Exploiting Locality in Maintaining Potential Causality

Sigurd Meldal* **Sriram Sankar** **James Vera**
Program Analysis and Verification Group
Computer Systems Laboratory
Stanford University
Stanford, California 94305

Abstract

In distributed systems it is often important to be able to determine the temporal relationships between events generated by different processes. An algorithm to determine such relationships is presented in [3] and [5]. This algorithm has many favorable attributes such as it allows for any kind of interprocess communication, and it requires no extra synchronization messages, additional communication links or central timestamping authority. The algorithm, however, requires $O(n)$ space for each process (where n is the number of processes). *i.e.*, it requires an overall space of $O(n^2)$. This can be a large overhead especially when there are a very large number of processes.

By cutting down on this generality, we can significantly decrease the amount of space required to determine temporal relationships. In this paper, we show how one may reduce the space requirements by assuming that the communication links between processes *is static and known ahead of time*; and also that one is interested only in determining the temporal ordering between messages arriving *at the same process*. We argue that these assumptions are reasonable to make for a large class of problems.

1 Introduction

There are two general activities that take place in a distributed system: *local activities* that are performed independently by each process and *synchronization activities* during which two or more processes interact with each other and possibly exchange information. These two activities are collectively referred to as *events*. Lamport [4] shows that the temporal ordering of events in a distributed system execution is a partial order. It is sometimes impossible to say that one of two events occurred first, hence in general this ordering cannot be a total order. Lamport also goes on to define a "happened before" relation which describes this partial order. In this definition, the only synchronization activities assumed are the sending and receiving of messages. The "happened before" relation is defined by the following three rules:

1. If a and b are events in the same process and a comes before b, then a "happened before" b.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then a "happened before" b.
3. If a "happened before" b and b "happened before" c, then a "happened before" c.

We use the term *potential causality* [5] to denote the "happened before" relation — *i.e.*, a potentially causes b is equivalent to a "happened before" b for the purposes of this paper.

Lamport provides an algorithm to determine a total ordering of events that is a consistent extention of the "happened before" relation. Subsequently, Fidge [3] and Mattern [5] developed similar algorithms to determine potential causality relationships between events. These algorithms mark each event performed with timestamp information; and potential causality between two events can be determined by comparing their respective timestamps. These algorithms have many favorable attributes such as that they require no extra synchronization messages, no additional communication links and no central timestamping authority. The algorithms require $O(n)$ space for each process (where n is the number of processes). This is because the algorithms require each process to maintain some information about each of the other processes. Hence the overall space requirement is $O(n^2)$. This amount of space may seem excessive, but Charron-Bost [2] has shown that this space is necessary. Since these algorithms are very similar, we shall refer to them collectively as Algorithm **FM**.

In many applications, it is not necessary to be able to determine potential causality relationships between any two arbitrary events. During our work on prototyping language design [1], we realized the need to determine potential causality information only between messages that were sent to the same process. Furthermore, the communication paths between processes are static and known ahead of time. In this paper, we present a modification of Algorithm **FM** which takes advantage of these properties and requirements. For a large class of architectures, our algorithm results in substantial savings in space and thus bandwidth (the amount of extra information to be tagged onto each message) — this savings is especially significant when the number of processes is large. In the worst case, our algorithm requires no more space than does Algorithm **FM**.

There are many situations where our algorithm can be used quite effectively. We shall illustrate the potential for

*On leave from the University of Bergen, Norway.

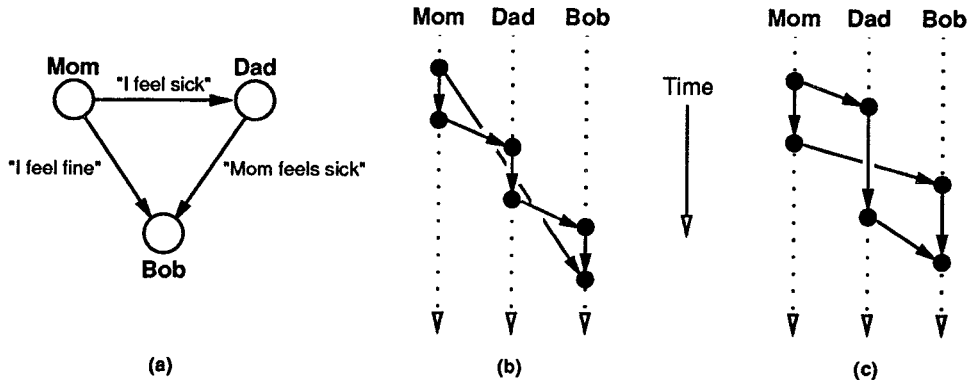

Figure 1: Messages from home

saving space using the following example of the world's postal system. In this system, the processes are the world's population and these processes communicate to each other only be sending letters. Assume that Bob receives a letter from his Mom saying "I feel fine" and a letter from his Dad saying "Mom feels sick". By determining the potential causality relationship between these two letters, Bob can determine which letter contains a more up-to-date status on his mother's health. Figure 1 illustrates this example — Figure 1(a) shows the letters that were sent. Figure 1(b) and 1(c) show the two different cases that Bob has to consider. We are assuming that the only means of synchronization is the sending of messages. There is no global clock that is visible to Mom, Dad, and Bob.

