The Dissertation Committee for Chakarat Skawratananond
Certifies that this is the approved version of the following dissertation:

A Framework for Distributed Applications on Systems with Mobile Hosts

Committee:

Vijay K. Garg, Supervisor

Craig M. Chase

Gustavo A. De Veciana

Mohamed G. Gouda

San-Qi Li

A Framework for Distributed Applications on Systems with Mobile Hosts

by

Chakarat Skawratananond, B.Eng., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2002

To my family

Acknowledgments

First and foremost, I am deeply indebted to my supervisor Prof. Vijay Garg. His guidance has been invaluable in shaping this dissertation and taking it to completion. I am also grateful to Dr. De Veciana, Dr. Craig Chase, Dr. Mohamed Gouda, and Dr. San-qi Li for graciously serving on my dissertation committee.

I also would like to thank my colleagues in the Parallel and Distributed Systems Lab. In particular, Neeraj Mittal contributed significantly to this dissertation. He co-authored the chapter on causal message ordering algorithm. Alper Sen, J. Roger Mitchell, Om Damani, Ashis Tarafdar, and Venkatesh Murty provided encouraging and insightful comments that has greatly improved my work.

Lastly, and most importantly, my family has been an endless source of support throughout my graduate study, and all through my life. My parents have always given me the freedom to pursue my interests, and have always backed up my choice with tremendous confidence. My wife has always given me the motivation to complete this doctoral study whenever I need it. This dissertation is dedicated to all of them.

Chakarat Skawratananond

The University of Texas at Austin

May 2002

A Framework for Distributed Applications on Systems with Mobile Hosts

Publication No. _____

Chakarat Skawratananond, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Vijay K. Garg

Mobile hosts have fundamentally different limitations than stationary hosts. For example, mobile hosts generally operate on limited battery power and memory resources. Distributed programs executing on the system with mobile hosts therefore require various modifications to compensate for these factors. In this dissertation, we present a framework that facilitates developing distributed applications for systems with mobile hosts.

Radio spectrum is a scarce resource in mobile systems, efficient allocation of frequency channels is therefore critical for the system performance. The first part of the dissertation addresses this problem. We present a distributed *update-based* algorithm that imposes lower message complexity, while requiring smaller storage overhead than existing algorithms.

In the second part of the dissertation, we propose an efficient way to implement causal message ordering in mobile computing systems. Causally ordered message delivery is a required property for several distributed applications particularly those that involve human interactions (such as teleconferencing and collaborative work). Our message overhead in wired network is independent of the number of

mobile hosts, therefore, the proposed algorithm is scalable and can easily handle dynamic change in the number of participating mobile hosts in the system. Our algorithm, when compared to previous proposals, offers a low unnecessary delay, low message overhead and optimized handoff cost.

Determining order relationship between events in the computation is a fundamental problem in distributed systems with applications in distributed monitoring and fault-tolerance. Fidge and Mattern's vector clocks capture the order relationship with vectors of size N in a system with N processes. Due to limited resources in mobile computing systems, it is natural to ask if this overhead can be reduced. The third part of the dissertation addresses this problem. We present efficient timestamping algorithms suitable for mobile computing systems.

Contents

Chapter 1

Introduction

The emergence of personal digital assistants and handheld personal computers with communication capabilities has had a significant impact on distributed systems. These devices provide users freedom to move anywhere under the service area while retaining network connection via wireless channels. Wireless networks have fundamentally different properties than typical wired networks, including higher error rates, lower bandwidth, nonuniform transmission propagation, increased usage costs, and increased susceptibility to interference and disconnection. Similarly, mobile devices behave differently and have fundamentally different limitations than stationary devices. For example, the point of connection to the network of the moving host is dynamic, and mobile hosts generally operate on limited battery power and memory resources. Distributed programs that run on the system with mobile devices therefore require many modifications to compensate for these factors.

1.1 Mobile Network Architectures

The mobile computing system considered in this research consists of two kinds of processing units: *mobile hosts*, and *mobile support stations*. A mobile host (MH) is a host that can move while retaining its network connections. A mobile support stations (MSS) is a machine that can communicate directly with mobile hosts.

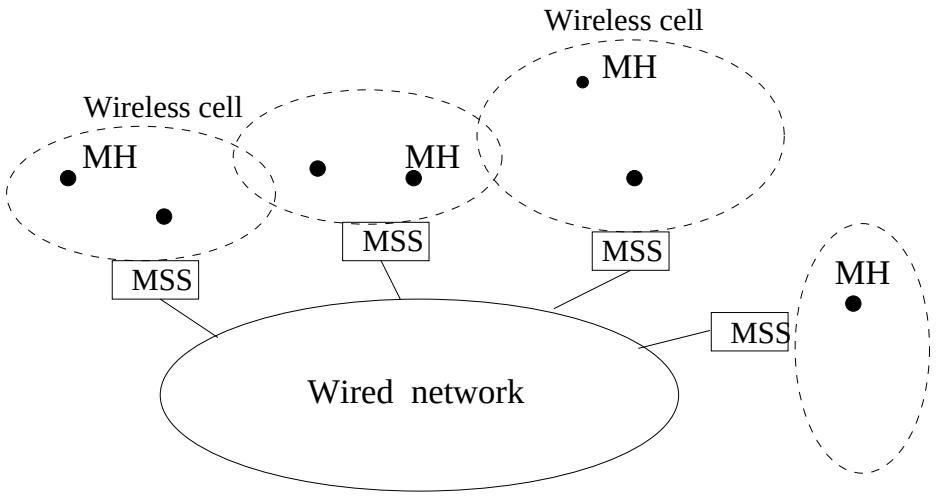


Figure 1.1: A mobile computing system.

The coverage area under an MSS is called a *cell*. Even though cells may physically overlap, an MH can be directly connected through a wireless channel to at most one MSS at any given time. An MH can communicate with other MHs and MSSs only through the MSS to which it is directly connected. All MSSs and communication paths between them form the *wired network*. Figure 1.1 illustrates a mobile computing system.

In this research, we develop a framework that facilitates developing applications for distributed systems with mobile devices. This framework consists of two layers: Channel allocation Layer (CL) and Message delivery Layer (ML). We also present an efficient monitoring system suitable for mobile computing systems with a large number of mobile hosts. Specifically, we introduce an alternative to *vector clocks* for characterizing causality relationship in distributed systems.

1.2 Channel Allocation

In the first layer, CL, we provide an efficient distributed channel allocation algorithm. Each mobile host can establish communication with other entities of the

network only through its local base station via a wireless channel. Availability of radio channels, therefore, plays an important role in achieving good system performance. A radio channel can be used by any two cells at the same time without co-channel inteference if the reuse constraint is satisfied. Here, we use the distance between cells as the reuse constraint.

One simple approach is to have a central controller assign channels to cells so that there is no interference. Obviously, this centralized solution does not scale well, and it suffers from a single point of failure. Another approach is to distribute tasks and responsibilities to each MSS in the system. In this distributed approach, each MSS exchanges information with its neighbors within the co-channel inteference range so that it can make a decision about channel allocation in its cell based on its local knowledge. Therefore, distributed algorithm is more robust and scalable.

We present a distributed algorithm in which each MSS bears an equal amount of responsibility for channel allocation control. Each cell is required to communicate only with a small subset of its neighbors within the co-channel inteference range in order to acquire a channel. No responses are ever deferred, therefore, the communication set up time is relatively small. Moreover, the algorithm is simple to implement, and requires less storage and message overhead than existing algorithms.

1.3 Message Delivery

The second layer (ML) deals with message ordering in distributed computation with mobile hosts. A lightweight algorithm ensuring causal message ordering is presented. Causal message delivery is required in many distributed applications such as management of replicated data [BJ87, BSS91], resource allocation [RST91], and multimedia systems [AS95].

Algorithms to implement causal message ordering in systems with static hosts have been extensively studied. These algorithms, however, require high message and

memory overheads; therefore, they are not suitable for systems with mobile devices. We propose an algorithm for causal message ordering in which message overhead is small compared to those for static systems and limited resources on mobile hosts are efficiently utilized. Moreover, the proposed algorithm is well-suited for dynamic systems since the size of message overhead is constant, even when the number of participating mobile hosts varies.

1.4 Causality Tracking

Determining the order of events is an essential problem in distributed systems. A distributed computation is modeled as a partially ordered set (poset) $(E, \rightarrow)$ where E is the set of events in the computation and $\rightarrow$ is the *happened before* relation [Lam78]. Fidge [Fid89] and Mattern [Mat89] independently introduced vector clocks to timestamp events such that "happened before" relationship between any two events can be determined by examining their timestamps. However, vector clock mechanism does not scale well because it imposes $O(N)$ of local storage on each process and $O(N)$ message overhead in a system with N processes. Due to the limited resource on mobile devices and wireless links, the traditional vector timestamp is not suitable for systems with mobile devices.

We propose an efficient scheme for characterizing "happened before" relationship in distributed systems. Our first proposal is based on drawing connections between vector clocks and dimension theory of partially ordered sets. Dimension provides an encoding scheme (or a timestamping scheme) for a partial order based on representing it as an intersection of some number of *chains*. We introduce the concept of *string dimension* which leads to a more efficient encoding. In particular, the number of bits required to encode a poset using *strings* is smaller than the number of bits required to encode a poset using *chains*.

The first lower bound argument on the size of the vector clocks is due to

Charron-Bost [CBMT96]. Her results show that in the worst case the timestamps may require N-dimensional vector clocks. However, they do not exclude timestamps which use less than N coordinates for interesting subset of computations on N processes. Our second proposal is based on this observation. In particular, we show that timestamping can be done more efficiently in distributed computations that use *synchronous* messages.

1.5 Overview of the Dissertation

The rest of this dissertation is organized as follows. Chapters 2 and 3 describe the channel allocation and message delivery layers, respectively. Chapter 4 provides our implementation of the two layers, and a simple shared object application. In Chapter 5, we present an efficient approach to capture causality and concurrency between events. Finally, we draw conclusions in Chapter 6.

Chapter 2

Channel Allocation Layer

In this chapter, we present an efficient distributed channel allocation algorithm for mobile systems. The material presented here also appears in [SG99].

2.1 Introduction

The radio spectrum is a scarce resource in mobile communication systems. An efficient reuse of the radio spectrum allocated to the system is required as the population of mobile users continue to grow at the tremendous rate. Each mobile host can establish communication with other entities of the network only through its local base station via a wireless channel. Therefore, availability of radio channels plays an important role in achieving good system performance. A major prohibiting factor in radio spectrum reuse is co-channel interference. A radio channel can be reused by any two cells without co-channel interference if the distance between them is at least a required value D_{min}. Spectrum reuse efficiency can be increased by making use of efficient channel assignment techniques.

The radio spectrum is divided into a number of frequency units. If *channels* are those units, then each channel is used to support only one call in a cell at a time. If *carriers* are used as a unit, then the entire bandwidth of a carrier assigned to a cell is time-shared by all the calls initiated within the cell. We choose to use

the term channel throughout this chapter.

A channel can be reused in a spatially disjoint cell if *reuse constraints* are satisfied. Reuse constraints are conditions used to determine when a channel can be reused by other cells without causing an interference called *co-channel interference*. The reuse constraint we consider is the distance between cells.

To avoid co-channel interference each cell must [DL97]

1. compute the set of available channels,
2. select one channel from the set of available channels, and
3. acquire the selected channel.

The procedure that performs task (1) and (3) is referred to as the *channel allocation algorithm*, and the one that performs task (2) referred to as the *channel allocation strategy*. Channel allocation strategies have been extensively studied in the context of cellular telephone systems. A comprehensive survey on channel allocation strategies can be found in [KN96].

Generally, channel allocation strategies can be divided into three different categories: fixed (FCA), dynamic (DCA), and hybrid (HCA). In FCA, channels are assigned to each cell according to some fixed reuse pattern so that the co-channel interference is avoided. This strategy is simple, but it does not adapt to the changing traffic conditions and user distribution. To overcome these disadvantages, DCA strategies are introduced. Channels in DCA can be assigned to any calls as long as there is no co-channel interference. DCA provides traffic adaptability at the cost of higher complexity. However, DCA is less efficient than FCA under high load conditions [KN96]. HCA strategy, a mixture of FCA and DCA, is introduced to overcome this drawback. Channels in HCA are divided into *fixed* and *dynamic* sets. Each cell is assigned with a fixed set as in FCA, and these channels are preferred for use in their respective cells. The dynamic set is shared by all users in the system to

increase flexibility.

There are many different ways to perform task (1) and (3). One simple approach is to have each cell request a channel from a central controller who will ensure that the co-channel interference does not occur. This centralized approach does not scale well, and it has a single point of failure. Another approach is to distribute tasks and responsibilities to each MSS in the system. In this distributed approach, each MSS exchanges information with its neighbors within the co-channel interference range so that it can make a decision about channel allocation in its cell based on its local knowledge. Therefore, distributed algorithm is more robust and scalable.

A distributed channel allocation algorithm should aim at minimizing the channel acquisition delay.[1] It must also aim at minimizing the amount of information (size and number of messages) that needs to be exchanged per request. Also, the algorithm must be able to cope with failures of the components of the network. One criterion to measure this property is the *failure number* which measure the number of cells affected by a faulty support station. Clearly, the goal is to minimize the failure number of the algorithm.

We present a distributed algorithm in which each MSS bears an equal amount of responsibility for channel allocation control. Each cell is required to communicate only with a small subset of its neighbors within the co-channel interference range in order to acquire a channel. No responses are ever deferred, therefore, the communication set up time is relatively small. The failure number of the algorithm is kept at minimum. It is one-half of the size of the interfering neighbors in the system with the smallest D_{min}. Moreover, the algorithm is simple to implement, and requires less storage and message overhead than existing algorithms.

The problem of distributed channel allocation can be viewed as a variant of

[1] The elapsed time between a cell's sending a request message and its acquisition of a channel.

distributed mutual exclusion problem, where each channel corresponds to a critical section. In distributed channel allocation, however, multiple cells can simultaneously use the same channel (enter a critical section at the same time) if the reuse constraint is satisfied. Algorithms previously proposed for channel allocation are more or less based on techniques used in mutual exclusion problems. Next, we briefly describe them, and compare them in detail with the algorithm presented here in the following section.

The rest of the chapter is organized as follows. The related work is given in Section 2.2. Section 2.3 presents notation and definitions that we used in this chapter. System with the smallest possible D_{min} is investigated in Section 2.4. The technique given in Section 2.4 is generalized in Section 2.5. An optimization of the proposed algorithm is given in Section 2.7. Experimental results are given in Section 2.8.

2.2 Related Work

Prakash et al. [PSS95] proposed an algorithm based on *deferral* technique used in Ricart and Agrawala [RA81]. In this algorithm, each MSS decides which channel it can use after gathering information from each cell in its interference neighborhood. Each MSS replies with the information containing its channel use if it is not requesting a channel, or if it is requesting with a larger timestamp than the timestamp of the received request. Otherwise, the response is deferred. With all the received information, the MSS may have to borrow a non-busy channel allocated to its neighbors. With the permission from each neighbor that owns this channel, the MSS can transfer the channel and then uses it to support a communication session. Otherwise, the MSS must select a new channel to transfer, and repeat another borrowing procedure. This algorithm will be referred to as PK.

Choy and Singh [CS96] proposed a new borrowing scheme (referred to as

CS) that reduces the number of failed transfers. In this scheme, a randomly selected group of channels is exclusively locked by the MSS before the actual transfer occurs. Each MSS obtains a lock on the selected group of channels by executing an existing dining philosophers algorithm [CS95]. This ensures that no two interfering MSSs lock the same group of channels simultaneously. This algorithm also guarantees that each lock request is eventually granted. The optimum size of the group varies under different load distributions. It was shown in [CS96] by simulations that CS significantly outperforms PK when the traffic load is high and non-uniformly distributed. This is primarily because pessimistic approach acquires a group of channels, and more requests therefore can be served per each acquisition. This saves a lot of communication and processing overhead

Both PK and CS fall into the *search* category according to [DL97] since each MSS does not maintain the information about the channel use of its neighbors. When a channel is needed, the MSS searches all neighboring cells to compute the set of currently available channels. Due to deferring mechanism used to resolve conflicts, acquisition delay in search algorithms will significantly increase as traffic in the network grows. In particular, the response of the request message received by each cell will be deferred even when the cell is requesting a different channel.

Instead of gathering information each time a channel is needed, each MSS could maintain a set of available[2] channels by informing cells in its interference neighborhood each time it acquires and releases a channel. This is called *update* scheme [DL97]. A conflict occurs if two cells within each other interference neighborhood concurrently request the same channel. Therefore, to request a channel, each cell must send a request message to each cell in its interference neighborhood. The request is successful if each response is a *grant* message. This is known as the *basic update* scheme. This scheme reduces acquisition delay at the expense of higher

[2]Channels that cells can use without co-channel interference

message complexity. Algorithms presented in [IC93, IC94] take this approach.

Dong and Lai [DL97] proposed an update algorithm (referred to as DL). They reduced message complexity by using an *inquire-some mutual exclusion model*. This algorithm is designed to work with a class of HCA strategies such that carriers[3] assigned to each cell with different priorities. In particular, each cell is initially given a set of preferred carriers so that they are always selected whenever available before other carriers. This a priori information about carrier status is referred to as *channel reuse planning*. To request a carrier, each cell sends request messages only to a small subset of its interference neighborhood, depending on the carrier being requested. Thus, each cell has to maintain a list of cells for each carrier to which request messages must be sent.

Garg et al. [GPT96] proposed two update algorithms. In the first algorithm, all MSSs are synchronized such that no two neighboring MSSs choose frequencies at the same time. This is achieved by using the same synchronization mechanism as in CS. Each MSS informs its neighbor which channels are selected. The neighbors are also informed when the channels are released. Again, the response time of this algorithm is high due to the fact that a cell can be blocked by its neighbors if they are requesting simultaneously even though the requested channels are different. This algorithm will be referred to as G-Det. The second algorithm is intended for the system where the number of channels available to the system is much larger than the number of channels requested by each cell. In this algorithm (referred to as G-Ran), a group of channels is randomly picked, and the MSS must ensure that the selected channels are not currently used by its neighbors. If so, the channels are dropped. Note that the channels selected in G-Det will never be dropped.

The algorithm presented here (referred to as QB[4]) uses Maekawa's technique [Mae85] to reduce message complexity in the basic update algorithm. Similar to

[3] *Carrier* is the basic unit of resource used in their algorithm.

[4] QB comes from quorum-based.

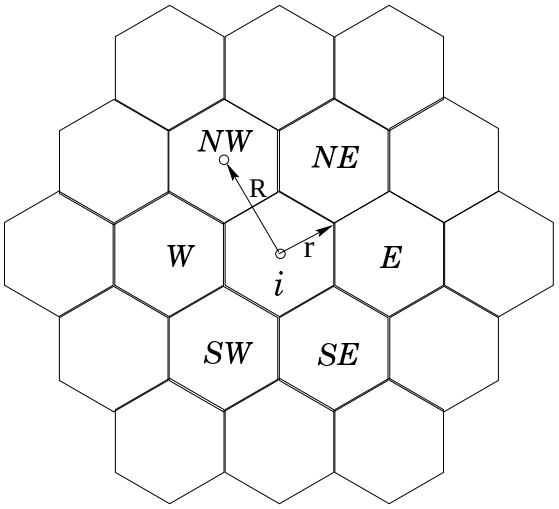


Figure 2.1: Hexagonal Cellular Model

DL, request messages are sent to only a small subset of cells in the interference neighborhood. However, this subset does not depend on the requested channels. This algorithm therefore requires less memory space, and is simpler to implement. Unlike DL, this algorithm can be used with any channel reuse planning. Moreover, the generalization of the update algorithm to the system where the interfering cells is more than one hop away is also presented.

2.3 Notation

It is assumed that the allocated radio spectrum is divided into a number of channels. A unique ID is assigned to each channel. Each cell also has a unique ID represented by a pair (x, y), the position of cell on a 2-dimensional plane. Each cell i is logically modelled as a hexagon of radius r; therefore, it has six neighbors, denoted by $E_i, W_i, NE_i, NW_i, SE_i, SW_i$, according to its direction from i. Figure 2.1 displays a cellular network model.

The distance between cells is defined as the distance between the two centers. If R is the distance between two neighboring cells, then $R = \sqrt{3}r$. A channel can

be reused in any two cells if the distance between them is at least D_{min}. We say that two cells are in each other's *interference neighborhood* if the distance between them is less than D_{min}. Note that D_{min} must be greater than R.

We also assume that all MSSs are connected and every message sent between them is eventually received in FIFO order. Each MSS maintains Lamport's logical clock [Lam78] so that events at different MSSs are totally ordered.

Given a cell c, $IN(c)$ denote the set of all the cells whose distance to c is less than D_{min}.

Definition 1

$$IN(c) = \{c' | dist(c, c') < D_{min}\}$$

where $dist(c, c')$ denotes the distance between c and c'. We say that two cells, i, j, interfere with each other if the distance between them is less than D_{min}. We also say that i is in the interference neighborhood of j if $i \in IN(j)$.

The goal of any channel allocation algorithm is to ensure that no two interfering cells simultaneously use the same channel. To achieve this goal, any two requests from two interfering cells must be known to at least one of the arbitrators. If we assume that cell i obtains a permission from each member of a set S_i (*request set*) such that $S_i \subset IN(i)$, there must be at least one common cell between a pair of S_i and S_j for any two interfering cells i, j. We refer to this property as the pairwise non-null intersection property ($\mathcal{PN}$).

We next present a scheme to construct the request set such that $\mathcal{PN}$ is satisfied for a given D_{min}.

2.4 3-Cell Cluster System

In this system, a channel cannot be reused within the same cell or the neighboring cells. This is because $R < D_{min} \leq \sqrt{3}R$. Consequently, there are exactly 6 members in each cell's interference neighborhood. To satisfy $\mathcal{PN}$, we assign to each cell i the

following request set: $S_i = \{i, NW_i, SW_i, E_i\}$. We call this 1-*hop request set*. It is easy to see that the following property can be derived from 1-hop request set. Let S_{ij} denote $S_i \cap S_j$.

Property 2 *For any two interfering cells* i, j

- $|S_{ij}| = 1$, *and*
- $j \in S_{ij}$ *or* $i \in S_{ij}$

Next we present a distributed channel allocation algorithm for 3-cell cluster system using 1-hop request set.

2.4.1 The Algorithm

We now provide an informal description of the 3-cell cluster algorithm shown in Figure 2.2. Let $\mathcal{C}$ denote the set of channels shared by each cell in the system. Each cell c maintains two sets, U_c and I_c. U_c is the set of channels currently used by c. I_c is the set of interfered channels at c. Initially, every channel $k \in \mathcal{C}$ is available for use by c. A channel becomes an *interfered* channel for c if it is acquired by a cell in $IN(c)$.

When a cell needs a channel to support a communication session, it selects a channel k from $(\mathcal{C} - (I_c \cup U_c))$ based on the underlying channel allocation strategy. If the underlying strategy is HCA, then channels in the fixed set are used whenever it is available. To acquire k, a cell sends a request message ($REQ(k)$) to each cell in its request set except itself. Once c receives a grant ($GRNT(k)$) from every cell in its request set, and if at this moment k does not belong to I_c, then channel k can be used to support a communication session in cell c, and is added into U_c. Next, c sends an acquisition message ($ACQ(k)$) to each cell in its interference neighborhood except those in its request set to inform about its use of channel k. When a cell d receives $ACQ(k)$ message, k is added into its set I_d.

MSS$_i$::

var
I_i: the set of interfered channels at i;
U_i: the set of used channels at i;
S_i: the request set of i;
$\mathcal{C}$: the set of channels shared by all cells;

(A) To support a communication session
MSS$_i$ selects a channel $k \in (\mathcal{C} - (I_i \cup U_i))$, according to the underlying allocation strategy, and sends a $REQ(k)$ to each cell in S$_i$

(B) Upon receiving a $REQ(k)$ from cell j
MSS$_i$ replies with a $REJ(k)$ if either $k \in U_i$ or
MSS$_i$ is also requesting for k with a smaller timestamp;
otherwise, it replies with a $GRNT(k)$ and add k into I_i

(C) The request for channel k is successful if
(C1) Every response is a grant, and
(C2) $k \notin I_i$

(D) Upon the request for channel k is successful
send $ACQ(k)$ to each cell $c \in IN_i - S_i$
add k in U_i

(E) Upon receiving a $REJ(k)$ from j or conditions (C1) and (C2) are false
send a $REL(k)$ to each cell $c \in S_i$ except j, and execute (A)

(F) Upon a communication session using channel k terminates,
MSS$_i$ sends a $REL\_TERM(k)$ to each cell in IN_i

(G) Upon receiving a $REL(k)$ or $REL\_TERM(k)$
remove k from I_i

(H) Upon receiving a $ACQ(k)$
add k into I_i

Figure 2.2: The 3-cell Cluster Algorithm

On receiving a request for channel k', c replies with a reject message (REJ) if either c is using this channel ($k' \in U_c$) or c is also requesting k' with a smaller timestamp. Otherwise, c replies with a grant message ($GRNT$), and k' is added into I_c. Note that k' is added into I_c immediately after c has granted the request for k'. When the communication session using k is terminated, c sends a REL_TERM message to each cell in IN_c.

A cell realizes that its acquisition of channel k is failed when (1) a $REJ(k)$ has been received, or (2) a grant has been received from each cell in S_c but $k \in I_c$. Once the acquisition of channel k is failed, c sends release messages ($REL(k)$) to those cells that have already granted k to c. On receiving $REL(k)$ or $REL\_TERM(k)$, k is removed from I_c.

Once an acquisition is failed, it means that a *conflict* has occurred. A conflict occurs when there are at least two cells in each other's interference neighborhood concurrently request the same channel.

2.4.2 Correctness Proof

Theorem 3 *(Safety) No two cells within each other's interference area concurrently use the same channel.*

Proof: Let there be two cells i, j within each other's interference area. We show that i and j cannot simultaneously use the same channel k. Without loss of generality, we assume that the timestamp of i's request is smaller than that of j. Due to Property 2, there are only 2 cases to be considered.

- $j \in S_i$: If i is using k, k must be in I_j and then j will not be able to use k unless i releases k. Similarly, if j is using k, then i will receive a REJ message from j. Therefore, i's request will fail. If both i, j are in the process of requesting k, i must receive a grant from j since its timestamp is smaller

than j's. Therefore, j's request will fail even though it receives $GRNT(k)$ responses from other cells in S_j.

- $i \in S_j$: If either i or j is using k, then the other cannot use k by the same argument as in the first case. If both i and j are in the process of requesting k, then j will receive a $REJ(k)$ message from i because i's timestamp is smaller than j's. ■

Theorem 4 *(Concurrency) The requests of a channel k from any two non-interfering cells will be granted.*

Proof: Consider two non-interfering cells, c_1 and c_2. Assume that both of them are requesting for a channel k. There are two cases. (1) $S_1 \cap S_2 = \emptyset$. In this case, the request for channel k by c_1 will not fail the request by c_2. (2) $S_1 \cap S_2 = c_3$. If c_3 is requesting k with timestamp higher than those of c_1 and c_2 or c_3 is not using k, c_3 sends grant message to both c_1 and c_2 (step (B)). ■

Theorem 5 *(Deadlock freedom) The algorithm is deadlock-free.*

Proof: In this algorithm, each cell may either reject or grant a request message, but it never defers its response. Thus, the algorithm is deadlock-free. ■

We next prove that *livelock* never occur in the 3-cell cluster algorithm. We call an event livelock if more than one cells within each other's interference neighborhood compete for the same channel, and each one of them fails to acquire the channel.

Theorem 6 *(Progress) A conflict on any channel k is always resolved in such a way that there is exactly one successful acquisition.*

Proof: We need to prove that if there are x cells within each other's interference neighborhood requesting for a channel k, then there will be exactly one cell c successfully acquiring k.

Since $R < D_{min} \leq \sqrt{3}R$, x can be either 2 or 3. For $x = 2$, let those two cells be c_1 and c_2. From Property 2, we know that either c_1 will receive the request message from c_2 or vice versa. From step (B) in the algorithm, we know that there will be exactly one successful acquisition.

For $x = 3$, assume that cells c_1, c_2, c_3 are within each other's interference neighborhood and requesting a channel k with timestamps t_1, t_2, t_3, respectively. Without loss of generality, we assume that $t_1 < t_2 < t_3$. Let $p \rhd q$ be an abbreviation for *"a request from p is received by q"*. It follows from the configuration of 1-hop request set that either (1) $c_1 \rhd c_2$, $c_2 \rhd c_3$, and $c_3 \rhd c_1$, or (2) $c_1 \rhd c_3$, $c_3 \rhd c_2$, and $c_2 \rhd c_1$.

In (1), c_3's request will be rejected by c_1 since $t_3 > t_1$. c_2's request will fail because c_2 receives the request from c_1 which has a lower timestamp. Hence, only c_1's acquisition will be successful.

In (2), c_3's request will be rejected by c_2 since $t_3 > t_2$. c_2's request will be rejected by c_1 since $t_2 > t_1$. Again, only c_1's acquisition will be successful. ■

2.4.3 Performance Analysis

We now discuss message complexity and acquisition delay incurred by the 3-cell cluster algorithm. Let n_s denote the number of request messages sent per acquisition, and n_{if} denote the size of $IN(c)$ for each cell c. The number of messages required for a successful acquisition (excluding ACQ and REL_TERM) is $2{*}n_s = 6$. For a failed acquisition, the number of messages required is at most $3 * n_s$. Therefore, if a cell has to make m attempts before it finally acquires a channel, the number of messages required per a channel acquisition is at most $2{*}n_s + 3{*}n_s{*}(m-1) + (n_{if} - n_s) + n_{if}$.

The summation of $(n_{if} - n_s)$ and n_s is the total number of ACQ and REL_TERM messages. After substituting n_s and n_{if} with 3 and 6, respectively, the number of messages required is at most $9(m-1)+15$. In the basic update scheme [IC93, IC94], the number of messages required per a channel acquisition is exactly $12(m-1)+18$.

Let D denote the average communication delay between MSSs. The average acquisition delay for the algorithm is $2 * m * D$. The failure number of the 3-cell cluster algorithm is 3 since each support station receives request messages from exactly three neighboring cells.

Message complexity and acquisition delay of the 3-cell cluster algorithm are the same as DL.

2.5 Generalization

In this section, we present a general scheme to construct a request set for a given D_{min}. We also propose a distributed channel allocation algorithm to use in association with those request sets.

