Timestamping Messages in Synchronous Computations

Vijay K. Garg* and Chakarat Skawratananond

Electrical and Computer Engineering Department
The University of Texas at Austin
Austin, Texas 78712.

E-mail: {garg,skawrata}@ece.utexas.edu

Abstract

Determining order relationship between events in distributed computations is a fundamental problem with applications in distributed monitoring systems and fault-tolerance. Fidge and Mattern's vector clocks capture the order relationship with vectors of size N in a system with N processes. Since many distributed applications use synchronous *messages, it is natural to ask if the overhead can be reduced for these applications. In this paper, we present a new method of timestamping messages and events in synchronous computations that capture the order relationship with vectors of size less than or equal to the size of the vertex cover of the communication topology of the system. Our method is fundamentally different from that of Fidge and Mattern's technique. The timestamps in our method do not use one component per process but still guarantee that the order relationship is captured accurately. Our algorithm is online and only requires piggybacking of timestamps on program messages. It is applicable to all programs that either use programming languages which use synchronous communication such as CSP, or use synchronous remote procedure calls.*

1. Introduction

Determining order relationship between events in distributed computations is a fundamental problem with applications in distributed monitoring systems and fault-tolerance. For example, it is used to provide visualizations of the computation for debugging in systems such as POET [13], XPVM [12], and Object-Level Trace [11]. It is also used in the area of global property evaluation [5, 9]. In the area of fault-tolerance, the order relationship is used to determine if a process is *orphan* and needs to be rolled back [19, 2].

Vector clocks introduced by Fidge [5] and Mattern [15] (FM for short) is widely used to capture causality and concurrency between events in distributed systems. This relationship between events is based on Lamport's *happened before* relation [14]. However, vector clock mechanism does not scale well because it imposes $O(N)$ of local storage on each process and $O(N)$ message overhead in a system with N processes.

Charron-Bost [1] has shown that for every $N \geq 2$, there exists a computation on N processes such that vector clocks of size at least N are required to capture happened before relation and concurrency between events. In [8], we have shown that Fidge and Mattern's vectors are equivalent to *string* realizers of the poset corresponding to the distributed computation and the vectors of size equal to the string dimension [4, 8] of the poset are necessary and sufficient for timestamping events. However, timestamps that are determined using dimension theory cannot be used in an online manner because the knowledge of the entire poset is necessary to determine a realizer. Further, the problem of determining the size of the smallest realizer is NP-complete [24]. Although, these results show that in the worst case the timestamps may require N-dimensional vector clocks, they do not exclude timestamps which use less than N coordinates for interesting subset of computations on N processes. From the practical point of view, the natural question to ask is whether there exists an efficient algorithm for an interesting class of applications in which timestamping is scalable.

In this paper, we show that timestamping can be done more efficiently in distributed computations that uses *synchronous* messages. A message is called *synchronous* when the send is blocking, i.e., the sender waits for the message to be delivered by the receiver before executing further. Synchronous communication is widely supported in

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

many programming languages and standards such as CSP [10], Ada Rendezvous, and synchronous Remote Procedure Calls (RPC). While programming using asynchronous communication allows higher degree of parallelism and is less prone to deadlocks, algorithms using synchronous message-passing are easier to develop and verify. Also, the implementation of asynchronous communication requires buffer management and flow control mechanisms. Implementation of synchronous messages requires that the sender wait for an acknowledgment from the receiver before executing further.

It is known that a synchronous computation, i.e., a computation based on synchronous messages, is logically equivalent to a computation in which all messages are instantaneous. In other words, we can always draw the time diagrams for synchronous computations with *vertical* message arrows [1, 16]. If we ignore *internal* events in a computation, then timestamping events is equivalent to timestamping messages in the computation. We define a partial order $(M, \mapsto)$, where M is the set of messages in the computation and $\mapsto$ is the order relationship between messages (defined formally in Section 2). We then provide an algorithm that assigns vector clocks to the messages such that for any two messages m_1 and m_2,

$$m_1 \mapsto m_2 \iff vector(m_1) < vector(m_2)$$

Our method captures the order relationship with vectors of size less than or equal to the size of the vertex cover of the communication topology of the system. Our method is fundamentally different from that of Fidge and Mattern's technique. The timestamps in our method do not use one component per process but still guarantee that the order relationship is captured accurately. We use the notion of *edge decomposition* defined in Section 3 to partition edges in the communication topology graph of the system. We assign each component of the vector clock to each edge group in the edge decomposition. Our algorithm is online and only requires piggybacking of timestamps on program messages and acknowledgements. We exploit the communication topology of the system to reduce the size of vector clocks. For example, an integer is sufficient to timestamp in a system with a *star* or a *triangle* topology. For a client-server based system with a constant number of servers and a variable number of clients, vectors in our timestamps require a constant number of coordinates.