If Bob used Algorithm **FM**, he would need to keep track of information on every person in the world — an impossible task. It seems obvious that this amount of information is not really necessary. The reason why Algorithm **FM** requires this information is that it assumes any kind of communication is possible — for example, Mom may have communicated to Dad that she feels sick by forwarding the message through the entire world's population! The algorithms we present in this paper use our knowledge of the actual communication paths to reduce the space required to determine potential causality information.

We describe our algorithm for use on an asynchronous message passing system — though extending it to other distributed system architectures is quite simple. We therefore assume that there are events of only the following three kinds: (1) events corresponding to local activity on a process, (2) events corresponding to the sending of a message from one process to another, and (3) events corresponding to the receiving of a message by one process from another. In Section 2, we give an overview of Algorithm **FM**. We refine this algorithm in Sections 3 and 4. In Section 3 we show how the special nature of our problem can be used to reduce the amount of information necessary to determine potential causality. Section 4 uses the results of Section 3 to reduce the space requirements of Algorithm **FM** while still retaining enough information to determine potential causality between messages arriving at the same node. A series of algorithms are presented, each one improving on the previous algorithm. Section 5 illustrates the application of our algorithms on some common communication topologies. Section 6 summaries other work in the area. Section 7 concludes this paper.

2 Algorithm FM

In this algorithm, every process maintains a natural number to represent their local times. These numbers are referred to as *local clocks*. Each process initializes its local clock to 0 and increments it at least once before performing each event. When processes synchronize (in our case by the sending and receiving of messages), they pass on whatever local clock information they have to each other. In this manner, each process is updated on the local times of each of the other processes.

Hence each process maintains its own local clock information, and also whatever local clock information of the other processes it can obtain during synchronization. We find it convenient to represent the information maintained by each process as a mapping from processes to natural numbers. We refer to these mappings as *clocks* (as opposed to "local clock"). We refer to the clock maintained by process p as FM_p. Hence, $FM_p(q)$ (the value FM_p maps q to) is the most up-to-date information that process p has of the local clock of q.

A particular value of a clock is referred to as a *timestamp*. Every event is "timestamped" with the current value of the clock of the process performing this event. Also, every message is timestamped with the current value of the clock of the process sending the message. We refer to the timestamps of events e and messages m as FM_e and FM_m respectively.

For two events e_1 and e_2, their potential causality relationship is determined by comparing FM_{e_1} and FM_{e_2}. We use $e_1 \prec e_2$ to denote that e_1 occurs earlier than e_2 in the potential causality partial order (e_1 potentially caused e_2); and we use $e_1 \parallel e_2$ to denote that neither $e_1 \prec e_2$ nor $e_2 \prec e_1$ (events e_1 and e_2 happened concurrently).

Algorithm **FM** is now described in detail. The algorithm is divided into two parts — Propagation Algorithm 1, which maintains the clock information, and Determination Algorithm 1, which determines potential causality information based on this information.

Propagation Algorithm 1:
Initialization: For all processes p, set $FM_p(q)$ to 0 for all processes q.

Incrementing the local clock: Each process p increments its local clock *before* it performs each event. *i.e.*, $FM_p := FM_p[p \leftarrow FM_p(p) + 1]$.
When process p sends a message m: The current value of p's clock is attached to m. *i.e.*, $FM_m := FM_p$.
When a message m arrives at process q from process p: If m contains more recent clock information than q does of any process, q updates its clock with this information. However, $FM_m(p)$, which corresponds to the local clock value of p when it sent m, is incremented by 1 before q updates its clock. This is meant to take into account the message transmission delay and allows for a faster determination algorithm (see Determination Algorithm 1). We define the clock $sup(c_1, c_2)$, where c_1 and c_2 are themselves clocks as follows: For each process p, $sup(c_1, c_2)(p) = max(c_1(p), c_2(p))$. This step of the algorithm can then be represented as $FM_q := sup(FM_q, FM_m[p \leftarrow FM_m(p) + 1])$.

Determination Algorithm 1:
If e_1 and e_2 are two events performed by processes p_1 and p_2 respectively, then:

if $FM_{e_1}(p_1) < FM_{e_2}(p_1)$ **then**
    $e_1 \prec e_2$
elsif $FM_{e_2}(p_2) < FM_{e_1}(p_2)$ **then**
    $e_2 \prec e_1$
else
    $e_1 \parallel e_2$
end if;

Theorem 1 *If the clocks are maintained using Propagation Algorithm 1, then the potential causality information derived by Determination Algorithm 1 is correct.*

For a proof, please refer to [3].

3 Exploiting Graph Connectivity Information

Algorithm **FM** requires $O((1 + e_p) \times n)$ space for each process p, where e_p is the number of events whose timestamps p needs to keep track of. The algorithm also requires a bandwidth of $O(n)$ per message for propagating clock information. This can amount to a substantial overhead in space and bandwidth as the example in Section 1 illustrates.

