally ordered it follows that both these messages are comparable. The similar argument holds for the triangle topology.

Conversely, assume that the graph is not a star or a triangle. This implies that there exists two distinct edges (P_i, P_j) and (P_k, P_l) such that none of their endpoints is common. Consider a synchronously ordered computation in which P_i sends a synchronous message to P_j and P_k sends a synchronous message to P_l concurrently. These messages are concurrent and therefore the message

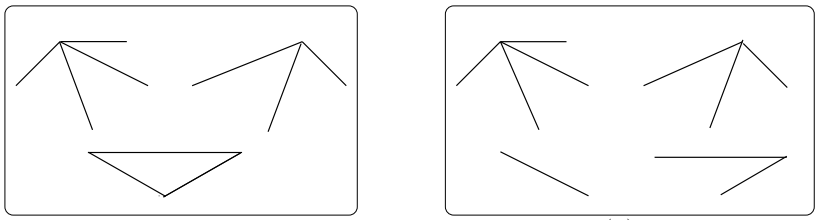

Figure 3: Edge decompositions of the fully-connected topology with 5 processes. (a) The first decomposition consisting of 2 stars and 1 triangle. (b) The second decomposition consisting of 4 stars.

set is not totally ordered. ∎

Note that the above Lemma does not claim that message set cannot be totally ordered for a topology that is neither a star nor a triangle. It only claims that for every such topology there exists a synchronously ordered computation in which messages do not form a total order. Now based on the definitions of star and triangle graphs, we are ready to define the edge decomposition of G.

Definition 2 (Edge Decomposition) *Let $G = (V, E)$ be communication topology of a system using synchronous communication. A partition of the edge set, $\{E_1, E_2, \ldots, E_d\}$, is called an edge decomposition of G if $E = E_1 \cup E_2 \cup \ldots \cup E_d$ such that:*

1. *$\forall i, j : i \neq j : E_i \cap E_j = \emptyset$, and*
2. *$\forall i : (V, E_i)$ is either a star or a triangle.*

We refer to each E_i in the edge decomposition as an edge group. In our algorithm, we will assign one component of the vector for every edge group. Note that there is possibly more than one decomposition for a topology. Our goal is to get the smallest possible decomposition. Consider a fully-connected system consisting of N processes. The first decomposition consists of $N - 3$ stars and 1 triangle. The second decomposition consists of $N - 1$ stars. Figure 3 presents the two decompositions of a fully-connected system with 5 processes.

The complete graph is the worst case for edge decomposition, resulting in $N - 3$ stars and 1 triangle. In general, the number of edge groups may be much smaller than $N - 2$. Given a tree-based communication topology consisting of 20 processes, Figure 4 shows how to decompose edges into three edge groups E_1, E_2, and E_3 where each group is a star.

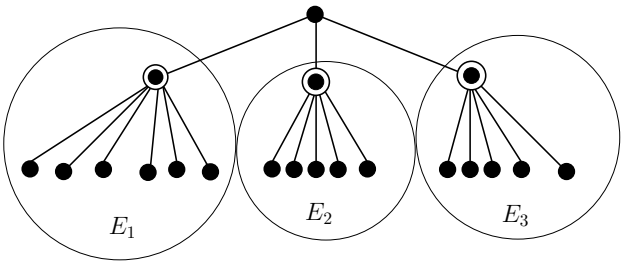

Figure 4: A tree-based topology with 20 processes.

We will discuss techniques for edge decomposition that minimize the number of edge groups in Section 4.

3.2 Timestamping Messages (Communication Events)

Each process maintains a vector of size d, where d is the size of the edge decomposition. We assume that information about edge decomposition is known to all processes in the system.

The online algorithm is presented in Figure 5. Due to the implementation of synchronous message ordering [25, 14], we assume that for each message sent from P_i to P_j, there exists an acknowledgment sent from P_j to P_i. Essentially, to timestamp each message, the sender and the receiver must first exchange their local vector clocks. Then, each process computes the component-wise maximum between its vector and the vector received (Lines (5) and (9)). Finally, both the sender and the receiver increment the g^{th} element of their vectors where the channel along which the message is sent belongs to the g^{th} group in the edge decomposition (Lines (6) and (10)). The resulting vector clock is the timestamp of this message. Intuitively, the g^{th} entry of the local vector clock at process P_i captures the number of messages that have been exchanged along the g^{th} edge group so far as per P_i.

Figure 6 shows a sample execution of the proposed algorithm on a fully-connected system with 5 processes. Edge decomposition consists of 2 stars (E_1 and E_2) and 1 triangle (E_3). For example, message sent from P_2 to P_3 is timestamped $(1, 1, 1)$ because the channel between P_2 and P_3 is in edge group E_2, and the local vector on P_2 and P_3 before transmission are $(1, 0, 0)$ and $(0, 0, 1)$, respectively.

Next, we prove that our online algorithm assigns vector timestamps to synchronous messages such that these timestamps encode poset $(M, \mapsto)$. The channel along which a message m_x is sent must be a member of a group in the edge decomposition. We use $m_x.g$ to denote the index of the

P_i ::
var
v_i : vector[1..d] of integers, initially **0**;
ED : edge decomposition,($\{E_1, \ldots, E_d\}$);

(01) On sending m to P_j;
(02) **send**(m, v_i) to P_j;

(03) On receiving (m, v) from P_j;
(04) **send** acknowledgment(v_i) to P_j;
(05) $\forall k : v_i[k] = max(v_i[k], v[k])$;
(06) $v_i[g]{+}{+}$ where edge $(i, j) \in E_g$;
(07) Timestamp of m is v_i;

(08) On receiving acknowledgment(v) of m from P_j;
(09) $\forall k : v_i[k] = max(v_i[k], v[k])$;
(10) $v_i[g]{+}{+}$ where edge $(i, j) \in E_g$;
(11) Timestamp of m is v_i;

Figure 5: An online algorithm for timestamping messages.

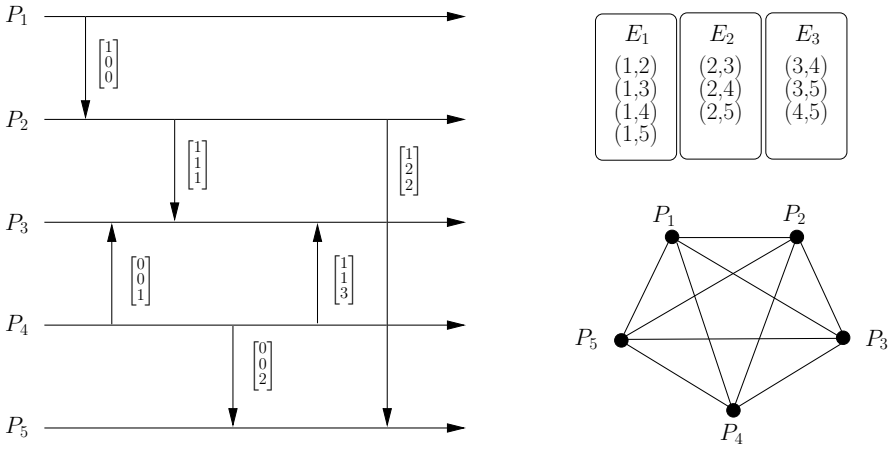

Figure 6: A synchronously ordered computation with 5 processes, and its edge decomposition.

group to which this channel belongs in the edge decomposition. Clearly,

Lemma 3 $m_i \| m_j \Rightarrow m_i.g \neq m_j.g$

Proof: Let c_i (resp. c_j) be an edge in the topology graph G that corresponds to the channel along which m_i (resp. m_j) is sent. Since $m_i \| m_j$, from Lemma 1, all messages in an edge group are totally ordered, we get that c_i and c_j must belong to different edge groups. Therefore, $m_i.g \neq m_j.g$. ∎

Theorem 4 *Given an edge decomposition of a system in which processes communicate using syn-*

chronous messages, the algorithm in Figure 5 assigns timestamps to messages such that $m_i \mapsto m_j \iff m_i.v < m_j.v$.