We also present an offline algorithm and show that timestamping of messages does not take more than $N/2$ components for any synchronous computation. This result is derived using dimension theory of posets.

The remainder of this paper is organized as follows. Section 2 provides technical background for the problem discussed in this paper. The online algorithm is given in Section 3. Section 4 describes the offline algorithm. In Section 5, we discuss how the proposed algorithms can be extended for timestamping internal events. Section 6 compares our work with others.

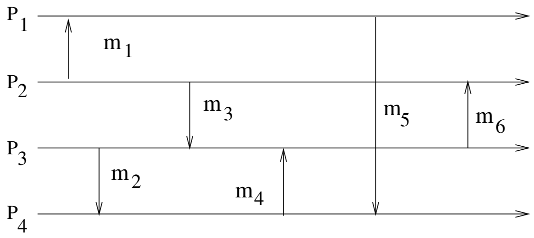

Figure 1. A synchronous computation with 4 processes.

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 synchronous computation. It can be shown that a computation is synchronous 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 sending and the receiving events associated with each message have the same timestamp. Therefore, the time diagram of the computation can be drawn such that all messages arrows are vertical [1] (see Figure 1).

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 synchronous computation forms a poset $\mathcal{M} = (M, \mapsto)$, where $\mapsto$ is the transitive closure of $\triangleright$ defined as follows.

$$m_1 \triangleright m_2 \iff \begin{cases} m_1.send \prec m_2.send & \text{, or} \\ m_1.send \prec m_2.receive & \text{, or} \\ m_1.receive \prec m_2.send & \text{, or} \\ m_1.receive \prec m_2.receive & \end{cases}$$

We say m_1 *synchronously precedes* m_2 when $m_1 \mapsto m_2$. And when we have $m_1 \mapsto m_2 \mapsto \ldots \mapsto m_k$, we say that there is a *synchronous chain* of size k from m_1 to m_k. We denote $m_1 || m_2$ when $m_1 \not\mapsto m_2$ and $m_2 \not\mapsto m_1$.

In the example given in Figure 1, $m_1 || m_2$, $m_1 \triangleright m_3$, $m_2 \mapsto m_6$, 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 need a timestamping mechanism that

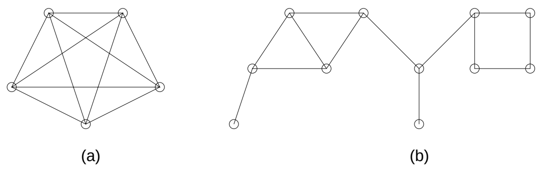

Figure 2. Examples of the communication topologies. (a) The system where every process can communicate directly with each other. (b) The system where not every pair of processes communicate directly with each other.

assigns a vector to each message m (or equivalently, the send and the receive event). Let $v(m)$ denote the vector assigned to message m. Our goal is to assign timestamps that satisfies the following property,

$$m_1 \mapsto m_2 \iff v(m_1) < v(m_2) \tag{1}$$

Given any two vectors u and v of size t, we define the relation $<$ as follows.

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

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

From Equations (1) and (2), one can determine if $m_1 \mapsto m_2$ by checking whether $v(m_1) < v(m_2)$. If $v(m_1)$ is not less than $v(m_2)$ and $v(m_2)$ is not less than $v(m_1)$, then we know that $m_1 || m_2$.

3. Online Algorithm

In this section, we give an algorithm to assign $v(m)$ for a message m such that Equation (1) is satisfied. Whereas FM vectors are based on the idea of assigning one component for each process, our algorithm assigns one component to each *edge group*. We first define the notion of *edge decomposition* and *edge group*.

3.1. Edge Decomposition

The communication topology of a synchronous 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 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 communicate 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 synchronous computation with these topologies are always totally ordered. In fact, we have the following.

Lemma 1 *The message sets for all synchronous 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 synchronous 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 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 synchronous 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 synchronous system. 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) $\forall i, j : E_i \cap E_j = \emptyset$, 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.

