

Dependency Sequences and Hierarchical Clocks: Efficient Alternatives to Vector Clocks for Mobile Computing Systems

RAVI PRAKASH¹ AND MUKESH SINGHAL²

¹*Department of Computer Science, University of Rochester
715 Computer Studies Building, Rochester, New York 14627-0226.
E-mail: prakash@cs.rochester.edu*

²*Department of Computer and Information Science, The Ohio State University
2015 Neil Avenue, Columbus, Ohio 43210.
E-mail: singhal@cis.ohio-state.edu*

Vector clocks have been used to capture causal dependencies between processes in distributed computing systems. Vector clocks are not suitable for mobile computing systems due to (i) lack of scalability: its size is equal to the number of nodes and (ii) its inability to cope with fluctuations in the number of nodes. This paper presents two efficient alternatives to vector clock, namely, sets of dependency sequences, and hierarchical clock. Both the alternatives are scalable and are immune to fluctuations in the number of nodes in the system.

Keywords: causal dependency, mobile computing, vector clock, dependency sequence, hierarchical clock.

1 Introduction

A mobile computing system divides the geographical region served by it into smaller regions called cells. Each cell is served by a static node, referred to as the mobile service station (*MSS*). All the *MSS*s in the system are connected to each other by a fixed wireline network. A mobile host (*MH*) can communicate with other units, mobile or static, only through the mobile service station of the cell in which it is present. From the mobile service station the signals can be forwarded along the fixed wireline network, or along another wireless channel, depending on whether the other party involved in the communication session is a unit outside the cell or a mobile host in the same cell, respectively.

The data rate achievable on wireless channels is significantly lower than the achievable data rate on the fixed wireline network [5]. Communication bandwidth available to mobile hosts is significantly less than the bandwidth available to nodes in traditional distributed systems that use wireline networks only. Sev-

eral communication protocols and application software that are suitable for such distributed systems are likely to have unacceptable performance in a mobile computing system due to the low data rate. Hence, there is a need to develop efficient alternatives [2].

Several mobile computing applications will be distributed in nature, consisting of multiple processes executing concurrently on different nodes. In order to track concurrency, and the resultant nondeterminism, in the application, it is important to determine the causal and temporal relationship between events that occur in the distributed computation [11]. Concurrency and nondeterminism tracking are important for analyzing, monitoring, debugging, and visualizing the behavior of a distributed system [11].

Applications running on distributed systems have employed vector clocks [4, 8] to track causal dependency relations between events on different processes. The vector clock is an integer vector with as many components as the number of processes in the application. It has been proved that a vector of this length is necessary to track causal dependency relationships [3]. For large distributed applications with many constituent processes the size of the vector clock is large. A message has to carry a *vector timestamp*: the value of the sending process' vector clock at the time the message is sent. The communication overheads imposed by vector timestamps are acceptable for high data rate fixed wireline networks. However, in a mobile computing system, sending vector timestamps with each message is not a feasible solution due to the low data rate of wireless channels. This is especially so for large distributed applications with several constituent processes.

The communication overheads of vector clocks can be moved away from the wireless channels, and onto the wireline network if the *MSSs* collectively take the responsibility for maintaining causal dependency information on behalf of all the *MHs*. In such a scenario, the *MSSs* are said to act as *proxies* for the *MHs*. Still, the size of each vector is likely to become prohibitively large to send it even over the higher bandwidth wireline network. Besides, the mobile host population fluctuates from time to time. Mobile hosts join the network, stay connected for a period of time, and then disconnect, either temporarily or permanently. Such fluctuations cannot be handled by vector clock implementations that employ static arrays of integers. This is because such implementations require that the number of nodes in the system does not change.

An alternative implementation of vector clocks could be to have sets of (*name, time*) pairs for each mobile host in the system, where the *time* component is an integer. When a mobile host joins the distributed computation, a (*name, time*) pair for it can be added to the set. However, if a mobile host disconnects from the network, should the (*name, time*) pair associated with it be maintained, or should the pair be discarded? If it is to be maintained, how long should it be maintained?

This is a difficult question to answer because if a mobile host disconnects, an independent observer cannot be sure of the duration of disconnection. Perhaps, the mobile host will rejoin the computation in a short while. Perhaps, the mobile host will not rejoin the computation before the application terminates. In such a situation, a number of *garbage* (name, time) pairs may have to be unnecessarily carried along with messages, long after they are no longer needed. Hence, there is a need for a representation of causal dependencies that can efficiently handle the addition as well as deletion of mobile hosts from a distributed application.

As all communication involving the mobile hosts takes place through the *MSS* of the cell in which they are present, it appears that causality relationships can be tracked by using vector clocks of n components, where n is the number of *MSSs* in the system. This would lead to significant reduction in communication overheads as the number of *MSSs* is usually a small fraction of the total number of nodes (*MHs* and *MSSs*) in the system.¹ However, using just n -integer vector clocks is insufficient for tracking all the causal dependencies, as will be shown in Section 3.

Therefore, there is a need for alternative means of tracking causal dependencies that impose as low an overhead as possible and are immune to fluctuations in the number of nodes in the system. This paper presents two dependency tracking schemes that meet these requirements. The fact that all communication involving *MHs* goes through the *MSSs* is exploited.

This paper is organized as follows. In Section 2 the system model is presented. Section 3 describes why more than n integers are needed to track all the causal dependencies. In Section 4, alternative representations for capturing all the dependencies in a mobile computing system are discussed. Section 5 presents *Dependency Sequences*, and Section 6 presents *Hierarchical Clocks*: two efficient alternatives to vector clocks. The two alternatives are compared in Section 7. Section 8 contains a brief description of causal dependency tracking of internal events. Finally, conclusions are presented in Section 9.

2 System Model

A mobile host can either be a cellular telephone, a laptop computer with a wireless modem, a personal data assistant, or any other mobile computing device capable of communicating over wireless channels.

¹The trend towards smaller cells (microcellular, picocellular) implies that the number of *MSSs* is on the rise. The main motivation for such a trend is the need to accommodate a rapidly growing *MH* population that requires wireless channels from a limited frequency spectrum for communication. Hence, despite the increase in the number of *MSSs*, their number will continue to be much smaller than the number of *MHs* in the system.

The links in the static wireline network support FIFO message communication. As long as an *MH* is connected to an *MSS*, the wireless channel between them also ensures FIFO communication in both directions. Message transmission through the wireline and wireless links takes an unpredictable, but finite amount of time. No messages are lost or modified in transit.

An application executing in a mobile computing system is assumed to be composed of N processes. These processes collectively execute a distributed computation. It is assumed that each process executes on a distinct node, either mobile or static, of the system. There is no globally shared memory. All communication between processes is through messages. There exists a logical communication channel between each pair of nodes. Process execution and message communication are asynchronous. The events in a process are executed in a sequential manner. These events can be classified into three categories: message *send* events, message *receive* events, and internal events. The *send* and *receive* events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.