Proof: ($\Rightarrow$) First, we show that $m_i \mapsto m_j \Rightarrow m_i.v < m_j.v$. Since the sender and the receiver of a message exchange their local vector clocks and compute the component-wise maximum of the two vector clocks, it is easy to see that if $m_i \rhd m_j$, then $m_i.v \leq m_j.v$. This in turn implies that if $m_i \mapsto m_j$ then $m_i.v \leq m_j.v$ because $\mapsto$ is the transitive closure of $\rhd$. We now claim that:

$$m_i \mapsto m_j \Rightarrow m_i.v[m_j.g] < m_j.v[m_j.g] \tag{3}$$

This is true because before the timestamp is assigned to m_j, $m_j.v[m_j.g]$ is incremented. Thus, we have $m_i \mapsto m_j \Rightarrow m_i.v < m_j.v$.

($\Leftarrow$) We now show the converse, $m_i \not\mapsto m_j \Rightarrow \neg(m_i.v < m_j.v)$. Due to the definition of vector order, it is sufficient to show that:

$$m_i \not\mapsto m_j \Rightarrow m_j.v[m_i.g] < m_i.v[m_i.g] \tag{4}$$

We do a case analysis.

(Case 1: $m_j \mapsto m_i$**)**

From Equation (3), by changing roles of m_i and m_j, we get that $m_j.v[m_i.g] < m_i.v[m_i.g]$.

(Case 2: $m_i \| m_j$**)**

We prove by induction on k, the size of the longest synchronous chain from a *minimal message* in the poset $(M, \mapsto)$ to m_j. A message m is minimal if there is no message m' in the computation such that $m' \mapsto m$.

(Base: $l = 1$**)** m_j is a minimal message.

From Lemma 3 and $m_i \| m_j$, $m_i.g \neq m_j.g$. Since m_j is a minimal message by the initial assignment of the vector clock, both sender and the receiver have 0 as the component for $m_i.g$ and the component-wise maximum also results in 0 for $m_i.g$. Further, since $m_i.g \neq m_j.g$ the component for $m_i.g$ is not incremented. Hence, $m_j.v[m_i.g] = 0$.

We now claim that $m_i.v[m_i.g] \geq 1$. This is true because we increment the component for $m_i.g$ before assigning the timestamp for m_i. Since the value of all entries are at least 0, it will be at least 1 after the increment operation.

From, $m_j.v[m_i.g] = 0$ and $m_i.v[m_i.g] \geq 1$, we get that $m_j.v[m_i.g] < m_i.v[m_i.g]$.

(Induction: $l > 1$)

Let m_k be any message such that $m_k \rhd m_j$. We know that $m_i \not\mapsto m_k$, otherwise $m_i \mapsto m_j$. By induction hypothesis,

$$m_i \not\mapsto m_k \Rightarrow m_k.v[m_i.g] < m_i.v[m_i.g]$$

To obtain $m_j.v$, the sender and receiver of m_j exchange timestamps of any immediately preceding message (if any). From induction hypothesis, we know that the $m_i.g^{th}$ component of vectors from both the sender and receiver are less than $m_i.v[m_i.g]$. Hence, it stays less after the component-wise maximum. Further, since $m_i.g \neq m_j.g$, the component for $m_i.g$ is not incremented. Therefore, $m_j.v[m_i.g] < m_i.v[m_i.g]$.

This establishes the Theorem. ∎

Given an edge decomposition of size d, our online timestamping algorithm uses a vector of size d at each process. Further, each message carries a vector of size d. It may appear that, with our timestamping approach, as many as d comparisons may have to be made in the worst case to determine the exact relationship between two messages, In the next section, we show that this time can actually be reduced to $O(1)$.

3.3 Reducing Time for Precedence Testing

One of the advantages of Fidge/Mattern's timestamps are that they can be used to test for precedence in $O(1)$ time. It turns out that our timestamps also satisfy the same desirable property. To reduce the time for precedence testing, we prove the following two Lemmas. The proof of both Lemmas uses the contrapositive of Equation (4), which was established while proving Theorem 4:

$$m_i.v[m_i.g] \leq m_j.v[m_i.g] \quad \Rightarrow \quad m_i \mapsto m_j \tag{5}$$

The first Lemma deals with the case when two messages are exchanged along channels that belong to the same edge group.

Lemma 5 *Assume $m_i.g = m_j.g$. Then,*

$$(m_i \mapsto m_j) \iff m_i.v[m_i.g] < m_j.v[m_i.g]$$

Proof: Assume that $m_i.g = m_j.g$.

($\Rightarrow$) We need to show that $m_i \mapsto m_j \Rightarrow m_i.v[m_i.g] < m_j.v[m_i.g]$. The implication follows from Equation (3), which was established while proving Theorem 4.

($\Leftarrow$) Now, we show the converse, that is, $m_i.v[m_i.g] < m_j.v[m_i.g] \Rightarrow m_i \mapsto m_j$. The implication follows from Equation (5) and the observation that $m_i.v[m_i.g] < m_j.v[m_i.g] \Rightarrow m_i.v[m_i.g] \leq m_j.v[m_i.g]$. ∎

The second Lemma deals with the case when two messages are exchanged along channels that belong to different edge groups.

Lemma 6 *Assume $m_i.g \neq m_j.g$. Then,*

$$(m_i \mapsto m_j) \iff m_i.v[m_i.g] \leq m_j.v[m_i.g]$$

Proof: Assume that $m_i.g \neq m_j.g$.

($\Rightarrow$) We need to show that $m_i \mapsto m_j \Rightarrow m_i.v[m_i.g] \leq m_j.v[m_i.g]$. Clearly, from Theorem 4, $m_i \mapsto m_j \Rightarrow m_i.v < m_j.v$. From the definition of vector order, it follows that $m_i.v < m_j.v \Rightarrow m_i.v[m_i.g] \leq m_j.v[m_i.g]$. Combining the two, we get the result.

($\Leftarrow$) The converse follows from Equation (5). ∎

Lemmas 5 and 6 enable us to determine the order relationship between two messages in $O(1)$ time provided we know the edge groups to which the two messages belong. Intuitively, edge groups play the same role in our approach as processes in Fidge/Mattern's approach.