Let r_i^j denote the set of cells that are j hops away from i in the east, northwest, and southwest directions relative to i. The *n-hop* request set of cell i (denoted by S_i^n) is defined as follows:

$$S_i^n = i \cup r_i^1 \cup \ldots \cup r_i^n$$

Figure 2.3 shows S_i and examples of S_{ij} for 2-hop request set. In the previous section, we have shown that 1-hop request set satisfies $\mathcal{PN}$ when $R < D_{min} \leq \sqrt{3}R$. The following lemma shows that 1-hop request set also satisfies $\mathcal{PN}$ even when $R < D_{min} \leq 2R$.

Lemma 7 *The 1-hop request set satisfies $\mathcal{PN}$ if $R < D_{min} \leq 2R$*

Proof: Since $R < D_{min} \leq 2R$, $R \leq dist(c', c) < 2R$ for any $c \in IN(c')$. From the configuration of 1-hop request set, it is easy to see that $S_c \cap S_{c'} \neq \emptyset$. ■

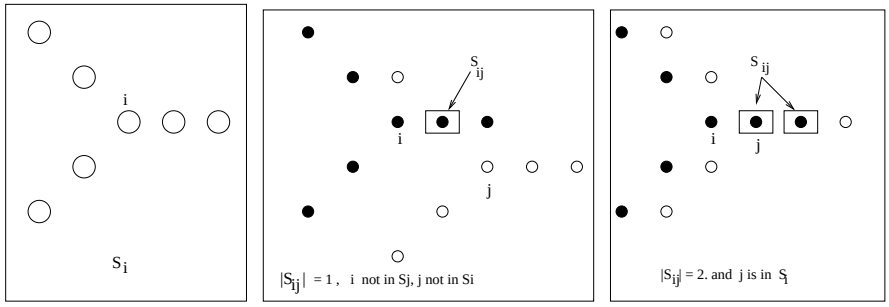


Figure 2.3: S_i and S_{ij} in 2-hop request set

Theorem 8 *For any integer $n \geq 1$, the n-hop request set satisfies $\mathcal{PN}$ if $nR < D_{min} \leq (n+1)R$*

Proof: We prove this theorem by induction on n. The base case is obtained from Lemma 7. Given cells c, c' such that $dist(c, c') < (n+1)R$, we need to show that $S_c \cap S_{c'} \neq \emptyset$. From induction hypothesis, for cell c' such that $dist(c, c') < nR$, we know that $S_c \cap S_{c'} \neq \emptyset$.

Let a pair (d, θ) represent the position of c' relative to c, where d is the distance between c and c', and θ is the angle between the line connecting c and c' and the horizontal axis. Hence, the position of c is $(0, 0)$. Given a cell $c' = (d, \theta)$ such that $nR \leq d < (n+1)R$, the position of cell $x \in S_c \cap S_{c'}$ is as follows:

Case (1) $\underline{d = nR}$:

(1.1) $\theta = 0$, $2\pi/3$, or $4\pi/3$

$\Rightarrow x = c'$

(1.2) $\theta = \pi/3$, π, or $5\pi/3$

$\Rightarrow x = c$

Case (2) $\underline{nR < d < (n+1)R}$:

Cells in this category can be divided into 6 groups. Each group contains exactly n cells, $c'_1, \ldots, c'_n$, such that $\theta_1 < \ldots < \theta_n$,

where θ_i is the angle of cell c'_i, for $i = 1, \ldots, n$.

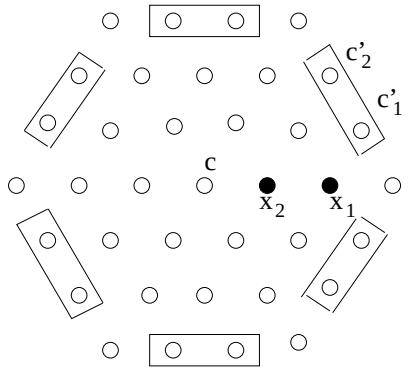


Figure 2.4: Positions of some $x \in S_c \cap S_{c'}$ for $D_{min} = 3R$.

(2.1) $0 < \theta < \pi/3$,

$\Rightarrow x_i = ((n-i+1)R, 0)$

(Figure 2.4 illustrates the positions of x_1 and x_2 for $n = 2$).

(2.2) $\pi/3 < \theta < 2\pi/3$,

$\Rightarrow x_i = (iR, 2\pi/3)$

(2.3) $2\pi/3 < \theta < \pi$,

$\Rightarrow x_i = ((n-i+1)R, 2\pi/3)$,

(2.4) $\pi < \theta < 4\pi/3$,

$\Rightarrow x_i = (iR, 4\pi/3)$

(2.5) $4\pi/3 < \theta < 5\pi/3$,

$\Rightarrow x_i = ((n-i+1)R, 4\pi/3)$

(2.6) $5\pi/3 < \theta < 2\pi$,

$\Rightarrow x_i = (iR, 0)$

According to the n-hop request set, it is easy to see that in each case, $x \in S_c$ and $x \in S_{c'}$. ■

The following property can be derived from the n-hop request set's configuration:

Property 9 *For any two interfering cells i, j,*

- *If $|S_{ij}| \geq 2$, then either $i \in S_j$ or $j \in S_i$.*

2.5.1 The Generalized Algorithm

We now provide an informal description of the generalized algorithm shown in Figure 2.5. This algorithm is intended for the system with $D_{min} > \sqrt{3}R$. In Figure 2.5, we use $*$ to denote *don't care*.

The definition of U_i and $\mathcal{C}$ here are the same as in Section 2.4. The request set S_i is determined by D_{min}. The set I_i is a set of *tickets*. A ticket is a *struct* containing four fields: $Status, Cell\_id, Ch\_id, TS$. Whenever cell i grants a request of channel k with timestamp ts from cell j, a *ticket* (UC, j, k, ts) must be added into I_i (UC for *unconfirmed*). $Status$ is a boolean variable, and becomes CF (*confirmed*) when i receives an $ACQ(k, ts)$ from j. An $ACQ(k, ts)$ is sent to each cell in the interference neighborhood if the acquisition of channel k is successful.

Unlike the previous algorithm, the response of each request can be either $GRNT$, C_GRNT, and REJ (Step `B`). A cell replies with C_GRNT to the request from cell j for a channel k if it has given at least one grant to another request from the cell within the interference neighborhood of j with a bigger timestamp. The ID's of those cells are also sent along with C_GRNT. A cell replies a channel k's request from cell j with $GRNT$ if either it is not using k, or it is not requesting k with a smaller timestamp, or it never grants the request of k to any cells that interfere with j before. Otherwise, the cell replies with REJ.

A cell determines if its request is successful by checking all three conditions given in Step `C`. If i receives $C\_GRNT(W, k)$ from p, it means that the request from each cell $w \in W$ have been granted by p and the request timestamp of w is greater than that of i. If $w \in S_i$ or $i \in S_w$, then w will receive i's request or i will receive w's request, respectively. In either case, w's request will be failed given that i's

request is not rejected by w. Checking if $i \in S_w$ can be done easily by calculating the distance and direction between i and w.

Once a cell realizes that its request has been rejected, it sends REL message to each cell in its request set except the cell(s) from which REJ message has been received. After a call has been terminated in cell i, REL_TERM is sent to each cell in IN_i. On receiving ACQ, REL or REL_TERM, set I_i is updated accordingly. Note that the algorithm presented in Section 2.4 is a special case of the one presented here.

2.5.2 Correctness Proof

Theorem 10 *(Safety) No two cells within each other's interference neighborhood simultaneously use the same channel.*

Proof: We show that any two cells i, j within each other's interference neighborhood cannot simultaneously use the same channel k. Without loss of any generality, assume that the timestamp of cell i's request (t_i) is smaller than that of cell j (t_j). According to the structure of the request set, there are 3 cases to consider: (1) $|S_{ij}| = 1$ and $i \in S_j$ or $j \in S_i$. (2) $|S_{ij}| = 1$ and $i \notin S_j$ and $j \notin S_i$. (3) $|S_{ij}| \geq 2$.

- Case (1): Same as the proof in Theorem 3.
- Case (2): Let $\phi \in S_{ij}$. If cell i's request arrives at ϕ before the request of j, then i will receive $GRNT(k)$ and j will receive $REJ(k)$ from ϕ. If cell j's request arrives at ϕ before i, then i will receive $C\_GRNT(j,k)$ and j will receive $GRNT(k)$ from ϕ. Therefore, i's request will fail because $i \notin S_j$.

  If i's request has been confirmed at ϕ before j's request arrives, then j's request will be rejected by ϕ. On the other hand, if j's request has been confirmed by ϕ before i's request arrives, then i's request will be rejected by ϕ.
- Case (3): From Property 9, we know that either $i \in S_j$ or $j \in S_i$.

MSS$_i$::
Var

struct *ticket* {
$Status$: boolean (CF or UC)
$Cell\_id$: cell ID
Ch_id : channel ID
TS : timestamp }
I_i : a set of *ticket*
$U_i, S_i, \mathcal{C}$: as defined in Figure 2.2.

(A) To support a communication session
(A1) MSS$_i$ selects a channel k from $\mathcal{C}$ such that $k \notin U_i$ and $t.Ch\_id \neq k, \forall t \in I_i$.
(A2) sends a $REQ(k)$ to each cell in S_i

(B) Upon receiving a $REQ(k)$ with timestamp TS from cell j
(B1) MSS$_i$ replies with a $REJ(k)$ if
$k \in U_i$ or
MSS$_i$ is also requesting for k with a smaller timestamp, or
$\exists (UC, j', k, ts) \in I_i : ts < TS$ and $j' \in IN(j)$,or
$\exists (CF, j', k, *) \in I_i$ and $j' \in IN(j)$
(B2) MSS$_i$ replies with a $C\_GRNT(W, k)$ if $W \neq \emptyset$, where $W = \{j' / (UC, j', k, ts) \in I_i \wedge ts > TS \wedge j' \in IN(j)\}$
(B3) otherwise, it replies with a $GRNT(k)$ and add (UC, j, k, TS) into I_i

(C) Cell i's request of channel k is successful if
(C1) Every response is either a $GRNT(k)$ or $C\_GRNT(CID, k)$,and
(C2) For each $C\_GRNT(W, k)$ message that cell i receives, $\forall w \in W : w \in S_i$ or $i \in S_w$, and
(C3) $\nexists t \in I_i : t.Ch\_id = k$

(D) Upon the request of channel k with timestamp t is successful
(D1) cell i sends an $ACQ(k)$ message to each cell in IN_i, and add k in U_i

(E) Upon conditions in (C) are false or a $REJ(k)$ has been received
(E1) send a $REL(k)$ to each cell in S_i except the one(s) rejecting the request, and go back to (A)

(F) Upon receipt of $ACQ(k)$ from cell j
(F1) if $(UC, j, k, *) \in I_i$ then turn UC to CF, otherwise, add $(CF, j, k, *)$ into I_i

(G) Upon receipt of $REL(k)$ or $REL\_TERM(k)$ from j
(G1) remove the corresponding *ticket* from I_i

(H) Upon the call using k is terminated
(H1) send $REL\_TERM(k)$ to each cell in IN_i

Figure 2.5: The Generalized Algorithm

- If $i \in S_j$, from Step (B), j's request will be rejected by i since $t_i < t_j$. If i is using k, j's request will be rejected. If j is using k, then k must be in I_i, and i will not select k.

- If $j \in S_i$, i's request will be granted by j. j'request will fail even though every response that j receives is either $GRNT$ or C_GRNT since condition (C3) is failed.

If j is using k, i's request will be rejected. If i is using k, then k must be in I_j, and j will not select k. ■

Theorem 11 *(Concurrency) Two requests from two non-interfering cells for the same channel will be granted.*

Proof: Consider any two cells x, y such that $x \notin IN(y)$, and cell z such that z is in the interference neighborhood of both x and y. Assume that z has granted the request of channel k from x. When cell z receives the request from y, and z is not using and not requesting a channel k, y's request will be granted (Step (B)) even though $\exists t \in I_z : t.ChID = k$ since $x \notin IN(y)$. ■

Theorem 12 *(Deadlock freedom) The generalized algorithm is deadlock-free.*

Proof: Each cell may either send REJ or $GRNT$ or C_GRNT, but it never defers its response. Thus, the algorithm is deadlock-free. ■

Later, we will show that livelock is possible in the generalized algorithm. Specifically, if more than 2 cells within each other's interference neighborhood request for the same available channel, it is possible that none of them will succeed.

2.5.3 Performance Analysis

We first describe the performance of the algorithm in term of message complexity and acquisition delay. Then, we give the comparison with the previous works.

Let n_s, n_{if}, and D be defined as in Section 2.4. If there is no conflict, the number of messages per channel acquisition is $2 * n_s + 2 * n_{if}$ which is the sum of n_s REQ messages, n_s $GRNT$ or C_GRNT messages, n_{if} ACQ messages, and n_{if} REL_TERM messages. The average delay is $2 * D$.

If there are conflicts, and a cell takes m requests before it acquires a channel. The average delay is $2 * D * m$. The number of messages becomes $2 * n_s + 3 * (m-1) * n_s + 2 * n_{if}$. It is easy to see that the generalized algorithm has n_s failure number.

2.6 Discussion

Here the detailed comparison between our algorithm and existing search and update algorithms is given.

Comparison with search algorithms

As mentioned earlier, search algorithms in general are more likely to suffer from the high acquisition delay during the high traffic load. In PK, the number of messages required per acquisition is smaller than that of update algorithms. The exchange of messages ($O(n_{if})$) is needed only when the currently allocated channels to a cell are not sufficient to support its call requests. Therefore, under light traffic load, message exchange is not necessary. However, during the high traffic load, the size of the response messages is larger than that of update algorithms. This is because response messages must carry the information about channels being used, allocated, or transferred in the neighboring cells as opposed to grant or reject in the update algorithms.

CS also suffers from the high acquisition delay under the high traffic load. When all the allocated channels in a cell are occupied, messages ($O(n_{if})$) are exchanged among the interfering support stations to avoid co-channel interference and starvation.

In the 3-Cell cluster system, the failure numbers of PK and CS are $|IN(c)| =$

6. This is because a request message is sent to each interfering neighbor. Simulation results that compare acquisition delays of our algorithm and search algorithms are given in the following section.

Comparison with update algorithms

Here, we give a detailed comparison between the proposed algorithm and Garg et al. In G-Det, a cell has to wait for all its neighbors to select channels in the worst case scenario [GPT96]. G-Det has failure number $|IN(c)|$. The synchronization mechanism used in G-Ran is similar to the one used in QB. Response messages are never deferred in G-Ran. However, the number of request messages used in QB is one-half of that used in G-Ran (in the 3-Cell cluster system). It is less than one-half in the system with arbitrary D_{min}. The failure number of G-Ran is $|IN(c)|$.

Here, the detailed comparison between QB and DL is as follows.

- The request set used in DL depends on the carrier[5] being requested. Therefore, each MSS needs additional memory space to keep a set of cell addresses for each carrier. On the other hand, each cell in our algorithm is required to maintain only one request set. Moreover, each cell in DL needs to know $IN(c)$ for any cell c from which it receives a request message. We only require that each cell can determine if $i \in IN(j)$ and $i \in S_j$ for any cells i, j. Hence, the memory overhead required in their algorithm is higher than that required in ours.

[5] In DL, a carrier is a basic unit of resource allocation , therefore, their 'carrier' is equivalent to 'channel' in our context

- The number of request messages used in DL is generally smaller than ours. The following table shows the size of the request set used in their algorithm and ours for $D_{min} = \sqrt{3}R, 2R, \sqrt{7}R, 3R$.

Algorithm \ D_{min}	$\sqrt{3}R$	$2R$	$\sqrt{7}R$	$3R$
DL	3	4	3	3 or 4
QB	3	3	6	6

  Nevertheless, our request messages in the n-hop request set are only sent to cells at most n hops away in exactly three directions: northwest, southwest, and east. Therefore, a sender could send only three request messages to its neighboring cells in those directions, and let each one of them forward the request message to the next destination. As a result, the routing for our request messages is simple, and the size of message overhead is smaller since only the direction and the number of hops are required.

- The failure number of DL is $|IN(c)|$. This is because each cell c must respond to the request message sent from another cell $j \in IN(c)$ if j is requesting a channel r, the primary channel of c.

- DL algorithm is designed to work with a specific channel reuse pattern. On the other hand, our algorithm can be used with any dynamic or hybrid allocation strategies including the one used in their algorithm.

Livelock

In the generalized algorithm, it is possible that $k \geq 3$ cells in each other's interference neighborhood request for the same available channel, but none of them will succeed. We call this *livelock* since each of these k cells need to make another request attempt even though there is a channel that can be used without interference. Livelock in the generalized algorithm can be illustrated by the following scenario. Let there be

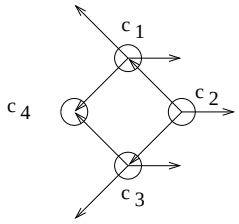


Figure 2.6: An example of livelock in 1-hop request set

three cells c_1, c_2, c_3 in the system with $D_{min} = 2R$ request for the same channel k with timestamps $t_1 < t_2 < t_3$, and these three cells are located in such a pattern depicted in Figure 2.6. If c_1's request arrives at cell c_4 after c_3's request, then c_1 will receive a $C\_GRNT(3, k)$. As a result, c_1's request will be failed because $c_1 \notin S_3$ and $c_3 \notin S_1$. c_2's request also fails since it is rejected by c_1. c_3's request also fails even though it is granted by every member of its request set if the receipt of the last $GRNT$ message occurs between the receipt of REQ and REL messages from c_2.

Note that this livelock scenario will not occur in our 3-cell cluster algorithm as shown in Theorem 6. Similarly, the algorithm proposed by Prakash et al. also suffers from livelock. An example of livelock scenario occurring in Prakash et al.'s algorithm was given in [CS96].

It will be seen from the experimental result given in the following section that the possibility for livelock to occur is extremely low. It will be even lower if a conflict reduction scheme[6] is applied. Therefore, livelock will have a very small impact on the performance of the proposed algorithms.

Starvation

In theory, it could occur in every algorithm except the one proposed by Choy and Singh that some cells may experience consecutive request failures or never be able to acquire a channel to support a call (even though the channel being requested is changed every time the cell makes another request attempt). However, as in livelock,

[6] A scheme that reduces the chance for cells within each other's interference neighborhood to select the same channel to request

the possibility for this to occur is extremely low.

2.7 Enhancement

A common weakness of update-based algorithms is high message complexity due to *acquisition* and *release* messages sent to each cell in the interference neighborhood. Given a cell c, the size of $IN(c)$ grows rapidly as D_{min} increases. For instance, if $D_{min} = 3R$, $|IN(c)| = 30$. We here propose a scheme to further reduce the number of messages required in our algorithms at the expense of higher acquisition delay.

We can reduce message complexity in the generalized algorithm by sending ACQ and REL_TERM messages only to S_c rather than $IN(c)$, and in the 3-cell cluster algorithm by omitting ACQ messages and sending REL_TERM messages only to S_c. It is easy to see that the modified algorithm still guarantee the safety property, that is, no two cells within each other's interference neighborhood will concurrently use the same channel. However, the number of conflicts will be higher since now conflicts are not only caused by the propagation delay of messages. With the proposed scheme, only cells in the request set of a cell c and c itself have knowledge of channels currently used by c. Hence, any cell in $IN(c) - S_c$ can request for a channel currently used by c, and later will be rejected by c. However, this will rarely occur if the traffic load is relatively low and/or the allocation strategy reducing the chance of selecting the same channel by interfering cells is applied. We conduct an experiment to investigate the increase of conflicts caused by this scheme, and the results are shown in Section 2.8.

As a result, the number of messages required per acquisition in the modified 3-cell cluster algorithm becomes $2n_s + 3n_s(m-1) + n_s$. The number of messages required per acquisition in the modified generalized algorithm becomes $2n_s + 3n_s(m-1) + n_s + n_s$.

Since the number of request attempts is likely to be greater than 1 in the

modified algorithm under the heavy traffic and equal to 1 in the original algorithm, it is important to calculate the value of m causing the number of messages required in the modified algorithm exceeds the number of those in the original algorithm with $m = 1$. If we let m^* denote the value of m when the number of messages required per acquisition in the modified algorithm is equal to the number of those in the original algorithm with $m = 1$.

For both 3-cell cluster and generalized algorithms,

$$m^* = \frac{2}{3}\left(\frac{n_{if}}{n_s} - 1\right) + 1$$

Once the number of request attempts in the modified algorithm is greater than m^*, the modified algorithm is outperformed by the original algorithm in both message complexity and acquisition delay. For example, $m^* = 3$ when $D_{min} = 2R$, and $m^* = \frac{5}{3}$ when $D_{min} = \sqrt{3}R$.

2.8 Experiments

2.8.1 Comparison with update algorithms

The system under simulation is a wrapped-around layout with 7×7 cells. There are totally 100 channels shared by all 49 cells. Each cell randomly selects a channel from its available set with the same priority. No special channel selection scheme is performed to prevent cells in each other's interference neighborhood request for the same channel simultaneously. If a request attempt fails, the cell randomly selects another channel.

We assume that the average one-way communication delay between any two MSSs is 2 milliseconds. The duration of a communication session during which a channel is in use is exponentially distributed with mean 120 seconds. Under spatially uniform traffic distribution, the arrival of channel requests to each cell are modelled by a Poisson process with rate λ. Under non-uniform traffic distribution,

the arrival process in each cell can be in either *normal* or *hot* state. This model of non-uniform traffic distribution was also used in [DL97]. The arrival rate in hot state is 5 times higher than normal state. The period of being in hot and normal state are exponentially distributed with mean 180 and 1800 sec., respectively. Handoff calls[7] and new calls are processed with the same priority.

Each run processes 98,000 calls, but data was collected after 49,000 calls had been processed in order to eliminate the impact of startup transients. The number of request attempts (m) for each call is collected, and the average value of m is calculated for each run. The mean value of five such runs, each with a different random number seed, corresponds to a single data in the figures.

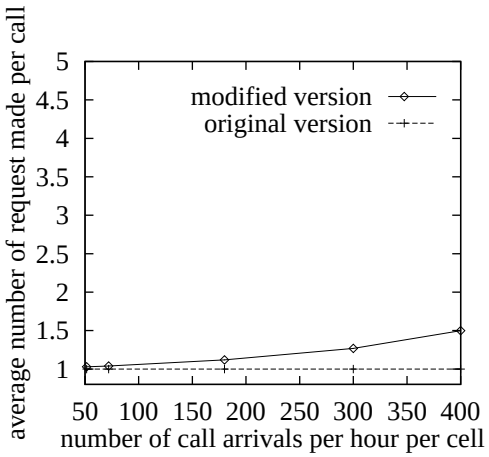
We run our simulation on two systems: $D_{min} = \sqrt{3}R$ with the 3-cell cluster algorithm and $D_{min} = 2R$ with the generalized algorithm. Each system is also run with the corresponding modified version for comparison with its original counterpart.

Figure 2.7 and 2.8 respectively illustrate the average number of request attempts per call under uniform and non-uniform traffic distributions.

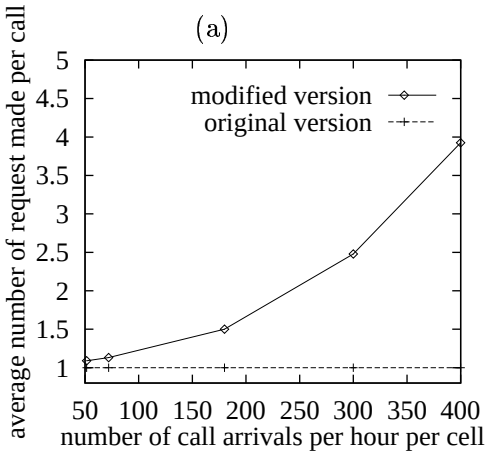
The results demonstrate that the average number of request attempts (m) in the system using the original algorithm under both uniform and non-uniform traffic distributions is very close to 1 (even though it appears in the graphs that it is always 1). For comparison purposes, we could assume that the result from DL algorithm would be similar to the result from our original algorithm. This is because every cell in their algorithm also informs each cell in its interference neighborhood each time it acquires a channel.

Under the modified algorithm, m increases as the traffic load grows; the rate at which m increases in $D_{min} = \sqrt{3}R$ is less than that of $D_{min} = 2R$. Specifically, the average number of request attempts of the system with $D_{min} = \sqrt{3}R$ exceeds the limit $m^* = 1.66$ at the higher channel demand than the system with $D_{min} = 2R$

[7] the ongoing calls which are transferred from one cell to another due to the mobility of MHs

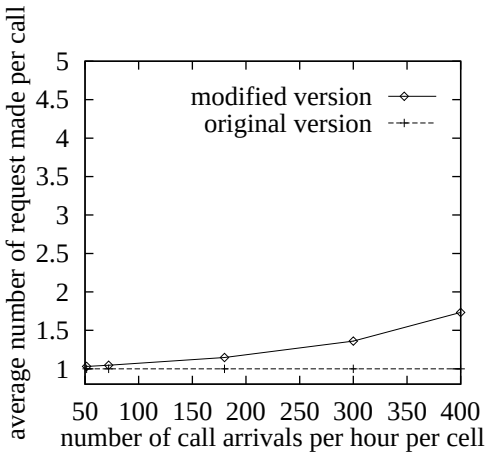


(a)

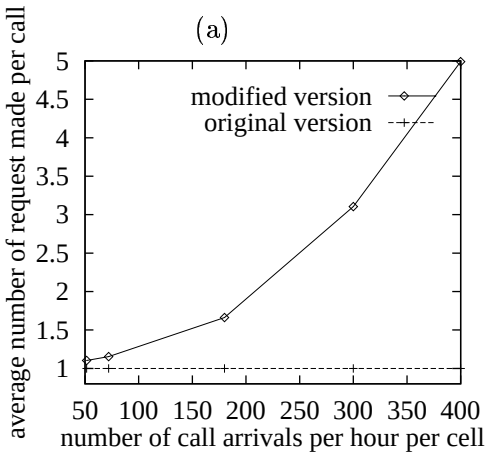


(b)

Figure 2.7: The result for uniform traffic distribution. (a) $D_{min} = \sqrt{3}R$, (b) $D_{min} = 2R$



(a)



(b)

Figure 2.8: The result for non-uniform traffic distribution. (a) $D_{min} = \sqrt{3}R$, (b) $D_{min} = 2R$

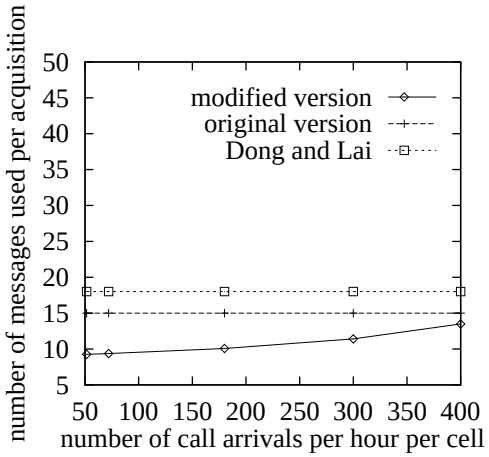
($m^* = 3$). This is because the size of the interference set , $IN(c)$, for each cell c in $D_{min} = \sqrt{3}R$ is smaller than $D_{min} = 2R$. The number of request attempts in the uniform traffic distribution increases at the lower rate than that of non-uniform distribution.

However, this higher acquisition delay can be reduced by allocating to each cell a set of preferred channels as in the hybrid channel allocation strategy. Note that preferred channels can only be used in the cell, and are always selected to support calls whenever available before other channels. Using these preferred channels also reduces message complexity since no request and response messages are required. In addition to preferred channels, we can further reduce conflicts by applying the allocation strategy that prevents interfering cells to select the same channel.

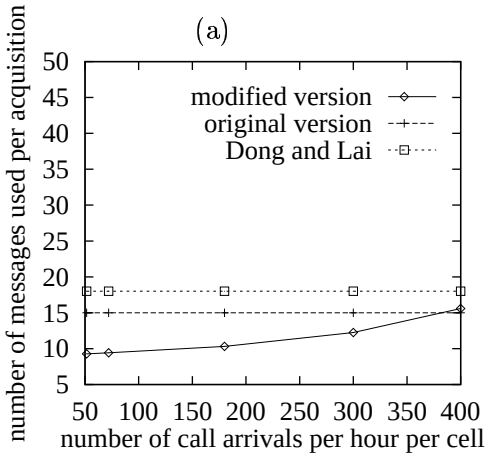
In Figure 2.9 and 2.10, we compare the number of messages used per acquisition in our algorithms with the algorithm proposed by Dong and Lai. Each point in the graphs of our algorithms is calculated from the corresponding data in Figure 2.7 and 2.8. For DL, each point is calculated from the formula presented in [DL97] given that each cell has no preferred channels and the average number of request attempts per call is 1. The results demonstrate that the modified algorithm requires less messages than both the original algorithm and DL. It remains lower until the channel demand reaches the point where the number of request attempts per call in the modified algorithm is equal to m^*.

Recall that the average acquisition delay is $2 * m * D$, where D is the average communication delay between support stations. Therefore, if D is small (which is true in the existing underlying networks), the impact of the slight increase of m on the acquisition delay should be negligible. As a result, message complexity in the update-based algorithms can be reduced at a slight increase in acquisition delay by using our modified algorithm.