The system does not have a global physical clock. Hence, the order in which two events occur at two different processes cannot be always determined solely on the basis of the time of occurrence. However, information about the partial order of occurrence of events belonging to the same application can be gathered based on the causal dependencies between them. Such dependencies can be expressed using the *happened before* relation (\rightarrow) between events, defined in [7] as:

- $a \rightarrow b$, if a and b are events in the same process and a occurred before b .
- $a \rightarrow b$, if a is the event of sending a message M in a process and b is the event of delivery of the same message to another process.
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$ (i.e., “ \rightarrow ” relation is transitive).

Let $Past(a)$ denote all events in the causal past of event a . That is,

$$Past(a) = \{b \mid b \rightarrow a\}.$$

$Past(a)$, along with a , constitutes an envelope of causal predecessor events across all processes in the distributed computation corresponding to the application program. This envelope is referred to as the *past cone of a* . If $a \not\rightarrow b$ and $b \not\rightarrow a$, then a and b are said to be concurrent and are represented as $a \parallel b$. In such a situation, a is not present in the past cone of b , and vice-versa.

Lamport’s *happened before* relation is employed to maintain a scalar clock [7] representing the partial order between events. Let $C(a)$ and $C(b)$ be the scalar clock values associated with two events a and b . Then, $a \rightarrow b \Rightarrow C(a) < C(b)$. However, the converse is not true.

Vector clocks, proposed independently by Fidge [4] and Mattern [8], guarantee that if $V(a)$ and $V(b)$ are the *vector timestamps* (values of the vector clocks when events occur) associated with events a and b , respectively, then $a \rightarrow b \Leftrightarrow V(a) < V(b)$. Thus, an isomorphism exists between the set of partially ordered events in a distributed computation and their vector timestamps [8, 9, 10]. If $V(a) < V(b)$, then event a is in the past cone of event b .

It is to be noted that in a mobile computing environment, or for that matter in any distributed environment, multiple independent distributed applications may be executing concurrently. However, causal dependency tracking is required only between events belonging to the same distributed application. Hence, for each of the independent distributed applications, there will be a distinct vector clock. In the subsequent sections, where we describe alternatives to vector clock for mobile computing applications, each application will have a distinct dependency tracking scheme and not share a common tracking scheme. While describing the alternatives to vector clocks, we will assume that only one application is being executed.

3 Insufficiency of n-component Vector Clocks

In mobile computing systems, all communications of the MH s take place through the MSS of the cell in which they are present. Even though two MH s in a cell may be communicating concurrently with entirely different sets of nodes, the MSS of the cell imposes a sequential relationship among all messages destined for or sent by MH s in its cell, based on the order in which they arrive at the MSS .

Causal dependencies between nodes are created solely by messages. So, an MSS is aware of all intra-cell causal dependencies between MH s in its cell, as well as inter-cell causal dependencies involving its MH s. This leads to an obvious question: Can all causal dependencies between nodes in the system (MH s and MSS s) be tracked using vector clocks of size n , where n is the number of MSS s in the system?

Lemma 1

Vector clocks with as many components as the number of MSS s in the system cannot accurately capture all the causal dependencies in a mobile computing system.

Proof: The proof is by a counter example shown in Figure 1.

Consider a two cell system, served by mobile service stations MSS_p and MSS_q . The mobile hosts MH_a and MH_b are present in the cell served by MSS_p . Mobile hosts MH_c and MH_d are served by MSS_q . As there are two MSS s in

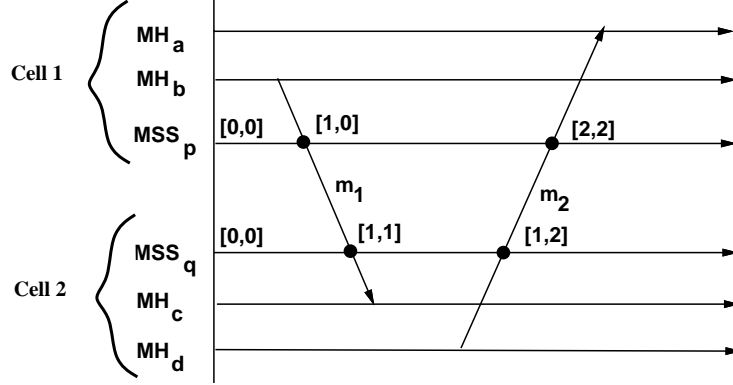


Figure 1: A vector clock with more than 2 components is needed for the 2-cell system.

the system, the vector clock has two components. Initially, the vector clocks at both the MSS s are set to $[0,0]$. MH_b has to send a message m_1 to MH_c . So, MH_b sends the message to MSS_p . MSS_p associates a vector timestamp of $[1,0]$ with the event $Send(m_1)$ (as it is the first message sent or received by MSS_p), and forwards the message to MSS_q . When m_1 arrives at MSS_q , MSS_q sets its vector clock to $[1,1]$, associates this timestamp with the event $Recv(m_1)$, (i.e., $V(Recv(m_1)) = [1,1]$) and forwards m_1 to MH_c .

MH_d has to send message m_2 to an arbitrary node in the network, say MH_a . So, MH_d sends m_2 to MSS_q . MSS_q receives m_2 after receiving m_1 . Hence, MSS_q updates its vector clock to $[1,2]$ and associates this timestamp with the event $Send(m_2)$, i.e., $V(Send(m_2)) = [1,2]$.

As there is no causal relationship between $Recv(m_1)$ and $Send(m_2)$, the two events are concurrent. However, $V(Recv(m_1)) < V(Send(m_2))$ implying that $Send(m_2)$ is causally dependent on $Recv(m_1)$: a break-down of the isomorphism property of vector clocks. ■

When one integer is used per cell to represent causality, the MSS of a cell increments its component of the vector clock for every *send* and *receive* event it observes even if these events are mutually concurrent. So, at the cell level, there is an implicit serialization of events. Even if no causal dependency exists between two events occurring at two different MH s in the same cell, the corresponding MSS is unable to maintain mutual concurrency information about those events.

A Geometric Interpretation

The contradiction shown above can be explained with the aid of Figure 2. Events

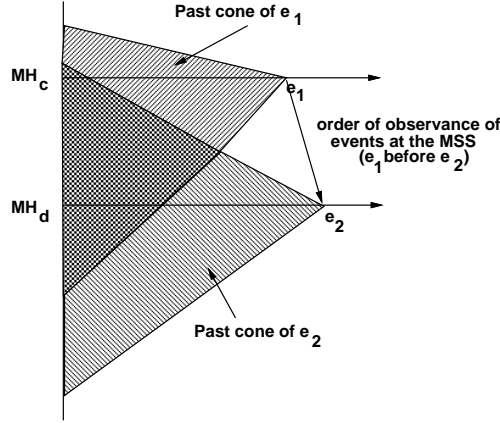


Figure 2: Past cones of two events occurring in cell 2.

e_1 and e_2 occur at mobile hosts MH_c and MH_d (present in the same cell), respectively. If the vector clock has only one component per cell, the MSS of the cell can erroneously interpret the past cone of an event, occurring at one MH , to be a union of the past cones of all events that have occurred at all the MH s present in the cell and have been observed by the MSS thus far. For example, event e_1 , in Figure 2, corresponds to $Recv(m_1)$ in the example described above, and e_2 corresponds to $Send(m_2)$. As e_2 is observed by MSS_q after e_1 , MSS_q erroneously interprets e_1 to be in the past cone of e_2 . There is no way to determine if two or more events occurred concurrently in a cell.

