TESTING AND MOBILITY ISSUES IN DISTRIBUTED SYSTEMS

APPROVED BY SUPERVISORY COMMITTEE:

Supervisor: ________________________________
TESTING AND MOBILITY ISSUES IN DISTRIBUTED SYSTEMS

by

Sridhar Alagar, B.Sc., B.E., M.S.

DISSERTATION
Presented to the Faculty of the Graduate School of
The University of Texas at Dallas
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS
January 1996
Acknowledgements

Words are not sufficiently powerful to convey my gratitude to my advisor, Dr. Venkatesan. His motivation, involvement, and commitment fueled my drive to publish papers and complete my dissertation. Under his tutelage, I solved problems, improved my skills for technical writing, and matured as a researcher. Initially he was my guru, but later became like a father to me on the technical side. It is needless to say that without him I would not have completed this work, but it is important to say that knowing him was one of the best things that have happened to me.

I thank Dr. Lawrence Chung, Dr. Joseph Cleveland, Dr. Simeon Ntafos, and Dr. Ivor Page for agreeing to be on the supervisory committee and for their valuable comments which improved the presentation of this dissertation.

I would like to thank the faculty of the computer science department whose teachings helped me broaden my knowledge in computer science.

I also thank the graduate secretaries, Sandra Bowen, Lynda Gary, and Susan Marsh, for their help during my graduate study.

I would like to thank the National Science Foundation and the Texas Advanced Technology Program for their financial support in the form of research grants.

Special thanks go to my friend and co-researcher Ramki for discussing with me often, criticizing my ideas, and constantly encouraging me. Without him, completing my graduate study would have been more difficult. I also thank
several other friends for expressing their confidence in me which helped me to complete this work.

Finally, and most importantly, I would like to thank my father, Alagar, my mother, Visalakshi, and my wife, Srimathi, for their unconditional support in all of my endeavors.

November, 1995
Testing distributed programs and coping with host mobility are among the important topics in distributed computing. Issues in testing distributed programs are considered first, and then problems posed by host mobility in distributed systems are considered.

For testing distributed systems, a hierarchy consisting of three levels is proposed. Level 1 involves selecting input data for the distributed program. Level 2 exists due to the asynchrony of the communication medium, and it consists of many runs for each input data selected at level 1. The difficulty at this level is that the number of runs for an input can be enormous. Distributed algorithms are presented to find and generate different runs.

At Level 3, a run selected from level 2 is tested. Testing at level 3 is difficult due to the combinatorial explosion of the state space. Several techniques are provided to tackle the state explosion problem while testing a run for errors that are expressed using arbitrary predicates. The techniques include space efficient algorithms, a parallel algorithm, and a method to increase the granularity of an execution step. Next, assuming that errors can be expressed using predicates in conjunctive normal form, an efficient on-line distributed algorithm is presented for detecting them.
Recording global states is useful in setting distributed breakpoints. A message optimal algorithm for recording a global state of a system with causal message ordering is presented.

Next, several distributed algorithms for communication support and fault tolerance in mobile systems are provided.

Conventional algorithms for causal message ordering cannot be used directly for mobile systems due to their energy constraints. Three algorithms are devised for causally ordered message delivery in mobile systems. The first algorithm handles the resource constraints of the mobile hosts. However, the algorithm is not easily scalable and does not handle hosts disconnections. The second algorithm eliminates the above disadvantages at the cost of inhibiting some messages. The third algorithm is a trade-off between the two algorithms. In all three algorithms, mobile support stations perform considerable amount of work on behalf of mobile hosts.

As mobile hosts are highly dependent on mobile support stations, coping with support station failures is important. Two schemes are presented to design mobile computing systems that can tolerate support station failures, and several related issues are discussed. Finally, a protocol to reliably broadcast messages in mobile wireless networks is provided.
Table of Contents
List of Figures
List of Tables
Chapter 1

Introduction

Distributed systems have gained their popularity in recent years because of declining hardware and communication costs. On-line banking, airline reservation systems, and telecommunication networks are some examples of the pervasiveness of distributed systems in everyday life. Inherent fault-tolerance through the redundancy of resources, increased performance by concurrently executing a single task on several computing modules, resource sharing, the mobility of hosts, and the ability to adapt to a changing environment are some of the advantages. In this dissertation, we consider problems in two important topics in distributed systems, namely, testing distributed programs and mobile computing.

1.1 Challenges in Testing Distributed programs

Although distributed systems have several advantages over centralized systems, they are harder to program and test than the sequential systems [?] for the following reasons. First, they consist of a number of concurrent processes. Second, a process may update its variables independently or in response to the actions of another process. Third, problems specific to the development of distributed applications are not suitably reflected by known programming languages and software engineering environments [?].

The problem of testing sequential programs has been considered by many,
and several techniques and tools are available. These techniques cannot be applied directly to testing distributed programs due to the presence of several interacting processes. Also, the communication medium introduces non-determinism, which is absent in sequential programs. The message arrival sequence at a process in one execution for a fixed "input data" may be different from another execution with the same input data yielding a different result. For completeness in testing a distributed program, all possible executions for a fixed input data may have to be tested (implicitly or explicitly). Then we can vary the input data. Also, given a single execution of a distributed program, checking if it exhibits an erroneous behavior is non-trivial. For high software reliability, the processes cannot be tested independently; the processes must be tested "together." But testing processes together leads to the combinatorial state space explosion problem. Clearly, testing distributed programs is hard and challenging.

1.1.1 Summary of Results

In this dissertation, we present a hierarchical approach for testing distributed programs. We also present several distributed algorithms for testing at various levels of the hierarchy. Figure 1.1 shows the three levels of the hierarchy.

Level 1 of the hierarchy deals with the selection of input data from the input space. The set of all input data for the distributed program constitutes the input space. This level is common to both sequential and distributed programs. Several existing techniques for sequential programs may be extended to distributed programs.

Level 2 of the hierarchy addresses the question of non-determinism due to the communication medium. The sequence of messages received by a process may be different from one execution to another (for the same input) due to the non-determinism. Thus the execution of a distributed program for a fixed input data is not unique. One execution may not reveal an error, and we may need to test
several executions. At this level, for a single input data selected from the input space in level 1, different executions are produced. We present a distributed algorithm for producing different executions for a fixed input data. Also, we show that the role of non-determinism due to the communication medium is drastically reduced in (1) distributed programs that run on unidirectional rings with FIFO channels and (2) distributed programs whose fundamental steps involve associative function computation.

Level 3 of the hierarchy involves testing a particular execution selected from the execution space in level 2. To achieve a comprehensive testing strategy, the states of the processes cannot be tested independently; “global states” have
to be tested. We propose three criteria, namely, testing all global computations, testing by global predicate detection, and restricted predicate based testing.

A single execution chosen from the execution space in level 2 may correspond to several *global computations*. A global computation is a sequence of global states starting from the initial global state and ending at the final global state of an execution. Testing all global computations is effective, but it is infeasible since the number of global computations can be exponential in the number of global states.

Global predicate detection is checking whether the given predicate is true in a global state of the system in a given execution. Since many properties about programs such as safety and liveness can be expressed in terms of predicates, a technique to detect whether a predicate is true at a global state is useful for testing distributed programs. Global predicate detection may involve exhaustive testing of all the global states. Testing all global states is adequate in several cases, but it may be difficult to perform due to the state explosion problem. We present several algorithms to tackle the state explosion problem in global predicate detection. First, we present an algorithm that uses $O(mn)$ space where $m$ is the total number of events in the computation and $n$ is the number of processes in the system. We further reduce the space complexity of the algorithm for detecting global predicates to $O(m)$ by not using vector clocks. We then parallelize our space efficient algorithm for detecting global predicates. Considering the explosive nature of global state space, a parallel algorithm can significantly reduce the time taken to detect a global predicate. A salient feature of our parallel algorithm is that the processors that test global states do not communicate (through messages or shared memory) to determine who has to test a particular global state. Our experimental results show that the speedup of our algorithm is close to the optimal value. We further improve the performance, both in time and space, of our algorithms by increasing the granularity of an execution step from an event to a sequence of events (*interval*). Instead of checking every global
state, we check every *global interval*. When the values of the variables related to
the global predicates are not changed “frequently,” the number of global intervals
can be substantially less than the number of global states, thereby reducing the
space and time requirements of our algorithms.

Restricted predicate based testing is to test for errors that can be ex-
pressed in a restricted form of predicates, e.g., predicates in CNF or DNF with
each clause consisting of variables from a single process. Predicate based testing
is feasible, but may be less effective. We present a distributed algorithm for de-
tecting predicates in CNF. The message complexity of the algorithm is \(O(kn^2)\)
where \(k\) is the total number of marked intervals at which a clause is true.

Another important problem in testing and debugging distributed pro-
grams is recording a snapshot (global state) of a distributed system. Recording
a snapshot is useful in setting “global breakpoints.” We present an optimal dis-
tributed algorithm to record a global state of a distributed system with causally
ordered message delivery. The message complexity of our algorithm is \(O(n)\) bits
where \(n\) is the number of processes in the system.