Network controllers can employ our modified algorithm to significantly re-

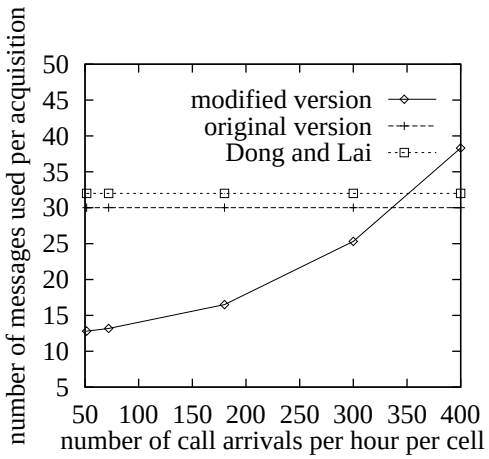


(a)

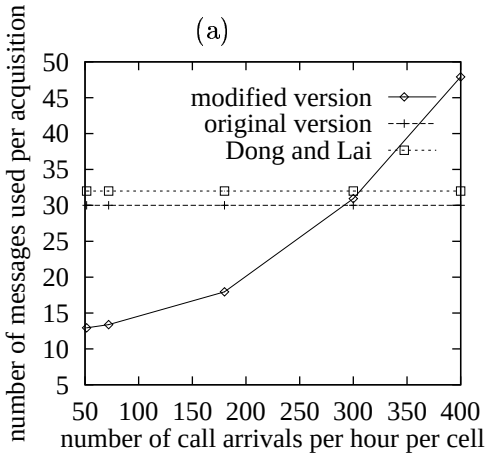


(b)

Figure 2.9: The number of messages required per acquisition for $D_{min} = \sqrt{3}R$. (a) uniform distribution, (b) non-uniform distribution



(a)



(b)

Figure 2.10: The number of messages required per acquisition for $D_{min} = 2R$. (a) uniform distribution, (b) non-uniform distribution

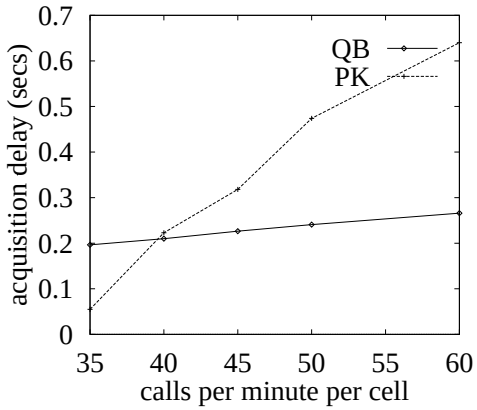
duce traffic load in the underlying wired network at a slight cost of acquisition delay. For example, each cell in the system can initially be set to use the modified algorithm. The heavily loaded cells will be controlled to switch to the original algorithm when the average number of request attempts m in the cell exceeds a predetermined threshold. However, this threshold must be less than m^* given in Section 2.7. The controller can also switch any cell back to its initial mode whenever its number of request attempts falls below the threshold again.

2.8.2 Comparison with search algorithms

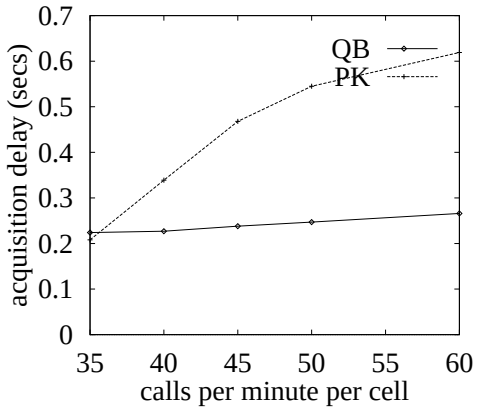
Here, we show the results from the simulation comparing the quorum-based algorithm with search algorithm PK.

The experimental setup used here is as follows: There are totally 450 channels in the system. One-way communication delay is 100 ms. The mean length of each call is 180 seconds. Only the 3-Cell cluster system is considered. In QB, two channels are exclusively allocated to each cell. We adopt the following dropping policies. In QB, calls that require more than 3 request attempts are dropped. In PK, if channel transfer is required, only one transfer attempt is allowed.

The simulations are performed under uniform and non-uniform traffic distribution. Figure 2.11(a) shows the simulation result when the mean arrival rate varies from 35 to 60 calls per minute per cell, and the mean call length is 180 secs. The acquisition delay of QB is rather stable as the traffic load increases. But PK's acquisition delay increases rapidly as load increases. Figure 2.11(b) shows the results of the non-uniform traffic distribution.



(a)



(b)

Figure 2.11: The acquisition delay of QB and PK. (a) uniform load distribution, (b) nonuniform load distribution

Chapter 3

Message Delivery Layer

In this chapter, we present an efficient algorithm implementing causal message ordering in the system with mobile hosts. The material presented here also appears in [SMG99]

3.1 Introduction

Causal message ordering was first proposed by Birman and Joseph [BJ87]. It deals with the notion of maintaining the same *cause and effect* relationship that holds among the send events of messages with the corresponding delivery events. The *cause and effect* relationship is also referred to as *causal dependency*. An event occurring at a process is causally dependent on every preceding event that has occurred at that process. Causal dependencies between events on different processes are established by message communication. Causal dependency can be expressed using Lamport's *happened before* relation [Lam78].

Figure 3.1 shows a violation of causal message ordering in a distributed computation. A computation is called *causally ordered* if it does not show the effect that a message is directly (or indirectly) bypassed by a chain of other messages. In Figure 3.1, m_1 is bypassed by a chain of m_2, m_3, and m_4.

Causal message delivery is required in many distributed applications such

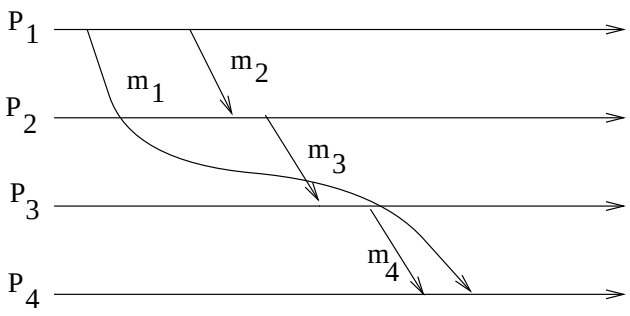


Figure 3.1: An example of the violation of causal message ordering.

as management of replicated data [BJ87, BSS91], distributed monitoring [BM93], resource allocation [RST91], distributed shared memory [AHJ91], and multimedia systems [AS95]. Algorithms to implement causal message ordering in systems with static hosts have been presented in [KS96, BSS91, MR93, RST91, RV95, SES89].

These algorithms, however, have high communication overhead. The message overhead of these algorithms is at least $\Theta(N^2)$ integers, where N is the number of processes in the system. Therefore, they are not scalable for mobile computing systems with a large number of mobile hosts. The scalability problem is exacerbated due to limitations of channel bandwidth, memory and energy supply. We propose an algorithm for causal message ordering in which message overhead does not vary with the number of processes in the system. Moreover, our algorithm efficiently uses limited resources provided on mobile hosts.

The rest of the chapter is organized as follows. Related Work is presented in Section 3.2. Section 3.3 presents the system model and the notation used in the chapter. Sufficient conditions for causal message ordering in mobile computing systems are presented in Section 3.4. We present our protocol in Section 3.5. We compare our protocol with the previous work in Section 3.6. The simulation results are presented in Section 3.7.

3.2 Related Work

While ordering of messages in distributed systems with static hosts has received wide attention, there has been little work on message ordering in mobile systems. Alagar and Venkatesan [AV94] presented an algorithm that enforces causal ordering among mobile support stations to achieve causal ordering among mobile hosts. Therefore, message overhead of this algorithm is reduced to $O(n^2)$, where n is the number of MSSs. However, some messages are delayed since they violate causal ordering in the MSS's point of view, even though they do not violate causal ordering in the mobile host's point of view. They also proposed a scheme to reduce this extra delay. In this scheme every physical MSS is partitioned into k logical MSSs. Each MH, entering a cell of an (physical) MSS, will be allocated to one of the logical MSSs depending on the load in each logical MSS. Since causal ordering is maintained among logical MSSs, messages to two MHs that belong to different logical MSSs will not have to wait for each other even though both MHs are in the same cell. As a result, message overhead for this scheme is increased to $O(k^2 * n^2)$.

Prakash, Raynal, and Singhal [PRS96] presented an algorithm to implement causal message ordering in which each message carries information only about its direct predecessors with respect to each destination process. Message overhead in their algorithm is relatively low; however, in the worst case, it can still be $O(N^2)$ where N is the number of mobile hosts in the system. Furthermore, the size of their message overhead varies when the number of participating processes dynamically changes.

In this chapter, we present an algorithm using the concept introduced by Alagar and Venkatesan. However, we lower the extra delivery delay, and at the same time maintain the size of message overheads at $O(n^2)$. The proposed algorithm is also well-suited for dynamic systems since the size of message overhead is constant, even when the number of participating mobile hosts varies. Our handoff algorithm

(the procedure executed when an MH changes cell) allows the MSS to deliver messages that it receives for a new MH sooner than Alagar's handoff algorithm. In their algorithm, the MSS is not allowed to deliver these messages until it receives from the previous MSS a control signal indicating that the handoff algorithm has been complete.

3.3 Notation

Each *process* (MH or MSS) in a computation generates an execution trace, which is a finite sequence of local *states* and *events*. A state corresponds to the values of all the variables and the program counter in the process. An event on a process can be classified into three types: *send* event (corresponds to send of a message by a process) , *receive* event (corresponds to arrival of a message at a process), and *local* event (which is not a send or a receive event). A *delivery* event is a local event that represents the delivery of a received message to the application or applications running on that process.

We use $\mathcal{H} = \{h_1, h_2, \ldots, h_{n_h}\}$ to denote a set of mobile hosts, and $\mathcal{S} = \{S_1, S_2, \ldots, S_{n_s}\}$ to denote a set of mobile support stations. In practice, $n_h \gg n_s$. Also, let $\mathcal{H}_i$ denote the set of MHs in the cell of MSS S_i. A mobile computation can be illustrated using a graphical representation referred to as *concrete diagram*. Figure 3.2 illustrates such a diagram where the horizontal lines represent MH and MSS processes, with time progressing from left to right. h_1 is in the cell of S_1. h_2 and h_3 are in the cell of S_2. A solid arrow represents a message exchanged between an MH and an MSS process. A dashed arrow represents a message sent from an MSS process to another MSS process. Filled circles at the base and the head of an arrow represent send and receive events of that message. A concrete diagram in which only MH processes are shown is referred to as an *abstract diagram*.

We denote the sequence of MSSs that an MH h_l visits by $\{S_k^l\}_{0 \leq k \leq n(h_l)}$,

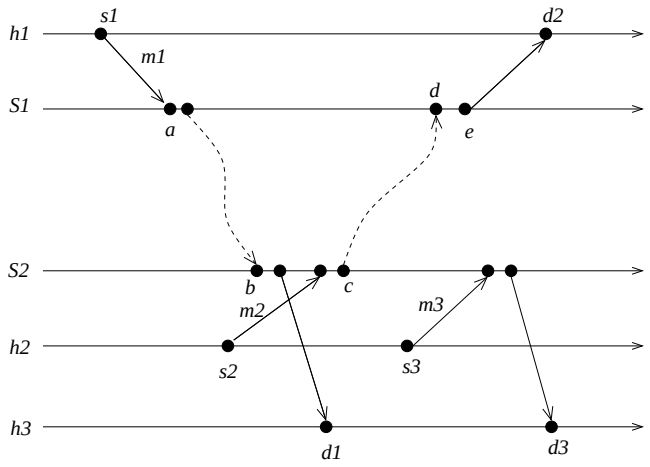


Figure 3.2: A concrete diagram of a mobile computation.

where $n(h_l)$ is the number of times h_l switches cell in a computation. Using this notation, S_0^l and $S_{n(h_l)}^l$ represent the *initial* and the *final* MSSs for h_l. Note that an MH can visit an MSS more than once. For a message m, let $m.src$ and $m.dst$ denote the source and destination processes. Moreover, $m.snd$, $m.rcv$ and $m.dlv$ denote the send event on the source process and the receive and the deliver events on the destination process respectively. We assume that a message sent to itself is immediately received by the sending process.

An *application* message is a message sent by an MH intended for another MH. Since MHs do not communicate with each other directly, an MH, say h_s, first sends an application message m to its MSS, say S_i, which then forwards m to the MSS, S_j, of the destination host, h_d. Using our notation, $m.src$ and $m.dst$ denote the source and the destination hosts respectively of m. In other words, $m.src = h_s$ and $m.dst = h_d$. Furthermore, $m.snd$ denotes the send event of m on h_s. Also, $m.rcv$ and $m.dlv$ denote the receive and delivery events respectively of m on h_d.

Let $\hat{m}$ denote the message which S_i sends to S_j (containing the application message m along with additional information for ensuring causality), requesting it to deliver m to h_d. Again using our notation, $\hat{m}.src$ denotes the MSS of h_s when m

was sent (in this case S_i). Similarly, $\hat{m}.dst$ denotes the MSS to which S_i forwards m (in this case S_j). As before, $\hat{m}.snd$ denotes the send event of $\hat{m}$ on the support station S_i. Similarly, $\hat{m}.rcv$ and $\hat{m}.dlv$ (when m becomes deliverable at S_j) denote the receive and delivery events respectively of $\hat{m}$ on S_j. Figure 3.3 illustrates our notation.

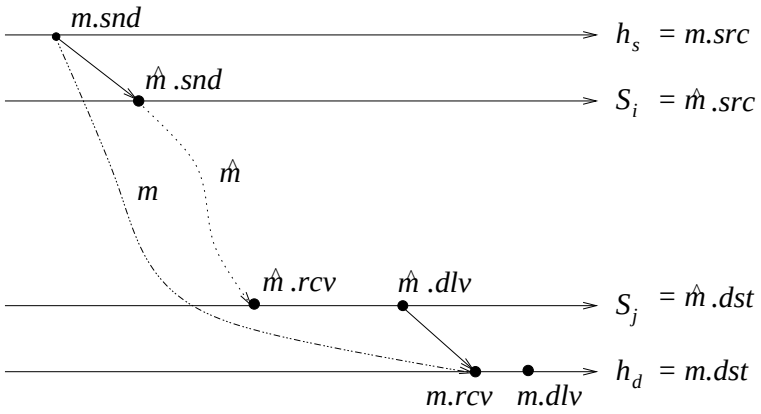


Figure 3.3: A figure illustrating the notation used in the paper.

An event e locally occurred before an event f in mobile host's view, denoted by $e \prec_h f$, iff e occurred before f in real-time on some mobile host. Similarly, an event e locally occurred before an event f in mobile support station's view, denoted by $e \prec_s f$, iff e occurred before f in real-time on some mobile support station. Let $\rightarrow_h$ and $\rightarrow_s$ denote the Lamport's happened before relation [Lam78] in abstract (on events on MHs) and concrete diagram (on events on MSSs) respectively. A mobile computation is causally ordered iff the following property is satisfied for any pair of application messages, m_i and m_j, in the system,

$$(CO) \qquad m_i.snd \rightarrow_h m_j.snd \;\Rightarrow\; \neg(m_j.dlv \prec_h m_i.dlv)$$

For convenience, $m_i \rightarrow_h m_j \stackrel{def}{=} m_i.snd \rightarrow_h m_j.snd$.

3.4 Sufficient Conditions

We next give the sufficient conditions for causally ordered message delivery in a mobile computation with static hosts.

Theorem 13 *: A mobile computation is causally ordered if*

(C_1) all wireless channels are FIFO,

(C_2) messages in the wired network are causally ordered, and

(C_3) each MSS sends out messages in the order they are received.

Proof: The condition C_2 can be formally expressed as,

$$(CO') \qquad \hat{m}_i.snd \rightarrow_s \hat{m}_j.snd \Rightarrow \neg(\hat{m}_j.dlv \prec_s \hat{m}_i.dlv)$$

We first prove that $m_i.snd \rightarrow_h m_j.snd \Rightarrow \hat{m}_i.snd \rightarrow_s \hat{m}_j.snd$. Let $\rightsquigarrow_h$ and $\rightsquigarrow_s$ relate the send and delivery events of the same message in abstract and concrete views respectively. Observe that due to C_1 and C_3, $m_i.snd \prec_h m_j.snd \Rightarrow \hat{m}_i.snd \prec_s \hat{m}_j.snd$. Moreover, since MHs communicate through MSSs therefore $m_i.snd \rightsquigarrow_h m_i.dlv \Rightarrow \hat{m}_i.snd \rightsquigarrow_s \hat{m}_i.dlv$, and $m_i.dlv \prec_h m_j.snd \Rightarrow \hat{m}_i.dlv \prec_s \hat{m}_j.snd$. Using induction on the definition of $\rightarrow_h$, it can be easily proved that $m_i.snd \rightarrow_h m_j.snd \Rightarrow \hat{m}_i.snd \rightarrow_s \hat{m}_j.snd$ (any causal chain from $m_i.snd$ to $m_j.snd$ is a combination of the three cases). Informally, if there is a causal path from m_i to m_j in the abstract diagram then there is a causal path from $\hat{m}_i$ to $\hat{m}_j$ in the concrete diagram.

Again, due to C_1, we have $m_j.dlv \prec_h m_i.dlv \Rightarrow \hat{m}_j.dlv \prec_s \hat{m}_i.dlv$. Using contrapositive, we get $\neg(\hat{m}_j.dlv \prec_s \hat{m}_i.dlv) \Rightarrow \neg(m_j.dlv \prec_h m_i.dlv)$. Thus, $m_i.snd \rightarrow_h m_j.snd \Rightarrow \hat{m}_i.snd \rightarrow_s \hat{m}_j.snd \Rightarrow \neg(\hat{m}_j.dlv \prec_s \hat{m}_i.dlv) \Rightarrow \neg(m_j.dlv \prec_h m_i.dlv)$. In other words, assuming C_1 and C_3, $CO' \Rightarrow CO$. ∎

Sufficient conditions given in Theorem 13 were implicitly used in [AV97]. For systems with static hosts, Theorem 13 gives a lightweight protocol for causal

message ordering. In the extreme case when the entire computation is in a single cell, causal ordering can be provided by simply using FIFO channels between MHs and their MSSs.

We now show that C_1, C_2, and C_3 are not necessary by a counter-example. In Figure 3.4, $s_1 \rightarrow_h s_3$ and $d_1 \prec_h d_3$. Therefore the computation in Figure 3.4 is causally ordered, although C_1 and C_2 do not hold.

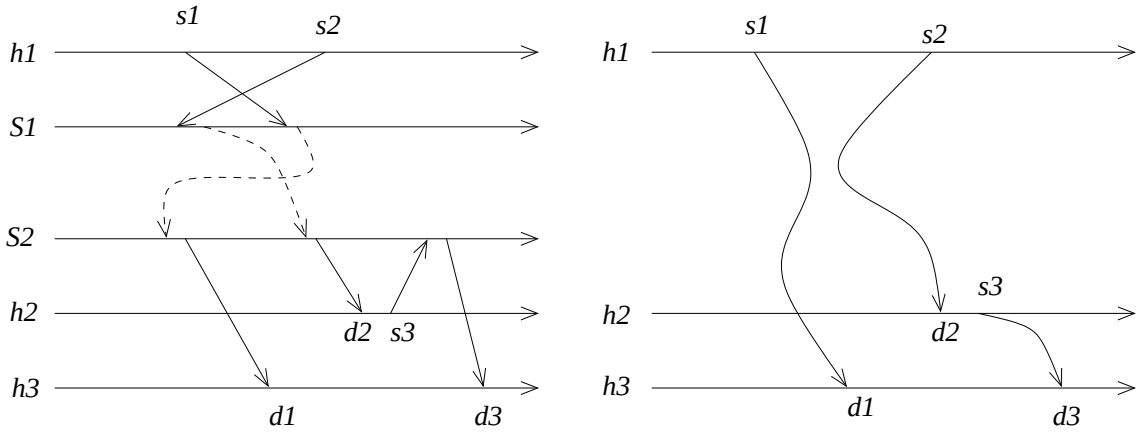


Figure 3.4: An example to show that C_1, C_2, and C_3 are not necessary for CO.

The algorithms presented by Alagar and Venkatesan ($AV2$ and $AV3$) [AV97] enforce CO' in order to achieve CO. Their algorithms delay messages that violate CO' even though they do not violate CO. This can be illustrated in a computation in Figure 3.5. In this example, message m_1 does not causally precede m_3 in the abstract view, but it does in the concrete view. Under CO', m_3 is unnecessarily delayed until m_1 is deliverable. Our goal is to reduce this unnecessary delay, while maintaining the message overhead in the wired network close to $O(n_s^2)$.

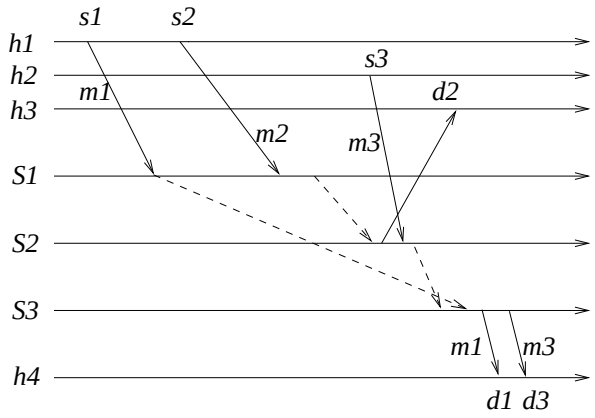


Figure 3.5: Unnecessary delay in $AV2$.

3.5 Algorithm

$AV2$ uses a single matrix for all MHs in a cell. This can create false causal dependencies between messages. In order to reduce these false causal dependencies and hence the unnecessary delay, we propose to use a separate matrix for each MH in a cell. The next two subsections describe the static and the handoff modules of our protocol. The static module is executed when an MH is in a particular cell. The handoff module is executed when an MH moves from one cell to another. We prove the correctness of both the modules in Section 3.5.3. Section 3.5.4 presents the condition characterizing the static module.

3.5.1 Static Module

For convenience, we first describe the static module assuming static hosts. In the next subsection, we describe the handoff module and the modifications that need to be made to the static module to incorporate mobile hosts.

Our static module is based on the algorithm proposed by Raynal *et al* [RST91]. For simple exposition of the protocol, we assume that the channels among the MSSs are FIFO. This assumption can be easily relaxed by implementing FIFO

S_i ::

var
$rcvQ$: queue of messages, initially ϕ;
$cell$: array [1..n_h] of 2-tuples $\langle mbl, mss \rangle$, initially $[\langle 0, S_0^k \rangle]_{1 \le k \le n_h}$;
$lastsent$, $lastrcvd$: array[1..n_s] of integers, initially $\mathbf{0}$;
M : set of matrices ($n_s \times n_s$), ($\{M_k \mid h_k \in \mathcal{H}_i\}$), each initially $\mathbf{0}$;
$ackQ$: set of FIFO queues of messages, ($\{ackQ_k \mid h_k \in \mathcal{H}_i\}$), each initially ϕ;
$sndQ$: set of FIFO queues of messages, ($\{sndQ_k \mid h_k \in \mathcal{H}_i\}$), each initially ϕ;
$canSend$: set of boolean variables, ($\{canSend_k \mid h_k \in \mathcal{H}_i\}$), each initially **true**;
$canDeliver$: set of boolean variables, ($\{canDeliver_k \mid h_k \in \mathcal{H}_i\}$), each initially **true**;

(A1) On receiving a data message m from h_s;
send an acknowledgement to h_s;
put m in $sndQ_s$;
call $process\_sndQ(h_s)$;

(A2) On calling $process\_sndQ(h_s)$;
if ($canSend_s$) then
while ($sndQ \neq \phi$) do
remove m from the head of $sndQ_s$;
let m be destined for h_d and S_j be $cell[d].mss$;
$lastsent[j]++$;
send $\langle m, M_s, lastsent[j] \rangle$ to S_j;
$M_s[i,j] := lastsent[j]$;
endwhile;
endif;

(A3) On receiving $\langle m, M, seqno \rangle$ from S_j;
$lastrcvd[j] := seqno$;
put $\langle m, M, seqno \rangle$ in $rcvQ$;
call $process\_rcvQ()$;

(A4) On calling $process\_rcvQ$;
repeat
forall $\langle m, M, seqno \rangle \in rcvQ$ do
let m be destined for h_d;
if ($canDeliver_d \wedge \langle \forall k :: lastrcvd[k] \geq M[k,i] \rangle \wedge$
$\langle \nexists \langle m', M', seqno' \rangle \in rcvQ :: (S_k$ sent m' for $h_d) \wedge$
$(seqno' \leq M[k,i]) \rangle$) then
remove $\langle m, M, seqno \rangle$ from $rcvQ$;
call $deliver(\langle m, M, seqno \rangle)$;
endif;
endforall;
until ($rcvQ = \phi$) $\vee$ (no more messages can be delivered);

(A5) On calling $deliver(\langle m, M, seqno \rangle)$;
let m be destined for h_d;
put $\langle m, M, seqno \rangle$ in $ackQ_d$;
send m to h_d;

(A6) On receiving an acknowledgement from h_d;
remove $\langle m, M, seqno \rangle$ from the head of $ackQ_d$ and let S_j sent m;
$M_d[j,i] := \max\{M_d[j,i], seqno\}$;
$M_d := \max\{M_d, M\}$;

Figure 3.6: The static module for a mobile support station S_i

among MSSs using sequence numbers. We also assume that every MSS knows about the location of the MHs. For each MH h_l, we maintain an $n_s \times n_s$ matrix M_l. $M_l[i,j]$ denotes the total number of messages h_l knows to have been sent by S_i to S_j. Assume that h_l is in the cell of S_i. In order to reduce the communication and computation overhead of h_l, the matrix M_l is stored at S_i. In addition, each S_i also maintains two arrays $lastsent_i$ and $lastrcvd_i$ of size n_s. The j^{th} entry of $lastsent_i$, $lastsent_i[j]$, denotes the number of messages sent by S_i to S_j. Similarly, the j^{th} entry of $lastrcvd_i$, $lastrcvd_i[j]$, denotes the number of messages sent by S_j that have been received at S_i.

Initially, all the entries in the matrices M_l, and arrays $lastsent_i$ and $lastrcvd_i$ are set to 0. To send a message m to another MH h_d, h_s first sends the message to its MSS S_i. Assume that h_d is in the cell of S_j. S_i increments $lastsent_i[j]$ by one and then sends $\langle m, M_s, lastsent_i[j] \rangle$, denoted by $\hat{m}$, to S_j. After that, S_i sets $M_s[i,j]$ to $lastsent_i[j]$.

S_j on receiving $\langle m, M, seqno \rangle$ from S_i meant for h_d first checks whether m is deliverable. m is *deliverable* if S_j has received all the messages on which m causally depends ($lastrcvd_j[k] \geq M[k,j]$ for all k), and there is no message destined for h_d on which m causally depends which is yet to be delivered to h_d ($\nexists \langle m', M', seqno' \rangle$ destined for h_d sent by S_k yet to be delivered such that $seqno' \leq M[k,j]$). If so, S_j transmits m to h_d. If m is not currently deliverable, it is kept in $rcvQ_j$ until it becomes deliverable. Like *YHH*, we do not update M_d immediately after delivering m to h_d, but we store m in $ackQ_d$. When h_d receives m, it sends back an acknowledge message, denoted by $ack(m)$, to S_j. On receiving $ack(m)$, S_j sets $M_d[i,j]$ to the maximum of its original value and $seqno$ (piggybacked on m). Then it sets each element in M_d to the maximum of its original value and the value of the corresponding element in M (also piggybacked on m). This prevents any outgoing message from h_d to become causally dependent on m that is sent before m is received

by h_d. Figure 3.6 gives a more detailed description of the static module.

3.5.2 Handoff Module

In order to ensure causally ordered message delivery, some steps have to taken during handoff after an MH moves from one cell to another. We now describe the handoff module. Each MH h_l maintains a *mobility number*, mbl_l, which is initially set to 0. It is incremented every time a mobile host moves. Intuitively, mbl_l denotes the number of times h_l has changed cell. In addition, every MSS maintains an array of 2-tuples, denoted by *cell*, with an entry for each MH. The l^{th} entry of $cell_i$, $cell_i[l]$, is a 2-tuple $\langle mbl, mss \rangle$, where the value of $cell_i[l].mss$ represents S_i's knowledge of the location of h_l and the value of $cell_i[l].mbl$ indicates how "current" the knowledge is.

Consider a scenario when an MH h_l moves from the cell of S_i to the cell of S_j. After switching cell, h_l increments mbl_l and sends $register(mbl_l, S_i)$ message to S_j to inform S_j of its presence. Also, h_l retransmits the messages to S_j for which it did not receive the acknowledge message from its previous MSS S_i. On receiving *register* message from h_l, S_j updates $cell_j[l]$ (its local knowledge about the location of h_l) and sends $handoff\_begin(h_l, mbl_l)$ message to S_i. The MSS S_i, on receiving $handoff\_begin(h_l, mbl_l)$ message, updates $cell_i[l]$ and sends $enable(h_l, M_l, ackQ_k)$ message to S_j. It then broadcasts $notify(h_l, mbl_l, S_j)$ message to all MSSs (except S_i and S_j), and waits for $last(h_l)$ message from all the MSSs to which it sent *notify* message. Meanwhile, if any message received by S_i for h_l becomes deliverable, S_i marks it as "old" and forwards it to S_j.

On receiving $enable(h_l, M_l, ackQ_l)$ message from S_i, S_j first delivers all the messages in $ackQ_l$. It also updates M_l assuming all the messages in $ackQ_l$ have been received at h_l. Then S_j starts sending the application messages on behalf of h_l. S_j also delivers all the messages for h_l that are marked "old" in the order in

which the messages arrived. However, messages destined for h_l that are not marked "old" are queued in $rcvQ_j$.

An MSS S_k, on receiving $notify(h_l, mbl_l, S_l)$ message, updates $cell_k[l]$ and then sends $last(h_l)$ message to S_i. Observe that since the channels among all the MSSs are assumed to be FIFO, after S_i receives $last(h_l)$ message from S_k there are no messages in transition destined for h_l that are sent by S_k to S_i. On receiving $last(h_l)$ message from all the MSSs (to which *notify* message was sent), S_i sends $handoff\_over(h_l)$ message to S_j. The handoff terminates at S_j after S_j receives $handoff\_over(h_l)$ message. S_j can now start delivering messages to h_l. Meanwhile, if S_j receives $handoff\_begin(h_l)$ message from some other MSS before the current handoff terminates, S_j responds to the message only after the handoff terminates.

Since we do not assume that the messages in the wired network are causally ordered, it is possible that a message m destined for h_l is sent to S_i (the old MSS of h_l), whereas its causally preceding message m', also destined for h_l, is sent to S_j (the new MSS of h_l). In order to prevent this, an MSS piggybacks additional information on all the message that contain application messages: messages destined for an MH (may or may not be marked as "old") and *enable* messages. On these messages, an MSS piggybacks its local knowledge of the location of all the mobile hosts that have changed their cells since it last communicated with the other MSS. On receiving this information, the other MSS updates its knowledge of the location of the MHs (its *cell*) based on their mobility number. In the worst case, this extra overhead could be as large as $O(n_h)$. In practice, we expect it to be much smaller. Let t_{snd} denote the mean inter-message generation time and t_{mov} be the mean inter-switch time for a MH. Then, the average extra overhead for uniform communication pattern (every MH has equal probability of sending a message to every other MH) is $\approx O(\frac{t_{snd}}{t_{mov}} n_s^2)$.