4 Alternative Dependency Representations

As shown in Section 3, using a vector clock of as many components as the number of MSS s, n , is insufficient to capture the causal dependencies between all the MH s. Yet, the presence of the MSS in each cell to sequentialize the message sends and receives for MH s in the cell indicates that a vector clock of size N (total number of nodes) is not necessary to capture the causal dependencies. If MSS s act as proxies for the MH s in their cells and augment the n -integer vector to capture some dependency information about their MH s, all causal dependencies in the system can be accurately represented.

The difference between the union of past cones of latest events of all the MH s in a cell, and the past cone of latest event e of a particular MH in the cell indicates the set of events, S_e , that have occurred in the cell and are not in the causal past of e . For example, in Figure 2, when using a 2-component vector clock, MSS_q erroneously interprets the past cone of e_2 to be the union of the past cones of e_1 and e_2 , even though e_1 and e_2 are mutually concurrent events. This is because

MSS_q observes event e_2 after e_1 . The actual past cone of e_2 is this unified cone minus the portion of e_1 's past cone that is not in e_2 's past cone. This portion of e_1 's past cone corresponds to the set S_e for event e_2 .

If all the dependency creating events (sends and receives) in a cell were to be sequenced in the order observed by the MSS , then S_e corresponds to those events in the sequence that are not causal predecessors of event e . Such events (that should be excluded from the sequence) are henceforth referred to as *gaps in the dependency sequence* of the cell with respect to event e . Thus, event e_1 belongs to the gap in the dependency sequence of the cell with respect to event e_2 . A means to represent the dependency gaps in all the cells, with respect to a particular event e , can capture all the events across the entire system that are in the causal past of e , and those that are not.

Two different approaches to representing the causal predecessors of an event e are as follows:

1. Sufficient dependency information is sent with each message so that each MH 's set of causal predecessors can be constructed quickly, based entirely on information available at the MSS of the cell in which it is present.
2. A small amount of dependency information is sent with each message. When the set of causal predecessors of an event is to be constructed, the dependency information available locally is used as a set of pointers to information that is used for the construction of the causal predecessor set.

Both these approaches have their tradeoffs. The first approach imposes higher communication overheads. However, time to construct the set of causal predecessors of an event is small. The second approach has low communication overheads. However, it takes longer to construct the set of causal predecessors of an event. The choice between the two approaches should be influenced by the frequency with which sets of causal predecessors need to be constructed and available communication bandwidth.

In the next two sections, dependency tracking schemes based on the two approaches are described.

5 Dependency Sequence Approach

The MSS of a cell acts as the proxy for all MH s in its cell with regard to maintaining dependency information. All dependency causing events (message sends and receives) in a cell, observed by the corresponding MSS , are assigned sequence numbers in a monotonically increasing fashion. Initially, the sequence counter is set to 0. If e_i and e_j are two successive dependency causing events

observed by an *MSS*, then the sequence number assigned to e_j is one greater than the sequence number assigned to e_i .

The causal predecessors of an event e are represented as a set of dependency sequences. Each dependency sequence in the set corresponds to a cell in the system, and consists of a sequence of non-negative integers. Pairs of these integers represent contiguous sequences of dependency causing events in a cell that are causal predecessors of e . As internal events of mobile hosts do not create causal dependencies, they are not considered. The cardinality of the set is equal to the number of cells in the system. Initially, each sequence in the set is empty. There is no upper bound on the length of a sequence. However, the length of each sequence is always even, and the integers in the sequence are arranged in non-descending order.

The first integer in the dependency sequence of e , with respect to a cell, denotes the first dependency causing event observed by that cell's *MSS* that is in the past cone of e . All the dependency causing events of the cell whose sequence numbers are greater than or equal to the first integer and less than or equal to the second integer in the sequence are in the past cone of e . All dependency causing events in the cell whose sequence numbers are greater than the second integer and less than the third integer are not in the past cone of e and constitute a *dependency gap*. Similarly, all dependency causing events with sequence numbers between the third and fourth integers in the sequence are in the past cone of e , and so on.

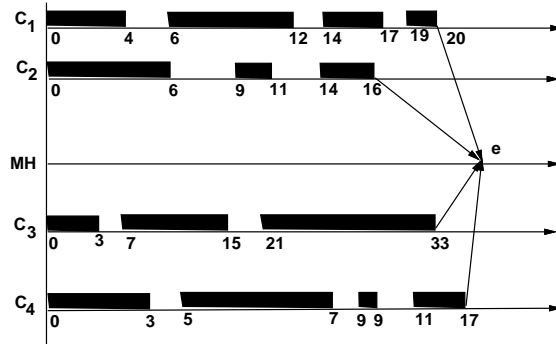


Figure 3: Dependency sequences for event e .

The significance of dependency sequences is illustrated with the help of Figure 3. Message send and receive events 0 – 4, 6 – 12, 14 – 17, and 19 – 20 in cell C_1 , represented by rectangular blocks, are causal predecessors of event e occurring at the mobile host shown in the figure. Events 5, 13, and 18 in cell C_1 are not causal predecessors of e , and correspond to gaps in the sequence. Similarly, the rectangular blocks on time-lines for cells C_2 , C_3 , and C_4 also represent the causal predecessors of e . Thus, the dependency sequences of e , for the four-cell system shown in Figure 3, are $\{0, 4, 6, 12, 14, 17, 19, 20\}$, $\{0, 6, 9, 11, 14, 16\}$, $\{0, 3,$

7, 15, 21, 33}, {0, 3, 5, 7, 9, 9, 11, 17}]. It is to be noted that the number of sequences in the set is equal to the number of cells in the system regardless of the number of MH s in the system. Thus, MH s may join and leave the system without having an impact on the number of sequences. Hence, the dependency sequence approach is immune to fluctuations in MH population.

5.1 Management of Dependency Sequences

The mobile service station of cell p , denoted by MSS_p , maintains the dependency sequences of all the MH s in the cell. The set of dependency sequences of MH_a is denoted by DS_a . DS_a has n components, where n is the number of cells in the system: one component for each cell. The p^{th} component, corresponding to cell p , is denoted by $DS_a[p]$. Each MSS also maintains an integer *counter* to keep track of the message send and receive events observed by it since the beginning of the computation. $DS_a[p]$ is initialized to an empty sequence, for $0 \leq p \leq n - 1$, and *counter* is initialized to 0.

The following functions are defined for sequences of integers:

last($DS_a[p]$): returns the last integer in the sequence $DS_a[p]$.

concatenate($seq1$, $seq2$): appends $seq2$ to the end of $seq1$, where both $seq1$ and $seq2$ are sequences of integers, and returns the resultant sequence.

When MH_a , resident in cell p has to send a message to MH_b , the message is first sent over a wireless channel to MSS_p .