However, we are only interested in comparing the potential causality relationship between messages that are sent to the same process. We also assume that the communication paths between processes are static and known ahead of time. These assumptions permit us to optimize Algorithm **FM** by decreasing the amount of clock information maintained by each process. As an illustration, in the example of Section 1, if Bob never received messages from Bush or Gorbachev (directly or indirectly), it would be unnecessary for Bob to keep track of the local clocks of Bush and Gorbachev. Note however, that these optimizations limit the amount of potential causality information that can be determined. For example, our algorithm will not permit us to determine the potential causality relationship between the sending of the letter from Mom to Bob and (say) the receiving of a letter by Gorbachev from Bush.

We model the communication path information as a directed graph, the nodes corresponding to the processes and the edges corresponding to the direct communication links. $p_1 \rightarrow p_2$ denotes the existence of an edge from p_1 to p_2. $\xrightarrow{+}$ denotes the transitive closure of $\rightarrow$.

As we develop our algorithms, we shall assume that the messages we are comparing are m_1 and m_2. We shall also assume that the senders of these messages are p_1 and p_2 respectively, and the recepient of these messages is p_R. The algorithms being developed will determine the potential causality relationship with respect to the sending of these messages.

We now present a refinement of Determination Algorithm 1, which makes use of the communication path information. This refinement is based on the following observations:

Lemma 2 *For two distinct processes p_1, p_2, if there is no path from p_1 to p_2, then Algorithm* **FM** *will never change $FM_{p_2}(p_1)$ from its initial value of* 0. *Hence $FM_{m_2}(p_1)$ will always be* 0 *in this situation.*

Justification. Consider the set of all processes p for which $FM_p(p_1)$ is non-zero. We shall refer to this set as S. We shall show (by induction) that at any time, for every member p of S, there is a path from p_1 to p.

Initial case: S is empty. Hence the property holds trivially.

When a new element is added to S, it is either p_1 itself (since the algorithm can increment $FM_{p_1}(p_1)$ in which case, the property still holds; or it can be a process p which receives a message m from process q such that $FM_m(p_1)$ is non-zero. This means that q is already in S, and by the induction hypothesis, there is a path from p_1 to q. Hence there is a path from p_1 to p also. □

Lemma 3 *The values of $FM_{m_1}(p_1)$ and $FM_{m_2}(p_2)$ are always greater than* 0.

Justification. Obvious from the algorithm definition. □

These observations tell us that if there is no path from p_1 to p_2, $FM_{m_2}(p_1)$ cannot be greater than $FM_{m_1}(p_1)$, and hence this comparison need not be made in Algorithm **FM**.

Determination Algorithm 2:

if $p_1 = p_2$ **then**
    order the messages by their sending sequence
elsif $p_1 \xrightarrow{+} p_2$ **and then**[1]
        $FM_{m_1}(p_1) < FM_{m_2}(p_1)$ **then**
    $m_1 \prec m_2$
elsif $p_2 \xrightarrow{+} p_1$ **and then**
        $FM_{m_2}(p_2) < FM_{m_1}(p_2)$ **then**
    $m_2 \prec m_1$
else
    $m_1 \parallel m_2$
end if;

Although Determination Algorithm 2 may appear more complex than Determination Algorithm 1, it reduces the amount of clock information required to determine potential causality information. It does this by first checking for the existence of paths. The clock information is used only if these paths exist in the graph. Furthermore, Determination Algorithm 2 can be optimized if the path information is known

[1] **and then** is the short-circuit **and** operator.

ahead of time. For example, if it is known that there is no path between p_1 and p_2 in either direction, Determination Algorithm 2 can be optimized to determine that $m_1 \parallel m_2$ without checking any timestamp values.

Determination Algorithm 2 (and the other determination algorithms that follow) handles $p_1 = p_2$ as a special case. The amount of clock information that needs to be maintained can be reduced by doing this. This case is handled by somehow determining the sending sequence of messages from the same sender. For example, if the communication paths are all FIFO, we can determine the sending sequence by observing the arrival sequence of the messages.

The four cases handled by Determination Algorithm 2 are illustrated in Figure 2. The straight arrows represent edges while the curved arrows represent paths.

4 Reducing the Space Requirements

4.1 Using Communication Path Information

Determination Algorithm 2 used the communication path information to reduce the amount of clock information required to determine potential causality. This in turn can reduce the amount of clock information that each process needs to maintain while still retaining enough information for Determination Algorithm 2. In this section, we shall develop algorithms for the purpose of reducing the amount of clock information to be maintained.

We shall extend our notion of clocks and timestamps to include *partial* mappings from processes to natural numbers. We shall refer to clocks and timestamps that are total mappings as *full* clocks and *full* timestamps respectively. In the algorithms we develop, each process p maintains a clock C_p, every event e is timestamped with T_e, and every message m is timestamped with T_m. When the C_p's, T_e's, and T_m's are full clocks and timestamps, our algorithms will reduce to Algorithm **FM**. We define $ClockSet_p$ to be the set of processes whose local clocks p needs to keep track of (*i.e.*, $dom(C_p) = ClockSet_p$).