Our handoff module is more efficient than the handoff module in $AV2$ and $AV3$ since we do not require the messages exchanged among the MSSs to be causally

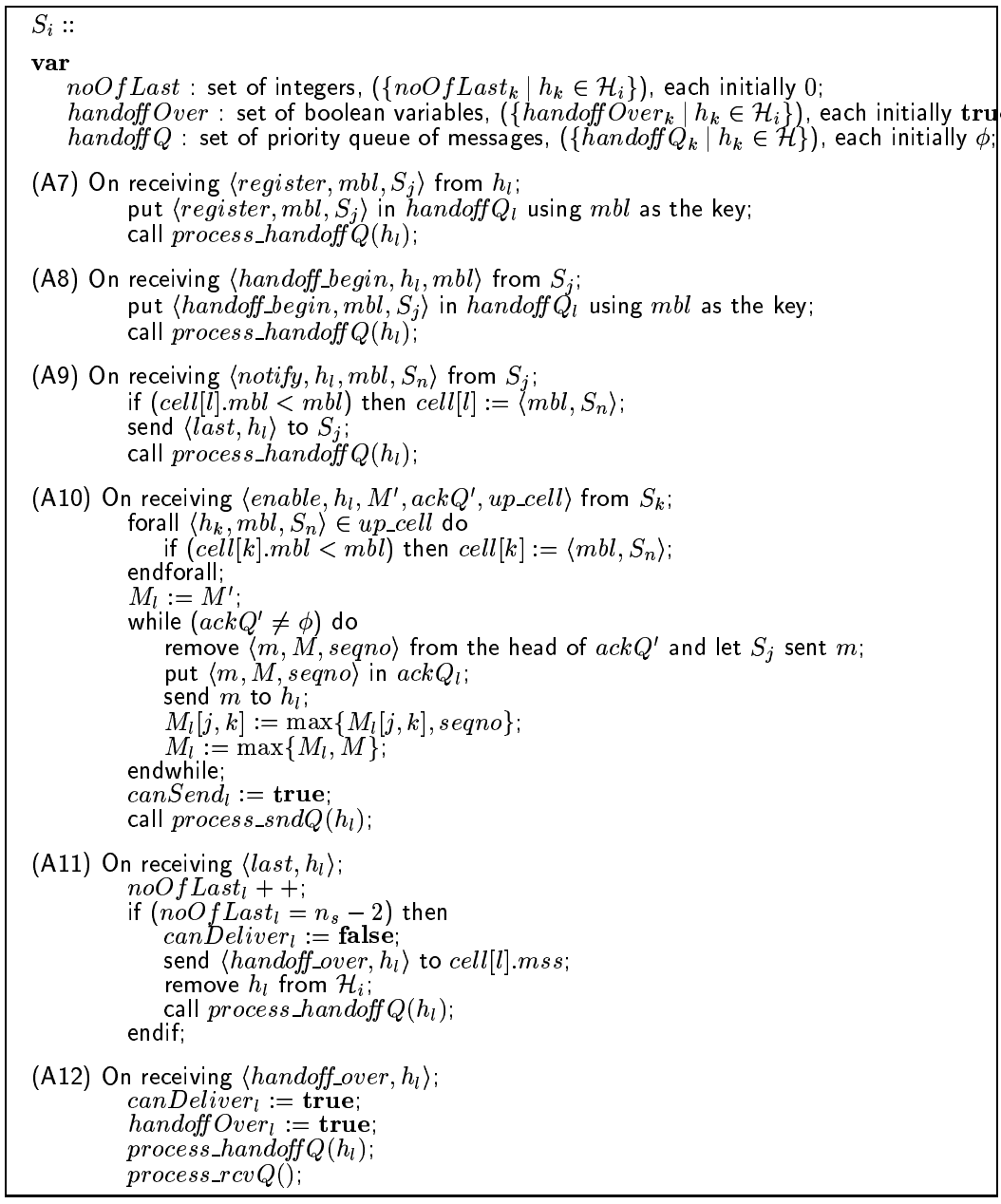


Figure 3.7: The handoff module for a mobile support station S_i

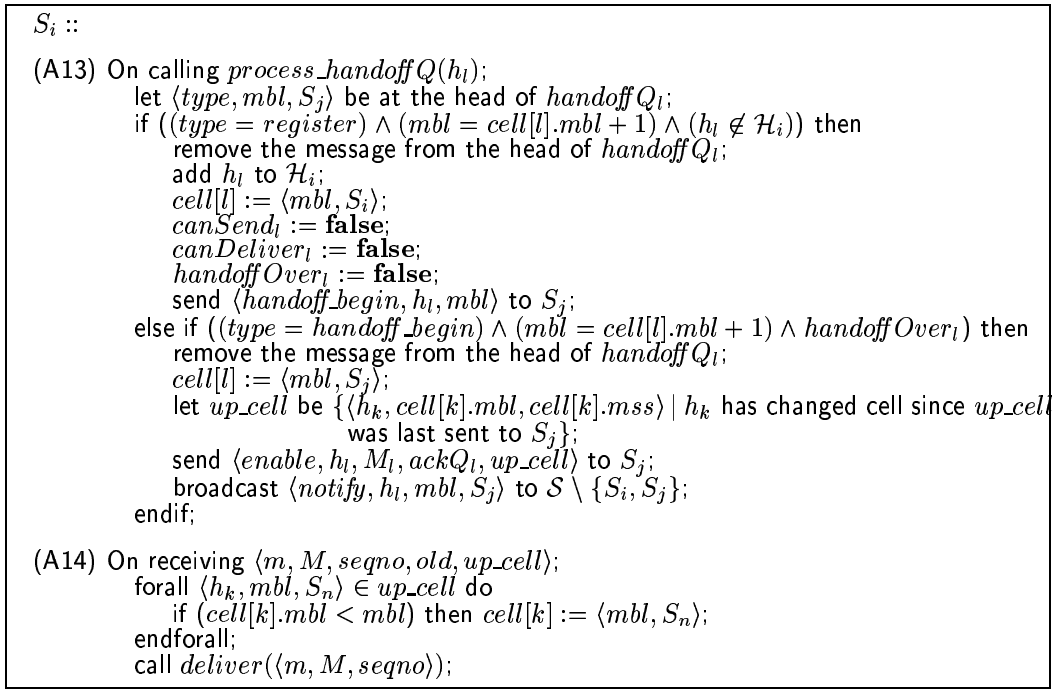


Figure 3.8: The handoff module for a mobile support station S_i (contd.)

ordered. Figure 3.7 and Figure 3.8 give a more detailed description of the handoff module. Figure 3.9 gives the modifications in the static module to incorporate mobile hosts.

Although we do not mention here but the mobility number has several usages. For instance, the messages exchanged between a MH and its MSS can also be tagged with the mobility number of the MH. It can then be used by an MSS to ignore messages received from an MH after it has sent *enable* message for that MH to the new MSS. It can also be used to correctly serialize the handoff procedures for an MH.

3.5.3 Proof of Correctness

We assume that a message sent to itself is immediately received by the sending process. Also, *enable*, *notify*, *last* and *handoff_over* messages are delivered as soon

```
S_i ::

(A2') On calling process_sndQ(h_s);
        if (canSend_s) then
            while (sndQ ≠ φ) do
                remove m from the head of sndQ_s;
                let m be destined for h_d and S_j be cell[d]mss;
                lastsent[j]++;
                let up_cell be {⟨h_k, cell[k].mbl, cell[k].mss⟩ | h_k has changed cell since
                                    up_cell was last sent to S_j};
                send ⟨m, M_s, lastsent[j], up_cell⟩ to S_j;
                M_s[i, j] := lastsent[j];
            endwhile;
        endif;

(A3') On receiving ⟨m, M, seqno, up_cell⟩ from S_j;
        forall ⟨h_k, mbl, S_n⟩ ∈ up_cell do
            if (cell[k].mbl < mbl) then cell[k] := ⟨mbl, S_n⟩;
        endforall;
        lastrcvd[j] := seqno;
        put ⟨m, M, seqno⟩ in rcvQ;
        call process_rcvQ();

(A5') On calling deliver(⟨m, M, seqno⟩);
        let m be destined for h_d;
        if (cell[d].mss = S_i) then
            put ⟨m, M, seqno⟩ in ackQ_d;
            send m to h_d;
        else
            let up_cell be {⟨h_k, cell[k].mbl, cell[k].mss⟩ | h_k has changed cell since up_cell
                                was last sent to cell[d].mss};
            send ⟨m, M, seqno, old, up_cell⟩ to cell[d].mss;
        endif;

(A6') On receiving an acknowledgement from h_d;
        remove ⟨m, M, seqno⟩ from the head of ackQ_d and let S_j sent m to S_k;
        M_d[j, k] := max{M_d[j, k], seqno};
        M_d := max{M_d, M};
```

Figure 3.9: The modification in static module in presence of host movement in mobile support station S_i

as they are received. The *register* and *handoff_begin* messages are delivered once the corresponding "if" condition is satisfied in (A13). Since the MSS does all the processing therefore for an application message m, $m.rcv = m.dlv$.

Note that since MHs are mobile and can change their cell, an application message m can be delivered to an MH by an MSS other than $\hat{m}.dst$ (either when m is received as on "old" message or m is in the acknowledgement queue of an *enable* message). In fact, m can be received multiple times by its destination MH. We consider m to be received (delivered) when the destination MH receives (delivers) it for the first time. Moreover, an application message can be sent multiple times (due to retransmission by the mobile host on failure to receive acknowledgement). We treat the retransmitted application message as a different application message and ignore the application message sent by an MH that is lost when the MH switched its cell. Here we assume that an MH can detect duplicate application messages and discard them. In our protocol, apart from the application message m, $\hat{m}$ also contains a matrix, denoted by $\hat{m}.M$, and a sequence number, denoted by $\hat{m}.seqno$. For convenience, $m.M = \hat{m}.M$ and $m.seqno = \hat{m}.seqno$. A matrix M_i is less than or equal to M_j, denote by $M_i \preceq M_j$, iff $\langle \forall\, k, l :: M_i[k,l] \leq M_j[k,l] \rangle$.

Let e be an event on an MSS S_i. We use $mbl(e)$ and $lastrcvd(e)$ to denote the value of the vectors $cell[1:n_h].mbl$ and $lastrcvd$ respectively at S_i on occurrence of e. The k^{th} entry of the vector v is denoted by $v.k$. A vector v_i is less than or equal to a vector v_j, denoted by $v_i \preceq v_j$, iff $\langle \forall\, k :: v_i.k \leq v_j.k \rangle$. We use the same operator $\preceq$ to compare the vectors and the matrices. Intuitively, the value $cell[k].mss$ at S_i represents S_i's knowledge of the location of h_k and $cell[k].mbl$ indicates how "recent" the knowledge is. For a message m sent by an MSS, $mbl(m) = mbl(m.snd)$. For an application message m, $mbl(m) = mbl(\hat{m})$. Since for all k, $cell[k].mbl$ is monotonically non-decreasing for every MSS therefore $e \prec_s f \;\Rightarrow\; mbl(e) \preceq mbl(f)$.

The following lemmas and theorems prove the correctness of both the static

and the handoff modules. The organization of the proof is as follows. We first prove Lemma 14-16 that are used in the proof of liveness and safety properties. Theorem 21 establishes the *liveness* property of the protocol, namely a message sent to a mobile host is eventually delivered. We prove the liveness property in two stages. We first prove that a message $\hat{m}$, carrying the application message m, is eventually delivered at its destination MSS (when m becomes deliverable at $\hat{m}.dst$ or $deliver(\hat{m})$ is called at $\hat{m}.dst$), $\hat{m}.dst$ (Lemma 20). Using it, we prove that the application message m is eventually delivered at its destination MH, $m.dst$. Theorem 25 establishes the *safety* property of the protocol, namely the modules implement causal ordering among mobile hosts.

In the lemmas that follow, let m_i and m_j be arbitrary application messages. Let $m_i.src = h_s$ and $m_i.dst = h_d$, and $m_j.src = h_{s'}$ and $m_j.dst = h_{d'}$. Let $\hat{m}_i.src = S_u$ and $\hat{m}_i.dst = S_v$, and $\hat{m}_i.src = S_{u'}$ and $\hat{m}_j.dst = S_{v'}$. For convenience, let $mbl(m_i).d = r$ and $mbl(m_j).d' = r'$. Note that $S_r^d = \hat{m}_i.dst = S_v$ and $S_{r'}^{d'} = \hat{m}_j.dst = S_{v'}$. Although, both S_v and S_r^d represent identical MSS $\hat{m}_i.dst$), we use S_r^d when we want to assert that S_v is the $r+1^{th}$ MSS of h_d and argue about the properties that hold during that time period. This usage is not limited to m_i.

3.5.3.1 Preliminary Lemmas

Let $\twoheadrightarrow_s$ denote the Lamport's happened before relation in the concrete view with respect to the messages on which *up_cell* is piggybacked. Observe that $\twoheadrightarrow_s \subseteq \rightarrow_s$. Let $enable(S_p^l)$ denote the enable message sent by S_p^l to S_{p+1}^l on processing *handoff_begin* message from S_{p+1}^l in the handoff module for h_l when h_l moves from the cell of S_p^l to the cell of S_{p+1}^l. Also, let $enable(S_p^l).M$ denote the matrix, M_l, piggybacked on the *enable* message. Since S_p^l does not process the *handoff_begin* message from S_{p+1}^l until it receives the *handoff_over* message from S_{p-1}^l and the protocol piggybacks *up_cell* on an *enable* message therefore,

(P 3.5.1) $$p < q \;\Rightarrow\; enable(S_p^l).snd \twoheadrightarrow_s enable(S_q^l).snd$$

Also, since M_l is monotonically non-decreasing, we have

(P 3.5.2) $$p \leq q \;\Rightarrow\; enable(S_p^l).M \preceq enable(S_q^l).M$$

Lemma 14 $m_i.snd \rightarrow_h m_j.snd \;\Rightarrow\; m_i.seqno \leq m_j.M[u,v]$

Proof: The proof is by induction on the number of messages, n, involved in the smallest causal chain (with respect to $\rightarrow_h$) from $m_i.snd$ to $m_j.snd$. Let $\hat{m}_i.src = S_a^s$ and $\hat{m}_j.src = S_{a'}^{s'}$, where $a = mbl(m_i).s$ and $a' = mbl(m_j).s'$. Note that $S_u = S_a^s$ and $S_{u'} = S_{a'}^{s'}$.

Base Case ($n = 0$): In this case, $m_i.snd \prec_h m_j.snd$. Observe that $h_s = h_{s'}$ and $a' \geq a$. There are two cases to consider depending on whether h_s switched its cell after sending m_i.

Case 1 [$a' = a$]: It can be verified from the protocol that as soon as $\hat{m}_i$ is sent, M_s is updated (A2'). Using monotonicity of M_s, we have $m_i.seqno \leq m_j.M[u,v]$.

Case 2 [$a' > a$]: Since S_a^s does not send forward any message on behalf of h_s to any MSS after sending the *enable* message, therefore $\hat{m}_i.snd \prec_s enable(S_a^s).snd$. Also, $S_{a'}^s$ does not forward any message on behalf of h_s until it receives $enable(S_{a'-1}^s)$, therefore $enable(S_{a'-1}^s).dlv \prec_s \hat{m}_j.snd$. Using P 3.5.2, monotonicity of M_s and $a' - 1 \geq a$, we get,

$$m_i.seqno \leq enable(S_a^s).M[u,v] \leq enable(S_{a'-1}^s).M[u,v] \leq m_j.M[u,v]$$

Thus, in any case, $m_i.seqno \leq m_j.M[u,v]$.

Induction Step ($n \geq 1$): Let m_k denote the last message in the smallest causal chain (with respect to $\rightarrow_s$) from $m_i.snd$ to $m_j.snd$. Then, by induction $m_i.seqno \leq m_k.M[u,v]$. Let $S_b^{s'}$ be the MSS which first delivered m_k to $h_{s'}$. Observe that $a' \geq b$. Let $ack(m_k)$ denote the acknowledgement message sent by $h_{s'}$ on receiving m_k. There are two cases to consider depending on whether $h_{s'}$ switched its cell after receiving (or delivering) m_k.

Case 1 [$a' = b$]: Since $m_k.dlv \prec_h m_i.snd$, therefore $ack(m_k)$ is received at $S_{a'}^{s'}$ before m_i. Moreover, when $ack(m_k)$ is received, m_k is at the head of $ackQ_{s'}$ (channel between an MH and its MSS is reliable and FIFO). On receiving $ack(m_k)$, $S_{a'}^{s'}$ updates $M_{s'}$ to reflect the "delivery" of m_k at $h_{s'}$ which involves taking component-wise maximum of $m_k.M$ and $M_{s'}$. Using monotonicity of $M_{s'}$, we have $m_k.M[u,v] \leq m_j.M[u,v]$.

Case 2 [$a' > b$]: Due to the movement of $h_{s'}$, it is possible that although $h_{s'}$ received m_k and sent $ack(m_k)$ to $S_b^{s'}$, $S_b^{s'}$ did not receive $ack(m_k)$ before it sends the *enable* message. Therefore, on receiving the *enable* message from $S_b^{s'}$, $S_{b+1}^{s'}$ updates $M_{s'}$ assuming that all the messages in $ackQ_{s'}$ have been received at $h_{s'}$ before proceeding further. Using P 3.5.2 and monotonicity of $M_{s'}$, we have $m_k.M[u,v] \leq m_j.M[u,v]$.

Thus, in any case, $m_i.seqno \leq m_j.M[u,v]$. Therefore by induction the lemma holds. ■

Lemma 15 $m_i.seqno \leq m_j.M[u,v] \Rightarrow \hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd$

Proof: Assume $m_i.seqno \leq m_j.M[u,v]$. The proof is by construction. We first prove the following property satisfied by m_i and m_j,

$$m_i.seqno \leq m_j.M[u,v] \;\Rightarrow\; (\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd) \vee \langle \exists\, m_k :: (m_i.seqno \leq m_k.M[u,v]) \wedge (\hat{m}_k.snd \twoheadrightarrow_s \hat{m}_j.snd) \rangle$$

Let $\hat{m}_j.src = S^{s'}_{a'}$ where $a' = mbl(m_j).s'$. Observe that $m_i.seqno \geq 1$ and $M_{s'}$ is initially $\mathbf{0}$. Since $M_{s'}$ is monotonically non-decreasing, therefore there exists an MSS where $M_{s'}$ was updated which made the inequality true. Let $S^{s'}_b$ be the first such MSS in the sequence $\{S^{s'}_l\}$, and e_k be the earliest event on it such that the inequality holds just after e_k. Note that $a' \geq b$ and $M_{s'}$ is updated only either due to a message sent by $h_{s'}$ or due to a message destined for $h_{s'}$. Let m_k denote the message involved in e_k. Observe that $m_k \neq m_j$. In the former case (the inequality became true due to a message sent by $h_{s'}$), $\hat{m}_k.src = S_u$ and $\hat{m}_k.dst = S_v$. Since *lastsent* on $S^{s'}_b$ is monotonically non-decreasing and $m_i.seqno \leq m_k.seqno$, therefore either $m_i = m_k$ or $\hat{m}_i.snd \prec_s \hat{m}_k.snd$. Moreover, if $a' = b$ then $\hat{m}_k.snd \prec_s \hat{m}_j.snd$, otherwise $\hat{m}_k.snd \prec_s enable(S'_b).snd$ and $enable(S^{s'}_{a'-1}).dlv \prec_s \hat{m}_j.snd$. Using P 3.5.1, we have $\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd$.

In the latter case, as before, if $a' = b$ then $e_k \prec_s \hat{m}_j.src$, otherwise $e_k \twoheadrightarrow_s \hat{m}_j.snd$ ($M_{s'}$ is not updated at $S^{s'}_b$ after it sends the *enable* message). Moreover, it can be verified from the protocol that $\hat{m}_k$ was in $ackQ_{s'}$ when $M_{s'}$ was updated. Let $\hat{m}_k.dst = S^{s'}_c$ where $c = mbl(m_k).s'$. Observe that $\hat{m}_k$ first enters $ackQ_{s'}$ either at $S^{s'}_c$ on occurrence of $\hat{m}_k.dlv$ or at $S^{s'}_{c+1}$ on being received as an "old" message. After that, it gets transferred to the next MSS piggybacked on the *enable* message. Since the messages containing application message, the messages tagged as "old" and the *enable* messages are piggybacked with *up_cell*, therefore $\hat{m}_k.snd \twoheadrightarrow_s e_k$. Thus, $\hat{m}_k.snd \twoheadrightarrow_s \hat{m}_j.snd$. There are again two cases to consider. The inequality became true either due to *seqno* of m_k or as a result of taking component-wise maximum

of $m_k.M$ and $M_{s'}$. In the first case, $\hat{m}_k.src = S_u$, $\hat{m}_k.dst = S_v$ and $m_i.seqno \leq m_k.seqno$. Therefore, either $m_i = m_k$ or $\hat{m}_i.snd \prec_s \hat{m}_k.snd$. Combining with earlier result, we get $\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd$. Finally, in the second case, $m_i.seqno \leq m_k.M[u,v]$.

Thus, $(\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd) \vee \langle \exists m_k :: (m_i.seqno \leq m_k.M[u,v]) \wedge (\hat{m}_k.snd \twoheadrightarrow_s \hat{m}_j.snd) \rangle$ holds. We can apply the same property to m_i and m_k since $m_i.seqno \leq m_k.M[u,v]$. We claim that at most n_h applications of the property establishes $\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd$. The proof is by contradiction. Assume the contrary. Then, there is a chain of messages, $m_{k_1}, m_{k_2}, \ldots m_{k_l}, m_j$ such that $\hat{m}_{k_1}.snd \twoheadrightarrow_s \hat{m}_j.snd$ ($\twoheadrightarrow_s$ is transitive) and $l > n_h$. Using the pigeon-hole principle, we can infer that at least two messages in the chain are sent by the same MH. Let the messages be m_{k_p} and m_{k_q}. Also, let e_{k_p} and e_{k_q} be the events used in the proof of the property. Since both events involve update of the MH matrix, therefore either $e_{k_p} \rightarrow_s e_{k_q}$ or $e_{k_q} \rightarrow_s e_{k_p}$ holds which contradicts the choice of e_{k_p} or e_{k_q}. Thus, the lemma holds.

■

Lemma 16 $\hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd \Rightarrow mbl(m_i) \preceq mbl(m_j)$

Proof: The lemma can be proved by induction on the number of messages, n, involved in smallest causal chain (with respect to $\twoheadrightarrow_s$) from $\hat{m}_i.snd$ to $\hat{m}_j.snd$. The proof is straightforward and is left to the reader. ■

3.5.3.2 Liveness Property

Lemma 17 *Every handoff procedure for a mobile host terminates.*

Proof: Let $handoff(l,p)$ denote the handoff procedure between S_p^l and S_{p+1}^l for h_l, when h_l moves from the cell of S_p^l to the cell of S_{p+1}^l. The lemma can be proved

easily by induction on p, $0 \leq p < n(h_l)$. ■

Let $handoff\_over(S_p^l)$ denote the *handoff_over* message sent by S_p^l to S_{p+1}^l in the handoff module for h_l, when h_l moves from the cell of S_p^l to the cell of S_{p+1}^l. Since S_p^l, does not process the *handoff_begin* message from S_{p+1}^l until it receives the *handoff_over* message from S_{p-1}^l, therefore we have,

(P 3.5.3) $\quad p < q \Rightarrow handoff\_over(S_p^l).snd \rightarrow_s handoff\_over(S_q^l).snd$

Let $\hat{m}_i.ercvd$ denote the earliest event on S_v such that $\hat{m}_i.M[1 : n_s, v] \preceq lastrcvd(\hat{m}_i.ercvd)$. Observe that $\langle \forall\ e\ :\ \hat{m}_i.ercvd \prec_s e\ :\ \hat{m}_i.M[1 : n_s, v] \preceq lastrcvd(e) \rangle$ is true. Intuitively, $\hat{m}_i.ercvd$ represents the earliest event on S_v when all the messages sent to S_v on which m_i causally depends (potentially) have been received at S_v.

Lemma 18 *$\hat{m}_i.ercvd$ occurs eventually. Moreover, if S_r^d is not the final mobile support station for h_d, i.e. $r < n(h_d)$, then $\hat{m}_i.ercvd \prec_s handoff\_over(S_r^d).snd$.*

Proof: Consider a message $\hat{m}_k$ destined for S_v such that $\hat{m}_k.seqno \leq \hat{m}_i.M[w, v]$, where $S_w = \hat{m}_k.src$. We claim that $\hat{m}_k$ is eventually received, i.e. $\hat{m}_k.rcv$ occurs eventually. Furthermore, if $r < n(h_d)$ then $\hat{m}_k.rcv \prec_s handoff\_over(S_r^d).snd$. Assume S_r^d is the final MSS for h_d. Since the channels among MSSs are reliable, therefore $\hat{m}_k$ is received eventually. Otherwise, assume $r < n(h_d)$. We have three cases to consider depending on the source MSS of $\hat{m}_k$. Let S_n denote the MSS to whose cell h_d moves after leaving the cell of S_r^d. Let $handoff\_begin(S_{r+1}^d)$ denote the *handoff_begin* message sent by S_{r+1}^d to S_r^d in the handoff procedure when h_d switches cell. Let $notify(S_r^d)$ represent the *notify* message broadcast by S_r^d to the MSSs in the handoff procedure and let $last(S_w, S_r^d)$ denote the corresponding *last* message sent by S_w to S_r^d.

Case 1 [$S_w = S_v$]: In this case, $\hat{m}_k.snd \prec_s handoff\_begin(S^d_{r+1}).dlv$. Assume the contrary. After processing the *handoff_begin* message, the value of $cell[d].mlb$ at S_v becomes $r+1$. Thus, $mbl(\hat{m}_k).d > r$. Using Lemma 15 and Lemma 16, we can infer that $r < mbl(\hat{m}_i).d$, a contradiction. Since the messages sent to itself are received immediately and $handoff\_begin(S^d_{r+1}).dlv \prec_s handoff\_over(S^d_r).snd$, therefore $\hat{m}_k.rcv \prec_s handoff\_over(S^d_r).snd$.

Case 2 [$S_w = S_n$]: In this case, $\hat{m}_k.snd \prec_s handoff\_begin(S^d_{r+1}).snd$. The proof is identical to the proof in Case 1. Since the channels are reliable and FIFO, therefore $\hat{m}_k.rcv \prec_s handoff\_begin(S^d_{r+1}).rcv$. Also, $handoff\_begin(S^d_{r+1}).rcv \prec_s handoff\_over(S^d_r).snd$. Thus, $\hat{m}_k.rcv \prec_s handoff\_over(S^d_r).snd$.

Case 3 [$S_w \in \mathcal{S} \setminus \{S_v, S_n\}$]: Finally, in this case, $\hat{m}_k.snd \prec_s notify(S^d_r).dlv$. Since the channels are reliable and FIFO, and $notify(S^d_r).dlv \prec_s last(S_w, S^d_r).snd$, therefore $\hat{m}_k.rcv \prec_s last(S_w, S^d_r).rcv$. Also, $last(S_w, S^d_r).rcv \prec_s handoff\_over(S^d_r).snd$. Thus, $\hat{m}_k.rcv \prec_s handoff\_over(S^d_r).snd$.

In any case, $\hat{m}_k.rcv \prec_s handoff\_over(S^d_r).snd$. Thus, for all $\hat{m}_k$ destined for S_v such that $\hat{m}_k.seqno \leq \hat{m}_i.M[w,v]$, where $\hat{m}_k.src = S_w$, we have $\hat{m}_k.rcv \prec_s handoff\_over(S^d_r).snd$. Since as soon as a message is received *lastrcvd* is updated, therefore $\hat{m}_i.ercvd \prec_s handoff\_over(S^d_r).snd$. Therefore the lemma holds. ■

Lemma 19 *$\hat{m}_i.rcv$ occurs eventually. Moreover, if S^d_r is not the final mobile support station for h_d, i.e. $r < n(h_d)$, then $\hat{m}_i.rcv \prec_s handoff\_over(S^d_r).snd$.*

Proof: The lemma can be proved by doing a case analysis identical to the one in Lemma 18. The proof is left to the reader. ■

Lemma 20 *$\hat{m_i}$ is eventually delivered (at its destination mobile support station S_v). Moreover, if S_r^d is not the final mobile support station for h_d, i.e. $r < n(h_d)$, then $\hat{m_i}.dlv \prec_s handoff\_over(S_r^d).snd$.*

Proof: Let $\mathcal{M}_A$ denote the set of messages which contain application messages (not tagged as "old") sent by a mobile support station to another mobile support station to be delivered to the destination mobile hosts. Let $\mathcal{M}_C$ be the set of messages on which *up_cell* is piggybacked. We first define a binary relation, $\sqsubset$, on $\mathcal{M}_A$ as follows,

$$\hat{m_i} \sqsubset \hat{m_j} \stackrel{def}{=} (h_d = h_{d'}) \wedge (S_v = S_{v'}) \wedge (\hat{m_i}.seqno \leq \hat{m_j}.M[u,v])$$

Observe that $\mathcal{M}_A \subseteq \mathcal{M}_C$ and $\sqsubset \subseteq \twoheadrightarrow_s$ (Lemma 15). Also, $(\mathcal{M}_C, \twoheadrightarrow_s)$ is a well-founded set. Thus, we can infer that $(\mathcal{M}_A, \sqsubset)$ is also a well-founded set. Let $\mathcal{P}.\hat{m_k}$ be "the lemma holds for $\hat{m_k}$". Assume $\langle \forall\, \hat{m_k} : \hat{m_k} \sqsubset \hat{m_i} : \mathcal{P}.\hat{m_k} \rangle$. There are two cases to consider: $r = n(h_d)$ or $r < n(h_d)$.