MSS_p 's action on receiving message from MH_a :

```
{ counter ← counter + 1
  if (last( $DS_a[p]$ ) = counter - 1)
    last( $DS_a[p]$ ) ← counter;
  else
     $DS_a[p]$  ← concatenate( $DS_a[p]$ , {counter, counter});
  send message and  $DS_a$  over the wireline network to  $MH_b$ 's service station, namely  $MSS_q$ ;
}
```

MSS_q 's action on receiving message and DS_a sent by MH_a for MH_b :

```
{ counter ← counter + 1;
  if (last( $DS_b[q]$ ) = counter - 1)
    last( $DS_b[q]$ ) ← counter;
  else
     $DS_b[q]$  ← concatenate( $DS_b[q]$ , {counter, counter});
  for all  $0 \leq i < n$ 
     $DS_b[i]$  ← merge( $DS_b[i]$ ,  $DS_a[i]$ );
}
```

The *merge* function works as follows: Let there be two dependency sequences $seq1 = \{S_1, S_2, \dots, S_{2i-1}, S_{2i}\}$ and $seq2 = \{R_1, R_2, \dots, R_{2j-1}, R_{2j}\}$ corresponding to the same cell. These sequences represent two different subsets of the dependency causing events of the cell observed by its *MSS*. The function $merge(seq1, seq2)$ returns a subset of the dependency causing events of the cell that correspond to $seq1 \cup seq2$. A detailed description of *merge* is presented in the Appendix.

Let $seq1 = [\{0,3,9,12,17,17\}, \{0,2,5,6,35,54\}]$, and $seq2 = [\{0,5,11,14,21,23\}, \{0,1,4,5,43,49\}]$ for a 2-cell mobile computing system. Let $seq1$ represent the dependency sequences of a mobile host MH_a at the time it sends a message m to another mobile host MH_b . Let $seq2$ represent the dependency sequences of MH_b when it receives m , and before it invokes *merge*. Then, $merge(seq1, seq2)$ returns the dependency sequences of MH_b after it has received m , and is equal to $[\{0,5,9,14,17,17,21,23\}, \{0,2,4,6,35,54\}]$. The first set in $seq1$ indicates that dependency causing events $e_0 - e_3$, $e_9 - e_{12}$ and e_{17} of cell C_1 are in the past cone of MH_a 's *message send*. Similarly, the first set in $seq2$ indicates that dependency causing events $e_0 - e_5$, $e_{11} - e_{14}$, and $e_{21} - e_{23}$ of cell C_1 are in the past cone of MH_b when m arrives. Therefore, on the delivery of m , MH_b 's past cone contains the following dependency causing events of C_1 : $e_0 - e_5$, $e_9 - e_{14}$, e_{17} , and $e_{21} - e_{23}$, and the following dependency causing events of C_2 : $e_0 - e_2$, $e_4 - e_6$, and $e_{35} - e_{54}$.

Correctness Proof

Lemma 2

Dependency sequences of a mobile host account for all the events in the mobile computing system that are causal predecessors of the current state of the mobile host.

Proof: The proof is by induction.

Base Case: Initially, no dependency causing event (message send or receive) has taken place at any of the nodes in the system. Hence, there are no causal predecessors of the initial state of a mobile host, and the dependency sequences are empty.

Induction Step: Let the set of dependency sequences of a mobile host MH_a , prior to a message send or reception, account for all the causal predecessors of MH_a 's state at that time. It is to be proved that the dependency sequences of MH_a continue to account for all causal predecessors of its state after the message send/receive. There are two possible cases:

1. *Message send:* The set of causal predecessors of MH_a 's current state, as defined earlier, contains the current state of MH_a . On message send, *counter* is incremented at its mobile service station MSS_p . At this point of time *counter* reflects the number of message sends and receives observed by MSS_p

since the beginning of the computation. The message send event, identified by the current value of *counter* is added to $DS_a[p]$. Thus, the updated set of causal predecessors of MH_a 's current state is the set of causal predecessors of its state prior to sending the message plus the message send event.

2. *Message receive*: Let MSS_p receive a message meant for MH_a that was sent by MH_b through its mobile service station MSS_q . Prior to message reception, DS_a accounts for all the causal predecessors of MH_a 's current state. At the time the message was sent by MSS_q , DS_b reflected all the causal predecessors of MH_b 's state, including the message send event. On the reception of the message, MSS_p updates DS_a as follows:

- the updated value of *counter* is added to $DS_a[p]$ to account for the message receive event,
- for every cell C_i in the system, the dependency sequence $DS_a[i]$ and $DS_b[i]$ (obtained with the received message) are merged and stored as the new $DS_a[i]$. The *merge* operation takes a union of the sets of events in cell C_i that are causal predecessors of MH_a 's state prior to receiving the message, or MH_b 's state at the time of sending the message. Thus, on message reception, the causal predecessors of MH_b 's state also become causal predecessors of MH_a 's current state. ■

5.2 Determination of Causal Relation

The set of dependency sequences associated with a mobile host at the time of the occurrence of an event e denotes the causal history [11] of e . The causal history contains all the events that causally precede e .

Let there be two sequences, $seq1 = \{S_1, S_2, \dots, S_{2i-1}, S_{2i}\}$ and $seq2 = \{R_1, R_2, \dots, R_{2j-1}, R_{2j}\}$. We define the *subset* function on two dependency sequences as follows:

Definition 1

$$seq1 \subseteq seq2 \text{ iff } \forall k \in \mathcal{N} : 2k \leq 2i, \exists l : 1 \leq l \leq j : S_{2k-1} \geq R_{2l-1} \wedge S_{2k} \leq R_{2l}$$

The definition states that $seq1$ is a subset of $seq2$ if and only if all events in $seq1$ are contained in $seq2$.

Let there be two dependency causing events e_1 and e_2 , that occur at two different mobile hosts MH_a and MH_b , respectively. At the time of occurrence of these events, let MH_a and MH_b reside in cells serviced by mobile service stations MSS_p and MSS_q , respectively. Let the dependency sequences associated with e_1 and e_2 be DS_a and DS_b , respectively.

Lemma 3

The following can be said about the causal relationship between events e_1 and e_2 :

- $e_1 \rightarrow e_2$ iff $\forall i : 1 \leq i \leq n, DS_a[i] \subseteq DS_b[i] \wedge \exists j : DS_a[j] \neq DS_b[j]$
- $e_1 \parallel e_2$ iff $e_1 \not\rightarrow e_2 \wedge e_2 \not\rightarrow e_1$

where $n =$ number of cells in the system.