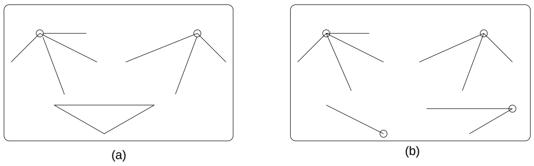

Figure 3. Edge decompositions of the fully-connected system with 5 processes. (a) The first decomposition consisting of 2 stars and 1 triangle. (b) The second decomposition consisting of 4 stars.

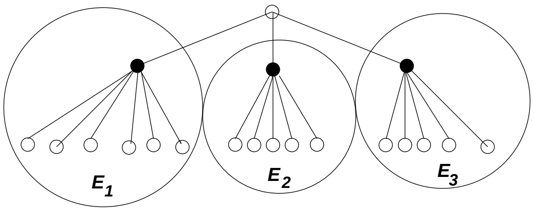

Figure 4. A tree-based computation with 20 processes.

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 synchronous system 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.

We will discuss techniques for edge decomposition that minimize the number of edge groups in Section 3.3.

3.2. Algorithm

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 by all processes in the system.

The online algorithm is presented in Figure 5. Due to the implementation of synchronous message ordering [16], we assume that for each message sent from P_i to P_j, there exists an acknowledgement sent from P_j to P_i. Essentially, to timestamp each message, the sender and the receiver must first exchange their vector clocks. Then, each process computes the component-wise maximum between the local vector and the vector received (Line (5) and (9)). Finally, both the sender and the receiver increment the g^{th} element of their vectors where the channel that the message is sent belongs to the g^{th} group in the edge decomposition (Line (6) and (10)). The resulting vector clock is the timestamp of this message.

P_i ::
var
  v_i : array[$1..d$] of integers, initially $\mathbf{0}$;
  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** acknowledgement(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 acknowledgement(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. The Online Algorithm.

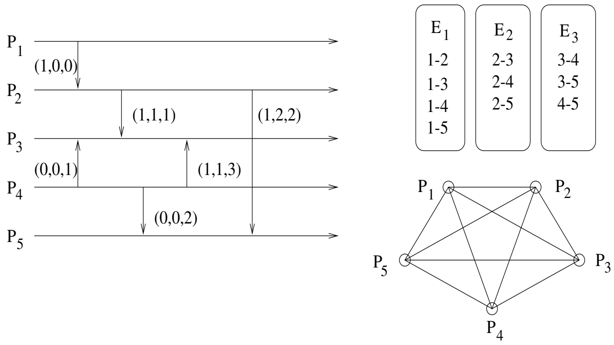

Figure 6. A synchronous computation with 5 processes, and its edge decomposition.

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 creates vector timestamps for messages in synchronous systems such that these timestamps encode poset $(M, \mapsto)$. The channel in which a message m_x is sent through must be a member of a group in the edge decomposition. We use $e(m_x)$ to denote the index of the group to which this channel belongs in the edge decomposition. Clearly,

Lemma 3 $m_i || m_j \Rightarrow e(m_i) \neq e(m_j)$

Theorem 4 *Given an edge decomposition of a synchronous* *system, the algorithm in Figure 5 timestamps messages such that* $m_1 \mapsto m_2 \iff v(m_1) < v(m_2)$

Proof: First, we show that $m_1 \mapsto m_2 \Rightarrow v(m_1) < v(m_2)$. Since vector clocks are exchanged between the sender and the receiver, and the component-wise maximum is computed between the received vector and the local vector, it is easy to see that if m_1 synchronously precedes m_2, then $v(m_1) \leq v(m_2)$. We now claim that

$$m_1 \mapsto m_2 \Rightarrow v(m_1)[e(m_2)] < v(m_2)[e(m_2)] \quad (3)$$

This is true because before the vector is assigned to m_2, $v(m_2)[e(m_2)]$ is incremented. Thus, we have $m_1 \mapsto m_2 \Rightarrow v(m_1) < v(m_2)$.

We now show the converse,

$$m_1 \not\mapsto m_2 \Rightarrow \neg(v(m_1) < v(m_2))$$

Due to the definition of vector order, it is sufficient to show that

$$m_1 \not\mapsto m_2 \Rightarrow v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$$

We do a case analysis.

(Case 1: $m_2 \mapsto m_1$**)**

From Equation (3), by changing roles of m_1 and m_2, we get that $v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$.