Case 1 $[r < n(h_d)]$: Using Lemma 19, we have $\hat{m_i}.rcv \prec_s handoff\_over(S_r^d).snd$. If S_r^d is the initial MSS of h_d, i.e. $(r = 0)$, then $canDeliver_d$ is true initially. Otherwise, using Lemma 17 we can infer that $handoff\_over(S_{r-1}^d).dlv$ eventually occurs at S_r^d after which $canDeliver_d$ is set to true. Moreover, $canDeliver_d$ remains true until S_r^d sends the *handoff_over* message to S_{r+1}^d. Let *canDeliver* be the earliest event on S_r^d after which $canDeliver_d$ is true. Then $canDeliver \prec_s handoff\_over(S_r^d).snd$. From Lemma 18, we can conclude that $\hat{m_i}.ercvd \prec_s handoff\_over(S_r^d).snd$. Consider $\hat{m_k}$ such that $\hat{m_k} \sqsubset \hat{m_i}$. Using definition of $\sqsubset$, Lemma 15 and Lemma 16, we have $m_k.dst = h_d$ and $mbl(\hat{m_k}).d \leq mbl(\hat{m_i}).d = r$. Therefore, using induction hypothesis and P 3.5.3, we get $\hat{m_k}.dlv \prec_s handoff\_over(S_r^d).snd$. Observe that after all mes-

sages $\hat{m_k}$ such that $\hat{m_k} \sqsubset \hat{m_i}$ have been delivered, then the last expression in the conjunct of the "if" condition in (A4) is never falsified. Let e be the latest of all the events in $\{\hat{m_i}.rcv, canDeliver, \hat{m_i}.ercvd\} \cup \{\hat{m_k}.dlv \mid \hat{m_k} \sqsubset \hat{m_i}\}$. Then $e \prec_s handoff\_over(S_r^d).snd$. After e, the "if" condition in (A4) evaluates to true for $\hat{m_i}$, and $deliver(\hat{m_i})$ is called. Thus, $\hat{m_i}.dlv \prec_s handoff\_over(S_r^d).snd$. Therefore $\mathcal{P}.\hat{m_i}$ holds.

Case 2 $[r = n(h_d)]$: In this case, we have to prove that $\hat{m_i}.dlv$ eventually occurs. The proof is quite similar to but simpler than the proof for Case 1.

Hence by induction, the lemma holds. ■

Theorem 21 *(***liveness***)* *m_i is eventually delivered (at its destination mobile host, h_d).*

Proof: We first show that $\hat{m_i}$ eventually enters $ackQ_d$. If S_r^d is the final mobile support station for h_d or $\hat{m_i}.dlv \prec_s handoff\_begin(S_r^d).dlv$ then $\hat{m_i}$ enters $acks_d$ as soon as $\hat{m_i}.dlv$ occurs. Otherwise, on occurrence of $\hat{m_i}.dlv$, $\hat{m_i}$ is sent to S_{r+1}^d where it is inserted into $ackQ_d$ on being received. Let S_t^d, $r \leq t \leq n(h_d)$ be the MSS such that h_d stays for sufficiently long time in the cell of S_t^d after $\hat{m_i}$ enters $ackQ_d$. Let $\mathcal{M}_{ack}$ be the set of messages that entered $ackQ_d$ at S_t^d (including messages that were already in $ackQ_d$ when $ackQ_d$ was transferred to S_t^d) before $\hat{m_i}$. Note that the messages are sent to h_d in the order in which they enter $ackQ_d$ ((A5') and (A10)). Moreover, after receiving $|\mathcal{M}_{ack}|$ acknowledgement messages from h_d, $\hat{m_i}$ will be at the front in $ackQ_d$. Since the channel between an MH and its MSS is reliable and FIFO, therefore S_t^d receives $|\mathcal{M}_{ack}|^{th}$ acknowledgement message from h_d if h_d does not switch cell for a sufficiently long time. Thus, m_i is delivered at h_d. ■

3.5.3.3 Safety Property

Lemma 22 *If $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$ then $m_i.dlv \prec_h m_j.dlv$.*

Proof: Note that $h_d = h_{d'}$. Let S_t^d and $S_{t'}^d$ denote the MSSs that delivered m_i and m_j respectively to h_d for the first time (t and t' exist due to Theorem 21). If $t < t'$ then it can be easily proved that $m_i.dlv \prec_h m_j.dlv$. Therefore, assume $t \geq t'$. Observe that in the protocol as soon as a message is inserted in $ackQ_d$ at $S_{t'}^d$, it is also dispatched to h_d ((A5') and (A10)). Thus, at $S_{t'}^d$, m_i is sent to h_d before m_j. Since the channel between an MH and its MSS is FIFO, therefore h_d receives m_i before m_j. Hence $m_i.dlv \prec_h m_j.dlv$. ■

Lemma 23 $mbl(m_i).d < mbl(m_j).d \;\Rightarrow\; \neg(m_j.dlv \prec_h m_i.dlv)$

Proof: If $h_d \neq h_{d'}$ then the consequent (and hence the lemma) is trivially true. Therefore assume $h_d = h_{d'}$. We first prove that $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$. If $\hat{m}_i.dlv \prec_s handoff\_begin(S_{r+1}^d).dlv$ then $\hat{m}_i$ enters $ackQ_d$ at S_r^d. Otherwise, on occurrence of $\hat{m}_i.dlv$, $\hat{m}_i$ is sent to S_{r+1}^d where it is inserted into $ackQ_d$ as soon as it is received. Using Lemma 20 and the fact that the channels among MSSs are FIFO, we can infer that $\hat{m}_i$ is received at S_{r+1}^d before $handoff\_over(S_r^d)$. Also, from the protocol we know that $\hat{m}_j$ cannot enter $ackQ_d$ before $handoff\_over(S_{r'-1}^d)$ is received. Thus, using P 3.5.3, we can conclude that in any case $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$. Finally, using Lemma 22, we have $m_i.dlv \prec_h m_j.dlv$. ■

Lemma 24 $(mbl(m_i).d = mbl(m_j).d) \wedge (m_i.snd \rightarrow_h m_j.snd) \;\Rightarrow\; \neg(m_j.dlv \prec_h m_i.dlv)$

Proof: If $h_d \neq h_{d'}$ then the consequent (and hence the lemma) is trivially true. Therefore assume $h_d = h_{d'}$. We first prove that $\hat{m}_i.dlv \prec_s \hat{m}_j.dlv$. Note that $S_v = S_{v'}$. From Lemma 14, we can conclude that $\hat{m}_i.seqno \leq \hat{m}_j.M[u, v]$. Observe

that $\hat{m}_j.ercvd$ cannot occur before $\hat{m}_i.rcv$. Moreover, after $\hat{m}_j.ercvd$ occurs, $\hat{m}_j$ cannot be delivered until $\hat{m}_i$ is delivered. Thus, $\hat{m}_i.dlv \prec_s \hat{m}_j.dlv$. If S_r^d is the final MSS for h_d then as soon as $\hat{m}.dlv$ occurs it is inserted into $ackQ_d$. Therefore, $\hat{m}_i$ is inserted into $ackQ_d$ before $\hat{m}_j$. Otherwise, there are three cases to consider:

Case 1 [$\hat{m}_i.dlv \prec_s \hat{m}_j.dlv \prec_s handoff\_begin(S_{r+1}^d).dlv$]: On occurrence of $\hat{m}_i.dlv$ ($\hat{m}_j.dlv$), $\hat{m}_i$ ($\hat{m}_j$) is inserted into $ackQ_d$. Hence $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$.

Case 2 [$\hat{m}_i.dlv \prec_s handoff\_begin(S_{r+1}^d).dlv \prec_s \hat{m}_j.dlv$]: On occurrence of $\hat{m}_i.dlv$, $\hat{m}_i$ is inserted into $ackQ_d$. On processing $handoff\_begin(S_{r+1}^d)$, $ackQ_d$ is piggybacked on the $enable(S_r^d)$ message and sent to S_{r+1}^d. Then, when $\hat{m}_j.dlv$ occurs, $\hat{m}_j$ is sent to S_{r+1}^d where it enters $ackQ_d$. Since the channels among MSSs are reliable and FIFO, therefore $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$.

Case 3 [$handoff\_begin(S_{r+1}^d).dlv \prec_s \hat{m}_i.dlv \prec_s \hat{m}_j.dlv$]: On occurrence of $\hat{m}_i.dlv$ ($\hat{m}_j.dlv$), $\hat{m}_i$ ($\hat{m}_j$ is sent to S_{r+1}^d tagged as on "old" message. On receiving $\hat{m}_i$ ($\hat{m}_j$), S_{r+1}^d inserts $\hat{m}_i$ ($\hat{m}_j$) into $ackQ_d$. Since the channels among MSSs are reliable and FIFO, therefore $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$.

In any case, $\hat{m}_i$ enters $ackQ_d$ before $\hat{m}_j$. Finally, using Lemma 22, we have $m_i.dlv \prec_h m_j.dlv$. ■

Theorem 25 *The protocol implements causal ordering among mobile hosts. In other words,*

$$m_i.snd \rightarrow_h m_j.snd \Rightarrow \neg(m_j.dlv \prec_h m_i.dlv)$$

Proof: The proof is a straightforward manipulation of the lemmas.

$$
\begin{array}{lll}
 & m_i.snd \rightarrow_h m_j.snd & \\
\Rightarrow & m_i.seqno \leq m_j.M[u,v] & \text{; Lemma 14} \\
\Rightarrow & \hat{m}_i.snd \twoheadrightarrow_s \hat{m}_j.snd & \text{; Lemma 15} \\
\Rightarrow & mbl(m_i) \preceq mbl(m_j) & \text{; Lemma 16} \\
\Rightarrow & mbl(m_i).d \leq mbl(m_j).d & \text{; definition of } \preceq, \\
 & & \text{; instantiation} \\
\equiv & (mbl(m_i).d < mbl(m_j).d) \vee (mbl(m_i).d = mbl(m_j).d) & \text{; definition of } \leq \\
\Rightarrow & (mbl(m_i).d < mbl(m_j).d) \vee ((mbl(m_i).d = mbl(m_j).d) \wedge (m_i.snd \rightarrow_h m_j.snd)) & \\
 & & \text{; use antecedent} \\
\Rightarrow & \neg(m_j.dlv \prec_h m_i.dlv) \vee \neg(m_j.dlv \prec_h m_i.dlv) & \text{; Lemma 23,} \\
 & & \text{; Lemma 24} \\
\Rightarrow & \neg(m_j.dlv \prec_h m_i.dlv) & \text{; idempotence of } \vee
\end{array}
$$

Thus, the theorem holds. ■

3.5.4 Characterization of Static Module

Here we state and prove the predicate that characterizes our static module. The static module in Section 3.5.1 implements,

$$
(\mathbf{CO''}) \quad \langle \exists\, m_k : \hat{m}_i.dst = \hat{m}_k.dst : (\hat{m}_i.snd \preccurlyeq_s \hat{m}_k.snd) \wedge (m_k.snd \rightarrow_h m_j.snd) \rangle \Rightarrow \\
\neg(m_j.dlv \prec_h m_i.dlv) \wedge \neg(\hat{m}_j.dlv \prec_s \hat{m}_i.rcv),
$$

where $e \preccurlyeq_s f$ iff $(e = f) \vee (e \prec_s f)$, under the assumption that the channels among MSSs are FIFO. Moreover, if the channels among MSSs are not FIFO then it implements,

$$
CO'' \wedge (\hat{m}_i.snd \prec_s \hat{m}_j.snd \Rightarrow \neg(\hat{m}_j.dlv \prec_s \hat{m}_i.rcv))
$$

For convenience, let $FO'' \stackrel{def}{=} \hat{m}_i.snd \prec_s \hat{m}_j.snd \Rightarrow \neg(\hat{m}_j.dlv \prec_s \hat{m}_i.rcv)$. For the following proofs, we define $m_i.P$ for an application message m_i as follows,

$$m_i.P[u,v] = \max\{\{m_k \mid (\hat{m}_k.src = S_u) \wedge (\hat{m}_k.dst = S_v) \wedge (m_k.snd \rightarrow_h m_i.snd)\}\},$$

where $\max\{S\}$ returns the message with the largest *seqno* in the set S. Also, $\max\{\phi\} = \perp$, where $\perp.seqno = 0$ and $\perp \rightarrow_h m_i$.

Lemma 26 *For an application message m_i, $m_i.M[u,v] = m_i.P[u,v].seqno$ for all u and v.*

Proof: Using Lemma 14 and definition of $m_i.P[u,v]$, we can infer that $m_i.P[u,v].seqno \leq m_i.M[u,v]$ (the inequality trivially holds if $m_i.P[u,v] = \perp$). Assume $m_i.M[u,v] > m_i.P[u,v].seqno$. We will derive a contradiction.
Let $m_i.M[u,v] = n, n > 0$. We first prove the following property for the application message m_i,

$$m_i.M[u,v] = n \Rightarrow \langle \exists m_k :: (((\hat{m}_k.src = S_u) \wedge (\hat{m}_k.dst = S_v) \wedge (m_k.seqno = n)) \vee (m_k.M[u,v] = n)) \wedge (m_k.snd \rightarrow_h m_i.snd) \rangle$$

Let $m_i.src = h_s$. Observe that $n > 0$ and M_s is initially $\mathbf{0}$. Since M_s is monotonically non-decreasing, therefore there exists an event on $\hat{m}_i.src$ when M_s was updated which made the equality, $M_s[u,v] = n$, true. Let e_k be the earliest event on it such that the equality holds just after e_k. Note that M_s is updated only either due to a message sent by h_s or due to a message received by h_s. Let m_k denote the application message involved in e_k. Observe that $e_k \prec_s \hat{m}_i.snd$. In the former case (the inequality became true due to a message sent by h_s), $\hat{m}_k.src = S_u$ and $\hat{m}_k.dst = S_v$. Moreover, $m_k.seqno = n$ and $m_k.snd \prec_h m_i.snd$. In the latter case, there are again two cases to consider. The equality became true either due to *seqno* of m_k or as a result of taking component-wise maximum of $m_k.M$ and M_s.

In the first case, $\hat{m}_k.src = S_u$, $\hat{m}_k.dst = S_v$ and $m_k.seqno = n$. In the second case $m_k.M[u,v] = n$. Moreover, in both cases, $m_k.snd \rightarrow_h m_i.snd$.

Thus, the property holds. If the second term of the "$\vee$" expression holds for m_k then we can apply the same argument since in that case $m_k.M[u,v] = n, n > 0$. We claim that at most n_h applications of the property establishes $\langle \exists\, m_k :: (\hat{m}_k.src = S_u) \wedge (\hat{m}_k.dst = S_v) \wedge (m_k.seqno = n) \wedge (m_k.snd \rightarrow_h m_i.snd) \rangle$. The proof is by contradiction. Assume the contrary. Then, there is a chain of messages, $m_{k_1}, m_{k_2}, \ldots m_{k_l}, m_i$ such that $m_{k_1}.snd \rightarrow_h m_i.snd$ ($\rightarrow_h$ is transitive) and $l > n_h$. Using the pigeon-hole principle, we can infer that at least two messages in the chain are sent by the same mobile host. Let the messages be m_{k_p} and m_{k_q}. Also, let e_{k_p} and e_{k_q} be the events used in the proof of the property. Then $e_{k_p} \rightarrow_s e_{k_q}$ or $e_{k_q} \rightarrow_s e_{k_p}$ holds which contradicts the choice of e_{k_p} or e_{k_q}. Thus, there exists an application message m_k such that $\hat{m}_k.src = S_u$, $\hat{m}_k.dst = S_v$, $m_k.seqno = n$ and $m_k.snd \rightarrow_h m_i.snd$. Also, $m_i.M[u,v] = n = m_k.seqno > m_i.P[u,v].seqno$ which contradicts the definition of $m_i.P[u,v]$. Hence $m_i.M[u,v] = m_i.P[u,v].seqno$ and the lemma holds. ■

Lemma 27 *For any two application messages m_i and m_j such that $\hat{m}_i.src = S_u$ and $\hat{m}_i.dst = S_v$, the static module satisfies,*

$$\langle \exists\, m_k : \hat{m}_i.dst = \hat{m}_k.dst : (\hat{m}_i.snd \preccurlyeq_s \hat{m}_k.snd) \wedge (m_k.snd \rightarrow_h m_j.snd) \rangle \equiv m_i.seqno \leq m_j.M[u,v]$$

Proof:

($\Rightarrow$)

(A.1) $m_i.snd \rightarrow_h m_j.snd \Rightarrow m_i.seqno \leq m_j.M[u,v]$

We prove (A.1) by induction on the number of messages,n, in the causal chain (with respect to $\rightarrow_h$) from $m_i.snd$ to $m_j.snd$.

Base Case ($n = 0$): In this case, $m_i.snd \prec_h m_j.snd$. On sending $\hat{m}_i$, S_u sets the $(u,v)^{th}$ entry of the host matrix to $m_i.seqno$. Since the wireless channels are FIFO and the host matrix is monotonically non-decreasing, therefore $m_i.seqno \leq m_j.M[u,v]$.

Induction Step ($n > 0$): Let $m_j.src = h_{s'}$. Let m_l be the last message in the causal chain. Using induction, we get $m_i.seqno \leq m_l.M[u,v]$. Observe that m_l is delivered to $h_{s'}$ before $m_j.snd$ occurs (to create the causal dependency). Since wireless channels are FIFO and reliable therefore acknowledge message for m_l, $ack(m_l)$, is received before m_j. On receiving $ack(m_l)$, $\hat{m}_j.src$ sets $M_{s'}$ to component-wise maximum of $m_l.M$ and $M_{s'}$. Hence, we have $m_l.M[u,v] \leq m_j.M[u,v]$. Thus, $m_i.seqno \leq m_j.M[u,v]$.

Thus, by induction, $m_i.snd \rightarrow_h m_j.snd \;\Rightarrow\; m_i.seqno \leq m_j.M[u,v]$.

(A.2) $\langle \exists\, m_k : \hat{m}_i.dst = \hat{m}_k.dst : (\hat{m}_i.snd \prec_s \hat{m}_k.snd) \wedge (m_k.snd \rightarrow_h m_j.snd) \rangle \;\Rightarrow$
$m_i.seqno \leq m_j.M[u,v]$

Since $\hat{m}_i.dst = \hat{m}_k.dst$ and $\hat{m}_i.snd \prec_s \hat{m}_k.snd$ therefore $m_i.seqno < m_k.seqno$. Moreover, since $m_k.snd \rightarrow_h m_j.snd$, using (A.1) we have $m_k.seqno \leq m_j.M[u,v]$. Combining both the results, we have $m_i.seqno \leq m_j.M[u,v]$.

($\Leftarrow$)

Assume $m_i.seqno \leq m_j.M[u,v]$. Using Lemma 26, we can infer that there exists a message m_l such that $m_l.seqno = m_j.M[u,v]$ and $m_l.snd \rightarrow_h m_j.snd$. Moreover, $\hat{m}_l.src = S_u, \hat{m}_l.dst = S_v$. Since $\hat{m}_i.src = S_u = \hat{m}_l.src$, $\hat{m}_i.dst = S_v = \hat{m}_l.dst$ and $m_i.seqno \leq m_j.M[u,v] = m_l.seqno$, therefore $\hat{m}_i.snd \preccurlyeq_s \hat{m}_l.snd$. ■

Theorem 28 *The static module implements CO'' under the assumption that the channels among mobile support stations are FIFO.*

Proof: Let $\mathcal{X}_{SM}$ and $\mathcal{X}_{CO''}$ be the set of executions accepted by the proposed static module and the condition CO'' respectively. To prove that the static module implements CO'', we need to show that $\mathcal{X}_{SM} = \mathcal{X}_{CO''}$ i.e. the executions generated by the static module satisfy the condition CO'' and vice versa. For convenience, let $m_i \mapsto m_j \stackrel{def}{=} \langle \exists m_k : \hat{m}_i.dst = \hat{m}_k.dst : (\hat{m}_i.snd \preccurlyeq_s \hat{m}_k.snd) \wedge (m_k.snd \rightarrow_h m_j.snd) \rangle$. Observe that $m_i \mapsto m_j \Rightarrow m_i \rightarrow_s m_j$. Therefore $\mapsto$ is acyclic.

(B.1) $\mathcal{X}_{CO''} \subseteq \mathcal{X}_{SM}$: Consider an execution $\mathcal{X}$ that satisfies CO''. Let $\rightarrow$ denote the Lamport's "happened before" relation on the set of events (on MHs and MSSs) in the execution $\mathcal{X}$. Since $\rightarrow$ is a partial order, it can be extended to some total order. Let E denote the sequence of events with respect to the total order and E_n be the prefix of E containing the first n events. We prove that for all n, E_n can be generated by the proposed static module. The proof is by induction on n. For the purpose of the proof, the events are either *deliver* or *non-deliver* events. Note that the static module controls only the deliver events on mobile support stations.

Base Case ($n = 1$): Observe that the first event cannot be a deliver event. Therefore E_1 can be generated by the static module.

Induction Step ($n > 1$): Using induction hypothesis, E_{n-1} can be generated by the static module. Assume n^{th} event, say e_n, is a deliver event on a mobile support station, say S_v, and let m_i be the application message involved in the event. We need to prove that m_i is deliverable according to our static module. Let $\mathcal{M}_R$ denote the set of messages destined for $m_i.dst$ that have been received but not yet delivered at S_v just before e_n occurs ($\mathcal{M}_R \neq \phi$ since $m_i \in \mathcal{M}_R$). Let

$chann(G, e_n)$ denote the set of messages sent to S_v in-transit (sent to S_v but not yet received at S_v) in the consistent cut G that includes e_n, and $\mathcal{M}_D$ be $\mathcal{M}_R \cup chann(G, e_n)$. We first show that m_i is minimal in $\mathcal{M}_D$ with respect to $\mapsto$ ($\mapsto$ is acyclic). Assume the contrary. Let m_k be the application message such that $m_k \mapsto m_i$. Then $m_k \in \mathcal{M}_R$ or $m_k \in chann(G, e_n)$. In either case, $\mathcal{X}$ does not satisfy CO'', a contradiction. Now we prove that m_i is deliverable according to the proposed static module. We prove the contrapositive, that is, if m_i is not deliverable then it is not minimal in $\mathcal{M}_D$. From the static module, it can be verified that either (1) $lastrcvd_v[u] < m_i.M[u, v]$ for some S_u, or (2) there exists an application message m_k in $rcvQ_v$, destined for $m_i.dst$, such that $m_k.seqno \leq m_i.M[u, v]$, where $\hat{m}_k.src = S_u$. In the first case, (1), using Lemma 26 we can infer that there exists a message m_k such that $\hat{m}_k.src = S_u$ and $\hat{m}_k.dst = S_v$. Also, $m_k.seqno \leq m_i.M[u, v]$ and $m_k \in chann(G, e_n)$. Using Lemma 27, we have $m_k \mapsto m_i$. In the second case, (2), $m_k \in \mathcal{M}_R$. Again using Lemma 27, we can conclude that $m_k \mapsto m_i$. In either case m_i is not minimal in $\mathcal{M}_D$, a contradiction. Thus, m_i is deliverable according to the static module.

Therefore, using induction, we can infer that the execution $\mathcal{X}$ can be generated by the static module.

(B.2) $\mathcal{X}_{SM} \subseteq \mathcal{X}_{CO''}$: Consider an execution $\mathcal{X}$ generated by the static module. We have to prove that $\mathcal{X}$ satisfies CO''. Let m_i and m_j be arbitrary application messages such that $m_i \mapsto m_j$. If $\hat{m}_i$ and $\hat{m}_j$ are destined for different MSSs then CO'' is trivially satisfied. Hence assume $\hat{m}_i.dst = \hat{m}_j.dst$. Let $\hat{m}_i.src = S_u$ and $\hat{m}_i.dst = \hat{m}_j.dst = S_v$. Using Lemma 27, we can conclude that $m_i.seqno \leq m_j.M[u, v]$. From the protocol it can be verified that when $\hat{m}_j.dlv$ occurs then $lastrcvd_v[u] \geq m_j.M[u, v]$. Therefore $lastrcvd_v[u] \geq m_i.seqno$ i.e. $\hat{m}_i.rcv$ has

already occurred. Thus, we have $\neg(\hat{m}_j.dlv \prec_s \hat{m}_i.rcv)$. If m_i and m_j are destined for different MHs then the first expression in the consequent of CO'' trivially holds. Therefore assume $m_i.dst = m_j.dst = h_d$. Again from the protocol it can be verified that when $\hat{m}_j.dlv$ occurs then $\hat{m}_i$ is not in $rcvQ_v$. Since $\hat{m}_i$ has been received (as argued before) therefore $\hat{m}_i.dlv$ has already occured at S_v (when $\hat{m}_j.dlv$ occurs). Moreover, the wireless channels are FIFO and reliable. Thus, we have $\neg(m_j.dlv \prec_s m_i.dlv)$. Hence $\mathcal{X}$ satisfies CO''.

Thus, $\mathcal{X}_{SM} = \mathcal{X}_{CO''}$ and the theorem holds. ■

Although we do not prove here but if we relax the FIFO assumption then it can be easily verified that the static module Section 3.5.1 implements $CO'' \wedge FO''$.

3.6 Comparison and Discussion

The proposed static module implements $CO'' \wedge FO''$ which is weaker than CO' implemented by $AV2$ ($CO' \Rightarrow CO'' \wedge FO''$). As a result, unnecessary delay in our protocol is lower than that imposed in $AV2$. In the worst case, message overhead in our protocol is $O(n_s^2 + n_h)$ but we expect it to be closer to $O(n_s^2)$ in practice. Our storage overhead in each MSS is $O(k \times n_s^2)$, where k is the number of MHs currently in the cell of the MSS. Even though this overhead is higher than that of $AV2$, it can be easily accommodated by MSSs due to their rich memory resources.

PSR [PRS96] is not suitable for systems where the number of mobile hosts dynamically changes because the structure of information carried by each message in their algorithm depends on the number of participating processes. In our protocol, the structure of the information carried by each message in the wired network does not vary with the number of MHs in the system. So, our protocol is more suitable for dynamic systems. PSR, however, incurs no unnecessary delay in message delivery.

We first give a scenario (in Figure 3.10) where YHH does not satisfy liveness

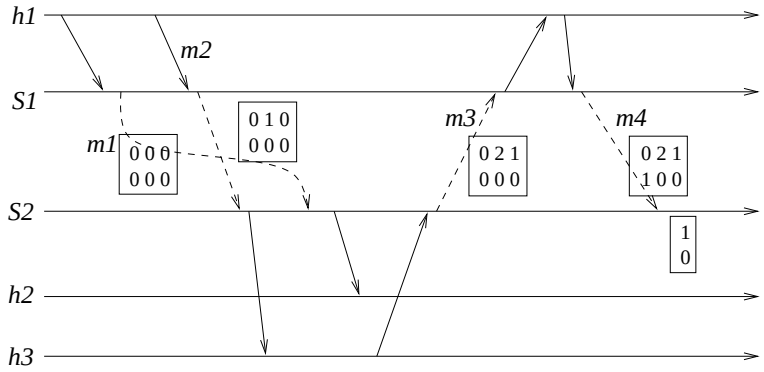


Figure 3.10: A mobile computation illustrating the liveness problem in *YHH*.

property. According to *YHH*, message m_4 will be delayed because $m_4.M[1,2] >$ MH_DELIV$_2$[1]. And since at the time when m_4 arrives at S_2, there are no messages in transit, m_4 is delayed indefinitely. The problem can be corrected by using sequence numbers. The static module in *YHH* (corrected) [YHH97] satisfies $\hat{m}_i.snd \rightarrow_s \hat{m}_j.snd \Rightarrow \neg(m_j.dlv \prec_h m_i.dlv)$. Their message overhead in the wired network is $O(n_s \times n_h)$. This overhead is higher than ours but lower than $AV1$. Their unnecessary delay is strictly lower than $AV2$. When comparing in terms of unnecessary delay, their delay is lower than ours in the average case which is expected because of their higher message overhead. However, there are cases where our protocol does not impose delivery delay but their protocol does. One can further reduce the unnecessary delay in *YHH* using the technique introduced in this paper. By assigning a matrix of size $n_s \times n_h$ to each host, the condition implemented by their static module can be weakened to,

$$\langle \exists\, m_k : m_i.dst = m_k.dst : (\hat{m}_i.snd \preccurlyeq_s \hat{m}_k.snd) \wedge (m_k.snd \rightarrow_h m_j.snd) \rangle \Rightarrow \neg(m_j.dlv \prec_h m_i.dlv)$$

Table 3.1 summarizes the comparison between our protocol and the previous work.

Algorithm	Message overhead	Well-suited for dynamic systems	Unnecessary Delay
$AV2$	$O(n_s^2)$	Yes	high
PSR	$O(n_h^2)$	No	none
YHH	$O(n_s \times n_h)$	No	low
Our Algorithm	$O(n_s^2 + n_h)$	Yes	moderate
n_h: the number of mobile hosts n_s: the number of mobile support stations			

Table 3.1: Comparison between our algorithm and the previous work.

3.7 Performance Evaluation

3.7.1 Simulation Environment

Simulation experiments are conducted for different combinations of *message size* and *communication patterns*. We use 512 bytes for the size of small messages, and $8K - 10K$ bytes for large messages. Two communication patterns are used in the simulation: *uniform*, and *nonuniform*. Nonuniform pattern is induced by having odd numbered hosts generate messages at three times the rate of even numbered hosts. For each application message m, we define *MH-to-MH Delay* as the elapsed time between $m.snd$ and $m.dlv$. Similarly, *MSS-to-MSS Delay* is the elapsed time between $\hat{m}.snd$ and $\hat{m}.dlv$.

The time between generation of successive messages at a mobile host is exponentially distributed with mean 100 ms. The destination host of each message is a uniformly distributed random variable. The throughput of a wired channel is assumed to be 100 Mbps, and the propagation delay in a wired channel is 7 ms. These two parameters are also used in [AV97]. For a wireless channel, the throughput and propagation delay are respectively assumed to be 20 Mbps and 0.5 ms. This throughput of wireless links is supported in European High Performance Radio Lo-

cal Area Network (HiperLAN). In each run, the ratio of the number of mobile hosts and support stations is varied from 1 to 150.

3.7.2 Results

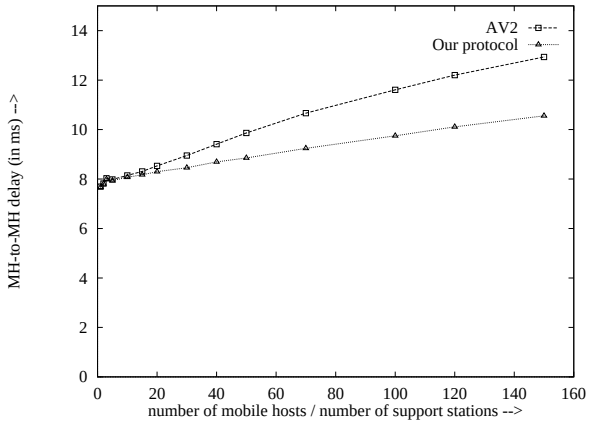
We plot the MH-to-MH and MSS-to-MSS delay from our static module against those from $AV2$.