Proof: This follows directly from the definition of causal history [11]. ■

Comparing two sets of dependency sequences to determine the causal relationship between a pair of events can be an expensive operation. However, causal relationship between two events can be determined in an inexpensive manner if information is available about the cell(s) in which the events occurred. This information can be easily maintained and is readily available. As we know that e_1 and e_2 occurred while the corresponding mobile hosts were in cells serviced by MSS_p and MSS_q , respectively, the causal relationship can be determined as follows:

1. $e_1 \rightarrow e_2$ iff $\{last(DS_a[p]), last(DS_a[p])\} \subseteq DS_b[p]$
2. $e_1 \parallel e_2$ iff $\{last(DS_a[p]), last(DS_a[p])\} \not\subseteq DS_b[p] \wedge \{last(DS_b[q]), last(DS_b[q])\} \not\subseteq DS_a[q]$

The conditions above state that for event e_1 to causally precede event e_2 , e_1 (whose occurrence is denoted by $last(DS_a[p])$ in the dependency sequence) should be in the causal history of event e_2 . If there is no dependency chain from e_1 to e_2 , $DS_b[p]$ will not contain $last(DS_a[p])$.

Therefore, determination of causal relationship between two events is reduced to searching for the position of two integers, namely, $last(DS_a[p])$ and $last(DS_b[q])$ in two dependency sequences, namely, $DS_b[p]$ and $DS_a[q]$, respectively. Let \mathcal{E} be the length (number of integers) of the longest dependency sequence. As both the dependency sequences are non-decreasing sequences of integers, the search can be performed using binary search or any other efficient searching method in time $O(\log \mathcal{E})$.

5.3 Managing Handoffs

When a mobile host MH_a moves out of cell p to a cell q , the responsibility of maintaining its causal dependencies shifts from MSS_p to MSS_q . This is handled as follows. During the handoff, MH_a registers with MSS_q and deregisters from MSS_p . As a part of the registration/deregistration process, the dependency sequences for MH_a , namely, DS_a , are transferred over the wireline network from MSS_p to MSS_q . As a result, now MSS_q has complete information about all the causal predecessors of MH_a 's current state. From this point on, if MH_a needs to

send a message, it communicates the message to MSS_q which forwards it towards the destination. Also, if some node in the system wishes to send a message to MH_a , the message is routed to MSS_q . On receiving the message, MSS_q uses the accompanying dependency sequences to update DS_a and forwards the message to MH_a over a wireless channel.

Thus, handoffs are straightforward to handle. The dependency sequences are transferred over the relatively high bandwidth wireline network. The low data rate wireless channels are not burdened by handoff traffic. Fluctuations in the number of mobile hosts in the system has no impact on the maintenance of dependency sequences as the number of such sequences depends solely on the number of MSS s, which can be safely assumed to remain fixed.

5.4 Storage Overheads

The storage overheads per mobile host is $O(n \times \mathcal{E})$, where n is the number of cells in the system, and \mathcal{E} is the length of the longest dependency sequence for an MH . Typically, n is a small fraction of the total number of nodes in the system (N). For each MH , there are n dependency sequences of integers. If there is heavy message traffic between the MH s: (i) there are long contiguous sequences of events in all cells that are causal predecessors of an MH 's current state and (ii) there are few gaps in the dependency sequence. So, the total number of integers in the sequence is small, *i.e.*, \mathcal{E} is small. Also, if most of the communication for an application is confined to just a few cells, the dependency sequences corresponding to other cells will be empty. Thus, the storage overheads are low.

It is to be noted that if an MSS handles several independent applications concurrently, it will maintain a distinct dependency sequence for each application. So, gaps will not be introduced in the dependency sequences due to an interleaving of independent dependency causing event streams.

The length of the dependency sequences can be bounded by periodic global checkpointing. Whenever the length of dependency sequences grows beyond a certain *threshold*, a global checkpoint of the system is taken. As the global checkpoint records all the causal dependencies generated so far, the dependency sequences can be reset to empty sequences and computation can proceed further. The threshold of dependency sequence length is a parameter of the algorithm, and can be chosen such that $O(n \times \mathcal{E})$ is asymptotically less than $O(N)$, the storage and communication overhead for vector clocks. So, the dependency sequence approach can potentially have lower overhead than vector clocks.

Global checkpointing can be an expensive operation. However, its utility is not limited to just reducing the size of dependency sequences. Global checkpoints are generally employed by a distributed application for several other purposes such as predicate evaluation, recovery from failure, etc. Failure recovery is a major concern for large mobile computing systems, where nodes often operate in hostile

conditions. Hence, if the cost of checkpointing is amortized over the numerous uses it can be put to, it is a reasonable price to pay.

Once the earlier part of the dependency sequences are stored in permanent storage during checkpointing, they do not have to be sent again in subsequent messages. Only the portions of the dependency sequences that were generated after checkpointing have to be sent. It is like factoring out the *common prefix* of the dependency sequence from the control information periodically, and reaping the advantages of such an elimination in all subsequent communications.

All dependency sequences of an *MH* are stored in the memory of the *MSS* of the *MH*'s current cell. An *MSS*'s memory is much larger than the memory of individual *MH*s. Although the available memory on some mobile hosts, like laptop computers, is growing rapidly, in a heterogeneous mobile computing environment there will also be other kinds of devices like PDAs that do not have as much memory as laptop computers. So, for the sake of uniformity of implementation, it is a good idea to move the responsibility of dependency sequence storage to *MSS*s. Also, the total number of *MH*s in the system, and fluctuations in this number over time, has no impact on the maintenance of dependency sequences.

6 Hierarchical Clock Approach

6.1 Process Abstraction

A mobile computing system of n cells can be modeled as a system of n processes, one per cell, where *a process is a partially ordered set of events*. The events of a process correspond to the union of the events of all the mobile hosts in the cell. The relative order of message send and receive events of the mobile hosts on the process' time-line is the order in which they are observed by the cell's *MSS*. The message send and receive events of different *MH*s in a cell may be causally independent. In such a situation, even though they occur in the same process (which abstracts the behavior of an entire cell), they are mutually concurrent.

For example, the two cell system with four *MH*s and two *MSS*s shown in Figure 4(a) can be abstracted by a two process system shown in Figure 4(b). The horizontal dotted lines in Figure 4(b) represent the two abstracted processes. The solid directed lines indicate causal dependencies. The lines labeled i indicate dependencies created by intra-process events, while the lines labeled m indicate dependencies created by inter-process message communication. There is no dependency between the event corresponding to the send of a message from MH_b to MH_d ($e_{p,3}$) and the event corresponding to the reception of a message by MH_a from MH_c ($e_{p,4}$). Hence, in Figure 4(b), even though they occur at the same abstracted process, there is no causal dependency between them. Similarly, there is no causal dependency between events $e_{q,1}$ and $e_{q,2}$, or between $e_{q,2}$ and $e_{q,3}$.

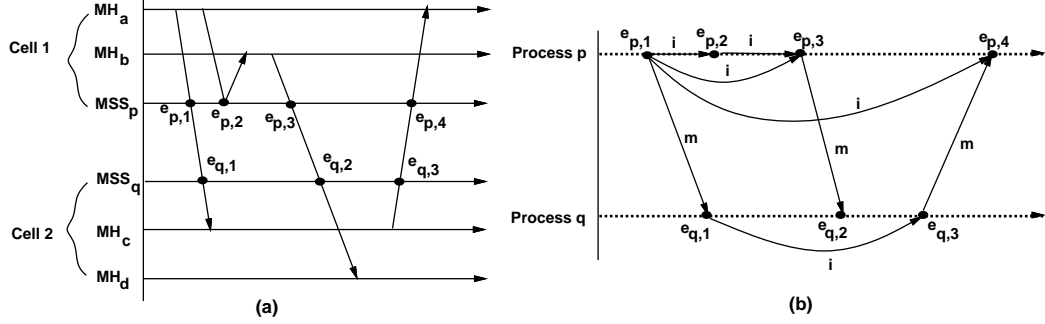


Figure 4: Process abstraction for hierarchical clocks.