### 1.1.2 Related Work

There have been several papers on debugging distributed programs, but there are
very few papers that address the issues in testing distributed programs. Taylor
and Kelly [?] propose a method to test concurrent programs by extending the
traditional program based testing technique. They define several criteria based on
the control flow graph and establish a hierarchy among these criteria. (Note that
the hierarchy proposed by Taylor and Kelly is based on the flow graph model of
the program and is different from the hierarchy proposed by us which is inherent
in dynamic testing of distributed programs.) The technique at the top of the
hierarchy, testing all the paths, is the most effective but it is not practical. Even
the restricted statement testing criterion is impractical for concurrent programs.
The added difficulty is due to the multiple number of processes. To bypass this difficulty, they restrict the criterion further, by choosing nodes that require closer examination. This criterion is less effective, but feasible. Carver and Tai [?] propose a static analysis technique to detect synchronization errors in concurrent programs. The proposed techniques of Taylor and Kelly [?] and Carver and Tai [?] are program based static analysis methods for testing concurrent programs (ADA and CSP). Our method is based on events and global states, and is dynamic in nature.

The concept of evaluating global predicates, which is useful in the third level of the hierarchy, evolved out of the event occurrence condition introduced by Spezialetti [?], and by Spezialetti and Kearns [?]. Cooper and Marzullo first proposed an algorithm for detecting whether an arbitrary predicate is true in a global state [?]. The time and space complexities of their algorithm, in the worst case, can be exponential in the number of processes. In this dissertation, we present an algorithm whose space complexity is linear in the number of processes. We also provide several techniques to reduce the time taken to detect arbitrary global predicates.

Several researchers have presented efficient algorithms for evaluating global predicates by applying some restrictions on the form of predicates. Manabe and Imase [?], Venkatesan and Dathan [?], and Garg and Waldecker [?] consider predicates in conjunctive normal form. We present an efficient on-line algorithm for detecting predicates in conjunctive normal form. Our technique uses vector clocks of Fidge and Mattern [?, ?] for global predicate detection, and is more efficient and decentralized than the techniques of [?, ?]. Venkatesan and Dathan’s [?] method for detecting global predicates requires $O(n^3)$ messages and is an off-line algorithm. Our algorithm uses $O(n^2)$ messages and is on-line in nature. But their algorithm does not use vector clocks while our algorithm uses vector clocks. Li and Dash [?] give an algorithm to detect predicate of the form $P$ unless $Q$. Miller and Choi [?], and Haban and Weigel [?] consider the problem of detecting
unstable linked predicates. A work by Hurfin, Plouzeau and Raynal [?] deals with a class of global predicates, called atomic sequence of predicates. These sequences are defined by a pair of sequences of local predicates: expected predicates and forbidden predicates [?].

Chandy and Lamport [?] define the notion of global states and present an algorithm to record a global state in distributed systems with FIFO communication channels. The message complexity of their algorithm is $O(m)$ bits where $m$ is the number of channels. Several algorithms exist in the literature for recording global states both in FIFO and NON-FIFO networks [?, ?, ?]. Acharya and Badrinath [?] first considered the problem of recording a global state in a distributed system that maintains “causal order” in message delivery. Since the causal order property is stronger than the FIFO property, global states might be recorded more efficiently in these systems than in systems with FIFO channels. The algorithm of Acharya and Badrinath [?] uses $O(n)$ control messages, but the messages are long (each message is $n$ integers in length where $n$ is the number of processes). In this chapter, we present a message-optimal algorithm to record global states. The message complexity is $O(n)$ and each message is of size $O(1)$ bits.

1.2 Mobility Issues in Distributed Systems

The advent of cellular communication, PCNs, and wireless LANs has made mobile computing realizable in practice. A mobile computing environment consists of a set of mobile hosts (MH) and mobile support stations (MSSs). A mobile host is a host that can move while retaining its connection with the network [?]. Typically mobile hosts are laptops or palmtops with severe energy and storage constraints. Many of them operate with AA batteries and a small amount of disk space [?]. A mobile host communicates with other hosts through a mobile support station.
A mobile support station is a static host and has the necessary infrastructure to communicate with mobile hosts. The geographical area within which an MSS supports mobile hosts is called a cell. Communication between a mobile support station and a mobile host located in its cell is through a wireless medium. All mobile support stations are connected through a wired network. The set of all mobile support stations and the links connecting them constitute the conventional (static) distributed computing environment.

With the introduction of host mobility several new issues arise in distributed computing [?]. At any time a mobile host can move from the cell of one MSS to the cell of another MSS. This leads to the problem of tracking the location of mobile hosts. Tracking a mobile host is essential in establishing a connection with the mobile host and in delivering messages to the mobile host [?, ?, ?, ?, ?].

Another issue in mobile computing is assigning wireless channels to mobile hosts to communicate with their MSSs. Two MSSs with a common cell boundary cannot use the same wireless communication channel to communicate with a mobile host in their respective cells [?]. Since the bandwidth of a wireless communication medium is limited, efficient channel allocation strategies are required.

Various other challenges arise because the existing communication protocols and distributed algorithms cannot be directly used in mobile systems due to the frequent changes in physical connectivity, resource constraints of mobile hosts, and limited bandwidth of the wireless links. Some of the main objectives in extending and designing new distributed algorithms for a mobile computing environment are as follows.

1. Handle host mobility efficiently.

2. Reduce the power consumption of a mobile host.

3. Cope with mobile host’s limited disk space.

4. Reduce the overall communication cost.
1.2.1 Summary of Results

In this dissertation, we design communication protocols and provide fault-tolerant features for mobile computing.

First, we present three algorithms for causally ordered message delivery in mobile systems. The first algorithm appends $O(n_h^2)$ integers to each message where $n_h$ is the number of mobile hosts. The algorithm copes with the resource constraints of the mobile hosts as the mobile support stations execute the algorithm on behalf of mobile hosts. But the algorithm is not easily scalable and does not handle hosts disconnections and connections well. Our second algorithm eliminates the above disadvantages at the cost of inhibiting some messages. The algorithm requires $O(n_s^2)$ integers to be appended to each message where $n_s$ is the number of mobile support stations. The performance of our second algorithm will be significantly better than the first algorithm since $n_h \gg n_s$. Our simulation results confirm this. The third algorithm is a trade-off between the first two algorithms.

We then consider the problem of designing mobile computing systems that tolerate mobile support station failures so that mobile hosts can continue to operate in spite of support station failures. We present two schemes to tolerate support station failures and discuss some important related issues. In both schemes, the information stored at a mobile support station is replicated at several “secondary” support stations. The two schemes differ in the way in which replication is achieved. When a mobile support station fails, the mobile hosts under its coverage can switch to one of their secondary support stations and continue their operation. The two schemes are compared by an experimental study. We also present several methods for selecting the set of secondary support stations for a mobile host and discuss various methods to cope with the failure of mobile hosts.

Finally, we present a protocol to reliably broadcast messages in mobile
wireless networks. A mobile wireless network consists of a set of mobile hosts distributed over a wide geographical area without any support stations. Reliable broadcast is achieved by selective flooding and handshaking.

1.2.2 Related Work

Designing communication protocols for mobile computing is a widely studied field in distributed computing systems. Ioannidis et al. [?] present an infrastructure that enables mobile hosts to keep their network connections even while they move. They present two IP-based protocols to provide continuous network connectivity without affecting higher-level protocols and software. Teraoka et al. [?] present a network architecture for providing host migration transparency at the network layer level. They achieve transparency by dividing the conventional network sublayer into two sublayers: virtual network sublayer and physical network sublayer. The physical address of a mobile host is cached at the physical network sublayer and propagating cache technique is used to update the change in physical address. Badrinath et al. [?] provide a general framework for designing distributed algorithms for mobile computing environments. They handle mobility by creating a proxy for each mobile host. The proxies for mobile hosts will be located at the MSSs. Causally ordered message delivery can be achieved using frameworks discussed above, but the solutions will be inefficient. We provide efficient algorithms for causal ordering by handling mobility explicitly in our algorithms.

Several researchers have worked on the fault-tolerant aspects of mobile computing [?, ?]. Acharya et al. present an efficient technique to checkpoint the states of mobile hosts [?]. Krishna et al. provide checkpointing and rollback techniques to recover from mobile hosts failure [?]. While these results consider the failures of mobile hosts, we consider the failures of MSSs and present techniques for providing continuous support to mobile hosts in spite of MSS failures.
Issues in mobile wireless networks have attracted the attention of several researchers. Perkins and Bhagwat present a routing scheme based on destination-sequenced distance-vector for hosts in mobile wireless networks [?]. Krishna et al. provide a cluster based approach for routing in mobile wireless networks [?]. Several other researchers [?, ?] have considered the problem of routing in mobile wireless networks. We consider the problem of reliably broadcasting messages to host which has not been considered by any other researcher.

The dissertation is organized as follows. System model and related definitions are discussed in Chapter 2. We identify a three level hierarchy inherent in testing distributed programs in Chapter 3. In Chapter 4, several techniques are presented to tackle the global state space explosion encountered in testing distributed programs. A distributed algorithm to record a global state in distributed systems with causal message ordering is presented in Chapter 5. Next we provide three algorithms to preserve causal ordering in mobile system in Chapter 6. In Chapter 7, various schemes are provided to tolerate base station failures in mobile computing. In Chapter ??, a protocol for reliable message broadcast in mobile wireless network is presented. Finally, Chapter ?? concludes the dissertation.

Parts of the results discussed in this dissertation appear in preliminary form in [?, ?, ?, ?, ?, ?]
Chapter 2

Model and Definitions

A distributed system is a collection of processes connected by a set of communication channels. The channels are bidirectional. Communication channels take an arbitrary but finite amount of time to deliver the messages (i.e., the channels are asynchronous). Execution speed of the processes is also asynchronous. Communication among the processes is by message passing only. Let $P = \{P_1, \ldots, P_n\}$ be the set of processes and $C_{ij}$ represent the channel between $P_i$ and $P_j$.