We extend the definition of the clock $sup(c_1, c_2)$ for partial mappings as follows: The domain of $sup(c_1, c_2)$ is the same as that of c_1. For each process p in $dom(c_1) \cap dom(c_2)$, $sup(c_1, c_2)(p) = max(c_1(p), c_2(p))$. For all other process p's, $sup(c_1, c_2)(p) = c_1(p)$. Intuitively, this can be thought of as updating the clock information in c_1 based on information available in c_2. Note that this definition makes *sup* non-commutative.

We now present Propagation Algorithm 2. In this algorithm, the method of choosing the $ClockSet_p$'s is not discussed. We shall return to this shortly.

Propagation Algorithm 2:
Initialization: For all processes p, set $C_p(q)$ to 0 for all processes q in $ClockSet_p$.
Incrementing the local clock: Each process p increments its local clock *before* it performs each event. *i.e.*, $C_p := C_p[p \leftarrow C_p(p) + 1]$. This step is performed only if $p \in dom(C_p)$[2].
When process p sends a message m: The current value of p's clock is attached to m. *i.e.*, $T_m := C_p$.
When a message m arrives at process q from process p: If m contains more recent clock information than q does of any process, q updates its clock with this information. *i.e.*, $C_q := sup(C_q, T_m[p \leftarrow T_m(p) + 1])$[3].

Lemma 4 *If for every p, $ClockSet_p$ is the set of all processes, then Propagation Algorithm 2 is identical to Algorithm* **FM**.

Justification. If for every p, $ClockSet_p$ is the set of all processes then $p \in dom(C_p)$ is always the case so the *Incrementing the local clock* steps of the two algorithms are equivalent. This being the only substantive difference between the two algorithms, means they become equivalent. □

We now present the first of several *clock allocation algorithms*, which are used to determine the *ClockSet*'s. This algorithm is motivated by Determination Algorithm 2. It attempts to reduce the size of the clock domains as much as possible while at the same time not impeding the functionality of Determination Algorithm 2.

Allocation Algorithm 1:

1. Initialize $ClockSet_p$ to the empty set for all processes p.
2. Then for each set of distinct processes p_1, p_2, and p_R such that $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, add p_1 to $ClockSet_p$ for every p that is in a path from p_1 to p_2.

Lemma 5 *For each set of distinct processes p_1, p_2 and p_R, where $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, there is a path from p_1 to p_2 if and only if Allocation Algorithm 1 includes p_1 in $ClockSet_{p_2}$*

Justification. Obvious. □

Based on this observation, we can rephrase Determination Algorithm 2 as follows:

Determination Algorithm 3:

if $p_1 = p_2$ **then**
    order the messages by their sending sequence
elsif $p_1 \in ClockSet_{p_2}$ **and then**
        $T_{m_1}(p_1) < T_{m_2}(p_1)$ **then**
    $m_1 \prec m_2$
elsif $p_2 \in ClockSet_{p_1}$ **and then**
        $T_{m_2}(p_2) < T_{m_1}(p_2)$ **then**
    $m_2 \prec m_1$
else
    $m_1 \parallel m_2$
end if;

The correctness of Determination Algorithm 3 follows directly from the correctness of Determination Algorithm 2 and Lemma 5 and Theorem 6.

Theorem 6 *If Propagation Algorithm 1 and Propagation Algorithm 2 (with Allocation Algorithm 1) were both running simultaneously, then for each process p, $FM_p(q)$ will be equal to $C_p(q)$ after each corresponding update of these clocks for every q in $ClockSet_p$.*

[2] If $p \notin dom(C_p)$ then our algorithms will also ensure that $p \notin dom(C_q)$ for any q.

[3] If $p \notin dom(T_m)$ then $C_q := sup(C_q, T_m)$.

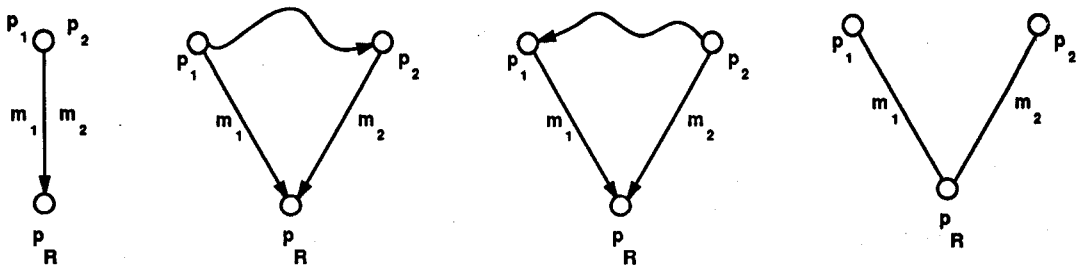

Figure 2: The four cases of Algorithm 2

Justification. The validity of Theorem 6 can be shown informally as follows: $FM_p(q)$ is the most recent information process p has of the local clock of q. The only way in which process p can get information on process q is via some path from q to p. Clock information is piggy-backed on messages along this path starting at q until it reaches p. If Allocation Algorithm 1 included q in $ClockSet_p$, then it will also ensure that all processes in paths from q to p also include q in their *ClockSet*'s. *i.e.*, any path from q to p (where $q \in ClockSet_p$) that Propagation Algorithm 1 can use to transmit clock information is also available for Propagation Algorithm 2 to transmit clock information. □

The paths that transmit clock information are called *update paths*. This is defined more precisely below.