Figure 3.11(a) and Figure 3.11(b) present MH-to-MH and MSS-to-MSS delays respectively under uniform communication pattern and small message size. The result shows that our static module can reduce the MH-to-MH delay by as much as 18.4%, and 20.7% for MSS-to-MSS delay.

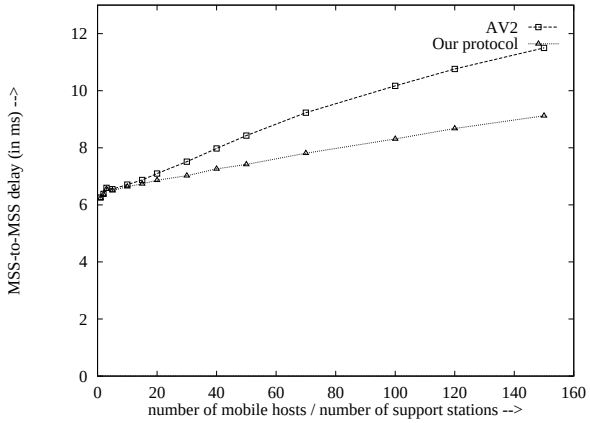
Figure 3.12(a) and Figure 3.12(b) present MH-to-MH and MSS-to-MSS delays respectively under uniform communication pattern and large message size. The result shows that our static module can reduce the MH-to-MH delay by as much as 11.02%, and 18.7% for MSS-to-MSS delay.

Figure 3.13(a) and Figure 3.13(b) present MH-to-MH and MSS-to-MSS delays respectively under nonuniform communication pattern and small message size. The result shows that our static module can reduce the MH-to-MH delay by as much as 18.9%, and 20.9% for MSS-to-MSS delay.

Figure 3.14(a) and Figure 3.14(b) present MH-to-MH and MSS-to-MSS delays respectively under nonuniform communication pattern and large message size. The result shows that our static module can reduce the MH-to-MH delay by as much as 12.11%, and 19% for MSS-to-MSS delay.

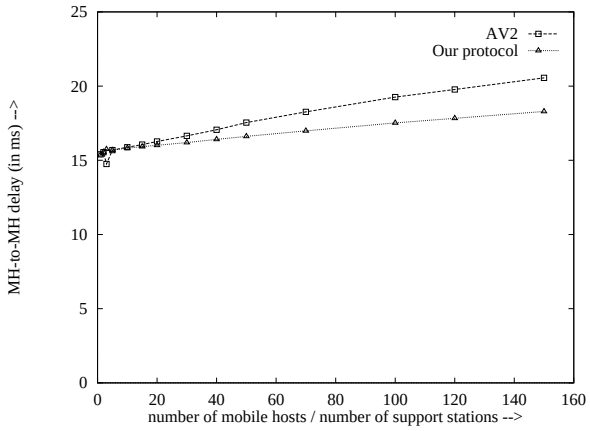


(a)

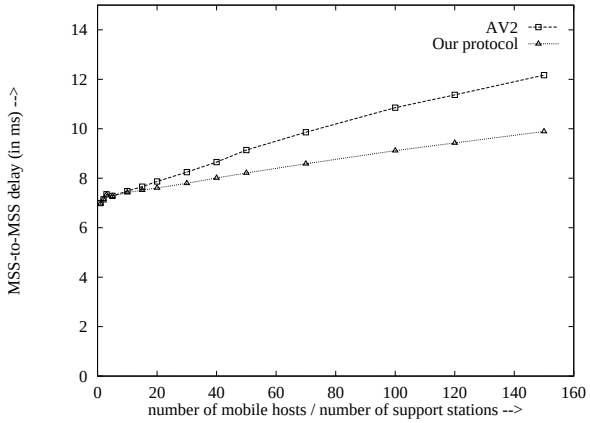


(b)

Figure 3.11: Delay under uniform communication pattern and small message size.

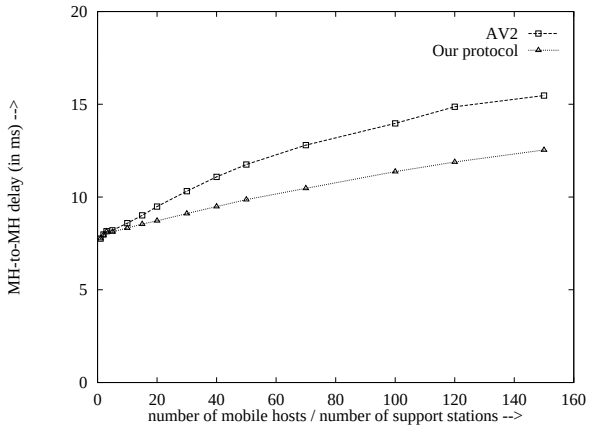


(a)

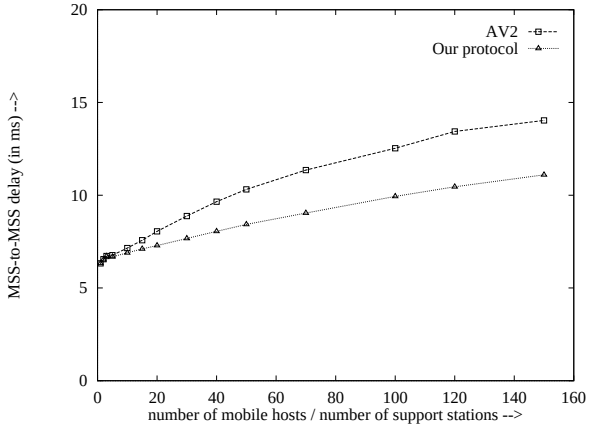


(b)

Figure 3.12: Delay under uniform communication pattern and large message size.

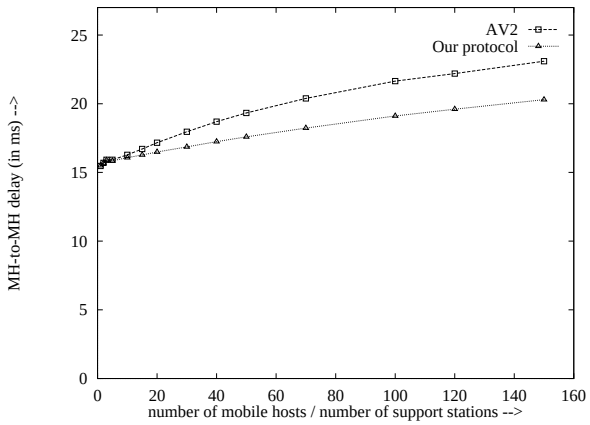


(a)

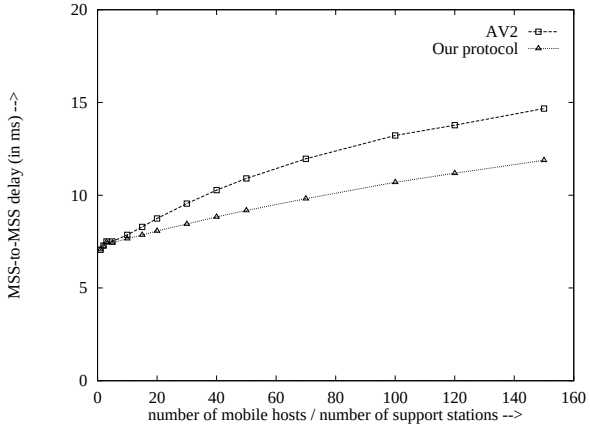


(b)

Figure 3.13: Delay under nonuniform communication pattern and small message size.



(a)



(b)

Figure 3.14: Delay under nonuniform communication pattern and large message size.

Chapter 4

Implementation

In this chapter, we describe the implementation of our channel allocation and causal message delivery layers.

4.1 Overview

We show one can use our proposed framework to build a simple shared object application. Our application supports two consistency criteria: Causal Consistency(CC) and Causal Serializability(CS) [RTKA96].

Our framework and the application are implemented in C++ on Sun Solaris 2.6. Messages exchanged between processes in the framework and the application are implemented using BSD Stream Sockets. We use POSIX PThreads when multi-threaded computation is needed. Graphic User Interface on mobile hosts is implemented using OSF/Motif Widget. Figure 4.1(a) shows the structure of our framework on each mobile host and support station.

Each mobile support station is implemented as a server process waiting for messages sent from other support stations or mobile hosts. Once a message is received, it is processed and then passed onto the appropriate destination. Each mobile host is a multi-threaded process. The first thread sleeps until the arrival of the message. The second thread waits for the input from the user. Our message

hierarchy are presented in Figure 4.1(b).

The rest of the chapter is organized as follows. The shared object model we consider is described in Section 4.2. Section 4.3 and Section 4.4 outline Causal Consistency and Causal Serializability, respectively. We discuss how we implement our application in Section 4.5.

4.2 Shared Object Model

We consider the system where mobile host processes, $MH_1, MH_2, \ldots, MH_n$, interacting through a finite set O of shared objects. Each object $o \in O$ can be accessed by a read or a write operation. The write operation assigning the new value v into object o is denoted $w(o)v$. The read operation on the object o returning value v is denoted $r(o)v$. Each MH_i executes transactions. A transaction t is a collection of read and write operations. Similar to [RTKA96], we assume that within each transaction the process first reads shared objects, then executes internal computation, and finally issues write operations on shared objects. Let $R(t)$ and $W(t)$ denote the set of objects read and written, respectively, by transaction t.

Let h_i denote the set of transaction executions issued by MH_i and $\rightarrow_i$ be the total order relation on transactions issued by MH_i. Thus, a history of a shared object system is a partial order $\hat{H} = (H, \rightarrow_H)$ such that:

- $H = \bigcup_i h_i$
- $t_1 \rightarrow_H t_2$ if
  1. $\exists P_i : t_1 \rightarrow_i t_2$
  2. $\exists w(x)v, r(o)v$ such that $w(o)v \in t_1$ and $r(o)v \in t_2$
  3. $\exists t_3 : t_1 \prec_H t_3$ and $t_3 \prec_H t_2$

Two transactions t_1 and t_2 are concurrent in $\hat{H}$ if $\neg(t_1 \rightarrow_H t_2)$ and $\neg(t_2 \rightarrow_H t_1)$.

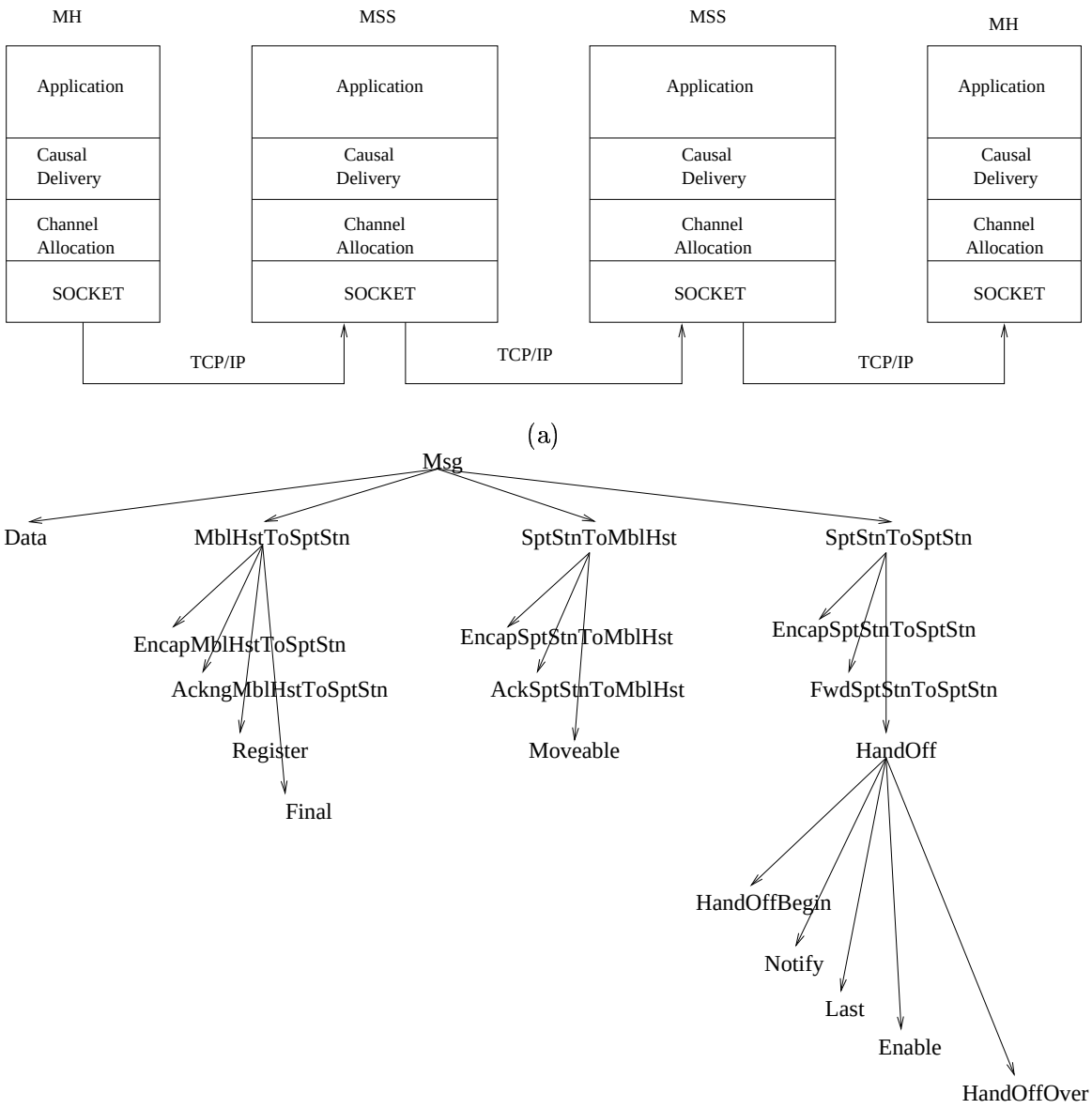


Figure 4.1: (a) Implementation Diagram, (b) Message Hierarchy

In shared object systems, a consistency criterion defines which is the value that must be returned to a process when it reads an object. The protocol that implements a consistency criterion describes how processes have to be synchronized in order to ensure that they satisfy the consistency criterion. *Serializability* is one of the most used consistency criterion in transactional systems. It requires that all processes have the same sequential view of the computation. This view is defined as a total order on operations issued by processes and an execution is correct if any read of an object gets the *last* value previously written into the object. The word "last" refers to the total order of operations defined by the common view. Consequently, strong synchronization among processes must be used to implement serializability. This may not be desirable in the system where communication between processes is not possible all the time (mobile hosts in disconnection mode).

Many applications, that support asynchronous interactions among widely distributed users, do not require strong consistency from the system. Causal consistency (CC) is the consistency criterion that meets the consistency requirement of such applications without the strong synchronization among processes. Specifically, two concurrent write operations can be perceived in different orders by different processes.

Another consistency criterion, causal serializability (CS), lies between serializability and CC. It is strong enough to satisfy a wide range of applications such as inventory control, distributed dictionaries, or cooperative work [RTKA96]. CS is CC plus the following constraint: all transactions writing into the same object must be perceived by all processes in the same sequential order.

In the following, we will provide the definition and the implementation of CC and CS.

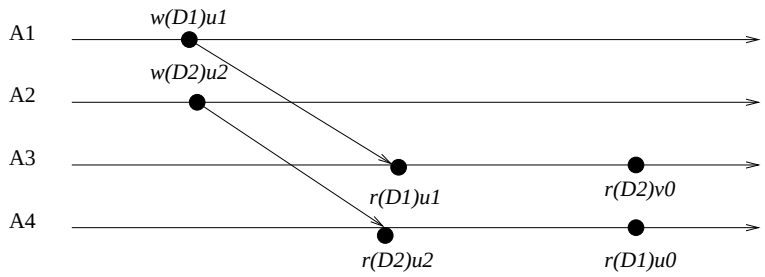


Figure 4.2: Causally Consistent History

4.3 Causal Consistency(CC)

CC allows each process to have its own sequential view of the execution $\hat{H}$ as long as the individual views preserve the causality relation $\rightarrow_H$. CC improves autonomy because the process is not required to agree with the views of others.

History $\hat{H}_1$ shown in Figure 4.2 is only possible in the causally consistent system. $\hat{H}_1$ consists of four authors, A_1, A_2, A_3, A_4, sharing documents D_1 and D_2. These two documents are written concurrently by A_1 and A_2. For example, D_1 could be the problem description and D_2 could be the motivation for the problem. As shown in the execution, A_3 is able to read the update u_1 of D_1 made by A_1 but the update u_2 by A_2 to D_2 is not seen by A_3 even after it reads D_1. Author A_4 sees the updates in the reverse order; it can read changes made to D_2 by A_2 but not the updates made by A_1. These executions are acceptable because the two documents are written concurrently and hence neither A_3 nor A_4 makes any assumptions about which document will be written first. CC exploits this and allows A_3 and A_4 to see these updates in different orders. On the other hand, serializability will not allow this execution because they require that the execution of all operations should be serializable. Disallowing such executions, which are acceptable in the application domain, requires unnecessary and expensive synchronization that is avoided by CC.

We use the protocol presented in [RTKA96] to implement CC in our application. When a process executes a transaction, it atomically does the following

sequence of actions. For each transaction t,

1. reads the value of all objects in $R(t)$.
2. performs the computation not involving shared objects.
3. updates all the objects in $W(t)$.
4. causally broadcast update message containing the new value of objects in $W(t)$.

4.4 Causal Serializability(CS)

With CC, when two update transactions that write into the same object are concurrent, they can be ordered differently by two processes in their views of the execution. This could lead to different final states of the system according to different processes. CS prevents such a possibility by adding the following constraint to CC. All transactions that update the same object must be perceived in the same order by all processes. This constraint ensures that, for each object, there is a unique "last" value on which all processes agree.

The history shown in Figure 4.3 is causally serializable. This is because each process has its own sequential view that preserves the causality relation $\rightarrow_H$. The view for P_1 is T_a, T_b, T_d. P_2's view is T_a, T_b, T_c, T_d. And P_3's view is T_a, T_d, T_e, T_b. Also, any pair of transaction writing into the same object are ordered in the same way (T_aT_b for x and T_aT_d for y) in the view of each process.

We again use the implementation presented in [RTKA96] in our application. Each shared object is associated with a token. In transaction t, the process must obtain all the tokens for each object in $W(t)$, and retain these tokens until it has broadcast the update message. To synchronize the token acquisition process, we assign a support station as the token coordinator. When a mobile host needs a token

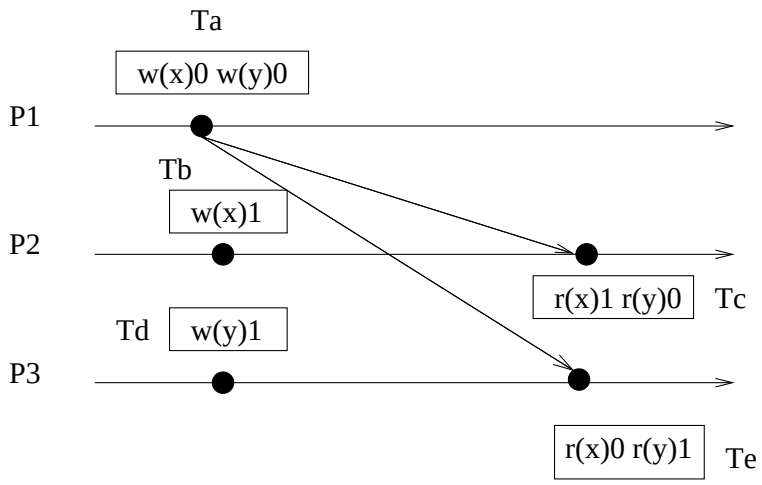


Figure 4.3: Causally Serializable History

for an object, it sends a request via its local support station to the coordinator. The token will be granted in the first-come first-serve basis.

4.5 The Application

Our application works in the fully replicated environment where all mobile hosts have copies of all the objects, and the user can choose between causal consistency or causal serializability. We allow only one operation for each transaction executed by each mobile host process.

The framework provides the following API for the application.

- `readObj()` : It implements *read* operation, and returns the value of the two integers.
- `writeObj()` : It implements *write* operation on the specified object under causal consistency criterion.
- `seqWriteObj()` : It implements *write* operation on the specified object under causal serializability criterion.

The framework also provides the API for *move* operation. This is to simulate mobility of mobile hosts. When invoked, the handoff protocol described earlier is executed.

Chapter 5

Causality Tracking

In this chapter, we present an efficient way to capture causality and concurrency between events in mobile computing. The material presented here also appears in [GS01, GS02].

5.1 Introduction

Determining order relationship between events in a distributed computation is a fundamental problem with applications in distributed monitoring systems and fault-tolerance. For example, it is used to provide visualizations of the computation for debugging in systems such as POET [KBTB97], XPVM [KG95], and Object-Level Trace [IBM]. It is also used in the area of global property evaluation [Fid89, GW94]. In the area of fault-tolerance, the order relationship is used to determine if a process is *orphan* and needs to be rolled back [SY85, DG96].

A distributed computation has been widely modeled as a partially ordered set (poset) $(E, \rightarrow)$ where E is the set of events in the computation and $\rightarrow$ is the *happened before* relation [Lam78]. Fidge [Fid89] and Mattern [Mat89] independently introduced vector clocks to timestamp events such that happened before relationship between any two events can be determined by examining their timestamps. In particular, in a distributed computation of N processes, vector timestamps provide

the following property:

$$\forall e, f \in E : e \rightarrow f \iff v(e) < v(f)$$

where $v(x)$ is the N-dimensional vector timestamp of any event x. In other words, the poset of events is isomorphic to the set of vectors in dimension N. However, vector clock mechanism does not scale well because it imposes $O(N)$ of local storage on each process and $O(N)$ message overhead in a system with N processes. Due to the limited resource on mobile devices and wireless links, it is natural to ask if there is an alternative mechanism with lower overhead.

Our first proposal is based on drawing connections between vector clocks used in distributed computing and dimension theory of partially ordered sets [Tro92]. The dimension of a partially ordered set (poset), first introduced by Dushnik and Miller [DM41], is defined as the least number of total orders such that the partial order is the intersection of these total orders. One of the advantages of this concept is that it provides an encoding scheme (or a timestamping scheme) for a partial order. If the dimension of a partial order on n elements is k, then each element can be assigned a code of size $k \log n$ such that the ordering between any two elements can be derived in k comparisons. Essentially, each element is represented by a k-tuple representing its position in each of the k orders.

We introduce the concept of *string dimension* which leads to a more efficient encoding of partial orders. In particular, the lower number of bits is typically required to encode when using string dimension. We show that the string dimension of a poset is exactly equal to the dimension of the poset whenever the string dimension is at least 2. This establishes a relationship between dimension theory and vector clock mechanisms which are more like strings. The efficient encoding scheme resulting from the concept of *string* is given.

The first lower bound argument on the size of the vector clocks is due to Charron-Bost [CBMT96]. Her result states that for all N, there exists a computa-

tion on N processes such that any assignment of events to $\mathcal{R}^k$ which captures the happened before relation (and its complement) must have $k \geq N$. We use standard results in dimension theory to derive results about vector clocks. We show that the theorem by Charron-Bost is a corollary of a result by Dushnik-Miller. Although, these results show that in the worst case the timestamps may require N-dimensional vector clocks, they do not exclude timestamps which use less than N coordinates for interesting subset of computations on N processes.

Our second proposal is based on this observation. We show that timestamping can be done more efficiently in distributed computations that uses *synchronous* messages. It is important to note that vectors of size equal to the string dimension of the poset are necessary and sufficient for timestamping events. However, timestamps that are determined using dimension theory cannot be used in an online manner because the knowledge of the entire poset is necessary to determine a realizer. Further, given a poset, the problem of determining the size of the smallest realizer is NP-complete [Yan82]. We present both online and offline algorithms for timestamping in synchronous computations.

This chapter is organized as follows. Related work is presented in Section 5.2. Section 5.3 provides background on poset and dimension theory. In Section 5.4, we introduce the concept of *string dimension*. The efficient encoding scheme resulting from the concept of *string* is discussed in Section 5.5. In Section 5.6, we use standard results in dimension theory to derive results about vector clocks. In Section 5.7, we show that timestamping can be done more efficiently in distributed computations that uses *synchronous* messages.

5.2 Related Work

In the area of dimension theory, Bouchet [Bou84] and Trotter [Tro92] introduced a generalization of the original dimension by restricting the length of chains used

in the realizer. This new dimension parameter is called k-dimension (denoted by $dim_k(P)$), when only the chains of length k are allowed in the realizer of P. Habib-Huchard-Nourine [HHN95] went further by allowing chains of different length in the realizer of the poset. They defined a new dimension parameter called *encoding dimension*. The encoding dimension of a poset P, denoted by $edim(P)$, is the least integer t such that $t = \sum_{i=1}^{i=p} \lceil \log_2 k_i \rceil$ and P can be embedded into $K_1 \times K_2 \times \ldots K_p$, where K_i denotes a chain of length k_i.

Different implementations of Fidge [Fid89] and Mattern [Mat89]'s vector clock have been proposed. Singhal and Kshemkalyani's [SK92] approach reduces the amount of data sent over the network. This is possible because of the increase in the amount of data stored by each process. Fowler and Zwaenepoel [FZ90] proposed an implementation where each process only keeps direct dependencies on others. Thus, only one scalar is required to represent a vector clock. However, for capturing transitive causal relations, it is necessary to recursively trace causal dependencies. This technique is therefore more suitable for applications where precedence test can be performed off-line.

Torres-Rojas and Ahamad [TRA96] introduced a class of scalable vector clocks called Plausible Clocks. It is scalable because it can be implemented using fixed-length vectors. Plausible Clocks do not characterize causality completely, that is, they do not guarantee that certain pairs of concurrent events will not be ordered. As a result, plausible clocks are useful for any application where imposing orderings on some pairs of concurrent events have no effects on the correctness of the results. Mutual consistency protocols for shared objects are examples of applications that can use plausible clocks.

Ward [War00] presented an algorithm to create vector timestamps whose size can be as small as the dimension of the partial order of execution. The algorithm incrementally builds a realizer using Rabinovitch and Rival's Theorem [RR79], and

then creates timestamp vectors based on that realizer. Therefore, vector timestamps that have already been assigned to events may have to be changed later. Further, all timestamps may not be of the same length. This leads to a complicated precedence test. Moreover, each coordinate is required to be a real number. Our algorithm does not suffer from any of these disadvantages.

A hierarchical cluster algorithm for online, centralized timestamp was presented in [WT01]. The algorithm is based on the fact that events within a cluster can only be causally dependent on events outside the cluster through receive events from transmissions that occurred outside the cluster. The precedence-test method in this algorithm is $O(c)$ where c is the size of the cluster.

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

5.3 Background

5.3.1 Partially Ordered Sets

A pair (X, P) is called a partially ordered set or poset if X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. We call X the *ground set* while P is a *partial order* on X. We write $x \leq y$ and $y \geq x$ in P when $(x, y) \in P$. Also, $x < y$ and $y > x$ in P means $x \leq y$ in P and $x \neq y$.

We use *Hasse diagrams*[1] to represent finite posets. If $x < y$ in P, then x appears lower than y in the diagram.

Let $x, y \in X$ with $x \neq y$. If either $x < y$ or $y < x$, we say x and y are *comparable*, and write $x \perp y$. On the other hand, if neither $x < y$ nor $x > y$, then

[1] The formal definition of Hasse diagrams can be found in [DP91].

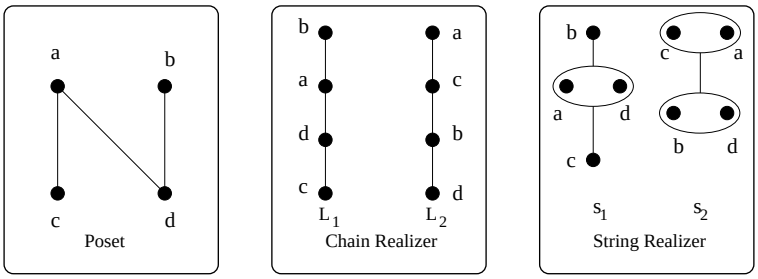


Figure 5.1: A poset (X, P)

we say x and y are incomparable, and write $x||y$. A poset (X, P) is called *chain* if every distinct pair of points from X is comparable in P. Similarly, we call a poset an *antichain* if every distinct pair of points from X is incomparable in P. A point $x \in X$ is called a *maximal* point (minimal point) if there is no point $y \in X$ with $x < y$ in P ($x > y$, respectively). We denote the set of all maximal points by $max(X, P)$, while $min(X, P)$ denotes the set of all minimal points.

A chain C of a poset (X, P) is a *maximum chain* if no other chain contains more points than C. We use similar definition for *maximum antichain*. The height of the poset P, denoted by $height(P)$, is the number of points in the maximum chain. Similarly, the width of the poset P, denoted by $width(P)$, is the number of points in a maximum antichain. We say (X, P) and (Y, Q) are isomorphic, if there exists a $1-1$ and onto map $f : X \longrightarrow Y$ so that $x_1 \leq x_2$ in P if and only if $f(x_1) \leq f(x_2)$ in Q.

5.3.2 Dimension

A family $\mathcal{R} = \{L_1, L_2, \ldots, L_t\}$ of linear orders on X is called a *chain realizer* of a poset (X, P) if $P = \cap\mathcal{R}$. $x < y \in L_i \cap L_j$ if $x < y$ in both L_i and L_j. We also say that $\mathcal{R}$ *realizes* (X, P). Figure 5.1 shows a poset (X, P) in which $\{L_1, L_2\}$ realizes (X, P).