3.4 Timestamping Internal (Non-Communication) Events

In this section, we show how internal events can be timestamped so that Lamport's happened-before relation between events [22] can be inferred from timestamps assigned to messages. Lamport's happened-before relation, denoted by $\rightarrow$, is defined as the smallest transitive relation satisfying the following properties [22]:

1. if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and
2. if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

Recall that for each synchronous message m sent from a process P_i to another process P_j, there is an acknowledgment sent from P_j to P_i. It is important to note that happened-before relation between events uses messages as well as their acknowledgments.

For an internal event e, let $e.p$ denote the process on which e is executed. Also, let $e.b$ denote the *last* message exchanged by $e.p$ *before* it executes e. If no such message exists, then $e.b$ is defined to be $\perp$. Finally, let $e.a$ denote the *first* message exchanged by $e.p$ *after* it executes e. If no such message exists, then $e.a$ is defined to be $\top$. We use $\underline{\mapsto}$ to denote the reflexive closure of $\mapsto$. Further, expressions $m \mapsto \perp$ and $\top \mapsto m$ evaluate to false for all messages m.

Theorem 7 $e \rightarrow f \iff (e \prec f) \vee (e.a \underline{\mapsto} f.b)$

Proof: ($\Rightarrow$) First, we have to prove that $e \rightarrow f \Rightarrow (e \prec f) \vee (e.a \underline{\mapsto} f.b)$. If e and f are on the same process, then $e \prec f$ and the implication trivially holds. Otherwise, since $e \rightarrow f$, there must be a causal chain of messages starting from e and ending at f. This in turn implies that either $e.a = f.b$ or there exists a synchronous chain of messages starting from $e.a$ and ending at $f.b$.

($\Leftarrow$) Conversely, we have to prove that $(e \prec f) \vee (e.a \underline{\mapsto} f.b) \Rightarrow e \rightarrow f$. Clearly, when $e \prec f$, $e \rightarrow f$. Therefore assume that $e.a \underline{\mapsto} f.b$. From the definition of $\perp$ and $\top$, $e.a \neq \top$ and $f.b \neq \perp$. Since e is executed before $e.a$ is exchanged and f is executed after $f.b$ is exchanged, there exists a causal chain of messages from e to f involving application messages and/or their acknowledgments. As a result, $e \rightarrow f$. ∎

From Theorem 7, timestamp for an internal event consists of two parts. The first part enables us to evaluate the first disjunct (whether $e \prec f$ holds) and the second part enables us to evaluate the second disjunct (whether $e.a \underline{\mapsto} f.b$ holds). For an event e, the first part can be realized using two integers: (1) identifier of the process on which e is executed, given by $e.p$, and (2) counter indicating the number of events that have been executed on $e.p$ before e, denoted by $e.c$. The second part can be realized using two vector timestamps: (1) vector timestamp for $e.b$ and (2) vector timestamp for $e.a$. This means that the timestamp for an internal event consists of $2d + 2$ integers. The size of the timestamp can be further reduced to only $d + 4$ integers using the following Lemma.

Theorem 8 $e.a \underline{\mapsto} f.b \iff (e.a \neq \top) \wedge (f.b \neq \perp) \wedge (e.a.v[e.a.g] \leq f.b.v[e.a.g])$

Proof: ($\Rightarrow$) Assume that $e.a \underline{\mapsto} f.b$ holds. From the definition of $e.a$ and $f.b$, we can infer that $e.a \neq \top$ and $f.b \neq \perp$. It remains to be shown that $e.a.v[e.a.g] \leq f.b.v[e.a.g]$. In case

$e.a = f.b$, the result clearly holds. Therefore assume that $e.a \mapsto f.b$. From Lemmas 5 and 6, either $e.a.v[e.a.g] < f.b.v[e.a.g]$ or $e.a.v[e.a.g] \leq f.b.v[e.a.g]$ holds. In either case, $e.a.v[e.a.g] \leq f.b.v[e.a.g]$ holds.

($\Leftarrow$) Assume that $(e.a \neq \top) \wedge (f.b \neq \bot) \wedge (e.a.v[e.a.g] \leq f.b.v[e.a.g])$ holds. In case $e.a.g = e.b.g$, from Lemma 5, we can infer that $e.a \underline{\mapsto} f.b$ holds. On the other hand, if $e.a.g \neq f.b.g$, then, from Lemma 6, we can deduce that $e.a \mapsto f.b$ holds. This in turn implies that $e.a \underline{\mapsto} f.b$ holds. ∎

Theorem 8 implies that the timestamp for e does not need to carry the vector timestamp of $e.a$. Rather it is sufficient to store two integers to be able to conduct the precedence test involving e: $e.a.g$ and $e.a.v[e.a.g]$. To summarize, the timestamp for e is given by five components: (1) $e.p$, (2) $e.c$, (3) $e.b.v$, (4) $e.a.g$ and (5) $e.a.v[e.a.g]$. The third component is defined only if $f.b \neq \bot$. The fourth and fifth components are defined only if $e.a \neq \top$. Theorem 8 also allows us to conduct the precedence test involving internal events in $O(1)$ time.

Observe that the timestamp for an internal event is not completely defined until the process to which the event belongs exchanges a message. This, however, does not create any problem as far as testing for precedence is concerned. It can be verified that the precedence test still produces correct result. Moreover, no other process in the system except the process to which it belongs would know about such an event (because it is an "internal" event). Therefore when a process exchanges a message, only timestamps stored locally may have to be updated. The change does not need to be propagated to other processes.

Remark 1 For a communication event e, we can define both $e.b$ and $e.a$ to be the message involved in the communication. It can be verified that Theorem 7 is still applicable as long as not both events are communication events of the same message. Therefore the precedence test described above can be used to compare any pair of events except when the two events are communication events of the same message. ∎

4 Decomposing Edges of a Communication Topology

As discussed in Section 3.2, the overhead of our algorithm is crucially dependent upon the size of the edge decomposition. Let $\alpha(G)$ denote the size of a smallest edge decomposition (note that there may be multiple edge decomposition of the same size). In our edge decomposition, we decompose the graph into stars and triangles. If we restricted ourselves to decomposing the edge set only in

stars then the problem is identical to that of vertex cover. A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both)

We can now provide a bound for the size of the vector clocks based on the vertex cover.

Theorem 9 *Let $G = (V, E)$ be communication topology of a system using synchronous communication. Let $\beta(G)$ be the size of the optimal vertex cover of G. Then, for $N \geq 3$, vectors of size $\min(\beta(G), N-2)$ are sufficient to timestamp messages.*

Proof: From the definition of vertex cover, every edge is incident on some vertex in the vertex cover. For every edge we assign some vertex to the vertex cover. If some edge has both the endpoints in the vertex cover, then we arbitrarily choose one. By the definition of vertex cover problem, all edges are partitioned in this manner into stars. When $\beta(G) = N - 1$, we can simply use trivial edge decomposition of $N - 3$ stars and one triangle. Thus, there exists an edge decomposition of size at most $\min(\beta(G), N - 2)$. ∎

Since vertex cover does not use triangles in edge decomposition, it is natural to ask how bad can a pure star decomposition be compared to star and triangle decomposition. We claim that $\beta(G) \leq 2\ \alpha(G)$. This bound holds because any decomposition of the graph into stars and triangles can be converted into a decomposition purely of stars by decomposing every triangle into two stars. The above bound is tight in general because if the graph consisted of just t disjoint triangles, then $\alpha(G) = t$ and $\beta(G) = 2t$.