We define an *update sequence for process p_1 at process p_{k+1} to a value c* as a sequence of messages m_1 from p_1 to p_2, m_2 from p_2 to p_3, ..., m_k from p_k to p_{k+1} where the arrival of message m_i causes an increase in $FM_{p_{i+1}}(p_1)$ to c for $i = 1, \ldots, k$. Such a path $(p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_{k+1})$ is called an *update path*.

Lemma 7 *All update paths are acyclic.*

Justification. Obvious since the definition of an update path requires each message m_i's arrival at process p_{i+1} to cause an increase in $FM_{p_{i+1}}(p_1)$ to a particular c. A particular $FM_{p_{i+1}}(p_1)$ can only be increased to c once, thus the paths are acyclic. □

4.2 Using Acyclic Communication Path Information

Lemma 7 allows us to further reduce on the amount of space while still retaining the validity of Theorem 6. Since all update paths are acyclic, we need only consider acyclic paths in our clock allocation algorithms:

Allocation Algorithm 2:

1. Initialize $ClockSet_p$ to the empty set for all processes p.
2. Then for each set of distinct processes p_1, p_2, and p_R such that $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, add p_1 to $ClockSet_p$ for every p that is in a *acyclic* path from p_1 to p_2.

Figure 3 illustrates the results of the two clock allocation algorithms presented so far for a particular communication network.

4.3 Reducing the Message Bandwidth

The next two algorithms — Allocation Algorithm 3 and Propagation Algorithm 3 address optimizating the bandwidth required to transmit messages. Consider why we need to tag clock information to messages:

- When a message m arrives at a process p, it updates its clock using the information in T_m. But we do not require all of T_m. Only that portion of the domain of T_m that is also part of the domain of C_p is of interest for this purpose.
- To determine potential causality information between messages, portions of timestamps are compared with each other.

The previous algorithms tag messages with more clock information than required for the above two operations. The following algorithms will reduce the amount of tagged information. We first define an *observation set* of processes $ObsSet_p$ for each process p. This set contains the processes whose clock information p needs to know about to perform the above operations. The following clock allocation algorithm extends Allocation Algorithm 2 to define the observation sets:

Allocation Algorithm 3:

1. Calculate the *ClockSet*'s as in Allocation Algorithm 2.
2. Initialize $ObsSet_p$ to be the same as $ClockSet_p$ for all processes p.
3. Then for each set of distinct processes p_1, p_2, and p_R such that $p_1 \rightarrow p_R$, $p_2 \rightarrow p_R$ and $p_1 \stackrel{+}{\rightarrow} p_2$, add p_1 to $ObsSet_{p_R}$.

We can now reduce the size of the timestamp a process needs to associate with an outgoing message:

Propagation Algorithm 3:
Initialization: As in Propagation Algorithm 2.
Incrementing the local clock: As in Propagation Algorithm 2.
When process p sends a message m to process q: The current value of p's clock is attached to m, but reduced to only the clock components of interest to q: $T_m := C_p \lceil (ClockSet_p \cap ObsSet_q)$[4].
When a message m arrives at process q from process p: As in Propagation Algorithm 2.

[4] The symbol $\lceil$ restricts the mapping on its left to the domain on its right.

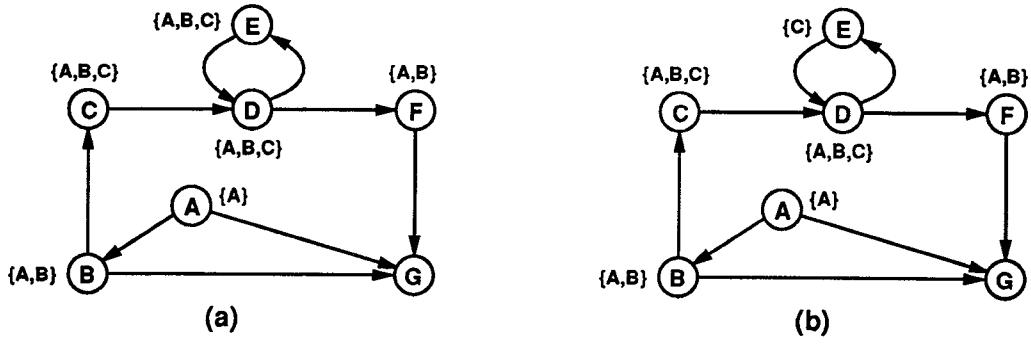

Figure 3: Clock allocations of (a) Allocation Algorithm 1 and (b) Allocation Algorithm 2

4.4 The Gateway Heuristic

Upto now, the algorithms that have been developed attempt to reduce the space requirements for each process and the bandwidth requirements for each communication path. In the worst case, these space requirements will remain the same as for Algorithm FM.

We now present a set of algorithms that modify the earlier algorithms based on *gateways* in the communication graph. Although the space and bandwidth requirements usually decrease by applying these algorithms when these algorithms are applied, there are situations where it can increase. We therefore call the method used in the following algorithms the *gateway heuristic*. We present an algorithm at the end of this section that can be used to determine when this heuristic should be applied to improve the space and bandwidth requirements.