The abstraction of a cell's behavior as a process with partial ordering between its events, as described above, is a deviation from the standard definition of a process where all its events are totally ordered. Thus, Lamport's *happened before* relation (\rightarrow) cannot completely capture the causal dependency relation. Instead, a new relation, \rightsquigarrow is defined over a set of events e_1 , e_2 , and e_3 as follows:

1. $e_1 \rightsquigarrow e_2$ if $e_1 \overset{i}{\rightsquigarrow} e_2$, i.e., events e_1 and e_2 occur on the same abstracted process (cell) and there is a causal dependency from e_1 to e_2 .
2. $e_1 \rightsquigarrow e_2$ if $e_1 \overset{m}{\rightsquigarrow} e_2$, i.e., e_1 is the send event of a message in one abstracted process (cell) and e_2 is the corresponding receive event in another abstracted process.
3. $e_1 \rightsquigarrow e_2$ if $\exists e_3 : e_1 \rightsquigarrow e_3$ and $e_3 \rightsquigarrow e_2$.

In Figure 4(b), the directed lines representing the $\overset{i}{\rightsquigarrow}$ relation have been labeled by the letter i , and those representing the $\overset{m}{\rightsquigarrow}$ relation have been labeled by letter m .

The order of message send and receive events actually observed by a cell's *MSS* represents one of the *potentially many possible* order of message sends and receives in the cell. As described earlier, as message send and receive events are the only events that can create causal dependencies, we will not consider internal events of *MHs* in our process abstraction. The process abstraction along with the \rightsquigarrow relation is an alternative to dependency sequences for representing the past cones of events.

In [1] also, a process modeled as a partially ordered set of events has been mentioned as a specific system model for distributed systems where all nodes are fixed. Such a model is then used in conjunction with two-way flush messages to partially recognize concurrent intra-process events.

6.2 Clock Description

Based on the process abstraction described above, each event of the abstracted process is assigned a hierarchical clock value to capture the \rightsquigarrow relation between events. Such a clock was also proposed in [6] for distributed systems where all nodes are fixed. The hierarchical clock ϕ is maintained at *MSSs* and has two components:

1. ϕ^i is a local clock representing the \rightsquigarrow^i relation. It is a variable length bit-vector with one bit for every event that has occurred in the process thus far. Each event (message send or message receive) observed by the abstracted process is assigned a unique sequence number in a monotonically increasing fashion. Let $e_{i,x}$ and $e_{i,y}$ be the x^{th} and y^{th} events of the abstracted process P_i , respectively. $\phi^i(e_{i,x})$ is x bits long and its x^{th} bit is set to 1. Also, $\phi^i(e_{i,x})$ is generated by bit-wise ORing the $\phi^i(e_{i,y})$ vectors for all events $e_{i,y}$ such that $e_{i,y} \rightsquigarrow^i e_{i,x}$. Thus, for every local event of process P_i that causally precedes $e_{i,x}$, the corresponding bit is set to 1 in $\phi^i(e_{i,x})$.
2. ϕ^m is a global clock representing the \rightsquigarrow^m relation. It is an integer-vector of n components, one for every abstracted process (cell) in the system. $\phi^m(e_{i,x})[k]$, the k^{th} component of event $e_{i,x}$'s global clock, identifies the last event on process P_k that causally precedes $e_{i,x}$.

For a message send event $e_{i,x}$, the *MSS* of the sending cell takes the following action to update the hierarchical clock of the corresponding abstracted process: $\phi^m(e_{i,x})[k]$, where $k \neq i$, is set to the maximum of the k^{th} components of all $e_{i,y}$ such that $e_{i,y} \rightsquigarrow^i e_{i,x}$. $\phi^m(e_{i,x})[i]$ is set to x . $\phi^m(e_{i,x})$ is sent with the message as its vector timestamp. For a message receive event $e_{i,x}$, the *MSS* of the receiving cell takes the following action: $\phi^m(e_{i,x})[k]$ ($k \neq i$) is set to the maximum of the k^{th} components of all $\phi^m(e_{i,y})$ such that $e_{i,y} \rightsquigarrow^i e_{i,x}$ and the k^{th} component of the vector timestamp carried by the message. $\phi^m(e_{i,x})[i]$ is set to x .

Figure 5 shows the ϕ^i (local clock) and ϕ^m (global clock) components of the hierarchical clock for each event in the distributed computation shown in Figure 4.

Each message sent between *MSSs* needs to carry only the global clock component ϕ^m . The ϕ^i and ϕ^m values of $e_{i,x}$ can be used to generate a bit matrix, ϕ , representing all events across the entire system that causally precede $e_{i,x}$. The i^{th} row of the bit matrix is equal to $\phi^i(e_{i,x})$. The j^{th} row of the bit matrix denotes the events in the abstracted process P_j (cell j) that causally precede $e_{i,x}$. Thus, the bits in row $\phi(e_{i,x})[j]$ that are equal to 1 indicate events that belong to at least one of the following categories:

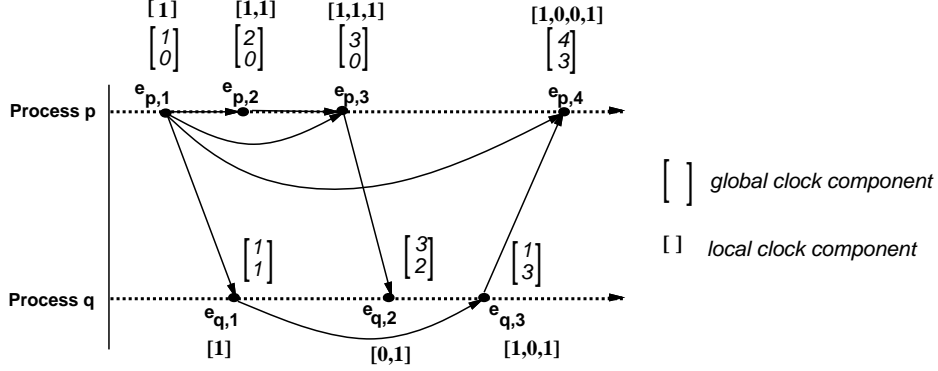


Figure 5: Local and global clock components for a distributed computation.

1. events in process P_j that are in the past cone of local events $e_{i,y}$ (local events are events that occur in the same cell as $e_{i,x}$) that causally precede $e_{i,x}$,
2. events in process P_j that are in the past cone of the latest events in all other abstracted processes (cells) that causally precede $e_{i,x}$.