Even for a connected topology, the ratio $\beta(G)/\alpha(G)$ can be made arbitrarily closed to two. Consider a communication topology of the form shown in Figure 7 consisting of t triangles. Any vertex cover of the topology has to contain at at least two vertices from each triangle. Therefore $\beta(G) \geq 2t$. However, the optimal edge decomposition of the topology consists of t triangles and 1 star. Therefore $\alpha(G) = t + 1$. As a result, the ratio $\beta(G)/\alpha(G) \geq 2t/(t+1) = 2 - 2/(t+1)$, which can be made arbitrarily close to 2 by choosing large enough value for t.

4.1 Complexity of Edge Decomposition Problem

It can be shown that the problem of optimal edge decomposition of a general graph is NP-hard. The proof of the following result was communicated to us in an email by Nirman Kumar who attributed it to Sariel Har-Peled. We have included the proof here for completeness sake.

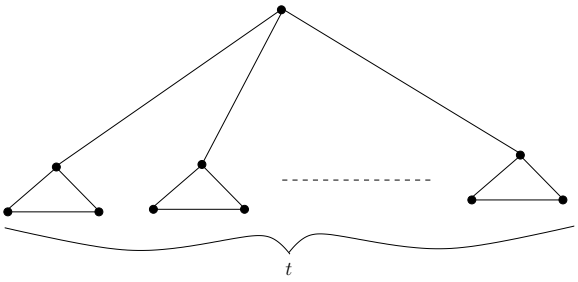

Figure 7: A communication topology for which the ratio $\beta(G)/\alpha(G)$ is close to 2.

Theorem 10 *Given an undirected graph G, and an integer k, determining whether there exists an edge decomposition of G of size at most k, is NP-complete in general.*

Proof: The problem is clearly in NP because given a partition of the edge set into stars and triangles it is easy to verify that it is a proper edge decomposition and its size is at most k.

To prove that the problem is NP-hard, we use the transformation from the vertex cover problem which is known to be NP-hard [13]. Given a graph G and a positive integer k, to determine whether there is a vertex cover of size k we transform it into the edge decomposition problem as follows. We construct a new graph H from G by replacing every edge $e = (x, y)$ in G with three edges: (x, x_e), (x_e, y_e) and (y_e, y) where x_e and y_e are new vertices added for this edge. Thus if the original graph G has n vertices and m edges, then H has $n + 2m$ vertices and $3m$ edges. Further, H does not have any triangles. We now claim that G has a vertex cover of size at most k if and only if H has an edge decomposition of size at most $k + m$.

First assume that G has a vertex cover of size k. For any edge $e = (x, y)$ either x or y is in the vertex cover. If only x is in the vertex cover for G, then we include x and y_e in the vertex cover for H. Similarly, if only y is in the vertex cover for G, then we include y and x_e in the vertex cover for H. If both x and y are in the vertex cover for G, then we include x, y and y_e in the vertex cover for H. It can be verified that the vertex cover for H has size at most $k + m$. Since a vertex cover is also an edge decomposition, it follows that there exists an edge decomposition of size at most $k + m$.

Now assume that H has an edge decomposition of size at most $k + m$. Because H has no triangles, any edge decomposition of H is equivalent to a vertex cover of H. By the construction of H any vertex cover of H must include at least one of the vertices from $\{x_e, y_e\}$ for all edges

e. If the vertex cover has both x_e and y_e, then we remove y_e from the vertex cover and add y to the vertex cover. This change ensures that there is a vertex cover with exactly m vertices from $\bigcup_e \{x_e, y_e\}$. The remaining vertices in the vertex cover of H forms a vertex cover of G. This set is of size at most k. ∎

4.2 An Approximation Algorithm for Edge Decomposition

We now present an algorithm that returns an edge decomposition which is at most twice the size of the optimal edge decomposition. Further, our algorithm returns an optimal edge decomposition when the graph is acyclic.

The algorithm is shown in Figure 8. It works by repeatedly deleting stars and triangles from the graph. The main **while** loop in line (02) has three steps inside. The first step chooses any node which has degree 1, say x which is connected to node y. It outputs a star rooted at y. When no nodes of degree 1 are left, the algorithm goes to the second step.

In the second step, the algorithm checks if there is a triangle (x, y, z) such that there are no edges in F which are incident to x or y other than those in the triangle. There may be other edges incident to z, but the degree of nodes x and y is exactly 2. Once all such triangles have been output, the algorithm goes to step three.

In the third step, the algorithm chooses an edge (x, y) with the largest number of adjacent edges. If there is more than one such edge, it chooses any one of them. Now it outputs two stars one rooted at x and the other rooted at y. After the third step, the algorithm goes back to the **while** loop to check if all edges have been accounted for.

Figure 9 shows the operation of our edge decomposition algorithm on the communication topology shown in Figure 2(b). Figure 9(b),(c), and (d) show the first, second, and third step, respectively, of the algorithm, respectively. In Figure 9(e), the execution loops back to the first step, edge (j, k) is output, and the program exits. Figure 9(f) shows the resulting edge decomposition consists of 4 stars and 1 triangle.

The algorithm has time complexity of $O(|V||E|)$ because in every step, the identification of the edge (Line (4), (8), and (12)) can be done in $O(|E|)$ time, which results in deletion of all edges incident on at least one vertex.

The following theorem shows that the algorithm produces an edge decomposition with a *ratio bound* of 2. The ratio bound is the ratio between the size of the edge decomposition produced by the algorithm and the size of an optimal edge decomposition.

```
Input: Undirected graph G = (V, E);
Output : edge decomposition,({E_1, ..., E_d});
// Each E_i is either a star or a triangle
(01) F := E;
(02) while F ≠ ∅ do
        //First Step:
(03)    while there exists a node x such that degree(x) = 1 do
(04)        Let (x, y) be the edge of F incident to x;
(05)        output star rooted at y and all incident edges to y;
(06)        remove from F all edges incident on y;
(07)    endwhile;
        //Second Step:
(08)    while there exists a triangle (x, y, z) with degree(x) = degree(y) = 2 do
(09)        output triangle (x, y, z) ;
(10)        remove from F the edges in the triangle;
(11)    endwhile
        //Third Step:
(12)    Let (x, y) be an edge of F with largest number of edges adjacent to it;
(13)    output star rooted at y and all incident edges to y;
(14)    output star rooted at x and all incident edges to x except (x, y);
(15)    remove from F all edges incident on x or y;
(16) endwhile;
```

Figure 8: An approximation algorithm for edge decomposition.

Theorem 11 *The algorithm in Figure 8 produces an edge decomposition with the approximation ratio bound of* 2.