(Case 2: $m_1 || m_2$**)**

We prove by induction on k, the size of the longest synchronous chain from a *minimal message* in the poset $(M, \mapsto)$ to m_2. A message m is minimal if there is no message m' in the computation such that $m' \mapsto m$.

(Base: $k = 1$**)** m_2 is a minimal message.

From Lemma 3 and $m_1 || m_2$, $e(m_1) \neq e(m_2)$. Since m_2 is a minimal message by the initial assignment of the vector clock, both sender and the receiver have 0 as the component for $e(m_1)$ and the componentwise maximum also results in 0 for $e(m_1)$. Further, since $e(m_1) \neq e(m_2)$ the component for $e(m_1)$ is not incremented. Hence, $v(m_2)[e(m_1)] = 0$.

We now claim that $v(m_1)[e(m_1)] \geq 1$. This is true because we increment the component for $e(m_1)$ before assigning the timestamp for m_1. Since the value of all entries are at least 0, it will be at least 1 after the increment operation.

From, $v(m_2)[e(m_1)] = 0$ and $v(m_1)[e(m_1)] \geq 1$, we get that $v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$.

(Induction: $k > 1$**)**

Let m_3 be any message such that $m_3 \rhd m_2$. We know that $m_1 \not\mapsto m_3$, otherwise $m_1 \mapsto m_2$. By induction hypothesis,

$$m_1 \not\mapsto m_3 \Rightarrow v(m_3)[e(m_1)] < v(m_1)[e(m_1)]$$

To obtain $v(m_2)$, the sender and receiver of m_2 exchange timestamps of any immediately preceding message (if any). We also know that the $e(m_1)^{th}$ component of vectors from both the sender and receiver are less than $v(m_1)[e(m_1)]$ due to induction hypothesis. Hence, it stays less after the component-wise maximum. Further, since $e(m_1) \neq e(m_2)$ the component for $e(m_1)$ is not incremented. Therefore, $v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$. ■

Given an edge decomposition of size d, our online algorithm has $O(d)$ message and space overhead.

3.3. Good Edge Decompositions

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 5 *Let* $G = (V, E)$ *be communication topology of a synchronous system. Let* $\beta(G)$ *be the size of the optimal vertex cover of* G*. Then vector clocks of size* $min(\beta(G), N - 2)$ *is sufficient to timestamp messages.*

Proof: From the definition of vertex cover, every edge is incident to some vertex in the vertex cover. For every edge we assign some vertex in 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 $\beta(G) = 2t$.

Since the problem of obtaining minimum vertex cover is NP-hard [7], it is unlikely that there exists an optimal algorithm for edge decomposition of a general graph. 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 7. 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 8 shows the operation of our edge decomposition algorithm on the communication topology shown in Figure 2(b). Figure 8(b),(c), and (d) shows the first, second, and third step of the algorithm, respectively. In Figure 8(e), the execution loops back to the first step, edge (j, k) is output, and the program exits. Figure 8(f) shows the optimal 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 the optimal edge decomposition.

Theorem 6 *The algorithm in Figure 7 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,

Input: Undirected graph $G = (V, E)$;
Output : edge decomposition,$(\{E_1, \dots, E_d\})$;
// Each E_i is either a star or a triangle
(01) $F := E$;
(02) **while** $F \neq \emptyset$ **do**
//First Step:
(03) **while** $\exists$ 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 7. Approximation algorithm for edge decomposition.

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 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 7 *The algorithm in Figure 7 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

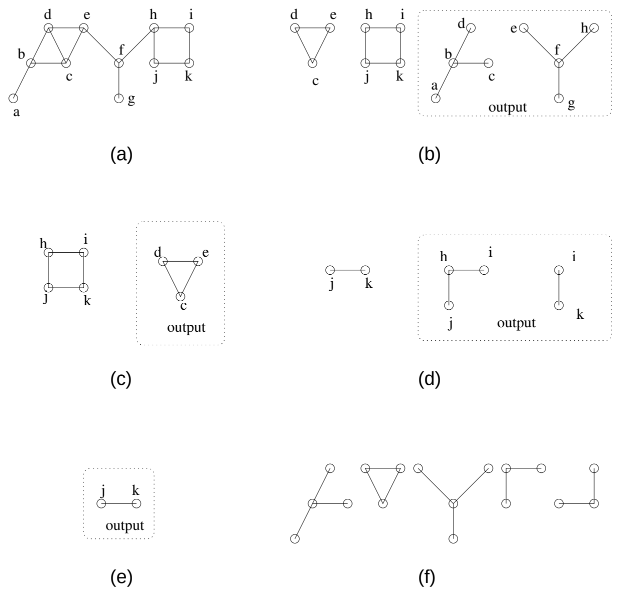

Figure 8. A sample run of the proposed decomposition algorithm.

one hop away from the leaves then we will eventually delete all the edges.