Hence, the j^{th} row, represented as $\phi(e_{i,x})[j]$ is evaluated using the following recurrence relation [6]:

$$\phi(e_{i,x})[j] = \left(\bigvee_{\forall k \neq i, \phi^m(e_{i,x})[k]=y} \phi(e_{k,y})[j] \right) \bigvee \left(\bigvee_{\forall z: \phi^l(e_{i,x})[z]=1} \phi(e_{i,z})[j] \right)$$

The first component is the bit-wise OR of the j^{th} rows of the bit matrices of latest events at other processes P_k that causally precede $e_{i,x}$. The second component is the bit-wise OR of the j^{th} rows of the bit-matrices of local events $e_{i,y}$ such that $e_{i,y} \rightsquigarrow^i e_{i,x}$. Thus, a dependency chain has to be traversed to generate the bit matrix.

6.3 Determination of Causal Relation

Let $|\phi(e_{i,x})[i]| = l$, and $|\phi(e_{j,y})[j]| = m$. This implies that $e_{i,x}$ is the l^{th} event in the abstracted process P_i , and $e_{j,y}$ is the m^{th} event in the abstracted process P_j . The ϕ matrix for an event in process P_j will reflect knowledge of the occurrence of $e_{i,x}$ only if that event in P_j is causally dependent on $e_{i,x}$. This knowledge will be reflected by the presence of a 1 in the l^{th} position of row i of the ϕ matrix for the event. Therefore,

1. $e_{i,x} \rightsquigarrow e_{j,y}$ iff $\phi(e_{j,y})[i][l] = 1$
2. $e_{i,x} \not\rightsquigarrow e_{j,y}$ iff $\phi(e_{j,y})[i][l] = 0$

3. $e_{i,x} \parallel e_{j,y}$ iff $e_{i,x} \not\prec e_{j,y} \wedge e_{j,y} \not\prec e_{i,x}$

Thus, in order to determine the causal relationship between two events, values of one bit in each of the dependency matrices, ϕ , needs to be checked. However, in the worst case scenario, prior to such a checking, both $\phi(e_{i,x})$ and $\phi(e_{i,y})$ may need to be recursively evaluated completely. The matrix generation effort can be reduced if after each iteration of the recurrence evaluation the partially generated matrices are compared to determine if the x^{th} bit $\phi(e_{j,y})[i]$ (corresponding to $e_{i,x}$) is equal to 1. If so, the evaluation of the recurrence relation can be stopped with the inference that $e_{i,x} \rightarrow e_{j,y}$.

6.4 Managing Handoff

When a mobile host MH_a moves from cell i to j , the handoff procedure is executed. MH_a deregisters from MSS_i by sending a message to MSS_i which carries information about MSS_j , its new mobile service station. On receiving the deregistration message, MSS_i sends a registration message to MSS_j over the wireline network. The registration message carries the current value of ϕ^m at MSS_i . In response to the registration message receive event, $e_{j,y}$, following actions are taken at MSS_j :

1. $\phi^i(e_{j,y})$ is updated to reflect the occurrence of $e_{j,y}$.
2. $\phi^m(e_{j,y})$ is computed by taking the component-wise maximum of ϕ^m at MSS_j before the reception of the registration message, and of ϕ^m received in the registration message.

As in the dependency sequence approach, fluctuations in the number of MH s has no impact on the hierarchical clock approach for the same reason: causal dependencies due to communication are abstracted at cell level rather than MH level, and the number of cells remains fixed.

6.5 Overheads

An integer vector of n components and a bit vector are associated with each dependency causing event. The length of the bit vector is equal to the number of dependency causing events that have occurred in the abstracted process (cell). However, as described earlier in the context of dependency sequences, the length of bit vector can be bounded by periodically taking global checkpoints and resetting bit vector size to zero.

Hierarchical clocks impose low run-time communication overheads. Each message communicated between MSS s over the high data rate wireline network has to carry a vector of n integers. As n is much smaller than the total number of

nodes in the system, N , this results in significant bandwidth savings as compared to the vector clock approach where each message would have to carry an N -integer vector.

7 Comparison of Alternatives

Dependency sequences and hierarchical clocks are two alternatives to vector clocks for representing causal dependencies in a mobile computing environment. Both approaches depend on *MSSs* to maintain the clocks and are immune to fluctuations in the number of mobile hosts in the system. Handoffs are easily handled in both cases.

In the dependency sequence approach, the amount of dependency information that is sent with each message on the wireline network could be large, especially if there are a large number of dependency gaps for message send events. In the hierarchical clock approach, on the other hand, each message needs to carry only a vector of n integers. However, with the hierarchical clock approach the recurrence relation stated in Section 6.2 has to be evaluated to determine the systemwide causal predecessors of an event. This can be time consuming as the dependency chains have to be traversed. Moreover, messages need to be sent between *MSSs*, over the fixed wireline network, for such a traversal. With the dependency sequence approach, the set of dependency sequences always indicates the systemwide set of causal predecessors of an event. No traversal of dependency chains is required.

Given two events, the causal precedence relation between them can be easily determined using the dependency sequence approach in time that is proportional to the log of the length of the longest dependency sequence in the set. However, when hierarchical clocks are used, such a determination may require the evaluation of the recurrence relation which may impose relatively high communication and computation overheads.

Thus there is a tradeoff between the two approaches: the dependency sequence approach has high run-time communication overheads, however, it incurs no extra effort or time delay to determine the causal predecessors of an event. The hierarchical clock approach has low run-time communication overheads, but determination of causal predecessors of an event incurs delay and communication overheads.

The choice between the two alternatives would depend on how often the past cone of an event has to be determined and how often causal dependency relations between pairs of events have to be evaluated. If such operations are to be performed frequently, high run-time overheads of the dependency sequence approach are justifiable. However, if such operations are performed infrequently, high over-

heads incurred by the hierarchical clock approach during such operations will be more than offset by its low run-time overheads.

8 Dependency Tracking for Internal Events

So far we have focused only on message *send* and *receive* events. This is because only such events lead to the creation of causal dependencies between nodes in the system. However, in the time interval between any pair of communication events at a node (mobile or static), internal events can also occur.

To determine the causal dependency between internal events at different *MHs*, we first have to determine which communication events (send/receive) bound the intervals during which they occurred. Let there be two such intervals of internal events, namely I_1 and I_2 , in two different cells. If the communication event marking the end of interval I_1 causally precedes the communication event marking the beginning of interval I_2 , then all the internal events in the interval I_1 causally precede all the internal events in the interval I_2 . If the end marking event of interval I_1 does not causally precede the beginning marking event of interval I_2 and the end marking event of I_2 does not causally precede the beginning marking event of I_1 , then all internal events in I_1 are said to be concurrent to all internal events in I_2 . As processes executing at each *MH* are assumed to be sequential in nature, causal precedence between internal events in an interval is determined by the order of their occurrence.

9 Conclusion

Using static array implementation of vector clocks to capture causal dependencies in mobile computing systems is not feasible for two reasons. First, the size of the vector may be large when a large number of mobile hosts are participating in a distributed computation. Second, the number of participating mobile hosts can change with time, while static array implementations of vector clocks require the number of nodes to remain fixed. Implementing vector clocks through a (*name, time*) pair for each node, static or mobile, raises serious garbage collection issues.