Proof: The algorithm creates edge groups in the first step (Lines (3)-(7)), the second step (Lines (8)-(11)) or the third step (Lines (12)-(15)). For every creation of an edge group, we identify an edge and include it in a set H. In the first step, we use the edge (x, y) the lone edge incident to x and put in the set H. In the second step, we use the edge (x, y) from the triangle and put it in H. Finally, for step 3, we put the edge chosen in line 12 in H. It is easy to verify that no two edges in H are incident to a common vertex. This is because any time we choose an edge in any of the steps, all adjacent edges are deleted from F. Since no two edges have any vertex in common, edges in H must all be in distinct edge groups in the optimal edge decomposition. However, the size of edge decomposition produced is at most twice the size of H. ∎

Note that in the above proof we have not used the fact that in step 3, we choose an edge with the largest number of adjacent edges. The correctness and the approximation ratio is independent

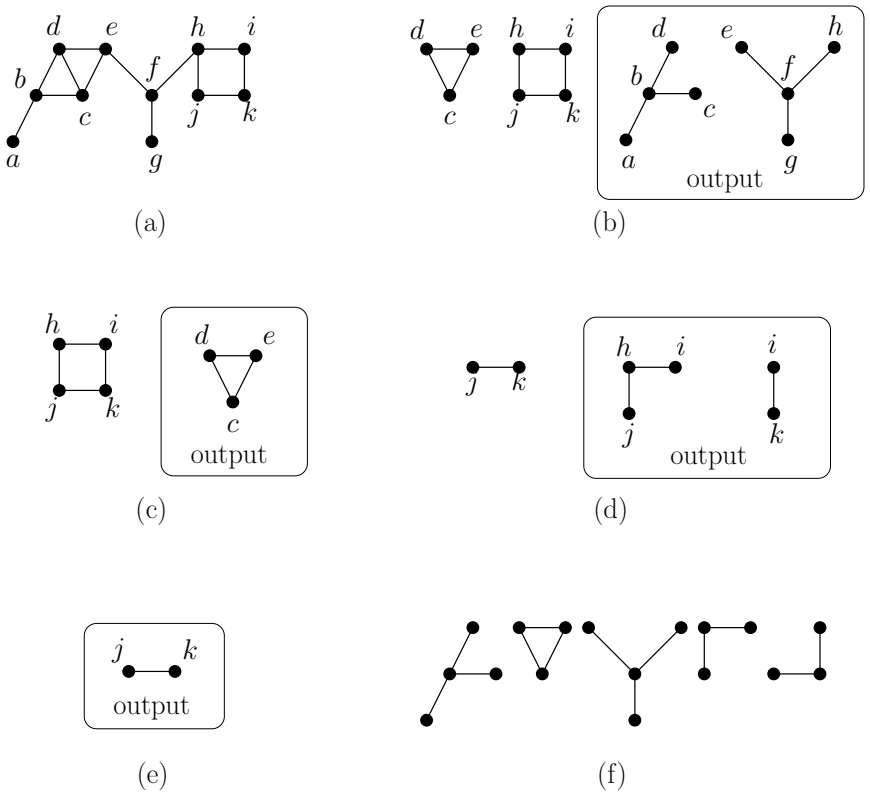

Figure 9: A sample run of the proposed decomposition algorithm. (a) The input topology. (b) In the first step, the algorithm outputs 2 stars. There are 7 edges remaining. (c) In the second step, the algorithm outputs a triangle (c, d, e). There are 4 edges remaining. (d) In the third step, two stars are output. Edge (j, k) is remaining. (e) The execution loops back to the first step again and edge (j, k) is output. The algorithm terminates. (f) The resulting edge decomposition consists of 4 stars and 1 triangle.

of that choice. However, by deleting as large number of edges as possible in each step, one would expect to have a smaller edge decomposition.

We now show that the above algorithm outputs optimal edge decomposition for acyclic graphs.

Theorem 12 *The algorithm in Figure 8 produces an optimal edge decomposition for acyclic graphs.*

Proof: First note that an acyclic graph can have only stars as edge groups. Further, when the algorithm is applied to an acyclic graph all the edges will be deleted in the **while** loop of the first step. In other words, if we take a forest (an acyclic graph is equivalent to a forest or a collection of trees) and repeatedly delete all edges that are adjacent or one hop away from the leaves then we will eventually delete all the edges.

Thus, the set H constructed in the proof of Theorem 11 consists of edges added only in step 1. Since we add exactly one edge group for every edge added to H, the optimality follows. ∎

4.3 Edge Decomposition for Common Topologies

Some of the common topologies that are used for communication in a distributed system are tree, ring, grid and hypercube. It can be shown that, for all these topologies, there exists a vertex cover of size at most $\lceil N/2 \rceil$. This implies that it is possible to timestamp messages and events of any synchronously ordered computation generated on these topologies using at most $\lceil N/2 \rceil + 4$ integers. For the sake of completeness, we briefly describe how to construct a vertex cover of size at most $\lceil N/2 \rceil$ for these topologies.

Tree Topology: A possible vertex cover consists of all vertices on even levels of the tree. Another vertex cover consists of all vertices on odd levels of the tree. Clearly, the size of one of these vertex covers is at most $\lceil N/2 \rceil$.

Ring Topology: Assume that the vertices in the ring are numbered sequentially in clockwise fashion starting from 1. Then the set of all odd-numbered vertices constitutes a vertex cover of the ring.

Grid Topology: Assume that vertices in each row are numbered sequentially from left to right starting from 1. A possible vertex cover for the topology can be constructed as follows. From every odd numbered row, pick all odd-numbered vertices. From every even numbered row, pick all even-numbered vertices. It can be shown that the size of the vertex cover thus obtained is at most $\lceil N/2 \rceil$.

Hypercube Topology: A vertex cover of a hypercube of size $N/2$ can be constructed by including all vertices with even parity in the bit representation of their labels. Since every edge in a hypercube connects vertices that differ in exactly one bit, one of the vertices adjacent to the edge has even parity. Hence this set covers all edges and contains exactly $N/2$ vertices.

5 An Offline Algorithm

We present an offline timestamping algorithm which takes a completed computation as an input and assigns a vector timestamp to each message in the given computation. Our offline algorithm is based on applying dimension theory to the poset formed by messages in the synchronously ordered computation. We first provide the technical background for dimension theory.

5.1 Background: Dimension Theory

A pair (X, P) is called an irreflexive partially ordered set or a poset if X is a set and P is an irreflexive, and transitive binary relation on X. A poset (X, P) is called *chain* if every distinct pair of points from X is comparable in P. Similarly, we call a poset an *antichain* if every distinct pair of points from X is incomparable in P. The width of poset (X, P), denoted by $width(X, P)$, is the size of the longest antichain of P.

A family of linear extensions of (X, P) denoted by $\mathcal{R} = \{L_1, L_2, \ldots, L_t\}$ is called a *realizer* of (X, P) if $P = \cap\, \mathcal{R}$. For any poset (X, P), the dimension of (X, P), denoted by $dim(X, P)$, is the least positive integer t for which there exists a family $\mathcal{R} = \{L_1, L_2, \ldots, L_t\}$ of linear extensions of P so that $P = \cap\, \mathcal{R} = \bigcap_{i=1}^{t} L_i$.

5.2 Offline Algorithm for Timestamping Messages

The offline algorithm is based on the result of the following theorem.