Thus, the set H constructed in the proof of Theorem 6 consists of edges added only in step 1. Since we add exactly one edge group for every edge added to H, the optimality follows. ■

While the size of the vector for the fully-connected system is still $O(N)$, the vector size of the system with tree-based topology may not grow considerably. In particular, if the number of processes in the system increases without changing the size of its *edge decomposition*, the size of our vector clocks is constant. This has a significant impact because tree is a popular structure used as a communication topology for distributed computing systems.

As another example, consider a client-server based system where (1) clients can only communicate with servers and (2) all interactions in the system are through synchronous RPC or RMI. In this case, the communication topology can be decomposed with one star rooted at each server. Thus, it is sufficient to use vector clocks of size equal to the number of servers to timestamp messages in the system.

4. Offline Algorithm

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

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

(1) Let w be the width of poset $\mathcal{M}$. From Theorem 8, $w \leq \lfloor \frac{N}{2} \rfloor$.

(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 [21])

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

Figure 9. Offline Algorithm.

4.1. 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 *chain 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 $P = \cap\mathcal{R} = \cap_{i=1}^{t} L_i$.

4.2. The Algorithm

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

Theorem 8 *Given a poset $(M, \mapsto)$ formed by messages in a synchronous computation with N processes, vector clocks of size $\lfloor \frac{N}{2} \rfloor$ can be used to encode poset $(M, \mapsto)$.*

Proof: For any subset $L \subseteq M$ such that $|L| > \lfloor \frac{N}{2} \rfloor$, 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 $\lfloor \frac{N}{2} \rfloor + 1$ 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 $\lfloor \frac{N}{2} \rfloor$. From Dilworth's theorem [3], for any poset P, $dim(P) \leq width(P)$. Hence, $dim(M, \mapsto) \leq \lfloor \frac{N}{2} \rfloor$. ■

As a result from Theorem 8, we get the offline algorithm as shown in Figure 9.

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

5. Timestamping Events

Thus far, we had focused our attention on timestamping send/receive (external) events in synchronous systems. We now show how to extend our algorithm to timestamp internal events such that the resulting timestamps capture Lamport's happened before relation.

For simple exposition, let us first assume that we have exactly one internal event between any two external events. Later we show how this algorithm can be extended easily to handle the general case. Recall that for each synchronous message m sent from a process P_i to another process P_j, there is an acknowledgement sent from P_j to P_i. It is important to note that happened before relation between events uses messages and their acknowledgements as well.

We now give the timestamping algorithm for internal events. Each event e is assigned with a tuple $(prev(e), succ(e))$ where $prev(e)$ is the timestamp of the message immediately prior to e, and $succ(e)$ is the timestamp of the message immediately after e. If there is no message before e, $prev(e)$ is a zero vector (denoted by $\mathbf{0}$). If there is no message after e, $succ(e)$ is a vector where all elements are ∞. Observe that an internal event can be assigned a timestamp only after the process knows the timestamp of the message after e.

In the following, we show that the proposed timestamps capture causal relationship between events in the synchronous systems. That is,

$$e \rightarrow f \iff succ(e) \leq prev(f)$$

where $\rightarrow$ denotes Lamport's happened before relation. We say that there is a causal chain of size k between e_1 and e_k when $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_k$.

We now ready to prove the property of the proposed timestamp algorithm.

Theorem 9 $e \rightarrow f \iff succ(e) \leq pref(f)$

Proof: First, we have to prove that

$$e \rightarrow f \Rightarrow succ(e) \leq prev(f) \quad (4)$$

If e and f are on the same process then the result is trivially true. Otherwise, since $e \rightarrow f$, there must be a causal chain between e and f. If m_e is the message immediately after e, and m_f is the message immediately before f, we know that $m_e \mapsto m_f$ or $m_e = m_f$. From Theorem 4, $succ(e) \leq prev(f)$.