Hence, there is a need for efficient representation of causal dependencies. In this paper we proposed two alternatives to vector clocks, namely, *set of dependency sequences* and *hierarchical clocks*. Dependency sequences always contain complete information about all the causal predecessors of an event, but can have high run-time communication overheads. Hierarchical clocks have low run-time communication overheads, but determination of all causal predecessors of an event

requires traversal of a dependency chains which may be incur delays and have message overheads.

Both approaches abstract the collective behavior of all nodes in a cell into a process whose events can be partially ordered, instead of being totally ordered. Migration of *MHs* from one cell to another, and disconnected operations of *MHs* for indeterminate periods of time can be tolerated by such an abstraction. Hence, both dependency sequence approach and hierarchical clock approach are immune to fluctuations in the number of nodes in the system, and in their distribution among the cells.

The communication overheads of the dependency sequence approach will be low if there is either high or low volume of communication between nodes in the system. In the former case, there will be very few gaps, and therefore short dependency sequences. In the latter case, there will be very few dependency sequences. The actual overheads will depend on the communication pattern. If there is high communication activity in a cell, its bit-vector used in the hierarchical clocks may become long.

The communication overheads of both the approaches can be kept within practical limits by taking global snapshots of the system at regular intervals, and resetting the dependency sequence or bit-vectors to an empty set and a vector of length zero, respectively. Therefore, the communication overheads of both the approaches can be less than the vector clock approach where each message has to carry a vector of as many integers as the total number of nodes, fixed and mobile. In the dependency sequence approach, the most expensive operation is the *merge* of pairs of integer sequences with time complexity of the order of the lengths of these sequences. In the hierarchical approach, clock updates require bit-wise OR operations to update ϕ^i and $O(n)$ integer comparisons to update ϕ^m .

Acknowledgment

The authors wish to thank Professor Friedemann Mattern of Darmstadt Technical University, Germany for providing valuable comments on an earlier draft of this paper.

References

- [1] M. Ahuja, T. Carlson, A. Gahlot, and D. Shands, Timestamping events for inferring “affects” relation and potential causality, Technical Report OSU-CISRC-5/91-TR13, The Ohio State University, Computer and Information Science Research Center, 1991.
- [2] B. R. Badrinath, A. Acharya, and T. Imielinski, Structuring distributed algorithms for mobile hosts, *Proceedings of the 14th International Conference on Distributed Computing Systems*, pp. 21–28, June 1994.
- [3] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, *Information*

- Processing Letters*, 39(1991), July, pp. 11–16.
- [4] J. Fidge, Timestamps in message-passing systems that preserve the partial ordering, *Proceedings of the 11th Australian Computer Science Conference*, pp. 56–66, February 1988.
- [5] G. H. Forman and J. Zahorjan, The challenges of mobile computing, *IEEE Computer*, 27(4)(1994), April, 38–47.
- [6] A. Gahlot and M. Singhal, Hierarchical clocks, Technical Report OSU-CISRC-93-TR19, Computer and Information Science Research Center, The Ohio State University, 1993.
- [7] L. Lamport, Time, clocks and the ordering of events in a distributed system, *Communications of the ACM*, 21(7)(1978), July, 558–565.
- [8] F. Mattern, Virtual time and global states of distributed systems, *Proceedings of the Workshop on Parallel and Distributed Algorithms*, M. Cosnard et. al., eds, Elsevier Science Publishers B.V.(North-Holland), 1989, pp. 215–226..
- [9] F. Mattern, On the relativistic structure of logical Time in distributed systems, *Datation et Controle des Executions Reparties, Bigre 78 (ISSN 0221-525)*, pp. 3–20, 1992.
- [10] M. Raynal and M. Singhal, Logical time: capturing causality in distributed systems, *IEEE Computer*, February 1996, pp. 49–56.
- [11] R. Schwarz and F. Mattern, Detecting causal relationships in distributed computations : in search of the holy grail, *Distributed Computing*, 7(3)(1994), 149–174.

Appendix

The function *merge* is defined as follows:

```

merge(seq1, seq2)
{ int i1, i2, j1, j2;
  sequence_of_integers S;
  S ← {};
  i1, j1 ← 1;
  i2, j2 ← 2;
  while (not end of seq1 ∧ not end of seq2)
    { if (Si2 < Rj1 - 1)
      { S ← S ∪ {Si1, Si2};
        i1 ← i1 + 2, i2 ← i2 + 2;}
      else if (Rj2 < Si1 - 1)
      { S ← S ∪ {Rj1, Rj2};
        j1 ← j1 + 2, j2 ← j2 + 2;}
      else if (Si1 ≤ Rj1 ≤ Rj2 ≤ Si2)
      { S ← S ∪ {Si1, Si2};
        i1 ← i1 + 2, i2 ← i2 + 2;
        j1 ← j1 + 2, j2 ← j2 + 2;}
      else if (Rj1 ≤ Si1 ≤ Si2 ≤ Rj2)
      { S ← S ∪ {Rj1, Rj2};
        i1 ← i1 + 2, i2 ← i2 + 2;
        j1 ← j1 + 2, j2 ← j2 + 2;}
      else if (Rj1 - 1 ≤ Si2 ≤ Rj2)
      { S ← S ∪ {Si1, Rj2};
        i1 ← i1 + 2, i2 ← i2 + 2;
        j1 ← j1 + 2, j2 ← j2 + 2;}
    }
}

```

```

        else if ( $S_{i_1} - 1 \leq R_{j_2} \leq S_{i_2}$ )
            {  $S \leftarrow S \cup \{R_{j_1}, S_{i_2}\}$ ;
               $i_1 \leftarrow i_1 + 2, i_2 \leftarrow i_2 + 2$ ;
               $j_1 \leftarrow j_1 + 2, j_2 \leftarrow j_2 + 2$ ;}
        } /* end while */
    if ( $i_2 > 2i$ )
         $S \leftarrow \text{concatenate}(S, \{R_{j_1}, \dots, R_{j_2}\})$ 
    else if ( $j_2 > 2j$ )
         $S \leftarrow \text{concatenate}(S, \{S_{i_1}, \dots, S_{i_2}\})$ 
    compact(S);
    return(S);
}

```

where $\text{compact}(S)$ is defined as follows: for every sequence of integers in S of the form $\{\dots, S_{2i+1}, S_{2i+2}, S_{2i+3}, S_{2i+4}, \dots\}$ such that $S_{2i+2} = S_{2i+3}$ or $S_{2i+2} = S_{2i+3} - 1$, delete S_{2i+2} and S_{2i+3} from the sequence.

Function merge requires the sequences seq1 and seq2 to be traversed once. The length of the sequence produced as a result is less than or equal to the sum of the lengths of seq1 and seq2 . This resultant sequence is then scanned once during the compact operation. So, the computational complexity of the $\text{merge}(\text{seq1}, \text{seq2})$ function is $O(|\text{seq1}| + |\text{seq2}|)$.