Theorem 13 *Given a poset $(M, \mapsto)$ formed by messages in a synchronously ordered computation with N processes, vector clocks of size $\lfloor N/2 \rfloor$ can be used to encode poset$(M, \mapsto)$.*

Proof: For any subset $L \subseteq M$ such that $|L| > \lfloor N/2 \rfloor$, there exists $m_i, m_j \in L : m_i \mapsto m_j$ or $m_j \mapsto m_i$. This is because each message involves two processes. From a set of $\lfloor N/2 \rfloor + 1$ messages, there must be at least two messages that share a common process. Hence, the size of the longest antichain of $(M, \mapsto)$ (or width$(M, \mapsto)$) is at most $\lfloor N/2 \rfloor$. From Dilworth's theorem [7], for any poset P, $dim(P) \leq width(P)$. Hence, $dim(M, \mapsto) \leq \lfloor N/2 \rfloor$. ∎

As a result from Theorem 13, we get the offline algorithm as shown in Figure 10.

As an example, if we use offline algorithm to timestamp messages in the computation shown in Figure 6, 2-dimensional vectors are sufficient to capture concurrency as shown in Figure 11.

From a given poset $\mathcal{M}$,

(1) Let w be the width of poset $\mathcal{M}$. From Theorem 13, $w \leq \lfloor N/2 \rfloor$.

(2) Construct a set of linear extensions, $\{L_1, \ldots, L_w\}$, such that $\bigcap_{i=1}^{w} L_i = \mathcal{M}$.
(Procedure for constructing this linear realizer is given in [31])

(3) Timestamp each message m with V_m, where $V_m[i]$ is the number of elements less than m in L_i.

Figure 10: An offline algorithm for timestamping messages.

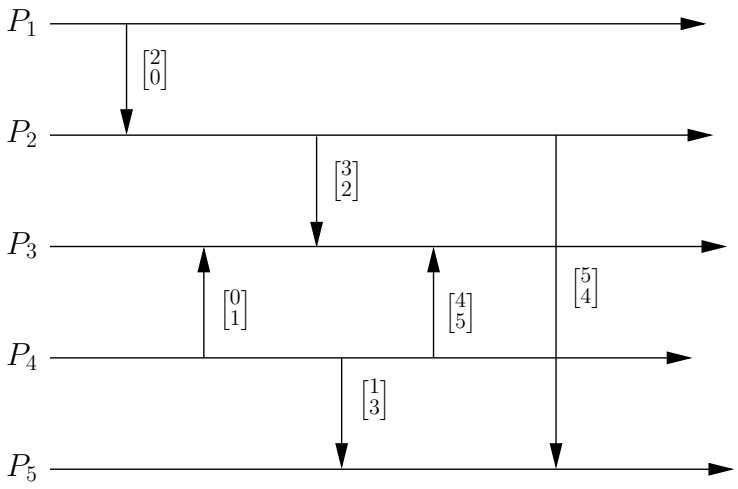

Figure 11: A sample run of the offline algorithm.

5.3 Lower Bound on Size of Message Timestamps

In this section, we show that every vector-based timestamping algorithm, in which vector timestamps are compared component-wise, has to use at least $2\lfloor N/6 \rfloor$ components for timestamping messages in a synchronously ordered computation on N processes, in the worst case.

Our proof uses a well-known poset in dimension theory known as the *standard example*. The standard example S_n for $n \geq 2$ consists of $2n$ elements $\{a_1, a_2, \ldots, a_n\} \cup \{b_1, b_2, \ldots, b_n\}$. The precedence relation is given by $a_i < b_j$ if and only if $i \neq j$, for $i, j = 1, 2, \ldots, n$. Figure 12 shows the diagram for S_5.

Dushnik and Miller [8] have shown that $dim(S_n) = n$. We construct a synchronously ordered computation involving N processes such that the poset on messages contains the standard example S_n with $n \geq 2\lfloor N/6 \rfloor$ as a subposet.

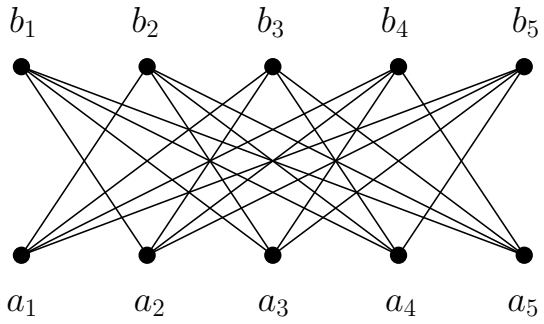

Figure 12: The standard example S_5.

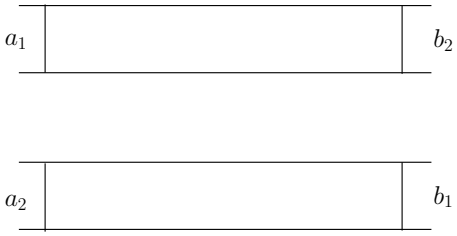

Figure 13: An example of a synchronously ordered computation that contains S_2 as a poset.

Our construction is recursive in nature. The result trivially holds for $N \leq 3$. For $4 \leq N \leq 12$, it is easy to construct a synchronously ordered computation that contains S_2 as a subposet. An example of such a computation is shown in Figure 13. We next show how to construct a synchronously ordered computation containing S_{n+2} as a subposet given a synchronously ordered computation containing S_n as a subposet by using only six additional processes. The construction is shown in Figure 14.

Besides messages a_{n+1}, a_{n+2}, b_{n+1} and b_{n+2}, we use fourteen additional messages to obtain the desired computation. The main idea behind the construction is as follows. Let C_n refer to the given computation and C_{n+2} refer to the resulting computation. For a message m, let $m.ps$ denote the set of processes involved in the exchange of m. Further, let A_n denote the set $\{a_1, a_2, \ldots, a_n\}$. The set B_n can be similarly defined. Observe that $a_{n-1} \mapsto b_i$ already holds for each $b_i \in B_{n-2}$ in C_n. This implies that there is a chain of messages (possibly empty) starting from a_{n-1} and ending at b_i for each $b_i \in B_{n-2}$ in C_n. Each chain starts from one of the processes in $a_{n-1}.ps$. Therefore, to ensure that $a_{n+1} \mapsto b_i$ holds for each $b_i \in B_{n-2}$ in C_{n+2}, we proceed as follows. We add messages s_{n+1} and t_{n+1} between one of the processes in $a_{n+1}.ps$ and both processes in $a_{n-1}.ps$ as shown in Figure 14. Each of the two messages is added after a_{n-1} but before any other message is exchanged by the respective process (of $a_{n-1}.ps$) in C_n. By the way of construction, it is easy to see that