Conversely, we have to prove that $succ(e) \leq prev(f) \Rightarrow e \rightarrow f$. We know that the vector timestamp of m_e is less than or equal to that of m_f. From the property of message timestamps (Theorem 4), we get that $m_e \mapsto m_f$ or $m_e = m_f$. From the definition of $\mapsto$, there must be a causal chain from e to f formed by either the application messages or the acknowledgements or both. ■

If there are more than one internal event between any two external events, the timestamp for each internal event becomes a triple $(prev(e), succ(e), c(e))$, where $c(e)$ is the value of $counter_i$ an integer maintained by each process P_i. Initially, $counter_i$ is zero, and is reset to zero whenever a new external event occurs in P_i. Further, $counter_i$ is incremented for each occurrence of an internal event. It is easy to verify that $e \rightarrow f \iff c(e) < c(f)$ when $prev(e) = prev(f)$ and $succ(e) = succ(f)$.

6. Related Work

Different implementations of Fidge [5] and Mattern [15] Vector Clock have been proposed. Singhal and Kshemkalyani's [18] approach reduces the amount of data sent over the network. This is possible because of the increase in the amount of data stored by each process. Fowler and Zwaenepoel [6] proposed an implementation where each process only keeps direct dependencies on others. Thus, only one scalar is required to represent a vector clock. However, for capturing transitive causal relations, it is necessary to recursively trace causal dependencies. This technique is therefore more suitable for applications where precedence test can be performed off-line.

Torres-Rojas and Ahamad [20] introduced a class of scalable vector clocks called Plausible Clocks. It is scalable because it can be implemented using fixed-length vectors. Plausible Clocks do not characterize causality completely, that is, they do not guarantee that certain pairs of concurrent events will not be ordered. As a result, plausible clocks are useful for any application where imposing orderings on some pairs of concurrent events have no effects on the correctness of the results. Mutual consistency protocols for shared objects are examples of applications that can use plausible clocks.

Ward [22] presents an algorithm to create vector timestamps whose size can be as small as the dimension of the partial order of execution. The algorithm incrementally builds a realizer using Rabinovitch and Rival's Theorem [17], and then creates timestamp vectors based on that realizer. Therefore, vector timestamps that have already been assigned to events may have to be changed later. Further, all timestamps may not be of the same length. This leads to a complicated precedence test. Moreover, each coordinate is required to be a real number. Our algorithm does not suffer from any of these disadvantages.

A hierarchical cluster algorithm for online, centralized timestamp was presented in [23]. The algorithm is based on the fact 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. The precedence-test method in this algorithm is $O(c)$ where c is the size of the cluster.

Our proposal generates vector timestamps that completely captures the relations between synchronous messages. We exploit the configuration of the system topology to reduce the size. The length of our vector clocks is never changed during the execution of the algorithm. Once the timestamp is assigned, it is never changed. Our precedence test is therefore straightforward.

References

[1] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. *Distributed Computing*, 9:173–191, September 1996.

[2] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115, Hong Kong, may 1996. IEEE.

[3] R. P. Dilworth. Decomposition theorem for partially order sets. *Ann. Math.*, 51:161–165, 1950.

[4] B. Dushnik and E. Miller. Partially ordered sets. *Amer. J. Math.*, 63:600–610, 1941.

[5] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, January 1989.

[6] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 131–141. IEEE Computer Society Press, 1990.

[7] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[8] V. K. Garg and C. Skawratananond. String realizers of posets with applications to distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing(PODC'01)*, pages 72–80, May 2001.

[9] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, Mar. 1994.

[10] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

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

[12] J. A. Kohl and G. A. Geist. The pvm3.4 tracing facility and xpvm 1.1. Technical report, Comp. Science and Math. Division Oak Ridge National Lab, TN, USA, 1995.

[13] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8), 1997.

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

[15] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed algorithms*, pages 215–226, 1989.

[16] 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, April 1995.

[17] I. Rabinovitch and I. Rival. The rank of distributive lattice. *Discrete Math.*, 25:275–279, 1979.

[18] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[19] R. E. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[20] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *Proc. 10th Int. Workshop on Distributed Algorithms*, pages 71–88. Springer-Verlag LNCS, October 1996.

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

[22] P. Ward. A framework algorithm for dynamic, centralized, dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, Toronto, Canada, November 2000.

[23] P. Ward and D. J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *IEEE 21st International Conference on Distributed Computing Systems*, Phoenix, April 2001.

[24] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, September 1982.