A process is a collection of events that form a total order. An event in a process is an action that changes the state of the process. An event may be a send event resulting in sending of a message to one or multiple processes, a receive event resulting in the receipt of a message from a process, or an internal event in which no sending or receiving of a message is involved. The $k^{th}$ event of process $P_i$ is denoted by $e_i^k$.

Let $E$ be the set of all events in a particular execution. Events in $E$ are ordered based on the causal relation happened before, $\to$, introduced by Lamport [?]. For any two events $e$ and $e'$, $e \to e'$ is true if any of the following holds.

1. $e$ and $e'$ are two events in the same process and $e$ occurs before $e'$.
2. \( \varepsilon \) corresponds to sending a message \( m \) and \( \varepsilon' \) corresponds to the receipt of \( m \).

3. There exists an event \( \varepsilon'' \) such that \( \varepsilon \rightarrow \varepsilon'' \) and \( \varepsilon'' \rightarrow \varepsilon' \).

The set \( E \) and the binary relation \( \rightarrow \) on \( E \) defines an execution. \( (E, \rightarrow) \) is a partially ordered set. An execution of a distributed program can be represented by a space-time diagram. The space-time diagram for a sample execution is shown in Figure 2.1.

To determine the causality between any two events, we use the vector clock introduced by Fidge [?] and Mattern [?]. Let \( V_i \) be the vector clock maintained by \( P_i \). The value of \( V_i \) varies as \( P_i \) executes events. Let the timestamp of an event \( e_i \), denoted as \( TS(e_i) \), be the value of \( V_i \) after executing \( e_i \). Whenever a process sends a message, it \emph{timestamps} the message by appending the current (updated) clock value to it. The following operations are performed on the clock \( V_i \) by \( P_i \) when it executes an event \( e \).

1. For each event \( e \), \( P_i \) first increments the \( i^{th} \) component of its clock, i.e.,
\[
V_i[i] = V_i[i] + 1.
\]

2. If \( e \) is a send event, \( TS(e) \) is appended to the message.

3. If \( e \) is a receive event and \( T \) is the time stamp of the message, \( V_i[j] = max(V_i[j], T[j]) \) for all \( j \).

Consider the example shown in Figure 2.1. The initial clock value of all three processes is \([0, 0, 0]\). When \( P_1 \) sends its first message to \( P_3 \), \( P_1 \) changes its clock value to \([1, 0, 0]\) (by operation 1) and timestamps it with \([1, 0, 0]\). On receiving this message, \( P_3 \) changes its clock value to \([1, 0, 2]\) (by operations 1 and 2).

We say that \( TS(e_i^x) < TS(e_j^y) \) if \( TS(e_i^x)[x] \leq TS(e_j^y)[x] \) for all \( x \) and there exists a \( y \) such that \( TS(e_i^x)[y] < TS(e_j^y)[y] \). It is easy to show that \( e_i^x \rightarrow e_j^y \) iff
The frontier for a cut is unique, and hence we sometimes represent a cut by its frontier. In Figure 2.1, cut $C$ ($< e_1^1, e_2^2, e_3^3 >$) is a consistent cut; whereas cut $C'$ is inconsistent since $e_1^2$ is included in $C'$ and $e_2^2$ is not included in $C'$ and $e_2^2 \rightarrow e_1^2$.

A process state is represented as a sequence consisting of the initial state followed by a sequence of events occurred in that process and the process state is the state after executing the sequence of events from the initial state.

A channel state is a sequence of messages sent along the channel that are consistent. The frontier for a cut is unique, and hence we sometimes represent a cut by its frontier. In Figure 2.1, cut $C$ ($< e_1^1, e_2^2, e_3^3 >$) is a consistent cut; whereas cut $C'$ is inconsistent since $e_1^2$ is included in $C'$ and $e_2^2$ is not included in $C'$ and $e_2^2 \rightarrow e_1^2$.

A process state is represented as a sequence consisting of the initial state followed by a sequence of events occurred in that process and the process state is the state after executing the sequence of events from the initial state.

A channel state is a sequence of messages sent along the channel that are
not yet received. A global state of a distributed system is a collection of process states and channel states. We say that an event e is in a global state G if e is in a process state of the global state G. Similarly, a message m is in a global state G if m is in a channel state of the global state G. For a message m, let send(m) be the event that corresponds to the sending of m and recv(m) be the event that corresponds to the receipt of m. A global state G is consistent if, for a message m sent by P_i to P_j,

\[ m \text{ is in } G \text{ or } \text{recv}(m) \text{ is in } G \implies \text{send}(m) \text{ is in } G. \]

A global state G is complete if, for a message m sent by P_i to P_j,

\[ \text{send}(m) \text{ is in } G \implies m \text{ is in } G \text{ or } \text{recv}(m) \text{ is in } G. \]

For every cut \(<e^1_{k_1}, \ldots, e^n_{k_n}>\), there is a unique global state \(<s^1_{k_1}, \ldots, s^n_{k_n}>\) where \(s^i_{k_i}\) is the state of \(P_i\) after executing \(e^i_{k_i}\). (The global state is unique because the state after every event in the cut is unique.) Also, for every global state, there is a unique cut. In Figure 2.1, the global state corresponding to cut \(C\) is \(<s^1_{k_1}, s^2_{k_2}, s^3_{k_3}>\). A global state \(<s^1_{k_1}, \ldots, s^m_{k_m}>\) is a consistent global state if its corresponding cut is a consistent cut. Hereafter, the term “global state” refers to “consistent global state.”

Mattern [?] has shown that the set of all global states of a given execution (run) forms a lattice. The lattice for the execution shown in Figure 2.1 is shown in Figure 2.2. A point in a lattice represents a global state. Two global states \(G\) and \(G'\) are connected by a directed edge if \(G'\) can be reached from \(G\) by executing one event in any of the processes. There is a path from a global state \(G\) to \(G'\) in the lattice if \(G'\) is reachable from \(G\) by executing several events. Finally, a path from the initial global state to the final global state represents one global computation and is a possible computation of the system. Note that there may be several paths from the initial global state to the final global state depending on the relative order in which the events take place within the processes.
Several other definitions used in this dissertation are not discussed here for the sake of brevity. They will be defined as and when they are required.
Chapter 3

Hierarchy in Testing Distributed Programs

In this chapter, we identify a three level hierarchy that is inherent in testing distributed programs. Level 1 involves the selection of an input data set for the given distributed program. Level 2 deals with selecting an execution for a given input, and level 3 is concerned with testing an execution selected in level 2. Level 1 is common to both sequential and distributed programs. Level 2 is non-existent in testing sequential programs, and is present in testing distributed programs due to the non-determinism of the communication medium. Level 3 testing is more involved in distributed programs than sequential programs due to the presence of multiple number of processes. We merely identify the existent of Level 1; the major contribution of this chapter is to Level 2 and Level 3.

The organization of this chapter is as follows. Certain issues related to selecting input data (level 1) are discussed in Section 3.1. In Section 3.2, we consider the problem of generating different executions for a given input data (level 2). Issues related to level 3 of the testing hierarchy are considered in Section 3.3. Section 3.4 concludes the chapter.
3.1 Selecting Input Data (Level 1)

In this level, input data is selected from the input space of a distributed program. In a distributed program, for each read statement executed by a process, a value must be chosen and given. Thus, an input data for a distributed program is a set of sequences of input values, one sequence per process, such that when a process needs data, the next value from the sequence associated with that process is given.

This level is common to both sequential program testing and distributed program testing, and it is more involved in the case of distributed programs. Several existing techniques for sequential programs can be extended to distributed programs. One such possible technique is random testing. Input data from the input space are selected by random sampling. As stated earlier, this dissertation concentrates more on issues in level 2 and level 3 than in level 1. Some issues in level 1 are extending techniques available for sequential program testing and studying their feasibility and effectiveness in testing distributed programs.

3.2 Coping with the Communication Medium (Level 2)

In a sequential program, the input determines the output. In a distributed program, the output of a process not only depends on the input data but also on the sequence of the messages received by it. A different message arrival sequence may produce a different result for the same input data. Thus, for a given input data, an error in a distributed program may not be revealed in a particular execution while the error may be revealed in a different execution. We illustrate this point with an example.

Consider a voting-based distributed mutual exclusion program. Every process has one vote. A process trying to enter its critical section must get a majority of votes from the other processes. (For simplicity, we do not consider
deadlocks and livelocks [?].) Assume that in an implementation, a process trying to enter its critical section waits for exactly half the number of votes instead of waiting for more than half the number of votes (due to a typing error). We consider two executions; one reveals this error and the other does not reveal it.

Figure 3.1 represents one execution of the mutual exclusion program outlined above. Initially, processes $P_2$ and $P_3$ try to enter the critical section. $P_2$ broadcasts a request vote message (message $r'$ in Figure 3.1), $P_3$ broadcasts a request vote message (message $r$), and both of them wait for the votes. $P_1$ and $P_4$ receive the request from $P_3$ first, hence they send their votes to $P_3$ (messages $g$). Since $P_3$ gets half the number of votes, it enters its critical section (denoted as CS in the Figure). $P_2$ has only one vote (its own vote), so it does not enter its critical section. After leaving its critical section, $P_3$ returns the votes (messages $t$). Now, $P_1$ and $P_4$ grant their votes to $P_2$, and $P_2$ enters its critical section.