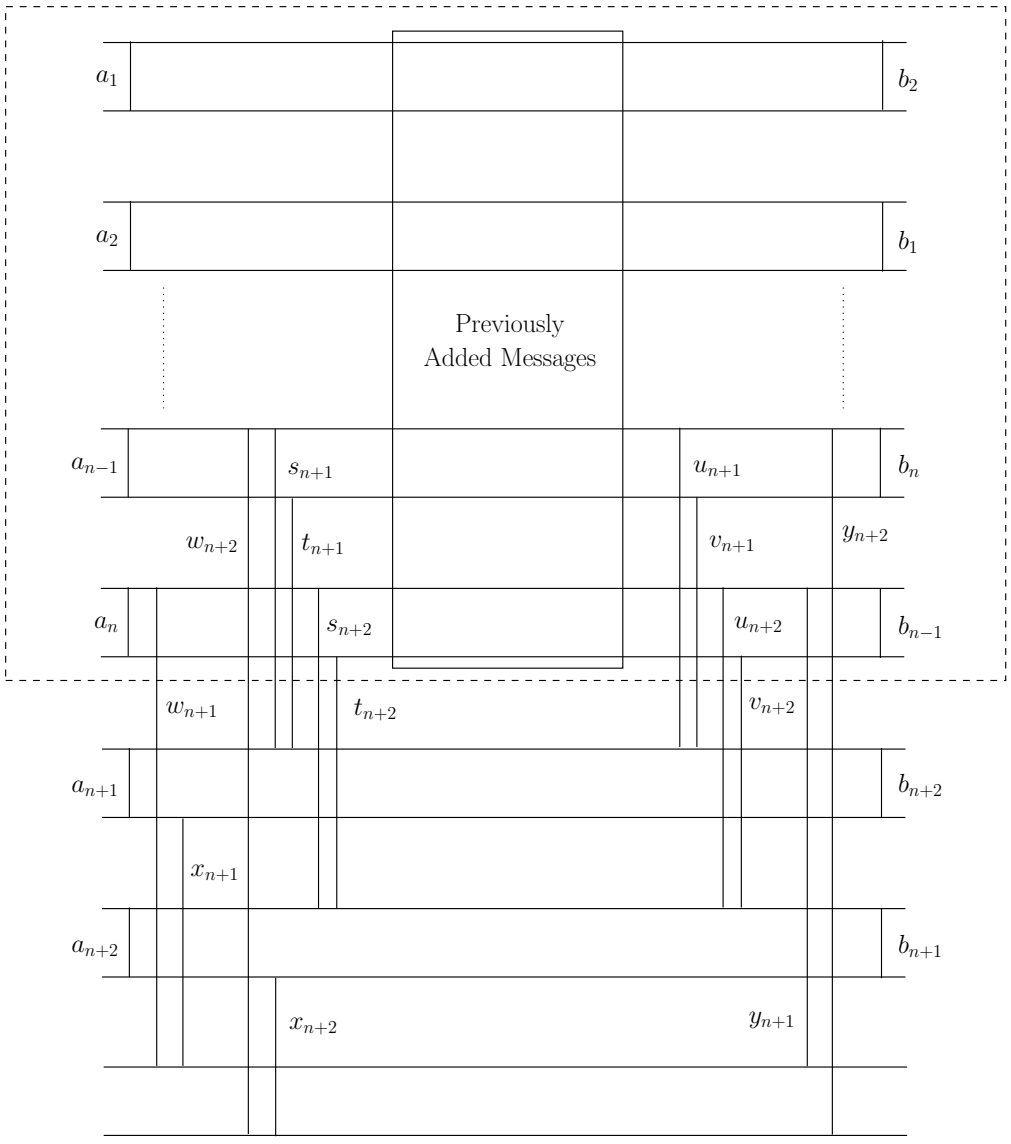

Figure 14: Constructing S_{n+2} from S_n using six additional processes.

$a_{n+1} \mapsto b_i$ for each $b_i \in B_{n-2}$ in C_{n+2}. We now summarize the function of all fourteen messages.

- Messages s_{n+1} and t_{n+1} ensure that $a_{n+1} \mapsto b_i$ for each $b_i \in B_{n-2}$ and $a_{n+1} \mapsto b_n$. Messages x_{n+1} and y_{n+1} ensure that $a_{n+1} \mapsto b_{n-1}$.

- Messages s_{n+2} and t_{n+2} ensure that $a_{n+2} \mapsto b_i$ for each $b_i \in B_{n-1}$. Messages x_{n+2} and y_{n+2} ensure that $a_{n+2} \mapsto b_n$.

- Messages u_{n+1} and v_{n+1} ensure that $a_i \mapsto b_{n+2}$ for each $a_i \in A_{n-1}$. Messages w_{n+1} and x_{n+1} ensure that $a_n \mapsto b_{n+2}$.

- Messages u_{n+2} and v_{n+2} ensure that $a_i \mapsto b_{n+1}$ for each $a_i \in A_{n-2}$ and $a_n \mapsto b_{n+1}$. Messages w_{n+2} and x_{n+2} ensure that $a_{n-1} \mapsto b_{n+1}$.

It can be verified that, even after adding the fourteen messages, $a_i \| b_i$ still holds for each $i = 1, 2, \ldots, n+2$. Therefore the poset induced on messages in $A_{n+2} \cup B_{n+2}$ by the synchronously precedes relation $\mapsto$ actually corresponds to the standard example S_{n+2}. We have,

Theorem 14 *For every $N \geq 2$, there exists a synchronously ordered computation on N processes such that the poset $(M, \mapsto)$ has a dimension of at least $2\lfloor N/6 \rfloor$.*

Observe that, in our construction, no process exchanges messages with more than six processes. Therefore the lower bound holds even if the communication topology is sparse and contains only $\Theta(N)$ edges.

6 Related Work

Fidge, in his paper on timestamping events in a distributed computation [9, 11], also describes a method for timestamping synchronous communication events (which is equivalent to timestamping synchronous messages) using traditional vector clocks. As opposed to processes in our model, processes in [11] are allowed to communication using both asynchronous and synchronous messages. We, on the other hand, assume that all messages are synchronous and our focus is on timestamping messages and events efficiently for such a computation.

Several techniques have been proposed to reduce the overhead imposed by Fidge/Mattern's vector clocks [9, 10, 11, 24]. Singhal and Kshemkalyani [27] present a technique to reduce the amount of data piggybacked on each message. The main idea is to only send those entries of the vector along with a message that have changed since a message was last sent to that process. Hélary *et al* [18] further improve upon Singhal and Kshemkalyani technique and describe a suite of algorithms that provide different trade offs between space overhead and communication overhead. The ideas described in the two papers are actually orthogonal to the ideas presented in this paper and, therefore, can also benefit our timestamping algorithm by reducing its overhead.

Fowler and Zwaenepoel [12] propose a variant of vector clocks in which each process only keeps *direct dependencies* on others. Although each process maintains a vector of size equal to the number of processes, only one integer is piggybacked on a message. For capturing transitive causal relations, however, it is necessary to recursively trace causal dependencies. This technique is therefore more

suitable for applications where precedence test can be performed offline. Jard and Jourdan [19] propose an algorithm that allows only relevant events to be tracked using a variation of direct depending mechanism, which they refer to as *adaptive timestamping*. Torres-Rojas and Ahamad [30] introduce another variant of vector clocks called *plausible clocks*. Unlike traditional vector clocks, plausible clocks are scalable because they can be implemented using fixed-length vectors independent of the number of processes. However, plausible clocks do not characterize causality completely because two events may be ordered even if they are concurrent. As a result, plausible clocks are useful only when imposing ordering on some pairs of concurrent events has no effect on the correctness of the application.

In [3], Basten *et al* introduce the notion of an *abstract event*. An abstract event is a non-empty subset of primitive events. Basten *et al* define two precedence relations on abstract events, namely strong precedence and weak precedence [3]. They also present techniques for timestamping abstract events to accurately capture the two precedence relations.