Consider Figure 4(a). The labels on this graph illustrate the *ClockSet*'s calculated by Allocation Algorithm 3. In this figure, E is required to maintain the local clock information of A, B, and C. This is to retain the ability to compare messages from E that arrive at D with messages from A, B, and C that arrive at D. In a communication network such as shown in Figure 4 the potential causality links that we need to test for are the sending of a message from A, B, or C to D, followed by a message from D to E, which in turn is followed by a message from E to D. In this case, then the message from E happens after the message from A, B, or C as the case might be. The idea behind the optimization presented in this section is that if we maintain the arrival times of the messages from A, B, and C at D, we can still determine the potential causality relationships between messages arriving at D. By doing this, we can avoid having to maintain the local clock information of A, B, and C at D and E. The algorithms presented in this section will modify the *ClockSet*'s to what is shown in Figure 4(b).

If E represents a large number of processes, then D is in effect a gateway between such a "subnet" and another part of the network. Since communication between the two sides of the gateway are serialized at the gateway, processes on one side of the gateway do not have to maintain local clock information of processes on the other side of the gateway.

This kind of connectivity can be expected in sparse nets organized hierarchically. In our prototyping language, the block structure and visibility rules promotes these kinds of structures, and hence this is of particular interest to us.

We formally define *gateway* as a process p_G such that there is a loop[5] in the graph that contains p_G, and that there is at least one process p not in this loop such that $p \rightarrow p_G$.

The following clock allocation algorithm extends Allocation Algorithm 3 to consider gateways in the graph:

Allocation Algorithm 4:

1. Initialize $ClockSet_p$ to the empty set for all processes p.
2. Then for each set of distinct processes p_1, p_2, and p_R such that $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, add p_1 to $ClockSet_p$ for every p that is in a acyclic path from p_1 to p_2 *and which does not include* p_R.
3. Now, for each loop L and process $p \notin L$ such that $p \rightarrow p_G$ for some $p_G \in L$ (p_G is a gateway) such that there is at least one process in L whose *ClockSet* does not include p, add p_G to the *ClockSet*'s of every process in L.
4. The *ObsSet*'s are calculated just as in Allocation Algorithm 3, except that the *ClockSet*'s determined by this algorithm is used.

Allocation Algorithm 4 will give us the allocation of Figure 4(b) as opposed to that in Figure 4(a). This algorithm will also improve on the clock allocation shown in Figure 3(b).

As we mentioned earlier, we are able to perform the optimization of Allocation Algorithm 4 at the cost of having to maintain arrival times of messages at each gateway[6]. If message m arrives at a gateway, then Arr_m is its arrival time based on the local clock of the gateway. We modify Propagation Algorithm 3 to maintain the arrival times as shown below:

Propagation Algorithm 4:
Everything is exactly the same as Propagation Algorithm 3 except for the following extra step:

[5] For the purposes of this paper, we assume that loops are closed paths such that no part of this path (other than the complete path itself) is closed

[6] Arrival times of messages can be used to determine the order of messages sent by the same process if communications channels are assumed to be FIFO (Section 3). In this case, maintaining arrival times is not an added cost.

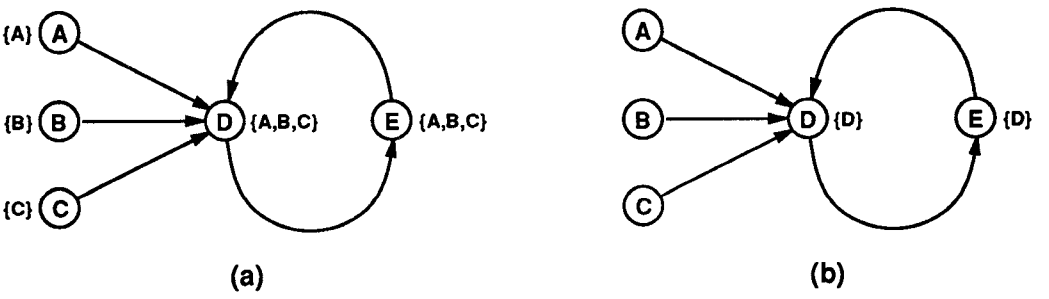

Figure 4: A gateway and subnet

When a message m arrives at process q from process p: If q is a gateway, then the arrival time of m at q is tagged onto m. *i.e.*, $Arr_m := C_q(q)$.