Consider another execution of the same program as shown in Figure 3.2. In this execution, $P_1$ gets the request from $P_2$ first (message $r'$ in Figure 3.2), so it sends its vote to $P_2$. $P_4$ grants its vote to $P_3$ as in the earlier execution. Now, $P_2$ and $P_3$ have half the number of votes, hence both may enter the critical section “at the same time” (as shown by the consistent cut GS). Thus, for a given input data, there may be several runs (some may exhibit errors) with different outputs depending on the several possible message arrival sequences.

A test case for a sequential program is the input data alone. A **distributed test case** is a collection of input values (for each process) and the sequence of messages received by each process such that the behavior of the distributed system is identical for each run of the distributed test case [?]. Thus, generating test cases for distributed programs involves generating input data, as in sequential programs, and considering different message patterns. Note that by generating a different run for a fixed input data, we generate a different distributed test case.

We next present an algorithm to generate different runs (distributed test cases)
for a fixed input data.

3.2.1 Producing Different Runs

Let the distributed program be executed once, and \( E_i = < e'_1, e'_2, \ldots > \) be the sequence of consecutive events of \( P_i \). \( E_i \) is a process run for process \( P_i \). \( E = \{ E_1, \ldots, E_n \} \) is a run of the distributed system.

For a process \( P_i \), a process run \( E'_i \) is different from a process run \( E_i \) if the sequences of messages received by \( P_i \) in the two process runs \( E'_i \) and \( E_i \) are different (even if the input data is the same for \( E_i \) and \( E'_i \)). Unless mentioned otherwise, the term “different run” means a different run for the same input data. A run \( E' = \{ E'_1, \ldots, E'_n \} \) of a distributed system is different from a run \( E = \{ E_1, \ldots, E_n \} \) if and only if there exists at least one process \( P_i \) such that \( E'_i \)
is different from \( E_i \). Let \( \text{recv}(m) \) denote the receive event initiated by the receipt of message \( m \), and let \( \text{send}(m) \) be the event that involves the sending of \( m \).

**Lemma 1** Let \( E = \{E_1, \ldots, E_n\} \) be a run of the distributed system for a given input data, and let \( M_i = < m_1, m_2, \ldots> \) be the sequence of messages received by \( P_i \) during process run \( E_i \in E \). There is a process run \( E'_i \) different from \( E_i \), for the same input data, if and only if there are two consecutive messages \( m_{i-1} \) and \( m_i \) received by \( P_i \) in \( E_i \) such that \( \text{recv}(m_{i-1}) \not\rightarrow \text{send}(m_i) \) in \( E \).

**Proof** (\( \Rightarrow \)) Let \( E \) and \( E' \) are two different runs of the distributed system. Let \( S = \{e'_1, \ldots, e'_n\} \) be a set of events such that \( e'_j \) is an event in \( E'_j \) (of process \( P_j \)) and \( e'_j \) is the earliest event of \( E'_j \) that differs from its corresponding event \( e_k \) in \( E_j \). Note that the first event in \( E'_j \), for all \( j \), is the first event in \( E_j \) since the input data is the same for both the runs.

**Fact 1:** There exists an event \( e'_{k_y} \in S \) such that \( e'_{k_z} \not\rightarrow e'_{k_y} \) for all \( z \neq y \). (That is, no event of \( e'_{k_z} \in S \) happens before/precedes \( e'_{k_y} \).)

Since our model is event driven (where an event \( e \) is determined solely on its previous event and, if \( e \) is a receive event, on the contents of the message received), \( e'_{k_y} \) is a receive event. Now, choose \( i = y \) (\( P_i = P_y \)). Let \( \text{recv}(m') = e'_{k_y} \) and \( \text{recv}(m_q) = e_{k_i} \) where \( m_q \) is a message received by \( P_i \) in \( E_i \) (See Figure 3.3).

Since \( e'_{k_i} \neq e_{k_i} \), \( m' \neq m_q \). Now, there exists a message \( m_t \in M_i \) such that \( m_t = m_q \) and \( t > q \). (Recall that \( e'_{k_i} \) is the earliest event in \( E'_i \) that differs from \( e_{k_i} \) in \( E_i \) and, from Fact 1, \( e'_{k_i} \) is not preceded by any other event of \( S \).) Clearly, \( m_t \) is received before \( m_q \) in \( E'_i \). Therefore, \( \text{recv}(m_q) \not\rightarrow \text{send}(m_t) \) in \( E \). (Call this Claim 1.) Now, for any \( r \) (\( q \leq r < t \)), \( \text{recv}(m_q) \rightarrow \text{recv}(m_r) \) in \( E_i \). This implies that \( \text{recv}(m_{t-1}) \not\rightarrow \text{send}(m_t) \) in \( E_i \). (Otherwise \( \text{recv}(m_q) \rightarrow \text{send}(m_t) \), contradicting Claim 1 that was proved before.)

\( \lesssim \) Clearly, during \( E'_i \) message \( m_q \) can be received before \( m_{t-1} \) by \( P_i \) yielding a different process run.
Lemma 1 holds when the links are not FIFO. If the links are FIFO, the FIFO condition must be included.

**FIFO condition:** If two messages, $m_1$ and $m_2$, received by process $P_i$ are sent by the same process $P_j$ connected to $P_i$ by a bidirectional FIFO communication link, then $send(m_1) \rightarrow send(m_2) \Leftrightarrow recv(m_1) \rightarrow recv(m_2)$.

**Lemma 2** Let $E$ be a run (of the distributed system) for a given input data. Assume that the links are FIFO. Let $M_i = < m_1, m_2, \ldots >$ be the sequence of messages received by $P_i$ during process run $E_i \in E$. There is a process run $E'_i$ different from $E_i$, for the same input data, if and only if there are two consecutive messages $m_{t-1}$ and $m_t$ received by $P_i$ in $E_i$ such that (i) $recv(m_{t-1}) \not\rightarrow send(m_t)$ in $E$, and (ii) $send(m_{t-1})$ and $send(m_t)$ are events on two different processes in $E$.

**Proof:** It follows from Lemma 1 and the FIFO condition. 

Lemmas 1 and 2 are very significant. To produce a different run, we need to check only consecutive receive events to see if they can be swapped instead.

---

Figure 3.3: Different Runs
of checking every pair of receive events. Thus, the number of pairs checked for swapping is linear in the number of receive events.

We now explain the algorithms to produce different runs. Formal descriptions are in Figure 3.4 and Figure 3.5. We say that two messages $m_k$ and $m_l$ received by $P_i$ in $E_i$ can be permuted if $recv(m_k) \neq send(m_l)$ in $E$. We assume that the distributed program to be tested is reproducible – i.e., the behavior of each process during the first execution is identical to the behavior during subsequent executions of the same program under the same environment. (Several algorithms exist for reproducible execution and the reader is referred to [? , ?, ?, ?] for details.) During the first run, in addition to the data required for reproducible execution, each process stores the timestamp value $TS(recv(m))$ and $TS(m)$ for every message $m$ received. Algorithm output_pairs() shown in Figure 3.4 outputs all pairs of consecutive messages that can be permuted in a process. (Note that this algorithm can be executed on-line.) The user can then choose the pair of messages to be permuted in a process, and invoke algorithm produce_different_run($k$, $pid$). The parameters indicate that messages $m_k$ and $m_{k+1}$ have to be permuted in the process indicated by $pid$. If the id of the process is equal to $pid$, then the algorithm reproduces execution till $m_k$ and $m_{k+1}$ are received. It then processes $m_{k+1}$ before processing $m_k$ and broadcasts REPRODUCE_OVER to other processes indicating to them that a different run begins. All processes whose id is not equal to $pid$ reproduce execution till they receive the REPRODUCE_OVER message. Each process, after receiving the REPRODUCE_OVER message, continues its normal execution.

The number of possible test cases for a fixed input data may be large, and we do not have an upper bound. But, there are certain classes of programs for which the test case is unique.
output_pairs(); { executed by $P_i$ }
begin
    $k \leftarrow 0$;
    $\text{permute} \leftarrow \text{false}$;
    repeat
        $k \leftarrow k + 1$;
        $e \leftarrow \text{recv}(m_k)$;
        if $TS(e)[i] > TS(m_{k+1})[i]$ then (** $\text{recv}(m_k) \not\rightarrow \text{send}(m_{k+1})$ **) 
            output "k and $k + 1$th messages can be permuted"
            $\text{permute} \leftarrow \text{true}$;
        until process terminates
        if $\text{permute} = \text{false}$ then
            output "No different run exists."
    end.

Figure 3.4: Algorithm output_pairs for $P_i$

3.2.2 Restricted Cases

Unidirectional ring

Theorem 1 For fixed input data, the run in a unidirectional network with FIFO links is unique.

Proof: A process can receive a message from only one process and since the links are FIFO, it follows from Lemma 2 that the run is unique.

It is clear from Theorem 1 that testing distributed programs in unidirectional rings is simpler than in other networks since the input data alone determines the behavior of the processes. (A distributed test case consists of only the input data).
produce\_different\_run( k, pid); \{ executed by \( P_i \) \}

begin

if \( (P_i = \text{pid}) \) then

Reproduce execution \( E_i \) till messages \( m_k \) and \( m_{k+1} \) are received;

Process \( m_{k+1} \) before processing \( m_k \) and broadcast

REPRODUCE\_OVER message;

else

Reproduce execution until REPRODUCE\_OVER message is received;

Continue normal execution;

end.