It can be shown [Tro92] that $\mathcal{R}$ is a realizer of P iff for every $x, y \in X$ with

$x \parallel y$ (x incomparable to y) in P, there exists distinct integers i, j with $1 \leq i, j \leq t$ for which $x < y$ in L_i and $y < x$ in L_j. In the following, we write $x <_c y$ when $x < y$ in L_c,

Definition 29 *[Tro92] For any poset (X, P), the dimension of (X, P), denoted by $dim(X, P)$, is the least positive integer t for which there exists a family $\mathcal{R} = \{L_1, L_2, \ldots, L_t\}$ of linear extensions of P so that $P = \cap\mathcal{R} = \cap_{i=1}^{t} L_i$.*

The dimension of the poset in Figure 5.1 is 2. The concept of dimension provides us a way to encode a partial order. The elements of a partial order with dimension t can be encoded with a t-dimensional vector as follows. For any element x, the vector v_x is defined as follows: $v_x[i]$ = number of elements less than x in L_i, for $1 \leq i \leq t$. Given code for two elements v_x and v_y, we have the following order:

$$v_x < v_y \iff \forall i : v_x[i] < v_y[i] \tag{5.1}$$

For example, the code for a and b in the poset in Figure 5.1 is $(2, 3)$ and $(3, 1)$ based on the realizer. Based on the code and (5.1), it can be easily determined that a and b are concurrent. We call the order given by (5.1) the *chain order.*

The dimension of a poset can be arbitrarily large. Consider a poset (X, P) where $X = \{a_1, a_2, \ldots, a_n\} \cup \{b_1, b_2, \ldots, b_n\}$, and $a_i < b_j$ in P if and only if $i \neq j$, for $i, j = 1, 2, \ldots, n$. This class of posets is known as *the standard example* and denoted by S_n. Figure 5.2 shows the diagram for S_5. The following Theorem is due to Dushnik and Miller [DM41].

Theorem 30 *[DM41] $dim(S_n) = n$.*

Let $L_i = [a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, b_i, a_i, b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_n]$, where a_1 is the lowest element, and b_n is the highest element in chain L_i Then $\mathcal{R} = \{L_1, L_2, \ldots, L_n\}$ is a realizer of S_n.

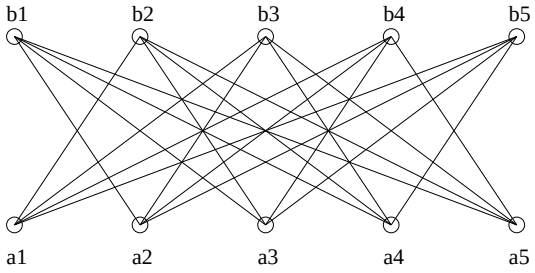


Figure 5.2: S_5

5.4 String and String Dimension

In Section 5.3 we saw that classical dimension theory provides lower bounds on the dimension of vectors when the comparison is based on the *chain order*. On the other hand, the vector clocks in distributed computing use vector ordering given by the following (5.2) which we call *vector order*.

$$\begin{aligned} u < v \quad \equiv \quad & \forall k : 1 \leq k \leq N : u[k] \leq v[k] \wedge \\ & \exists j : 1 \leq j \leq N : u[j] < v[j] \end{aligned} \tag{5.2}$$

Consider a distributed system in which the code of elements is determined in a decentralized fashion. In this case the relationship between two events may not be known globally. Thus, if event e happened before f, this relationship may be known only to a single process. From the perspective of other processes, e and f may be indistinguishable (for example, when both are internal to the process). This is more easily captured in the vector order where a vector u is deemed as smaller than vector v even when u is smaller than v in just one component and same in all the other components. Since chain order requires that all the coordinates in code of event e are strictly less than all the respective coordinates in code of event f, it is difficult to use chain order in a distributed system. In this section, we generalize the concepts in dimension theory so that the ordering used between codes is identical to (5.2).

We first give the definition of a *string*.

Definition 31 (string) *A poset (X, P) is a string if and only if $\exists f : X \rightarrow \mathcal{N}$ (the*

set of natural numbers) such that $\forall x, y \in X : x < y$ *iff* $f(x) < f(y)$

The set of elements in a string which have the same f value is called a *knot*. For example, a poset (X, P) where $X = \{a, b, c, d\}$ and $P = \{(a, b), (a, c), (a, d), (b, d), (c, d)\}$ is a string because we can assign $f(a) = 0, f(b) = f(c) = 1$, and $f(d) = 2$. Here, b and c are in the same knot. The difference between a chain and a string is that a chain requires existence of a *one-to-one* mapping such that $x < y$ iff $f(x) < f(y)$. For strings, we drop the requirement of the function to be *one-to-one*. We represent a finite string by the sequence of knots in the string. Thus, P is equivalent to the string $\{(a), (b, c), (d)\}$.

A chain is a string in which every knot is of size 1. An anti-chain is also a string with exactly one knot. Note that a string drops the distinction between elements which have the same order relationship with all other elements. Thus, two elements x and y have the same code $f(x) = f(y)$ iff for any element z, (1) $x < z$ iff $y < z$, and (2) $z < x$ iff $z < y$. This is a more natural concept for ordered sets.

A string gives more efficient encoding of the partial order than the use of chains. At an extreme, the range of f may be finite even when the domain of f is infinite. For example, the following order {all even numbers} < {all odd numbers} on natural numbers can be encoded by assigning 0 to all even numbers and 1 to all odd numbers. Such a poset cannot be assigned codes using the classical dimension theory.

We write $x \leq_s y$ if $x \leq y$ in string s, and $x <_s y$ if $x < y$ in string s.

Definition 32 (String Realizer) *For any poset* (X, P)*, a set of strings* $\mathcal{S}$ *is called a string realizer iff* $\forall x, y \in X : x < y$ *in* P *if and only if*

1. $\forall s \in \mathcal{S} : x \leq_s y$*, and*
2. $\exists t \in \mathcal{S} : x <_t y$.

The definition of less-than relation between two elements in the poset based on the strings is identical to the less-than relation as used in vector clocks. This is one of the motivation for defining string realizer in the above manner. A string realizer for the poset in Fig. 5.1 is given by two strings

$$s_1 = \{(c), (d, a), (b)\} \qquad s_2 = \{(d, b), (c, a)\}$$

There are two important differences between definitions of string realizers and chain realizers. First, if $\mathcal{R}$ is a chain realizer of a poset P, then P is simply the intersection of linear extensions in $\mathcal{R}$. This is not true for a string realizer (see Fig. 5.1). Secondly, all the total orders in $\mathcal{R}$ preserve P, i.e., $x < y$ in P implies that $x < y$ in all chains in $\mathcal{R}$. This is not true for string realizer. For example, $d < a$ in poset P of Fig. 5.1, but (d, a) appears as a knot in the string s_1. We are only guaranteed that a will not appear lower than d in any string - they may appear in the same knot.

Now, analogous to the dimension we define

Definition 33 (String Dimension) *For any poset (X, P), the string dimension of (X, P), denoted by $sdim(X, P)$, is the size of the smallest set of strings $\mathcal{S}$ such that $\mathcal{S}$ is a string realizer for (X, P).*

Example 1 *Consider the standard example S_n. The following function f can be used to create a string realizer of S_n. For all $k, i = 1, 2, \ldots, n$,*

$$f_k(a_i) = \begin{cases} 0 & \text{if } k \neq i \\ 1 & \text{otherwise} \end{cases}$$

$$f_k(b_i) = \begin{cases} 0 & \text{if } k = i \\ 1 & \text{otherwise} \end{cases}$$

For example,

$$a_1 = (1, 0, 0, \ldots, 0), \quad b_1 = (0, 1, 1, \ldots, 1)$$
$$a_2 = (0, 1, 0, \ldots, 0), \quad b_2 = (1, 0, 1, \ldots, 1)$$

In this example, the length of each string is 2 *and thus each element requires only* n *bits for encoding. If we use classical dimension based on total orders, each element would require* $n * \log n$ *bits.*

Example 2 *Consider the poset* (X, P) *as follows.*
$X = \{\emptyset, \{a\}, \{b\}, \{a, b\}, \{a, c\}, \{a, b, c\}\}$
$P = \{(A, B) \in X \times X : A \subseteq B\}$.

A string realizer for the poset can be obtained as follows. For each set $A \in X$*, we use a bit vector representation of the set* A*. Thus,* $\{a, c\}$ *is represented by* $(1, 0, 1)$ *and the set* $\{a, b\}$ *is represented by* $(1, 1, 0)$*. This representation gives us a string realizer with three strings such that every string has exactly two knots.*

We now establish the relationship between string dimension and chain dimension. It may appear, at first, that the string dimension of a poset may be much smaller than the chain dimension. However, this is not the case as shown by the following result.

Theorem 34 (Equivalence Theorem) *For any poset* (X, P) *such that* $sdim(X, P)$ *is greater than* 1*,*

$$sdim(X, P) = dim(X, P)$$

Proof: There are two cases.
$\underline{sdim(P) \leq dim(P)}$.

It is sufficient to show that for any chain realizer of size k, there exists a string realizer of equal or smaller size. Given a chain realizer $\mathcal{C}$, we construct the string realizer as follows. Each chain is simply viewed as a string. Our obligation is to show that the order generated from the string realizer is the same as the one based on chain realizer (recall that the definition of *less than* for string realizer is different from *less than* in a chain realizer.) In this proof, let $x \mapsto y \iff \forall s \in \mathcal{S}$:

$x \leq_s y \ \wedge \ \exists t \in \mathcal{S} : x <_t y$. It is sufficient to show that $x < y \iff x \mapsto y$. First, we show that $x < y \ \Rightarrow \ x \mapsto y$.

$$\begin{aligned} x < y \quad &\Rightarrow \quad \forall c \in \mathcal{C} : x <_c y \\ &\Rightarrow \quad \forall s \in \mathcal{S} : x <_s y \\ &\Rightarrow \quad \forall s \in \mathcal{S} : x \leq_s y \ \wedge \ \exists t \in \mathcal{S} : x <_t y \\ &\Rightarrow \quad x \mapsto y \end{aligned}$$

Next, we show that $\neg(x < y) \ \Rightarrow \ \neg(x \mapsto y)$. There are two cases.

$$\begin{aligned} \text{case A:} \quad y < x \quad &\Rightarrow \quad y \mapsto x \quad \text{(From case I)} \\ &\Rightarrow \quad \neg(x \mapsto y) \\ \text{case B:} \quad x \parallel y \quad &\Rightarrow \quad \neg(x < y) \ \wedge \ \neg(y < x) \\ &\Rightarrow \quad \exists c \in \mathcal{C} : x <_c y \ \wedge \ \exists d \in \mathcal{C} : y <_d x \\ &\Rightarrow \quad \exists s \in \mathcal{S} : x <_s y \ \wedge \ \exists t \in \mathcal{S} : y <_t x \\ &\Rightarrow \quad \neg(y \mapsto x) \ \wedge \ \neg(x \mapsto y) \\ &\Rightarrow \quad \neg(x \mapsto y) \end{aligned}$$

$\underline{dim(P) \leq sdim(P)}$.

Given a string realizer of P, $\mathcal{S}$, we construct the corresponding chain realizer. We achieve this by untying knots of the string to form a chain.

First consider the case when two elements x and y belong to the same knot in all strings. We will combine these elements into one element say z. After finding the chain realizer of the new set, we replace z with x and y. Further, in one chain we keep x less than y and in another chain we keep y less than x. Observe that we can do this because there are at least two chains due to our assumption of $sdim(P) \geq 2$.

Now assume that there are no two elements as in the first case. Consider any knot $\{x_1, x_2, \ldots, x_m\}$ in any string s_1. Now we determine for all pairs (x_i, x_j) of the elements in the knot.

1. If $(\forall s \in \mathcal{S} - \{s_1\} : x_i \leq_s x_j) \ \wedge \ (\exists t \in \mathcal{S} - \{s_1\} : x_i <_t x_j)$, then we get $x_i < x_j$.

   or

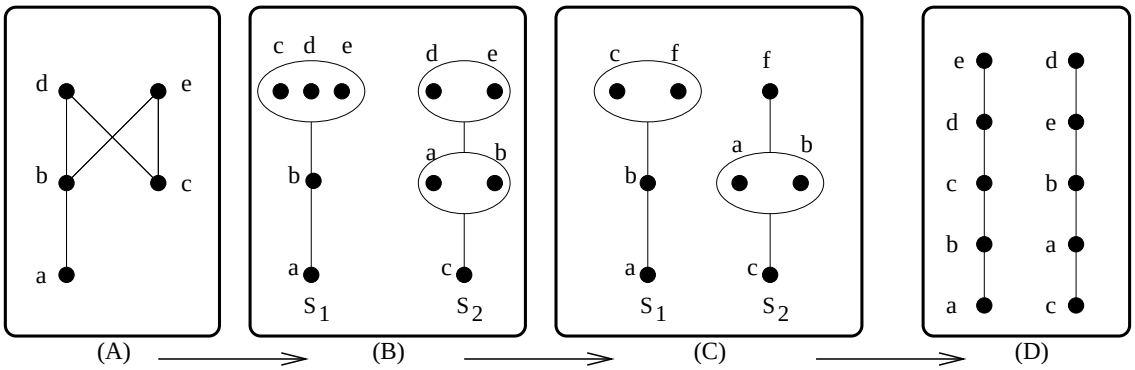


Figure 5.3: An example of untying mechanism.

2. If $\exists s, t \in S - \{s_1\} : (x_i <_s x_j) \ \wedge \ (x_j <_t x_i)$, then we get $x_i \parallel x_j$.

Then, we can untie this knot by performing the topological sort. By repeating this process, all knots on s_1 can be untied, and we obtain the chain. ■

Figure 5.3 shows an example of this untying mechanism. Since d and e appear in the same knot in all strings, we first combine d and e into one element f. As a result we get strings in Figure 5.3(C). We then untie the knot (c, f) by keeping c less than f in s_1 and untie the knot (a, b) in s_2 by keeping a less than b. We now have the chain order. Now we replace f by d and e, keeping d less than e in s_1 and e less than d in s_2 to get the chains in Figure 5.3(D).

5.5 Encoding Partial Orders

The concept of string realizer has the advantage over chain realizer that it generally requires less number of bits to encode a partial order using string realizer. Formally, consider the following problem. Given a partial order (X, P), define a coding function $code : X \longrightarrow \{0,1\}^k$ and a binary relation $<$ on codes such that $\forall x, y \in X : x < y$ in $P \iff code(x) < code(y)$. Note that the order relation may be any arbitrary order (not necessarily vector order). The only requirement is that it can only use the bits in $code(x)$ and $code(y)$ to determine the order. It is clear

that any partial order can be coded using $log(n) + n$ bits per element as follows. For every element, we store a binary array of size n. Further, each element is assigned a unique index into the array. Let $index(x)$ be the index of x in $1..n$ and $x.v$ be the n bit array for element x. Then, we determine the order between x and y as follows. $x < y$ iff $(x.v[index(y)] < y.v[index(x)])$.

Using dimension theory, partial orders of lower dimensions can be encoded much more succinctly. If a partial order has dimension k, then it can be encoded using $k * log(n)$ bits. However, when the dimension is large (as for the standard example), this method may take up to $n/2 * log(n)$ bits per element.

String realizers typically result in a lower number of bits for encoding. From Theorem 34, we know that for coding purposes, the total number of coordinates based on total orders and strings are the same. The difference lies in the number of bits required to code a single coordinate. Given a string realizer R. If R has k strings each of length less than or equal to l, then (X, P) can be coded using $k * log(l)$ bits. l is clearly less than or equal to $|X|$. Depending upon the structure of the poset, $log(l)$ may be much smaller than $log(n)$ as seen for the case of the standard example.

In general, we have the following result.

Theorem 35 *Every partial order (X, P) on $n \geq 2$ elements can be encoded using a string realizer in at most* $\log(height(P) + 1) * width(P)$ *bits.*

Proof: For convenience, let $w = width(P)$. We use Dilworth's chain covering theorem which states that (X, P) can be partitioned into w chains $C_1, C_2, ..., C_w$. We then use the transitively reduced diagram of (X, P) with w processes as given by the chain decomposition. Further, we use Fidge and Mattern's algorithm to assign vector timestamp for each event when the poset diagram is viewed as a computation. These vector timestamps determine a string realizer with w coordinates such that

no coordinate is greater than $height(P) + 1$. ■

There is a small change in application of Fidge and Mattern's algorithm in above construction. Their algorithm assumes that initial events of all processes are incomparable and assigns the initial event at process i a vector timestamp as follows:

$$\forall j : j \neq i : v[j] = 0;$$

$$v[i] = 1;$$

In our construction (in the proof of Theorem 35), all the initial events of chains may not be incomparable. To solve this problem, it is sufficient to add a special initial event for each chain whose smallest event is not a minimal event in the partial order. For example, consider the poset in Figure 5.4. This poset can be decomposed into three chains $\{a, b, c\}, \{d, e\}$, and $\{f, g\}$. However, d is not a minimal element of the poset. Hence, to apply Fidge and Mattern's algorithm we may assume an event smaller than d which is incomparable to a and f in Process 2 with vector clock equal to $(0, 1, 0)$. Then, to compute the vector at d, we compute the maximum of vectors for a, f and $(0, 1, 0)$. Thus, the vector clock for all events can be derived as

$v(c) = (3, 0, 0); v(e) = (2, 2, 1); v(g) = (1, 1, 2);$

$v(b) = (2, 0, 0); v(d) = (1, 1, 1); v(f) = (0, 0, 1);$

$v(a) = (1, 0, 0)$

This results in the following string realizer:

$s_1 = \{(f), (a, d, g), (b, e), (c)\},$

$s_2 = \{(a, f, b, c), (d, g), (e)\}$, and

$s_3 = \{(a, b, c), (d, e, f), (g)\}.$

Observe that some strings may be longer than others and we need not use the same number of bits to encode positions in all the strings. The total number of

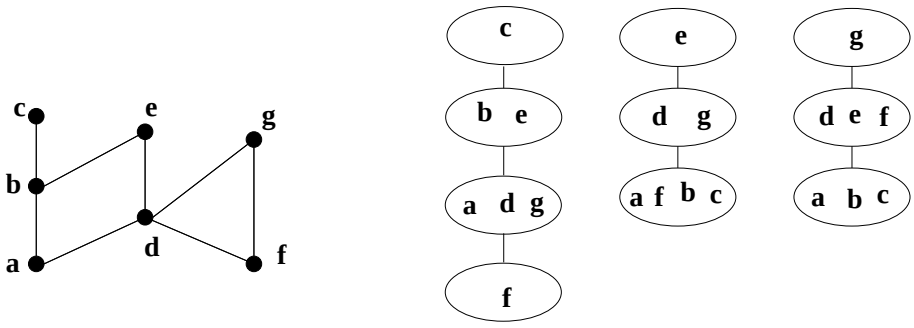


Figure 5.4: A Poset and its String Realizer

bits required for a realizer with t strings is

$$\sum_{i=1}^{i=t} \lceil log(length(s_i)) \rceil$$

We note here that Bouchet [Bou84] and Trotter [Tro92] introduced a generalization of the original dimension by restricting the length of chains used in the realizer. This new dimension parameter is called k-dimension (denoted by $dim_k(P)$), when only the chains of length k are allowed in the realizer of P.

The k-dimension of P, $k \geq 2$, is the smallest positive integer t for which P is isomorphic to a subposet of K^t (ie. K^t is the product of t chains of length k). Therefore, the 2-dimension is the size of the smallest hypercube in which P can be embedded.

Obviously [HHN95],

$$dim(P) \leq dim_{k-1}(P) \leq \ldots \leq dim_2(P)$$

One interesting question is to determine the smallest integer k, $2 \leq k \leq |P|$, such that $dim(P) = dim_k(P)$. Habib-Huchard-Nourine [HHN95] went further by allowing chains of different length in the realizer of the poset. They defined a new dimension parameter called *encoding dimension* as follows.

The encoding dimension of a poset P, denoted by $edim(P)$, is the least

integer t such that $t = \sum_{i=1}^{i=p} \lceil \log_2 k_i \rceil$ and P can be embedded into $K_1 \times K_2 \times \ldots K_p$, where K_i denotes a chain of length k_i.

It is shown in [HHN95] that when P is an antichain, then $edim(P) = 2 \log |P|$. This is equal to the number of bits required in the Dushnik-Miller's dimension. However, by using string realizers, we can use only one bit to encode each element in an antichain.

A key distinguishing feature of our work is that we allow order equivalent elements to have the same code. This is more natural concept for posets. Further, it allows hierarchical representation of orders. Two elements may have the same code at one level, but different at the other level when they are not distinguishable at coarser granularity but can be distinguished with finer granularity of the order. For example, in a distributed computation, all internal events between two external events may be assigned the same code at the coarser level of granularity.

5.6 Lower Bound on Dimension of Vector Clocks

As we have mentioned before, the definition of a string realizer is identical to the definition for vector clocks in distributed systems. A distributed computation on N processes can be modeled as a poset of events $(E, \rightarrow)$ of width N. Fidge and Mattern's vector clocks are simply string realizers of the poset $(E, \rightarrow)$. For example, consider the poset in Fig. 5.3 which has width two. We can view it as a computation on two processes, the first process executes events a, b and d in that order, and the second process executes c and e in that order. By viewing b and c as send events received at e and d respectively, we get the following vector clocks for all events:

$$v(a) = (1, 0); v(b) = (2, 0); v(d) = (3, 1);$$

$$v(c) = (0, 1); v(e) = (2, 2)$$

This corresponds to two strings

$$s_1 = \{(c), (a), (b, e), (d)\} \quad \text{and} \quad s_2 = \{(a, b), (c, d), (e)\}$$

This is a different string realizer than shown in Figure 5.3, but has the same dimension.

Now that we have established equivalence of dimension and string dimension for non-string posets, we can use existing results from dimension theory to prove results on dimension of vector clocks.

We first consider lower bounds on the (string) dimension of vector clocks. Charron-Bost [CB91] has shown that we require at least N-dimensional vector timestamps to capture concurrency in the distributed computation consisting of N processes. The proof is by constructing a computation in which any timestamping scheme with less than N coordinates is not able to capture concurrency accurately. The following proof uses dimension theory and our equivalence theorem.

Theorem 36 *For every N, there exists a distributed computation $(E, \rightarrow)$ on N processes such that any assignment from E to $\mathcal{N}^k$ that captures concurrency relation on E has $k \geq N$.*

Proof: The result is trivially true for N equal to 1. For any $N \geq 2$, consider the standard example S_N shown in Figure 5.2. Define a_i and $b_{(i \bmod N)+1}$ to be on process P_i. This computation is on N processes. By Dushnik and Miller's Theorem, this poset has dimension N. From Theorem 34, the computation has string dimension also equal to N. Any assignment from E to $\mathcal{N}^k$ that captures concurrency relation, results in a string realizer with k strings. Since the string dimension is N, it follows that $k \geq N$. ∎

Although this result proves that there cannot be a uniform timestamping mechanism of less than N coordinates, it does not exclude timestamping mechanism which may use less than N coordinates for a particular computation.

As an extreme example, consider a system of N processes, where $N > 3$. Assume that processes do not send any messages to each other. We can timestamp each event j on process i by the vector $v_i(j) = (i, n-i, j)$. It is easy to see that this timestamping mechanism captures concurrency relation accurately[2].

5.7 Efficient Vector Timestamps for Synchronous Computations

In this section, we show that timestamping can be done more efficiently in distributed computations that use *synchronous* messages. A message is called *synchronous* when the send is blocking, i.e., the sender waits for the message to be delivered by the receiver before executing further. Synchronous communication is widely supported in many programming languages and standards such as CSP [Hoa85], Ada Rendezvous, and synchronous Remote Procedure Calls (RPC). While programming using asynchronous communication allows higher degree of parallelism and is less prone to deadlocks, algorithms using synchronous message-passing are easier to develop and verify. Also, the implementation of asynchronous communication requires buffer management and flow control mechanisms. Implementation of synchronous messages requires that the sender wait for an acknowledgment from the receiver before executing further.

A computation that uses only synchronous messages is called a synchronous computation. It can be shown that a computation is synchronous if it is possible to timestamp send and receive events with integers in such a way that (1) timestamps increase within each process and (2) the sending and the receiving events associated with each message have the same timestamp. Therefore, the time diagram of the computation can be drawn such that all messages arrows are vertical [CBMT96] (see Figure 5.5).

Determining the order of messages is crucial in observing distributed systems.

[2]In fact, this partial order can be encoded using vector clocks of dimension 2.

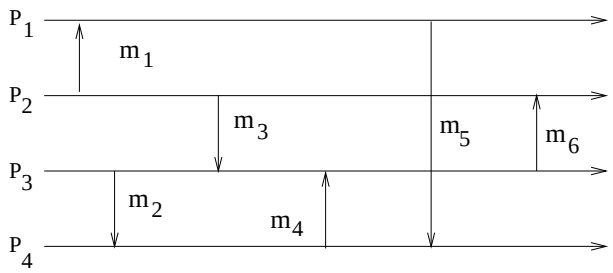


Figure 5.5: A synchronous computation with 4 processes.

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

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

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

In the example given in Figure 5.5, $m_1 || m_2$, $m_1 \rhd m_3$, $m_2 \mapsto m_6$, and $m_3 \mapsto m_5$. There is a synchronous chain between m_1 and m_5 of size 4.

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

$$m_1 \mapsto m_2 \iff v(m_1) < v(m_2) \tag{5.3}$$

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

$$u < v \iff \begin{cases} \forall k : 1 \leq k \leq t : u[k] \leq v[k] \wedge \\ \exists j : 1 \leq j \leq t : u[j] < v[j] \end{cases} \tag{5.4}$$

We call the relation given in Equation (5.4) *vector order*.

From Equations (5.3) and (5.4), one can determine if $m_1 \mapsto m_2$ by checking whether $v(m_1) < v(m_2)$. If $v(m_1)$ is not less than $v(m_2)$ and $v(m_2)$ is not less than $v(m_1)$, then we know that $m_1 || m_2$.

5.7.1 Online Algorithm

We now give an algorithm to assign $v(m)$ for a message m such that Equation (5.3) is satisfied. Whereas FM vectors are based on the idea of assigning one component for each process, our algorithm assigns one component to each *edge group*. We first define the notion of *edge decomposition* and *edge group*.

5.7.1.1 Edge Decomposition

The communication topology of a synchronous system that consists of N processes, $P_1, \ldots, P_N$, can be viewed as an undirected graph $G = (V, E)$ where $V = \{P_1, \ldots, P_n\}$, and $(P_i, P_j) \in E$ when P_i and P_j can communicate directly. Figure 5.6(a) gives the communication topology of a system in which every process can communicate directly with each other. Figure 5.6(b) gives the communication topology of another system in which not every pair of processes communicate directly with each other.

Some particular topologies that will be useful to us are the *star* and the *triangle* topologies defined below.

Definition 37 (Star) *An undirected graph $G = (V, E)$ is a star if there exists a vertex $x \in V$ such that all edges in E are incident to x. We call such a star as* rooted *at node x.*

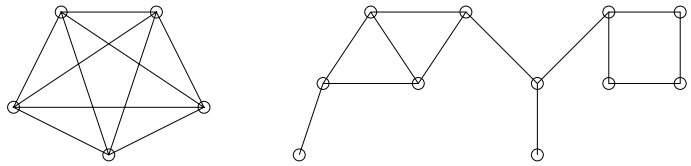


Figure 5.6: Examples of the communication topologies. (a) The system where every process can communicate directly with each other. (b) The system where not every pair of processes communicate directly with each other.

Definition 38 (Triangle) *An undirected graph $G = (V, E)$ is a triangle if $|E| = 3$, and these three edges form a triangle. We denote a triangle by a triple such as (x, y, z) denoting its endpoints.*

The star and triangle topologies are useful because messages in a synchronous computation with these topologies are always totally ordered.

Lemma 39 *The message sets for all synchronous computations in a system with $G = (V, E)$ as the communication topology are totally ordered if and only if G is a star or a triangle.*

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

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

■

Note that the above Lemma does not claim that message set cannot be totally ordered for a topology that is neither a star nor a triangle. It only claims that for every such topology there exists a synchronous computation in which messages do not form a total order.

Now based on the definitions of star and triangle graphs, we are ready to define the edge decomposition of G.

Definition 40 (Edge Decomposition) *Let $G = (V, E)$ be communication topology of a synchronous system. A partition of the edge set, $\{E_1, E_2, \ldots, E_d\}$, is called an edge decomposition of G if $E = E_1 \cup E_2 \cup, \ldots, \cup E_d$ such that*

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

We refer to each E_i in the edge decomposition as an edge group. In our algorithm, we will assign one component of the vector for every edge group. Note that there is possibly more than one decomposition for a topology. Our goal is to get the smallest possible decomposition. Consider a fully-connected system consisting of N processes. The first decomposition consists of $N - 3$ stars and 1 triangle. The second decomposition consists of $N - 1$ stars. Figure 5.7 presents the two decompositions of a fully-connected system with 5 processes.

The complete graph is the worst case for edge decomposition, resulting in $N-3$ stars and 1 triangle. In general, the number of edge groups may be much smaller than $N - 2$. Given a tree-based synchronous system consisting of 20 processes, Figure 5.8 shows how to decompose edges into three edge groups E_1, E_2, and E_3 where each group is a star.

We will discuss techniques for edge decomposition that minimize the number of edge groups in Section 5.7.1.3.

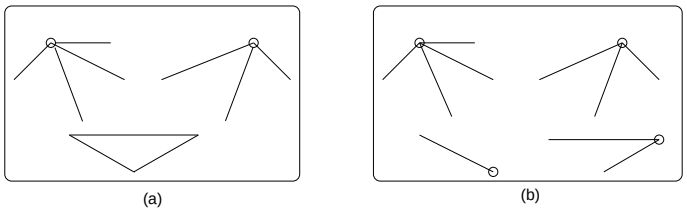


Figure 5.7: Edge decompositions of the fully-connected system with 5 processes. (a) The first decomposition consisting of 2 stars and 1 triangle. (b) The second decomposition consisting of 4 stars.

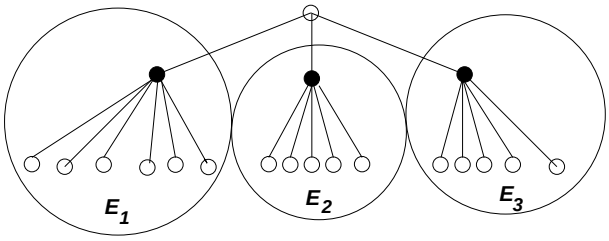


Figure 5.8: A tree-based computation with 20 processes.

5.7.1.2 Algorithm

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

The online algorithm is presented in Figure 5.9. Due to the implementation of synchronous message ordering [MG95], we assume that for each message sent from P_i to P_j, there exists an acknowledgement sent from P_j to P_i. Essentially, to timestamp each message, the sender and the receiver must first exchange their vector clocks. Then, each process computes the component-wise maximum between the local vector and the vector received (Line (5) and (9)). Finally, both the sender and the receiver increment the g^{th} element of their vectors where the channel that the message is sent belongs to the g^{th} group in the edge decomposition (Line (6) and (10)). The resulting vector clock is the timestamp of this message.