Lemma 5 is not valid with respect to Allocation Algorithm 4. Instead we have:

Lemma 8 *For each set of distinct processes p_1, p_2 and p_R, where $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, there is a path from p_1 to p_2 not passing through p_R if and only if Allocation Algorithm 4 includes p_1 in $ClockSet_{p_2}$*

Justification. Obvious. □

Lemma 9 *If two messages m_1 and m_2 arrive at a gateway p_G from processes p_1 and p_2 respectively, where $p_G \in ClockSet_{m_2}$ and $T_{m_2}(p_G) > Arr_{m_1}$, then $m_1 \prec m_2$.*

Justification. Obvious from the fact that at the time of message m_1's arrival $Arr_{m_1} = C_{p_G}(p_G)$, and the algorithm for increasing $C_{p_G}(p_G)$ and $C_{p_1}(p_G)$. □

Based on the above observations, we now modify Determination Algorithm 3 as follows:

Determination Algorithm 4:

```
if p1 = p2 then
    order the messages by their sending sequence
elsif pR ∈ ClockSet_p2 and then
          Arr_m1 < T_m2(pR) then
    m1 ≺ m2
elsif pR ∈ ClockSet_p1 and then
          Arr_m2 < T_m1(pR) then
    m2 ≺ m1
elsif p1 ∈ ClockSet_p2 and then
          T_m1(p1) < T_m2(p1) then
    m1 ≺ m2
elsif p2 ∈ ClockSet_p1 and then
          T_m2(p2) < T_m1(p2) then
    m2 ≺ m1
else
    m1 || m2
end if;
```

In Determination Algorithm 4, the first two *elsif* clauses handle the case where p_R is a gateway. If the condition being tested ($p_R \in ClockSet_{p_i}$) is true, then p_R is guaranteed to be a gateway and we can assume that all messages arriving at p_R have been tagged with arrival times.

As we mentioned earlier, Allocation Algorithm 4 may increase the space requirements in some cases. The final algorithm in this paper is one of many possible approaches to address this problem. The idea here is to first perform Steps 1 and 2 of Allocation Algorithm 4. We then consider each gateway in the graph one by one and weigh the costs of applying Allocation Algorithm 4 with respect to applying Allocation Algorithm 3. We choose the cheaper option in each case.

Allocation Algorithm 5:

1. Initialize $ClockSet_p$ to the empty set for all processes p.
2. Then for each set of distinct processes p_1, p_2, and p_R such that $p_1 \rightarrow p_R$ and $p_2 \rightarrow p_R$, add p_1 to $ClockSet_p$ for every p that is in a acyclic path from p_1 to p_2 and which does not include p_R.
3. Now visit each gateway process in some order. For each gateway process p_G, compare the costs of applying each of the following steps, and perform the cheaper of the two steps:
   - For each loop L that contains p_G and process $p \notin L$ such that $p \rightarrow p_G$ and such that there is at least one process in L whose *ClockSet* does not include p, add p_G to the *ClockSet*'s of every process in L. *A reasonable cost estimate for this step may be the sum of the increases in the sizes of the ClockSet's in L plus the amount of extra space required to store the arrival time of each message of interest at p_G (see footnote 6.*
   - For each pair of distinct processes p_1, p_2 such that $p_1 \rightarrow p_G$ and $p_2 \rightarrow p_G$, add p_1 to $ClockSet_p$ for every p that is in a acyclic path from p_1 to p_2 *which includes p_G.*

     A reasonable cost estimate for this step may be the sum of the increases in the sizes of the ClockSet's.
4. The *ObsSet*'s are calculated just as in Allocation Algorithm 3, except that the *ClockSet*'s determined by this algorithm is used.

Propagation Algorithm 4 and Determination Algorithm 4 need to be modified to take into account that all gateways have not been handled in the same manner by Allocation Algorithm 5. In Propagation Algorithm 4, arrival times of messages need only be maintained at gateways handled by the first option of step 3 of Allocation Algorithm 5. Also,

the tests ($p_R \in ClockSet_{p_i}$) in Determination Algorithm 4 should be replaced by a test to check that p_i is a gateway and has been handled by the first option of step 3 of Allocation Algorithm 5.

Further Lemma 8 is not valid with respect to Allocation Algorithm 5 but could be reexpressed by taking into account the cost metric used in Allocation Algorithm 5.

5 Some Example Topologies

We shall now apply our algorithms to some common connection networks. First, we analyze a star network where there is a central process p and n satellite processes $p_1, \ldots, p_n$. The satellite processes can communicate in both directions with the central process. Using Allocation Algorithm 5, we get the central process p to be a gateway, and the counter and observation sets for all processes is $\{p\}$. In this situation, our algorithm reduces both space and bandwidth by a factor of $(n+1)$.

The second network is what we call the *hierarchical triangle*. Our prototyping language has exactly this kind of connection network. Figure 5 illustrates this network. Each triangle depicts a declarative region, the process at the apex corresponding to the declarative region itself and the processes on the base corresponding to the entities (which may be nested declarative regions) declared within this declarative region. Within a triangle, all processes can communicate in both directions with all other processes. Using Allocation Algorithm 5, the counter set and observation set of a process p are the set of all processes in the triangles it is part of. Each process can be part of at most two triangles. For example, in Figure 5, the counter set and observation set of p_{01} includes p_0, $p_{01}, \ldots, p_{0m}$ and all processes in the triangle below p_{01}. Every inner process (processes other than the root and the leaves) is a gateway between the two triangles it is part of and therefore maintains arrival times of all messages coming to it. The space and bandwidth requirements for our algorithms can be as little as $O(n)$ (where n is the number of processes), a factor of n savings over Algorithm **FM**.