Figure 3.5: Algorithm \text{produce\_different\_run} for \( P_i \)

---

**Associative distributed programs**

Many of the distributed algorithms such as algorithms for breadth first search, shortest paths, maximum flow, maximum matching, sorting, median finding, and constructing snapshots use the following simple paradigm [?] for reducing the message complexity: (a) preprocess the network, and (b) decompose the algorithm into several phases such that the following computation takes place in each phase: the value of an associative function \( F \) of the private values (which may change from phase to phase) of each process in the network is distributively computed. The coordinator then advances the computation. We call these algorithms \textit{af-based} (associative function based) algorithms [?]. The computation of the associative function is carried out using a spanning tree \( T \).

The root of \( T \) (the coordinator) initiates the computation. A leaf node computes the function using its private value only and sends the value to its parent. A non-leaf node (including the root) waits for a value from each child, computes the function using its private value and the values received from all of
its children, and sends the computed value to its parent (if any).

![Diagram](image)

**Figure 3.6: Computation of MIN function**

For example, consider a six processes system computing the MIN function where $T$ is as shown in Figure 3.6(a). The private value of a process is shown in italics, the arrows denote messages, and the number on each arrow denotes the value sent. Figure 3.6(b) shows an execution. Consider process $P_2$. In the beginning, $P_2$ has its local value 8 and it receives three messages with values 5, 3 and 6. Clearly, the order in which $P_2$ receives these messages is immaterial as far as computing the MIN function is concerned. Thus, even if the three messages had arrived at a different order, the behavior of $P_2$ after receiving all of them is the same. A similar reasoning applies to $P_1$ when it receives the values sent by $P_2$ and $P_6$. Clearly, in this example, the random communication delays cannot alter a process’s behavior that is visible to the other processes. This idea is generalized as follows.

Let $S$ be a state of $P_i$ and $X = [m_1, \ldots, m_x]$ be a sequence of messages. Let $RS_i(S, X)$ be the resulting state of $P_i$ and $out_{P_i}(S, X)$ be the sequence of
messages sent by $P_i$ to other processes when $P_i$ starts from state $S$ and processes the sequence of messages $X$. Note that $RS_i(S, X')$ and $outm_i(S, X')$ are undefined if (a) $RS_i(S, X''')$ is undefined or (b) $P_i$ cannot receive $m''$ while in state $RS_i(S, X''')$ where $X' = X''m''$, and $X'$ and $X''$ are prefixes of $X$.

Let $\{m_1, \ldots, m_x\}$ be a set of messages sent (but not yet delivered) to process $P_i$ such that $\forall s, t$ and $1 \leq s, t \leq x$, $m_s$ and $m_t$ are concurrent*. Let $perm(X)$ be the set of all permutations of $X$. A process $P_i$ is said to execute an associative distributed program with respect to its process state $S$ and a message sequence $X$ if $RS_i(S, X) = RS_i(S, Y)$ and $outm_i(S, X) = outm_i(S, Y)$ for each $Y \in perm(X)$. A distributed application program is said to be an associative distributed program if each process $P_i$ executes an associative distributed program with respect to $S$ and $M$, where $S$ is any legal state of $P_i$ and $M$ is any sequence of messages.

It is easy to see that all af-based distributed programs are associative distributed programs. For these programs, level 2 is considerably less involved than the general case. Clearly, for all $i$, the global state at the end of the $i^{th}$ phase of the associative function does not depend on the random communication delays. Thus, the variation of the “behavior” of the process with random message transmission delays is localized and is not visible outside.

3.3 Testing a run (Level 3)

Sequential programs are tested by running the test cases and by comparing the (intermediate and) final states or the output values with the expected values. Testing a distributed program is complicated because of the presence of multiple processes. Testing the local states of all of the processes in isolation is insufficient;

*Two messages $m_s$ and $m_t$ are concurrent if $m_s$ does not “affect” $m_t$ and $m_t$ does not “affect” $m_s$. 
we need to test the global states. The necessity of testing global states is illustrated with an example. Consider again the execution of the mutual exclusion program shown in Figure 3.2. In the consistent global state $GS$ (denoted by the dotted line in the Figure) both $P_2$ and $P_3$ are in their critical section which is clearly an error. We cannot detect errors of this nature by testing the local states of the processes alone. Also, it is not sufficient to test the final global state alone since the transient errors occurring in the intermediate global states may not be detected.

### 3.3.1 Testing all Global Computations

Recall from Chapter 2 that an execution forms a lattice. Each path from an initial global state to the final global state (of a lattice) represents a global computation that could have been observed by an external observer. One way to test a distributed program is to test all possible paths from the initial global state to the final global state in the lattice. This corresponds to testing all the global computations of an execution. Clearly, this technique can ensure correctness, but, in the worst case, the number of paths may be exponential in the number of global states. Hence, this criterion is not feasible in all cases.

### 3.3.2 Testing Global Predicates

Given a run and a predicate $\Phi$, we test the run by asking the question “Does there exist a global state (consistent cut) where the predicate $\Phi$ holds?” This question is posed as POSSIBLY($\Phi$) in [?, ?] and as weak predicates in [?]. If $\Phi$ represents an error condition, then POSSIBLY($\Phi$) represents a possible occurrence of an error. Since many properties about programs can be expressed in terms of predicates, a technique to detect POSSIBLY($\Phi$) is useful in detecting whether any (safety) property of the program is violated. The predicates may be constructed using the specification of the distributed program with some input
from the implementation group. To thoroughly test the program, sufficient number of predicates covering all aspects of specification and implementation must be constructed. We do not discuss methods for constructing predicates since they are specific to the application program.

To detect $\text{POSSIBLY}(\Phi)$, for any arbitrary $(\Phi)$, we can test every global state. (It appears that detecting an arbitrary global predicate involves exhaustive checking of every global state.) Thus in this technique, we test all consistent global states instead of testing all possible paths. Unfortunately, the worst-case number of global states may be exponential in the number of processes. For a given number of events (on all processes), the number of global states may not be high when a large number of messages are sent since messages create dependencies. The number of global states is very high when the number of messages transmitted is low. In Chapter 4 we provide several algorithms to tackle the state explosion problem while detecting $\text{POSSIBLY}(\Phi)$.

### 3.3.3 Restricted Predicate based testing

If the number of global states is high, we may have to restrict the testing methodologies to properties that can be expressed by predicates in a restricted form. For example, if the distributed program implements a solution to the mutual exclusion problem, then the property to be detected is “multiple processes are in the critical section in a global state.” This can be expressed as “$\Phi = (P_1 \text{ is in critical section}) \land (P_2 \text{ is in critical section})$” in a system consisting of two processes.

In this section, we consider only CNF predicates where each conjunct is any boolean expression using constants and variables of a single process. Thus, a conjunctive predicate may be represented as $C_1 \land C_2 \land \ldots \land C_n$, where conjunct $C_i$ is any boolean expression involving constants and variables local to process $P_i$ only. We next present a distributed algorithm for evaluating $\text{POSSIBLY}(\Phi)$. 
Testing POSSIBLY(\(\Phi\))

An event may be the result of the execution of several statements or subprograms, and several variables may be updated in an event. Note that no messages are sent or received during any part of an event except in the beginning, and each action that results in sending a message or receiving a message marks the beginning of a new event. If the conjunct \(C_i\) becomes true at any time \(t\) of execution of the program at process \(P_i\), then \(P_i\) marks the event that occurs during time \(t\). Clearly, the marked events of \(P_i\) represent those events during which \(C_i\) becomes true at least once. Since \(C_i\) refers to variables of \(P_i\) only, no messages are needed in marking events.

The events are marked on-line by all of the processes and the marked events along with their timestamps are stored in memory. For every process there will be a corresponding monitor process which will execute the algorithm for detecting POSSIBLY(\(\Phi\)). All the monitor processes communicate among themselves and can execute the distributed algorithm on-line. For ease of exposition, hereafter we use the term process instead of monitor process whenever there is no confusion. Each process merges consecutive marked events into a maximal sequence of marked events. Let \(S_i\) be a maximal sequence of consecutive marked events of \(P_i\). Let the indices of the first and last events of \(S_i\) be \(f_i\) and \(l_i\) respectively. The event immediately before the first event of \(S_i\) and the event immediately after the last event of \(S_i\) are unmarked events. If \(C_i = \emptyset\), then \(S_i\) contains all events of \(P_i\).

Testing POSSIBLY(\(\Phi\)) is equivalent to checking if there exists a consistent cut (global state) such that each component of the cut occurs within a marked sequence of events. If no event of \(P_i\) is marked, then POSSIBLY(\(\Phi\)) is false. An algorithm for detecting POSSIBLY(\(\Phi\)) is presented in Figure 3.7. Initially, a process \(P_i\) sets its \(S_i\) to the first sequence of marked events. All the processes distributively find whether there exist a consistent cut, \(< e_k^1, \ldots, e_k^n >\), such
1. For each process $P_i$, $S_i$ is set to the first sequence of marked events of $P_i$.

2. Run the procedure $detect$ of Figure 3.8.

3. If $flag_i = true$, $status_i \leftarrow true$, else remove the current $S_i$ from consideration and set $S_i$ to the sequence of marked events of $P_i$ that occurs just before the currently removed $S_i$ and $status_i \leftarrow false$. If no such sequence exists, $status_i \leftarrow over$.

4. Broadcast $status_i$, and receive $status_j$ from all $P_j$.

5. If $status_j = true$ for all $P_j$, POSSIBLY($\Phi$) is true; halt. If $status_j = false$ for any $P_j$, goto step (2). If $status_j = over$ for any $P_j$, POSSIBLY($\Phi$) is false, halt.