Figure 5.10 shows a sample execution of the proposed algorithm on a fully-

P_i ::

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

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

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

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

Figure 5.9: The Online Algorithm.

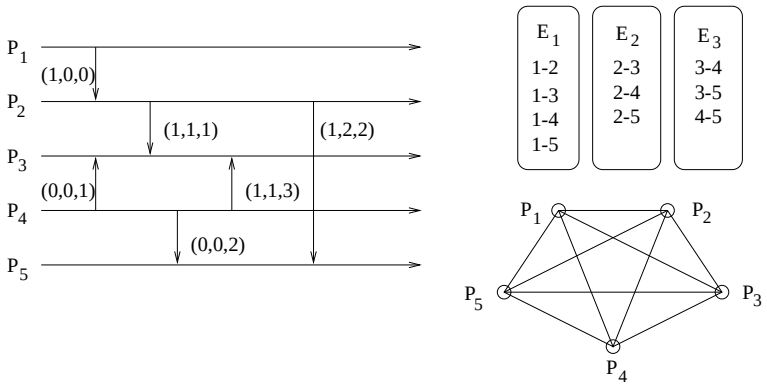


Figure 5.10: A synchronous computation with 5 processes, and its edge decomposition.

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

Next, we prove that our online algorithm creates vector timestamps for messages in synchronous systems such that these timestamps encode poset $(M, \mapsto)$. The channel in which a message m_x is sent through must be a member of a group in the

edge decomposition. We use $e(m_x)$ to denote the index of the group to which this channel belongs in the edge decomposition.

Lemma 41 $m_i||m_j \Rightarrow e(m_i) \neq e(m_j)$

Proof: Let $c_i(resp.c_j)$ be an edge in the topology graph G that corresponds to the channel on which m_i (resp. m_j) is sent. Since $m_i||m_j$, from Lemma 39, all messages in an edge group are totally ordered, we get that c_i and c_j must belong to different edge groups. Therefore, $e(m_i) \neq e(m_j)$. ■

Theorem 42 *Given an edge decomposition of a synchronous system, the algorithm in Figure 5.9 timestamps messages such that* $m_1 \mapsto m_2 \iff v(m_1) < v(m_2)$

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

$$m_1 \mapsto m_2 \Rightarrow v(m_1)[e(m_2)] < v(m_2)[e(m_2)] \tag{5.5}$$

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

We now show the converse,

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

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

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

We do a case analysis.

(Case 1: $m_2 \mapsto m_1$)

From Equation (5.5), by changing roles of m_1 and m_2, we get that $v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$.

(Case 2: $m_1 || m_2$)

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

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

From Lemma 41 and $m_1 || m_2$, $e(m_1) \neq e(m_2)$. Since m_2 is a minimal message by the initial assignment of the vector clock, both sender and the receiver have 0 as the component for $e(m_1)$ and the componentwise maximum also results in 0 for $e(m_1)$. Further, since $e(m_1) \neq e(m_2)$ the component for $e(m_1)$ is not incremented. Hence, $v(m_2)[e(m_1)] = 0$.

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

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

(Induction: $k > 1$)

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

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

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

$v(m_2)[e(m_1)] < v(m_1)[e(m_1)]$. ■

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

5.7.1.3 Good Edge Decompositions

As discussed in Section 5.7.1.2, the overhead of our algorithm is crucially dependent upon the size of the edge decomposition. Let $\alpha(G)$ denote the size of a smallest edge decomposition (note that there may be multiple edge decomposition of the same size). In our edge decomposition, we decompose the graph into stars and triangles. If we restricted ourselves to decomposing the edge set only in stars then the problem is identical to that of vertex cover.

Definition 43 (Vertex Cover [CLR89]) *A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both)*

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

Theorem 44 *Let $G = (V, E)$ be communication topology of a synchronous system. Let $\beta(G)$ be the size of the optimal vertex cover of G. Then vector clocks of size $min(\beta(G), N - 2)$ is sufficient to timestamp messages.*

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

most $min(\beta(G), N-2)$. From Theorem 42, we know that with this edge decomposition, the online algorithm generates vector timestamps that capture synchronously precede relation between messages. ■

Since vertex cover does not use triangles in edge decomposition, it is natural to ask how bad can a pure star decomposition be compared to star and triangle decomposition. We claim that

$$\beta(G) \leq 2\ \alpha(G)$$

This bound holds because any decomposition of the graph into stars and triangles can be converted into a decomposition purely of stars by decomposing every triangle into two stars. The above bound is tight in general because if the graph consisted of just t disjoint triangles, then $\alpha(G) = t$ and $\beta(G) = 2t$.

Since the problem of obtaining minimum vertex cover is NP-hard [GJ79], it is unlikely that there exists an optimal algorithm for edge decomposition of a general graph. We now present an algorithm that returns an edge decomposition which is at most twice the size of the optimal edge decomposition. Further, our algorithm returns an optimal edge decomposition when the graph is acyclic.

The algorithm is shown in Figure 5.11. It works by repeatedly deleting stars and triangles from the graph. The main **while** loop in line (02) has three steps inside. The first step chooses any node which has degree 1, say x which is connected to node y. It outputs a star rooted at y. When no nodes of degree 1 are left, the algorithm goes to the second step.

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

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

```
Input: Undirected graph G = (V, E);
Output : edge decomposition,({E_1, ..., E_d}); // Each E_i is either a star or a triangle

(01) F := E;
(02) while F ≠ ∅ do

            //First Step:
(03)      while there exists a node x such that degree(x) = 1 do
(04)          Let (x, y) be the edge of F incident to x;
(05)          output the star rooted at y and all incident edges to y;
(06)          remove from F all edges incident on y;
(07)      endwhile;

            //Second Step:
(08)      while there exists a triangle (x, y, z) with degree(x) = degree(y) = 2 do
(09)          output triangle (x, y, z) ;
(10)          remove from F the edges in the triangle;
(11)      endwhile

            //Third Step:
(12)      Let (x, y) be an edge of F with largest number of edges adjacent to it;
(13)      output the star rooted at y and all incident edges to y;
(14)      output the star rooted at x and all incident edges to x except (x, y);
(15)      remove from F all edges incident on x or y;

(16) endwhile;
```

Figure 5.11: Approximation algorithm for edge decomposition.

Figure 5.12 shows the operation of our edge decomposition algorithm on the communication topology shown in Figure 5.6(b). Using our decomposition algorithm, we decompose the topology in Figure 5.6(b) into 5 stars and 1 triangle. It is easy to see that the optimal decomposition consists of only 4 stars and 1 triangle.

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

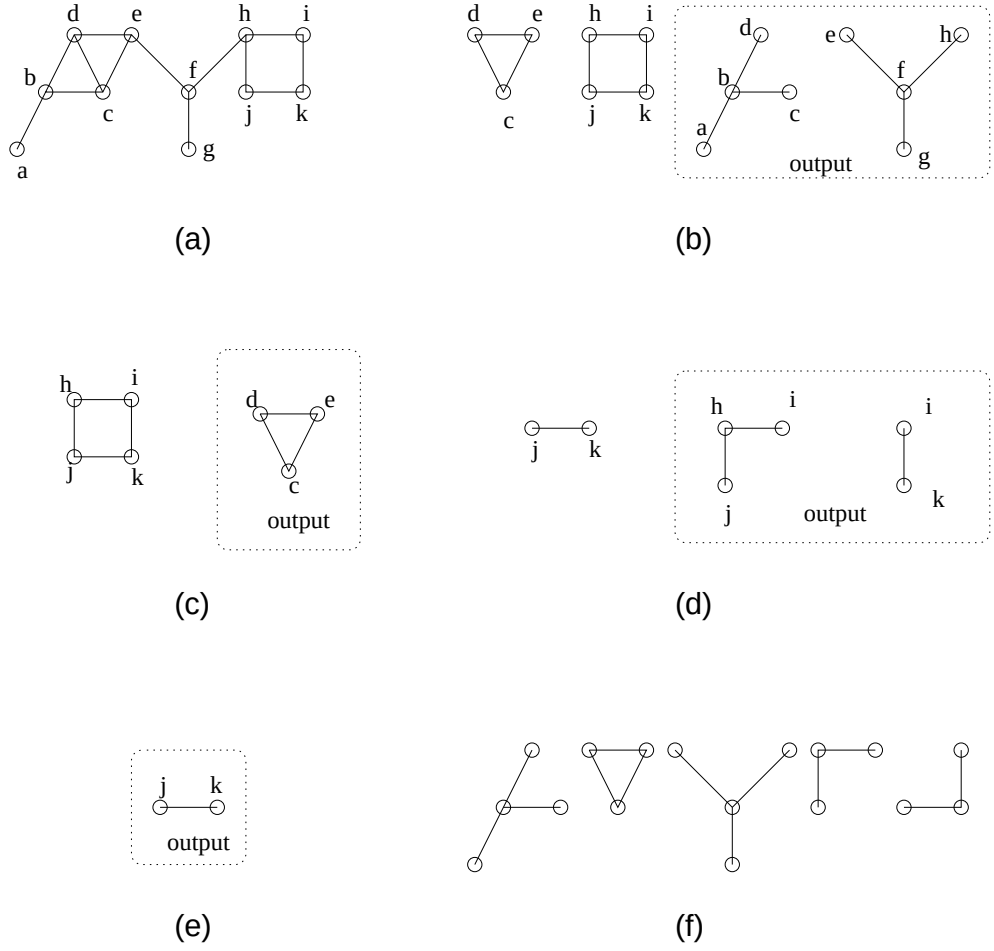


Figure 5.12: A sample run of the proposed decomposition algorithm. (a) The input topology. (b) In the first step, the algorithm outputs 2 stars. There are 7 edges remaining. (c) In the second step, the algorithm outputs a triangle (c, d, e). There are 4 edges remaining. (d) In the third step, two stars are output. Edge (j, k) is remaining. (e) The execution loops back to the first step again and edge (j, k) is output. The program exits. (f) The optimal edge decomposition consists of 4 stars and 1 triangle.

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

Theorem 45 *The algorithm in Figure 5.11 produces an edge decomposition with the approximation ratio bound of* 2.

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

Note that in the above proof we have not used the fact that in step 3, we choose an edge with the largest number of adjacent edges. The correctness and the approximation ratio is independent of that choice. However, by deleting as large number of edges as possible in each step, one would expect to have a smaller edge decomposition.

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

Theorem 46 *The algorithm in Figure 5.11 produces an optimal edge decomposition for acyclic graphs.*

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

Thus, the set H constructed in the proof of Theorem 45 consists of edges added only in step 1. Since we add exactly one edge group for every edge added to H, the optimality follows. ■

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

Example 3 (An Edge Decomposition for Mobile Systems) *Let us consider a client-server based mobile system shown in Figure 5.13 where (1) mobile hosts can only communicate with their local support stations and (2) all interactions in the system are through synchronous RPC or RMI [BB95, CKRH00, DSKF+00].*

In this case, the communication topology can be decomposed with one star rooted at each server because there is no direct communication between mobile hosts and all support stations are fully connected. Thus, it is sufficient to use vector clocks of size equal to the number of support stations to timestamp messages in the system.

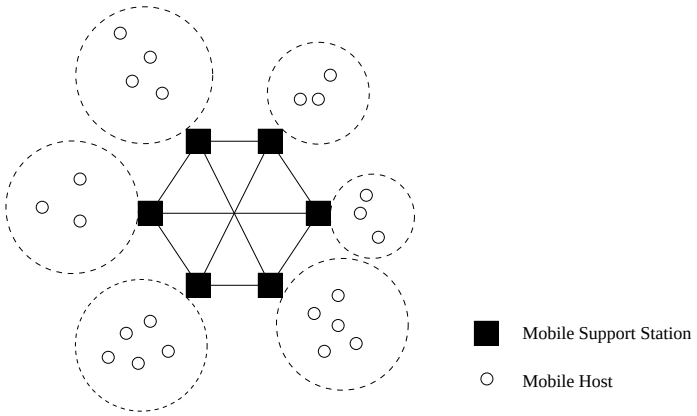


Figure 5.13: A Mobile Computing System.

5.7.2 Offline Algorithm

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

Theorem 47 *Given a poset* $(M, \mapsto)$ *formed by messages in a synchronous computation with* N *processes, vector clocks of size* $\lfloor \frac{N}{2} \rfloor$ *can be used to encode poset* $(M, \mapsto)$.

Proof: For any subset $L \subseteq M$ such that $|L| > \lfloor \frac{N}{2} \rfloor$, there exists $m_i, m_j \in L : m_i \mapsto m_j$ or $m_j \mapsto m_i$. This is because each message involves two processes. From a set of $\lfloor \frac{N}{2} \rfloor + 1$ messages, there must be at least two messages that share a common process. Hence, the size of the longest antichain of $(M, \mapsto)$ (or width$(M, \mapsto)$) is at most $\lfloor \frac{N}{2} \rfloor$.

Due to Dilworth's theorem [Dil50], for any poset P, $dim(P) \leq width(P)$. Hence, $dim(M, \mapsto) \leq \lfloor \frac{N}{2} \rfloor$. ■

As a result from Theorem 47, we get the offline algorithm as shown in Figure 5.14.

From a given poset $\mathcal{M}$,
(1) Let w be the width of poset $\mathcal{M}$. From Theorem 47, $w \leq \lfloor \frac{N}{2} \rfloor$.
(2) Construct a set of linear extensions, $\{L_1, L_2, \ldots, L_w\}$, such that $\bigcap_{i=1}^{w} L_i = \mathcal{M}$.
(The procedure for constructing this linear realizer can be found in [Tro92])
(3) Timestamp each message m with V_m,
where $V_m[i]$ = the number of elements less than m in L_i.

Figure 5.14: Offline Algorithm.

As an example, if we use offline algorithm to timestamp messages in the computation shown in Figure 5.10, 2-dimensional vectors are sufficient to capture concurrency as shown in Figure 5.15.

5.7.3 Timestamping Events in Synchronous Computations

We here show how to extend our algorithms to capture happened before relationship between internal events. So far, we had focused our attention on timestamping send/receive (external) events in synchronous systems. We now show how to ex-

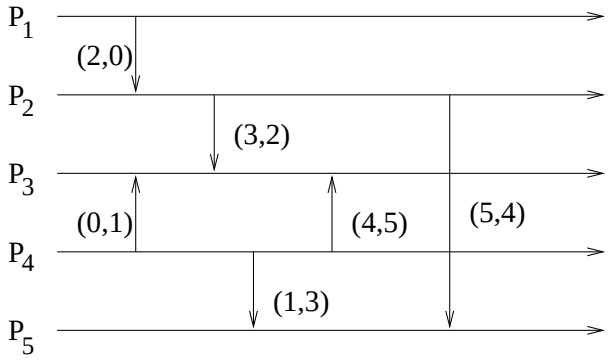


Figure 5.15: A sample run of the offline algorithm.

tend our algorithm to timestamp internal events such that the resulting timestamps capture Lamport's happened before relation.

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

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

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

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

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

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

Theorem 48 $e \rightarrow f \iff succ(e) \leq prev(f)$

Proof: First, we have to prove that

$$e \rightarrow f \Rightarrow succ(e) \leq prev(f) \tag{5.6}$$

If e and f are on the same process then the result is trivially true. Otherwise, since $e \rightarrow f$, there must be a causal chain between e and f. If m_e is the message

immediately after e, and m_f is the message immediately before f, we know that $m_e \mapsto m_f$ or $m_e = m_f$. From Theorem 42, $succ(e) \leq prev(f)$.

Conversely, we have to prove that $succ(e) \leq prev(f) \Rightarrow e \rightarrow f$. We know that the vector timestamp of m_e is less than or equal to that of m_f. From the property of message timestamps (Theorem 42), we get that $m_e \mapsto m_f$ or $m_e = m_f$. From the definition of $\mapsto$, there must be a causal chain from e to f formed by either the application messages or the acknowledgements or both. ■

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

$$e \rightarrow f \iff \begin{cases} succ(e) \leq prev(f) & \text{, or} \\ c(e) < c(f) & \text{and } (prev(e) = prev(f)) \text{ and } (succ(e) = succ(f)) \end{cases}$$

5.7.4 Multicast Communication

So far, we had assumed that the system under consideration uses only point-to-point communication. Here, we discuss how we can use the proposed online algorithm in the system supporting multicast communication.

In this case, the communication topology becomes an undirected hypergraph. Each hyperedge connects a subset of vertices that belong to the same multicast group. Note that two hyperedges corresponding to two different multicast groups are adjacent if these two multicast groups share at least one member. Given a hyperedge decomposition, we can run the online algorithm in Figure 5.9 to generate timestamps that capture synchronously-precede relation between messages. Theorem 42 is applicable even when edges are hyperedges.

Chapter 6

Conclusions and Future Work

We have presented a framework that help developing distributed applications for systems with mobile devices. This framework consists of two service layers: channel allocation and causal message delivery. We also introduced an efficient algorithm to timestamp events in the system. These timestamps accurately captures happened before relation.

In the channel allocation layer, we have proposed an efficient distributed algorithm for channel allocation based on the update approach. Quorum is employed to reduce message complexity from the basic update algorithm. This algorithm also requires less storage overhead than the existing algorithm using the same technique. The member of our request set for each cell is fixed, not dependent upon the channel being requested. For $nR < D_{min} < (n+1)R$, only $3n$ request messages are sent to cells at most n hops away. Furthermore, the response to each request message is sent back to the requesting cell without deferring. Therefore, this algorithm is suitable for applications requiring real-time response. Our algorithm can be used with any dynamic or hybrid channel allocation strategies. As a result, the algorithm adapts well to the dynamic change of the traffic load.

We also proposed a scheme to further reduce message complexity of our original algorithms at the expense of higher acquisition delay. The simulation is con-

ducted to investigate how fast the number of request attempts in the modified algorithms increases as the channel demand grows in both uniform and non-uniform distributions. From the experimental results, the average number of request attempts in the quorum algorithms with acquisition messages is relatively small. Therefore, it is reasonable to conclude that the effect of livelock and starvation can be negligible. The average number of request attempts in the modified quorum algorithm exceeds the point where the modified algorithm is outperformed by the original one at the very high channel demand. Therefore, a combination of the original and modified algorithms can be used to fine-tune mobile network throughput while reducing traffic load on the underlying wired network. Finally, the simulation results show that the acquisition delay of the quorum-based algorithm is significantly lower than that of search algorithms (PK and CS). An interesting future direction is to find update algorithms that do not suffer from livelock and starvation.

In the causal message delivery layer, we introduced a protocol that maintains the low message overhead while reducing unnecessary delivery delay imposed by Alagar and Venkatesan. Unlike [PRS96] and [YHH97], our proposed protocol is scalable and suitable for dynamic systems. It is scalable because message overhead does not depend on the number of mobile hosts. And it is suitable for dynamic systems because it is easy to adapt to the changes in the number of participating mobile hosts. Delivery delay is reduced at the cost of higher storage space required on each MSS. However, this can be accommodated by static hosts due to their rich storage resources. Unlike [AV94], our handoff protocol does not require causal ordering among application messages and messages sent as part of the handoff protocol. This further reduces the unnecessary delay in our protocol.

In addition to correctness proofs for static and handoff protocols, we also present the condition implemented by our static module. The conditions implemented by [AV94] and [YHH97] are also provided. Simulation results show that for

small messages, our protocol can reduce the end-to-end delay by as much as 18.9%, and by as much as 12.11% for large messages. In the future, as the throughput of wireless links keeps increasing, the reduction of the end-to-end delay achieved by our protocol will also be higher. Finally, we provide a case where the protocol presented in [YHH97] does not satisfy liveness property, that is, it is possible that a message is delayed indefinitely.

In the second layer, we only consider point-to-point communication. Therefore, an interesting future work is to extend the proposed algorithm to broadcast and multicast environments.

We implemented the framework and built a simple distributed application based on shared object model. In this application, the user can choose between the two consistency criteria; causal consistency and causal serializability. The implementation of these two consistency criteria relies upon causal message delivery provided in the framework.

For causality tracking, we introduced a new class of posets called string and define the notions of string realizer and string dimension. We showed that for distributed computing applications, these concepts are more natural than the corresponding classical concepts based on chains. In general, string encoding of partial orders is more efficient than chain encoding and easier to obtain in a distributed environment. We also established that the string dimension of a poset is the same as the chain dimension for any poset that is not a string.

We showed that vector clocks of size less than N can be used to characterize relationship between messages in synchronous computations with N processes. This message precedence has applications in visualization of distributed computations.

In the online algorithm, tracking of relationship between messages is performed while the computation is executing. The size of vector clocks is equal to the size of edge decomposition of the communication topology of the system. For

systems with tree-based topology, the size of the vector is equal to the size of vertex cover of the topology. In the offline algorithm, the size of our vector clocks is at most $\lfloor \frac{N}{2} \rfloor$ for the system with N processes.

We have presented an algorithm that returns an edge decomposition which is at most twice the size of the optimal edge decomposition. However, this approximation algorithm produces an optimal edge decomposition when the graph is acyclic. Since the problem of obtaining minimum vertex cover is NP-hard, it still remains an open question whether there exists an polynomial-time algorithm that returns an optimal edge decomposition. It is also interesting to extend the proposed algorithm to cope with the dynamic change of the system topology.

We have shown how to extend our offline algorithm to timestamp internal events in synchronous systems. This algorithm uses interval order to compare any two timestamps. These timestamps capture Lamport's happened before relation between events.

Bibliography

[AHJ91] M. Ahamad, P. Hutto, and R. John. Implementing and programming causal distributed memory. In *Proceedings of the* 11^{th} *IEEE International Conference on Distributed Computing Systems*, pages 271–281, 1991.

[AS95] F. Adelstein and M. Singhal. Real-time causal message ordering in multimedia systems. In *Proceedings of the* 15^{th} *International Conference on Distributed Computing Systems*, pages 36–43, June 1995.

[AV94] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. In *Proceedings of Workshop on Mobile Computing Systems and Applications*, pages 169–174, December 1994.

[AV97] S. Alagar and S. Venkatesan. Causal ordering in distributed mobile systems. *IEEE Transactions of Computers*, 6(3), 1997.

[BB95] A. Bakre and B. R. Badrinath. M-rpc: A remote procedure call service for mobile clients. In *Proceedings of the* 1^{st} *ACM Conference on Mobile Computing and Networking (Mobicom'95)*, pages 97–110, November 1995.

[BJ87] K. Birman and T. Joseph. Reliable communication in presence of failures. *ACM Transactions on Computer Systems*, 5(1):46–47, 1987.

[BM93] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms, 1993.

[Bou84] A. Bouchet. Codages et dimensions de relations binaires. *Annals of Discrete Mathematics 23, Orders: Description and Roles, (M. Pouzet, D. Richard eds)*, 1984.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic broadcast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[CB91] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[CBMT96] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous and asynchronous communication in distributed computations. *Distributed Computing*, 9:173–191, September 1996.

[CKRH00] S. Campadello, O. Koskimies, K. Raatikainen, and H. Helin. Wireless java rmi. In *Proceedings of the* 4^{th} *International Enterprise Distributed Objects Computing Conference (EDOC 2000)*, pages 114–123. IEEE Computer Society Press, September 2000.

[CLR89] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1989.

[CS95] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems*, 17(4):535–559, 1995.

[CS96] M. Choy and A. K. Singh. Efficient distributed algorithms for dynamic channel assignment. In *The Seventh IEEE International Symposium on Personal, Indoor and Mobile Rodio Communications*, pages 208–212, October 1996.

[DG96] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115, Hong Kong, may 1996. IEEE.

[Dil50] R. P. Dilworth. Decomposition theorem for partially order sets. *Ann. Math.*, 51:161–165, 1950.

[DL97] X. Dong and T. H. Lai. Distributed dynamic carrier allocation in mobile cellular networks: Search vs. update. In *Proceedings of the 17^{th} International Conference on Distributed Computing Systems*, pages 108–115, 1997.

[DM41] B. Dushnik and E. Miller. Partially ordered sets. *Amer. J. Math.*, 63:600–610, 1941.

[DP91] B.A. Davey and H.A. Priestley. *Introduction to lattices and orders.* Cambridge University Press, 1991.

[DSKF+00] O. Droegehorn, K. Singh-Kurbel, M. Franz, R. Sorge, R. Winkler, and K. David. A scalable location aware service platform for mobile applications based on Java RMI. In *Proceedings of the Third International IFIP/GI Working Conference: USM 2000*, pages 296–301, Munich, Germany, September 2000.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 183–194, January 1989.

[FZ90] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 131–141. IEEE Computer Society Press, 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1979.

[GPT96] N. Garg, M. Papatriantafilou, and P. Tsigas. Distributed list coloring: How to dynamically allocate frequencies to mobile base stations. In *Symposium on Parallel and Distributed Processing*, pages 18–25, 1996.

[GS01] V. K. Garg and C. Skawratananond. String realizers of posets with applications to distributed computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing(PODC'01)*, pages 72–80, Newport, Rhode Island, August 2001.

[GS02] V. K. Garg and C. Skawratananond. Timestamping messages in synchronous computations. In *Proceedings of the* 22^{nd} *International Conference on Distributed Computing Systems (ICDCS'02)*, Vienna, Austria, July 2002.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[HHN95] M. Habib, M. Huchard, and L. Nourine. Embedding partially ordered sets into chain-products. In *Proceedings of symposium on Knowledge Retrieval, Use and Storage for Efficiency*, pages 147–161, Santa Cruz, 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

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

[IC93] Chih-Lin I and Pi-Hui Chao. Local packing - distributed dynamic channel allocation at cellular base station. In *GLOBECOM*, pages 293–301, 1993.

[IC94] Chih-Lin I and Pi-Hui Chao. Distributed dynamic channel allocation algorithms with adjacent channel constraints. In *The Fifth IEEE International Symposium on Personal, Indoor and Mobile Rodio Communications*, pages 169–175, 1994.

[KBTB97] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. Poet: Target-system independent visualizations of complex distributed-application executions. *The Computer Journal*, 40(8), 1997.

[KG95] J. A. Kohl and G. A. Geist. The pvm3.4 tracing facility and xpvm 1.1. Technical report, Comp. Science and Math. Division Oak Ridge National Lab, TN, USA, 1995.

[KN96] I. Katzela and M. Naghshineh. Channel assignment schemes for cellular mobile telecommunication systems: A comprehensive survey. *IEEE Personal Communications*, pages 10–31, June 1996.

[KS96] A. D. Kshemkalyani and M. Singhal. An optimal algorithm for generalized causal message ordering. In *Proceedings of the* 15^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 87–88, Philadelphia, Pennsylvania, May 1996.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Mae85] M. Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems*, pages 145–159, May 1985.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed algorithms*, pages 215–226, 1989.

[MG95] V. V. Murty and V. K. Garg. Synchronous message passing. In *Proceedings of the International Symposium on Autonomous Decentralized Systems*, pages 208–214, Phoenix, Arizona, April 1995.

[MR93] A. Mostefaoui and M. Raynal. Causal multicasts in overlapping groups: Towards a low cost approach. In *Proceedings of the* 4^{th} *IEEE International Conference on Future Trends in Distributed Computing Systems*, pages 136–142, Lisbon, September 1993.

[PRS96] R. Prakash, M. Raynal, and M. Singhal. An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of the* 16^{th} *International Conference on Distributed Computing Systems*, pages 744–751, Hong Kong, May 1996.

[PSS95] R. Prakash, N. G. Shivaratri, and M. Singhal. Distributed dynamic channel allocation for mobile computing. In *Proceedings of the* 14^{th} *ACM Symposium on Principles of Distributed Computing*, pages 47–56, Ottawa, Canada, August 1995.

[RA81] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, January 1981.

[RR79] I. Rabinovitch and I. Rival. The rank of distributive lattice. *Discrete Math.*, 25:275–279, 1979.

[RST91] M. Raynal, A. Schiper, and S. Toueg. Causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, 1991.

[RTKA96] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *Proceedings of the* 5^{th} *Annual ACM Symposium on Principles of Distributed Computing*, page 310, Philadelphia, PA, May 1996.

[RV95] L. Rodrigues and P. Verissimo. Causal separators for large-scale multicast communication. In *Proceedings of the* 15^{th} *IEEE International Conference on Distributed Computing Systems*, pages 83–91, Vancouver, June 1995.

[SES89] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the* 36^{rd} *International Workshop on Distributed Algorithms*, pages 219–232, Berlin, 1989.

[SG99] C. Skawratananond and V. K. Garg. A quorum-based distributed channel allocation algorithm for mobile systems. In *Proceedings of the* 10^{th} *International Symposium on Personal, Indoor and Mobile Radio Communications*, Osaka, Japan, September 1999.

[SK92] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[SMG99] C. Skawratananond, N. Mittal, and V. K. Garg. A lightweight algorithm for causal message ordering in mobile computing systems. In *Proceedings of the ISCA* 12^{th} *International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 245–250, Florida, USA, August 1999.

[SY85] R. E. Strom and S. Yemeni. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[TRA96] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *Proc. 10th Int. Workshop on Distributed Algorithms*, pages 71–88. Springer-Verlag LNCS, October 1996.

[Tro92] W. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory.* The Johns Hopkins University Press, 1992.

[War00] P. Ward. A framework algorithm for dynamic, centralized, dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, Toronto, Canada, November 2000.

[WT01] P. Ward and D. J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *IEEE 21st International Conference on Distributed Computing Systems*, Phoenix, April 2001.

[Yan82] M. Yannakakis. The complexity of the partial order dimension problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, September 1982.

[YHH97] Li-Hsing Yen, Ting-Lu Huang, and Shu-Yuen Hwang. A protocol for causally ordered message delivery in mobile computing systems. *ACM/Baltzer Journal of Mobile Networks and Applications*, 2(4):365–372, 1997.

VITA

Chakarat Skawratananond was born in Bangkok, Thailand on July 6, 1969, the son of Somkiat Skawratananond and Ruttiporn Skawratananond. In June 1990, he received the Bachelor of Engineering degree in Electrical Engineering from Chulalongkorn University, Bangkok, Thailand. After that, he joined the graduate program in Electrical and Computer Engineering at the University of Texas at Austin, and received the Master of Sciences degree in May 1992. He was employed as an electrical engineer at The Communication Authority of Thailand from June 1992 to August 1993. In January 1994, he came back to the University of Texas to pursue the doctoral degree in Computer Engineering. From 1997 to 2000, he was also employed as a software engineer at Concero Inc. He has been with IBM since January 2001.

Permanent Address: 507/7-8 Petchburi Rd.
Phayathai, Bangkok 10400
Thailand

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.