Several centralized algorithms for timestamping events have also been proposed [32, 33, 34]. They are mainly used for visualizing a distributed computation. An important objective of these algorithms is to reduce the amount of space required to store timestamps for all events in a computation while maintaining the time required for comparing two events (to determine their relationship) at an acceptable level. Ward presents two centralized algorithms to create vector timestamps whose size can be as small as the dimension of the partial order of execution [32, 33]. The second algorithm is an online version of the first one. The main idea is to incrementally build a realizer using Rabinovitch and Rival's Theorem [26], and then create timestamp vectors based on that realizer. In the online algorithm, the vector timestamps that have already been assigned to events may have to be changed later on arrival of a new event. In fact, timestamp of an event may be changed multiple times. Further, all timestamps may not be of the same length. This leads to a somewhat complicated precedence test. Ward and Taylor present an offline algorithm for timestamping events based on decomposing processes into a hierarchy of clusters [34]. The algorithm exploits the observation that events within a cluster can only be causally dependent on events outside the cluster through receive events from transmissions that occurred outside the cluster. As a result, non-cluster receive events can be timestamped much more efficiently than cluster receive events.

Recently, Agarwal and Garg [1] have proposed a class of logical clock algorithms based on the notion of *chain clocks*. Chain clocks can be used for tracking dependencies between *relevant* events based on generalizing a process to any chain in the computation poset. Their approach reduces

the number of components required in the vector clock when the set of relevant events is a small fraction of the total events. The algorithm in this paper is not dependent on any notion of relevance. Moreover, the algorithm by Agarwal and Garg [1] is centralized whereas the algorithm in this paper is completely distributed.

7 Conclusion

In this paper, we have shown that, when communication is synchronous, messages and events can be assigned timestamps using fewer than N components for a distributed system consisting of N processes. The main idea is to decompose the communication topology into edge groups and to use one component in the vector for each edge group. If the size of the edge decomposition is d, then our timestamps for messages contain d integers and timestamps for events contain $d + 4$ integers. For many common topologies including tree, ring, grid and hypercube, $d \leq \lceil N/2 \rceil$. As a result, for these topologies, our timestamping approach significantly outperforms traditional vector clocks. We have also shown that the precedence test for our timestamping mechanism requires only $O(1)$ time.

When messages can be timestamped in an offline manner, we have proved that timestamping can be done using at most $\lceil N/2 \rceil$ integers. Moreover, we have shown that any vector-based timestamping algorithm requires at least $2\lfloor N/6 \rfloor$ integers in the worst case.

References

[1] A. Agarwal and V. K. Garg. Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems. In *Proceedings of the, ACM Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2005.

[2] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, August 2001.

[3] T. Basten, T. Kunz, J. P. Black, M. H. Coffin, and D. J. Taylor. Vector Time and Causality among Abstract Events in Distributed Computations. *Distributed Computing (DC)*, 11:21–39, 1997.

[4] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and Asynchronous Communication in Distributed Computations. *Distributed Computing (DC)*, 9:173–191, September 1996.

[5] IBM Corporation. IBM Distributed Debugger for Workstations. Available at http://www.ibm.com/software/webservers/appserv/doc/v35/ae/infocenter/olt/index.html.

[6] O. P. Damani and V. K. Garg. How to Recover Efficiently and Asynchronously when Optimism Fails. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 108–115, Hong Kong, May 1996.

[7] R. P. Dilworth. A Decomposition Theorem for Partially Ordered Sets. *Annals of Mathematics*, 51:161–166, 1950.

[8] B. Dushnik and E. W. Miller. Partially Ordered Sets. *American Journal of Mathematics*, 63:600–610, 1941.

[9] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial-Ordering. In K. Raymond, editor, *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, pages 56–66, February 1988.

[10] C. J. Fidge. Partial Orders for Parallel Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 183–194, January 1989.

[11] C. J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[12] J. Fowler and W. Zwaenepoel. Causal Distributed Breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 131–141. IEEE Computer Society, 1990.

[13] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1991.

[14] V. K. Garg. *Elements of Distributed Computing*. John Wiley and Sons, Incorporated, New York, NY, 2002.

[15] V. K. Garg and C. Skawratananond. String Realizers of Posets with Applications to Distributed Computing. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 72–80, Newport, Rhode Island, August 2001.

[16] V. K. Garg and B. Waldecker. Detection of Weak Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 5(3):299–307, March 1994.

[17] D. Haban and W. Weigel. Global Events and Global Breakpoints in Distributed Systems. In *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, pages 166–175, January 1988.

[18] J.-M. Hélary, M. Raynal, G. Melideo, and R. Baldoni. Efficient Causality-Tracking Timestamping. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1239–1250, 2003.

[19] C. Jard and G.-V. Jourdan. Dependency Tracking and Filtering in Distributed Computations. Technical Report 851, IRISA, Campus de Beaulieu – 35042 Rennes Cedex – France, August 1994.

[20] J. A. Kohl and G. A. Geist. The PVM3.4 tracing facility and XPVM 1.1. Technical report, Computer Science and Mathematics Division Oak Ridge National Lab, Tennesse, USA, 1995.

[21] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. POET: Target-System Independent Visualizations of Complex Distributed-Applications Executions. *The Computer Journal*, 40(8), 1997.

[22] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[23] K. Marzullo and L. Sabel. Efficient Detection of a Class of Stable Properties. *Distributed Computing (DC)*, 8(2):81–91, 1994.

[24] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[25] V. V. Murty and V. K. Garg. Synchronous Message Passing. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 208–214, Phoenix, Arizona, USA, April 1995.

[26] I. Rabinovitch and I. Rival. The Rank of Distributive Lattice. *Discrete Mathematics*, 25:275–279, 1979.

[27] M. Singhal and A. Kshemkalyani. An Efficient Implementation of Vector Clocks. *Information Processing Letters (IPL)*, 43:47–52, August 1992.

[28] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill and The MIT Press, 1994.

[29] R. E. Strom and S. Yemeni. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[30] F. J. Torres-Rojas and M. Ahamad. Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. In *Proceedings of the 10th Workshop on Distributed Algorithms (WDAG)*, pages 71–88. Springer-Verlag, 1996.

[31] W. T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. The Johns Hopkins University Press, Baltimore, MD, 1992.

[32] P. A. S. Ward. An Offline Algorithm for Dimension-Bound Analysis. In Dhabaleswar Panda and Norio Shiratori, editors, *Proceedings of the International Conference on Parallel Processing*, pages 128–136. IEEE Computer Society, 1999.

[33] P. A. S. Ward. An Online Algorithm for Dimension-Bound Analysis. In P. Amestoy *et al*, editor, *Proceedings of the Euro-Par*, Lecture Notes in Computer Science (LNCS), pages 144–153. Springer-Verlag, 1999.

[34] P. A. S. Ward and D. T. Taylor. A Hierarchical Cluster Algorithm for Dynamic, Centralized Timestamps. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 585–593, April 2001.

[35] M. Yannakakis. The Complexity of the Partial Order Dimension Problem. *SIAM Journal on Algebraic and Discrete Methods*, 3:351–358, 1982.