Figure 3.7: Algorithm POSSIBLY($\Phi$) for Process $P_i$

that each event $e^i_k$ is in $S_i$ by invoking the procedure $detect$ in Figure 3.8. If so, POSSIBLY($\Phi$) is true, and the algorithm terminates. Otherwise there exist an $S_j$ in $P_j$ and an $S_i$ in $P_i$ such that the first event of $S_j$ depends on an event that occurred later than the final event of $S_i$. Every such process $P_i$ sets $S_i$ to a sequence of marked events that occurs just after the current $S_i$, and the algorithm continues. If there exist no such sequence, then POSSIBLY($\Phi$) is false and the algorithm terminates.

Consider a sample run consisting of two processes shown in Figure 3.9. A sequence of marked events is shown by an ellipse in the Figure. The timestamp for each event is also shown in the Figure. Initially, $P_1$ sets $S_1$ to the first sequence of marked events and invokes procedure $detect$. (Process $P_2$ also executes the same steps.) Process $P_1$ then sends $TS(e^1_{f_1})[2] (= 0)$ to $P_2$ and waits for a message from
1. Let $f_i$ be the index of the first event of $S_i$ and $l_i$ be the index of the last event of $S_i$. Process $P_i$ sends $TS(e_{j_i}^i)[j]$ to each $P_j$.

2. Wait till $TS(e_{j_j}^j)[i]$ values sent by each $P_j$ is received and locally stored in $P_i$.

3. If $TS(e_{j_j}^j)[i] \leq l_i$ for each $j$, then set $flag_i$ to true else set $flag_i$ to false.

Figure 3.8: Procedure detect

![Diagram](image)

Figure 3.9: A sample run

$P_2$ (step 2 of procedure detect). Similarly, process $P_2$ sends $TS(e_{j_2}^2)[1]$ (= 4) to $P_1$. Since the value sent by $P_2$ is greater than $l_1$ (= 3), $flag_1$ is set to false. After returning from the procedure, $P_1$ advances $S_1$ to the next sequence of marked events since $flag_1$ is false. $P_2$ will not change $S_2$ since, at the end of procedure detect, $flag_2$ is true. After executing step 5 of the algorithm, $P_1$ and $P_2$ will go to step 2 and invoke procedure detect again. This time at the end of the procedure both $flag_1$ and $flag_2$ will be true and the algorithm will terminate. We next prove the correctness of our algorithm.
Lemma 3 For all i, let $S_i$ be a sequence of marked events in process $P_i$. Let $CC = \langle e_{k_1}^i, \ldots, e_{k_n}^i \rangle$ be a consistent cut such that $e_{k_i}^i$ is in $S_i$ for all i. Then $TS(e_{j_f}^i)[i] \leq l_i$ for any $i, j$.

Proof: Recall, that $e_{f_i}^i$ and $e_{i_0}^i$ are the first and last events of $S_i$. Since $CC$ is a consistent cut, by definition, $TS(e_{f_i}^i)[i] \leq TS(e_{i_0}^i)[i]$ for any $i, j$. As $e_{k_i}^i$ is in $S_j$, $TS(e_{k_i}^j)[i] \leq TS(e_{k_j}^j)[i]$ ($e_{j_f}^i \to e_{i_0}^i$). Similarly, $TS(e_{k_i}^i)[i] \leq l_i$ ($e_{k_i}^i \to e_{i_0}^i$). Hence, $TS(e_{j_f}^i)[i] \leq l_i$.

Lemma 4 For all i, flag$_i$ is set to true by the procedure detect if and only if there is a consistent cut $\langle e_{k_1}^i, \ldots, e_{k_n}^i \rangle$ such that $e_{k_i}^i$ is in $S_i$ for all i.

Proof: ($\Rightarrow$) Procedure detect sets flag$_i$ = true only if $TS(e_{j_f}^i)[i] \leq l_i$ for each $j$. Since flag$_i$ = true, $k_i = \max(TS(e_{j_1}^i)[i], \ldots, TS(e_{j_n}^i)[i]) \leq l_i$. Also, $k_i \geq f_i$, since $TS(e_{f_i}^i)[i] = f_i$. Hence event $e_{k_i}^i$ is in $S_i$ for each $i$. Now construct a cut $CC = \langle e_{k_1}, \ldots, e_{k_n} \rangle$ such that each $k_i = \max(TS(e_{j_1}^i)[i], \ldots, TS(e_{j_n}^i)[i])$ for all $i : 1, \ldots, n$. The supremum of any set of consistent cuts is a consistent cut (see [?]). Clearly, $CC$ is a supremum of $n$ consistent cuts denoted by the timestamps of $e_{j_f}^i$’s. Therefore, $CC, \langle e_{k_1}, \ldots, e_{k_n} \rangle$, is a consistent cut such that $e_{k_i}^i$ is in $S_i$ for all i.

($\Leftarrow$) Since there exists a consistent cut $\langle e_{k_1}, \ldots, e_{k_n} \rangle$ such that $e_{k_i}^i$ is in $S_i$ for all i, from Lemma 3, $TS(e_{j_f}^i)[i] \leq l_i$ for any $j$. Hence from step 3 of the procedure detect, flag$_i$ is set to true for each i.

In fact, Lemma 4 gives a method to determine a consistent cut at which POSSIBLY($\Phi$) holds.

Theorem 2 Algorithm POSSIBLY($\Phi$) of Figure 3.7 is correct.

Proof: We first show that if POSSIBLY($\Phi$) holds, our algorithm detects it. Since POSSIBLY($\Phi$) is true, by definition, there exists a consistent cut $CC =$
< e_{k_1}^i, \ldots, e_{k_n}^i > such that C_i is true at e_{k_i}^i for all i. Therefore, e_{k_i}^i must be in a marked sequence, say S_i'. From Lemma 3,
\[ TS(e_{j'}^i)[i] \leq l_i' \quad \forall i, j \] (3.1)
where \( j_i' \) and \( l_i' \) are indices of the first and last events of \( S_i' \) respectively. To prove that our algorithm detects POSSIBLY(\( \Phi \)), it is enough if we show that \( C_i \) is true at \( e_{k_i}^i \) for all \( i \) and step 2 of the algorithm is executed. The rest of the proof follows from Lemma 4. Assume to the contrary that some of the \( S_i' \) were eliminated. Let \( S' = \{ S_1', \ldots, S_n' \} \). Let \( S'' \subseteq S' \) be removed when step 3 of the algorithm is executed for the \( t^{th} \) time and for any \( t' < t \) no subset of \( S' \) were removed. Consider an \( S_i = S_i' \in S'' \). (\( S_i \) was removed when step 3 was executed for the \( t^{th} \) time.) \( S_i \) was removed because \( flag_i = false \). This implies that for some \( j \),
\[ TS(e_{j'}^i)[i] > l_i = l_i' \] (3.2)
Now, either \( S_j \in S'' \) or \( S_j \notin S'' \). If \( S_j \in S'' \), then \( S_j = S_j' \). If \( S_j \notin S'' \), then either \( S_j = S_j' \) or a sequence of marked events that occur before than \( S_j' \). Now there are only two cases to consider. Either \( S_j = S_j' \) or \( S_j \) occurs before \( S_j' \). \( S_j = S_j' \) implies \( f_j = f_j' \). Therefore from equation 3.1 \( TS(e_{j'}^i)[i] \leq l_i \) which is a contradiction to equation 3.2. Now if \( S_j \) is a sequence that occurs before \( S_j' \), then \( e_{j'}^i \rightarrow e_{j'}^i \) which implies \( TS(e_{j'}^i)[i] < TS(e_{j'}^i)[i] \). Therefore again from equation 3.1 \( TS(e_{j'}^i)[i] < l_i \) which is a contradiction to equation 3.2. Hence, if POSSIBLY(\( \Phi \)) is true our algorithm detects it.

If the algorithm detects that POSSIBLY(\( \Phi \)) is true, then \( flag_i = true \) for all \( i \). Hence it follows from Lemma 4 that POSSIBLY(\( \Phi \)) holds.

Message complexity: Each process broadcasts \( l_i \) and \( status_i \) and \( O(n) \) messages are used in broadcasting them. Thus, \( O(n^2) \) messages are sufficient for determining POSSIBLY(\( \Phi \)) if each process has one marked sequence. Otherwise, the message complexity is \( O(kn^2) \) where \( k \) is the total number of marked sequences in all of the processes.
3.4 Concluding Remarks

This chapter develops a hierarchical approach, which consists of three levels, to dynamic testing of distributed programs. At the first level, input data is selected. The second level deals with selecting different runs for an input data. An execution of the distributed program is tested for correctness at the third level.