A third kind of network is a ring network. In this case, our algorithm does not save either space or bandwidth as compared to Algorithm **FM**. Each process has to maintain a counter set that includes all processes in the network.

6 Related Work

Other work has been done to improve the efficiency of keeping track of potential causality. [3] and [5] have shown that in the case of a fixed number of processes, the FM mapping may be replaced by an array of counters where each process has a fixed index in the array.

[6] have given an algorithm whereby each process keeps track of the clock information it has sent to other processes by maintaining two additional vectors. A message from p to q then need only carry clock information which has changed at p since the last message from p to q. This can result in substantial savings in bandwidth and does not require a static communications architecture. However, it comes at the cost of losing the ability to determine the causal relationship between two messages arriving at a particular receiver based on the clock information the messages carry and the state of the receiver upon their arrival.

7 Conclusions

The algorithms presented in this article has all the nice attributes of Algorithm **FM**, namely they require no extra synchronization messages, no additional communication links and no central timestamping authority. By restricting the problem domain — the communication network is static and known ahead of time and that we are only interested in comparing messages that arrive at the same node, we are however able to save on the amount of space required for certain kinds of connectivity networks. Consider the hierarchical triangle network. If each triangle has no more than m processes, then each process needs to maintain at most $2m$ local clock values, and each message needs to be tagged with at most m amount of local clock information. This is a considerable decrease in the space and also the bandwidth requirements as compared to Algorithm **FM**.

During our work on prototyping languages, we have developed small examples in which there may be as many as 100 processes, each communicating with around 10 others in a hierarchical triangle network. Our algorithms can therefore save us space of around 80 units per process and a bandwidth of around 90 units per message even for these small examples. On larger sized examples, the amount of savings will be even more considerable.

Although our problem domain requires the comparison of only messages that arrive at the same process p_R, our algorithms allow us to compare any two events generated by p_1 and p_2 (the processes sending the messages) for potential causality relationships. Note that it is not important that there is a process p_R to which both p_1 and p_2 are connected. We could apply all our algorithms equally well by considering any two arbitrary processes p_1 and p_2 whose events we wish to compare.

Our algorithms will be useful in many distributed system applications where the communication network is static and reasonably sparse. Our algorithms begin to approximate Algorithm **FM** more closely as the communication network becomes more dense.

Possible uses for our algorithms are in the scheduling of resources and jobs in a distributed operating system, cache coherency algorithms, and in distributed databases. Furthermore, our algorithms may also be used to determine relationships other than temporal orderings. For example, tools to reason about any kind of partial order can make use of our algorithms.

8 Acknowledgements

We are especially grateful to Doug Bryan for his help in developing the initial versions of the algorithm. This work was motivated by the need for fast partial order determination algorithms in the implementation for our prototyping language at Stanford.

We are thankful to Jennifer Anderson and Alex Hsieh for many detailed comments on earlier drafts of this article.

This research was supported by the Defense Advanced Research Projects Agency/Information Systems Technology Office under the Office of Naval Research contract N00014-90-J1232, and by the Air Force Office of Scientific Research under Grant AFOSR83-0255. Sigurd Meldal is supported by the Norwegian Research Council for Science and the Humanities. James Vera is supported by a Graduate Fellowship from Bell Communications Research (Bellcore).

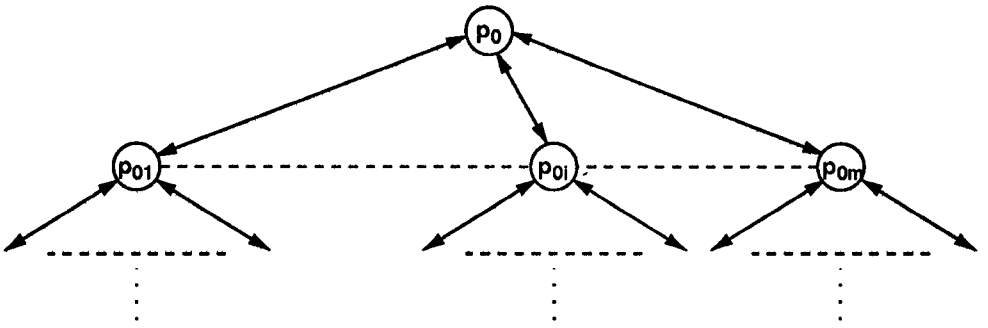

Figure 5: A hierarchical triangle network

References

[1] Frank Belz and David C. Luckham. A new approach to prototyping Ada–based hardware/software systems. In *Proceedings of the ACM Tri-Ada Conference*, Baltimore, December 1990.

[2] Bernadette Charron-Bost. *Concerning the Size of Clocks*, volume 469 of *Lecture Notes in Computer Science*, pages 176–184. Springer-Verlag, 1990.

[3] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10(1):55–66, February 1988.

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

[5] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proceedings of Parallel and Distributed Algorithms*. Elsevier Science Publishers, 1988. Also in: Report No. SFB124P38/88, Dept. of Computer Science, University of Kaiserslautern.

[6] M. Singhal and A. Kshemkalyani. An efficient implmentation of vector clocks. Technical report, Department of Computer and Information Science, The Ohio State University, October 1990.