There is no known way to generate a finite number of distributed test cases which will “expose” all errors, since the equivalent problem in the sequential case is known to be undecidable. Further research is needed to find reasonable criteria for selecting distributed test cases. Testing all paths in a lattice is infeasible, and enumerating and testing all global states may be difficult if the number of global states is enormous. Thus, one can settle for a lesser criterion like restricted predicate based testing. More study is required in this direction. Different approaches such as (1) placing restrictions on the type of distributed programs, and (2) using probabilistic methods are future directions for research.
Chapter 4

Techniques to Tackle State Space Explosion in Global Predicate Detection

In this chapter, we present several algorithms that are useful for testing in Level 3 of the hierarchy. We particularly consider the problem of detecting Possibly(Φ) where Φ is any arbitrary predicate. Cooper and Marzullo [?] present a centralized algorithm for detecting Possibly(Φ). In this chapter, we refer to this algorithm as the CM algorithm. The space and time complexities of the CM algorithm are exponential in the number of processes. Several researchers have presented polynomial time algorithms for detecting a global predicate by placing restrictions on the type of predicate [?, ?, ?, ?, ?]. However, the CM algorithm is important because (1) existing polynomial time algorithms are for restricted forms of predicates, and (2) the polynomial time algorithms are different for different kinds of predicates. In contrast, the CM algorithm may be used for any arbitrary predicate. It appears that detecting an arbitrary global predicate involves exhaustive checking of every global state. In this chapter, we present several techniques to tackle global state explosion while detecting an arbitrary predicate.

The organization of the chapter follows. We present some definitions in Section 4.1. Section 4.2 presents an algorithm for Possibly(Φ) that uses $O(mn)$
space where \( m \) is the total number of events in the computation and \( n \) is the number of processes in the system. This algorithm uses vector clocks. In Section 4.3, the space complexity of the algorithm for Possibly(\( \Phi \)) is further reduced to \( O(m) \) by not using vector clocks. We parallelize the first space efficient algorithm for detecting Possibly(\( \Phi \)) in Section 4.4. Section 4.5 improves the performance of the algorithms, both in time and space, by increasing the granularity of an execution step from an event to a sequence of events (\( interval \)). The chapter is concluded in Section 4.6.

### 4.1 Preliminaries

A global state \( S \) is represented as \( < k_1, \ldots, k_n > \) such that, for process \( P_i \), its state after executing \( k_i \) events is included in \( S \).

For a global state \( S = < k_1, \ldots, k_n > \),

\[
\text{value}(S) = (k_1, k_2, \ldots, k_n)
\]

For a set of global states \( GS \),

\[
\text{value}(GS) = \{ \text{value}(S) \mid S \in GS \}
\]

For any two global states \( S = < k_1, \ldots, k_n > \) and \( S' = < k'_1, \ldots, k'_n > \), \( \text{value}(S) > \text{value}(S') \) if there exists a \( j \) such that \( k_j > k'_j \) and \( k_i = k'_i \) for all \( i : 1 \leq i < j \).

Function \( \text{succ} \), the successor function, for a global state \( S \) is defined below.

\[
\text{succ}(S) = \{ S' \mid \text{system may change from global state } S \text{ to } S' \text{ by executing one event} \}
\]

\( \text{succ}(S) \) can contain at most \( n \) global states, since the number of possible events that can be executed at \( S \) is at most \( n \) (one event per process). Function \( \text{pred} \), the predecessor function, for a global state \( S \) is defined below.
\[
pred(S) = \{ S' \mid \text{system may change from global state } S' \text{ to } S \text{ by executing one event} \}
\]

\( \text{pred}(S) \) also contains at most \( n \) global states. For a global state \( S \), its \( \text{pred} \) and \( \text{succ} \) can be computed easily by using vector clocks [?, ?].

Throughout this chapter we assume that the computation terminates eventually.

### 4.2 Algorithm for Possibly(\( \Phi \))

We first discuss briefly why the CM algorithm for Possibly(\( \Phi \)) requires space exponential in the number of processes. The CM algorithm constructs the lattice of the execution to detect Possibly(\( \Phi \)). The number of nodes (global states) in the lattice can be exponential in the number of processes. The depth of the lattice is the total number of events in the run. Thus the average breadth of the lattice can be exponential in the number of processes, even when the total number of events in an execution is polynomial in the number of processes. The \textit{level} of a global state \( < k_1, \ldots, k_n > \) in a lattice is the sum \( k_1 + k_2 + \ldots + k_n \). The CM algorithm for Possibly(\( \Phi \)) traverses the lattice in a breadth first fashion. It computes all the global states in one level, stores and checks them and then proceeds to the next level. Hence the space required (to store nodes that are in one level) in the worst case can be exponential in \( n \).

In contrast, our algorithm traverses the lattice in a depth first fashion. We do not explicitly store the nodes that are already visited, as this will require exponential amount of space. Also, we do not visit a node more than once. We perform some computation to decide whether a node has already been visited or not.

**Main idea**

Assume that we are in global state \( S \) in the lattice. Let \( S' \) be a successor of \( S \).
in the lattice. Now we have to decide whether $S'$ can be visited (tested) from $S$. Global state $S'$ has at most $n$ predecessors in the lattice. All predecessors $S'$ can be ordered according to their values. If the value of $S$ is the maximum among all the predecessors of $S'$, then we visit $S'$ from $S$. The key rule for testing a global state exactly once is to visit the global state only from its predecessor that has the maximum value among all of its predecessors.

Our algorithm for Possibly($\Phi$) is executed by a monitor process. During the computation, all the processes, after executing an event, send the timestamp of the event and the value of the local variables related to $\Phi$ to the monitor. A formal description of the algorithm for Possibly($\Phi$) appears in Figure 4.1.

Algorithm POSSIBLY is invoked with a global state $S$. First, $\Phi$ is tested at $S$. If $\Phi$ holds, then the algorithm returns true. Otherwise $\text{succ}(S)$ is computed. For every successor $S'$ of $S$ such that $S$ is the predecessor of $S'$ with the maximum value, algorithm POSSIBLY($S'$) is invoked. Initially, the monitor invokes the algorithm with the initial global state $<0,\ldots,0>$. Before invoking the algorithm POSSIBLY with a global state $S$, if the monitor has not received the values of the variables corresponding to the state $S$, it waits for them and then invokes POSSIBLY with $S$. If the algorithm returns true, then Possibly($\Phi$) is true.

**An example**

We illustrate the working of the algorithm with a sample lattice shown in Figure 4.2(b). The execution corresponding to this lattice is shown in Figure 4.2(a). The numbers in bold show the order in which the lattice is traversed. The algorithm is invoked with the initial global state $<0,0>$. The value of $i$ is 1. After step 6, $S' = <1,0>$. Since, $<1,0>$ is a consistent global state and $\text{max}(\text{value}(\text{pred}(<1,0>)))$ is $<0,0>$, POSSIBLY($<1,0>$) is invoked.

To understand the algorithm further, assume that algorithm POSSIBLY is invoked with state $<0,1>$. After executing step 6 once, $S' = <1,1>$. The set $\text{pred}(<1,1>)$ is $\{<1,0>,<0,1>\}$, and $\text{max}(\text{value}(\text{pred}(<1,1>)))$ is
Algorithm POSSIBLY($S$)
begin
(1) if $\Phi$ is true at $S$ then
(2) \hspace{5mm} \text{return(true);}$
(3) \hspace{5mm} $i := 1;$
(4) \hspace{5mm} Let $S = <k_1, \ldots, k_i, \ldots, k_n>$$
(5) \hspace{5mm} \text{while (} i \leq n \text{) do}$
\hspace{10mm} begin
(6) \hspace{10mm} $S' = <k_1, \ldots, k_i + 1, \ldots, k_n>$$
(7) \hspace{10mm} \text{if } S' \text{ is a consistent global state then}$
(8) \hspace{10mm} \hspace{10mm} \text{if } value(S) = \max(value(pred(S')))$
(9) \hspace{10mm} \hspace{10mm} \hspace{10mm} \text{if } (\text{POSSIBLY}(S') = \text{true})$ then
(10) \hspace{10mm} \hspace{10mm} \hspace{10mm} \hspace{10mm} \text{return(true);}$$
(11) \hspace{10mm} i = i + 1;$
\hspace{10mm} end;
(12) \hspace{10mm} \text{return(false);}$$
end.$

Figure 4.1: Algorithm for detecting POSSIBLY($\Phi$)
Figure 4.2: A sample execution and its lattice.
<1, 0> and not <0, 1> (step 8). So, POSSIBLY(<1, 1>) is not invoked from <0, 1>. When step 6 is executed for the second time, S' becomes <1, 2>. But <1, 2> is not a consistent global state, and since there are no more successors of <0, 1>, the current invocation of the algorithm returns.

4.2.1 Correctness and Analysis

Theorem 3 Possibly(Φ) is true if and only if algorithm POSSIBLY returns true when invoked with the initial global state.

Proof(⇒) We prove by taking the contrapositive, i.e., if the algorithm returns false, then Possibly(Φ) is false. To prove this, it is sufficient to prove that all global states of the lattice are checked by the algorithm. The proof is by induction on the level of the lattice. The algorithm is initially invoked with the initial global state at level 0, so all global states at level 0 are checked. Assume that all the global states at level k are checked, for some k ≥ 0. Consider a global state S' at level k + 1. The global state S' has at least one predecessor at level k. Let S be a predecessor of S' such that S has the maximum value among all the predecessor of S' at level k. From step (8) of the algorithm it is clear that whenever S is checked, S' will also be checked. The global state S is at level k, and by induction hypothesis S is checked. Hence, if Possibly(Φ) is true, algorithm POSSIBLY returns true.

(⇐) Algorithm POSSIBLY returns true only if Φ is true at a global state. Hence Possibly(Φ) is true.

Theorem 4 The space complexity of Algorithm POSSIBLY is O(mn) where m is the total number of events in the given run and n is the number of processes.
Proof Observe that the depth of the recursion for the algorithm is at most $m$, the depth of the lattice. The size of data stored at each level of recursion is $O(n)$. (The number of components of the timestamp of an event is $n$). So, it is easy to see that $O(nm)$ is the space complexity for the algorithm.

We performed some experimental studies and their results are shown in Table 4.1. Some sample executions were generated for different number of processes. With 12 processes and 1200 events (depth) the maximum size of breadth is 875201. So a depth first approach can give significant savings in storage over the breadth first approach. Also when using the breadth first approach, the global states at the next level have to be computed and stored in a linked list. Before inserting a global state in the list, the list has to be searched to determine whether the global state is already present in the list. (A global state at the next level may be computed more than once because the global state can have more than one predecessor at the current level.) Maintaining and searching the list requires significant amount of time when the breadth of the lattice is large.

One advantage of breadth first approach is that Possibly($\Phi$) can be detected early if $\Phi$ is true at a global state at the top of the lattice. However if $\Phi$ is true at a global state that occurs near the bottom of the lattice, depth first approach detects $\Phi$ early as breadth first approach will be spending significant time in levels at the top of the lattice.

<table>
<thead>
<tr>
<th>Processes</th>
<th>Total No. of Events</th>
<th>No. of Global States</th>
<th>Max. Breadth</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>500</td>
<td>203801</td>
<td>1430</td>
</tr>
<tr>
<td>8</td>
<td>800</td>
<td>3238799</td>
<td>62609</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>18052003</td>
<td>270353</td>
</tr>
<tr>
<td>12</td>
<td>1200</td>
<td>22084953</td>
<td>875201</td>
</tr>
</tbody>
</table>

Table 4.1: Experimental Results
The space complexity of the algorithm can further be reduced to $O(m)$ without using vector clocks. The linear space algorithm is presented in the next section.

4.3 A Linear Space Algorithm

This algorithm does not use vector clocks. For every event $e$, the algorithm needs those events that immediately “happened before” $e$. Let the set of events that immediately happened before $e$ be denoted by $\text{dep}(e)$. If $e$ is an internal event or send event, $\text{dep}(e)$ is just the event that occurred immediately before $e$ in the same process. If $e$ is a receive event, $\text{dep}(e)$ has two events – the corresponding send event and the event that occurred immediately before $e$ in the same process. During the computation, we append, to every message, the event number of the corresponding send event and the process in which the send event occurred. Thus, $\text{dep}(e)$ can be easily computed on the fly.

Event $e \in C$ is incomparable if $e \not\rightarrow e'$ for any event $e' \in C$ ($C \subseteq E$). Let $\text{incomp}(S)$ denote the set of incomparable events in the cut corresponding to the global state $S$. The set $\text{incomp}(S)$ can have at most one event per process. We order the events in $\text{incomp}(S)$ according to the id of the process in which they occur. Let $\text{max(}\text{incomp}(S)\text{)}$ denote the event in $\text{incomp}(S)$ that occurred in the process with the largest id among all the events in $\text{incomp}(S)$.

**Lemma 5** Let $S$ be a global state in execution $E$. Global state $S'$ is a predecessor of $S$ in the lattice of $E$ if and only if $S$ is reachable from $S'$ by executing an event $e \in \text{incomp}(S)$.

**Proof:** ($\Rightarrow$) Let $C'$ and $C$ be the cuts corresponding to $S'$ and $S$ respectively. Since $S'$ is a predecessor of $S$, global state $S$ is reachable from $S'$ by executing an event $e$. Assume that $e \not\in \text{incomp}(S)$. This implies that there exists an event
\( e' \in C \) such that \( e \rightarrow e' \), \( e' \in C' \), because \( C' \subset C \) and \( C - C' = \{ e \} \). Now, \( e' \in C' \) and \( e \notin C' \) implies that \( C' \) is not a consistent cut and \( S' \) is not a consistent global state, a contradiction.

(\( \Leftarrow \)): Follows from the definition of predecessor.

The above lemma and the ordering of events in \( \text{incomp}(S) \) suggest a way to test the global state \( S \) exactly once. If we are in \( S' \), we will test \( S \) only if we execute event \( e \) at \( S' \) to reach \( S \) where \( e = \max(\text{incomp}(S)) \). For this, the knowledge of \( \text{incomp}(S) \) is needed. Assuming that we know \( \text{incomp}(S') \), the following lemma shows a way to compute \( \text{incomp}(S) \).

**Lemma 6** Let global state \( S \) be reachable from global state \( S' \) in the lattice of \( E \) by executing event \( e \). Then \( \text{incomp}(S) = \text{incomp}(S') - \text{dep}(e) + \{ e \} \).

**Proof:** First we show that for any event \( e' \), \( e' \in \text{incomp}(S) \Rightarrow e' \in \text{incomp}(S') - \text{dep}(e) + \{ e \} \). Assume that \( e' \in \text{incomp}(S) \) but \( e' \notin \text{incomp}(S') - \text{dep}(e) + \{ e \} \). Clearly, \( e' \neq e \). Also \( e' \neq e \) since \( e' (\in \text{incomp}(S)) \) is an incomparable event in \( C \) and \( e \in C \). Thus \( e' \notin \text{dep}(e) \). Therefore \( e' \notin \text{incomp}(S') \). As \( e' \) is not an incomparable event in \( C' \) there exists an event \( e'' \) in \( C' \) such that \( e' \rightarrow e'' \). Since \( C' \subset C \) and \( C - C' = \{ e \} \), \( e'' \in C \) also. Therefore, \( e' \) is not an incomparable event in \( C \). Hence \( e' \notin \text{incomp}(S) \), a contradiction.

Next, we show that \( e' \in \text{incomp}(S') - \text{dep}(e) + \{ e \} \Rightarrow e' \in \text{incomp}(S) \). If \( e' = e \), the claim follows from Lemma 5. Now, assume \( e' \neq e \). Clearly, \( e' \notin \text{dep}(e) \).

Otherwise \( e' \in \text{incomp}(S') - \text{dep}(e) + \{ e \} \) is not true, since \( \text{dep}(e) \) is subtracted from \( \text{incomp}(S') \). So, \( e' \in \text{incomp}(S') \). Now we claim that \( e' \neq e \). If \( e' \rightarrow e \), then either \( e' \) immediately happened before \( e \) or \( e' \rightarrow e'' \rightarrow e \) where \( e'' \in \text{dep}(e) \).

But \( e' \notin \text{dep}(e) \). Therefore, \( e' \rightarrow e'' \rightarrow e \) where \( e'' \in \text{dep}(e) \). Now \( \text{dep}(e) \subset C' \), since \( \text{dep}(e) \subset C \) and \( C = C' \cup \{ e \} \). \( \text{dep}(e) \subset C' \) and \( e' \in \text{incomp}(S') \) implies that \( e \neq e'' \). So, \( e' \neq e \). Therefore there does not exist any event \( e'' \) such that \( e'' \in C' \cup \{ e \} = C \) and \( e' \rightarrow e'' \). Hence \( e' \in \text{incomp}(S) \). 

\[ \square \]
We use the above lemma to reduce the space complexity. The algorithm is similar to the algorithm in Figure 4.1. The only difference is in the way the key rule to visit a global state is implemented. A formal presentation of the algorithm is shown in Figure 4.3. GS and INC are global variables. GS is initialized to the initial global state, and INC is initially empty. The set of incomparable events in GS is stored in INC. At any instant, GS contains the current global state being tested or considered for testing. At step 5, we execute the next event, say $e$, in process $i$ by incrementing the $i^{th}$ component of GS. Now to check whether the new value of GS is a consistent global state, it is enough if we check whether $\text{dep}(e)$ is included in GS since the previous value of GS is a consistent global state.

### 4.3.1 Correctness and Analysis

**Theorem 5** $\text{Possibly}(\Phi)$ is true if and only if algorithm POSSIBLY in Figure 4.3 returns true when invoked with the initial global state.

**Proof ($\Rightarrow$)** We prove by taking the contrapositive, i.e., if the algorithm returns false, then $\text{Possibly}(\Phi)$ is false. To prove this, it is sufficient if we prove that all global states of the lattice are checked by the algorithm. The proof is by induction on the level of the lattice. The algorithm is initially invoked with GS set to the initial global state. Assume that all the global states at level $k$ are checked, for some $k \geq 0$. Consider a global state $S$ at level $k + 1$. Let $e = \text{max}(\text{incomp}(S))$ and $e$ be an event in $P$. From Lemma 5, there exists a predecessor, say $S'$, from which $S$ can be reached by executing $e$. The global state $S'$ is at level $k$, and by induction hypothesis function possibly is invoked with $GS$ set to $S'$. Steps 5 through 9 of the algorithm ensures $S$ will also be checked. (From Lemma 6, it follows that the set $INC$ will be equal to $\text{incomp}(S)$ after step 8.) Hence, if $\text{Possibly}(\Phi)$ is true, algorithm POSSIBLY returns true.
Global variables:

$GS$ : array $[1, \ldots, n]$ of integers; \textit{contains the current global state}

$INC$ : set of events; \textit{contains the incomparable events in GS}

Function POSSIBLY()
begin
(1) if $\Phi$ is true at $GS$ then
(2) return(true);
(3) $i := 1$;
(4) while ($i < n$) do
\hspace{1em} begin
(5) $GS[i] = GS[i] + 1$; \textit{let} $\epsilon$ \textit{be the event executed in} $P_i$\end
(6) if $GS$ is a consistent global state then
\hspace{1em} $temp = INC \cap \text{dep}(\epsilon)$;
(7) $INC = INC - temp + \{\epsilon\}$;
(8) if $\max(INC) = \epsilon$
\hspace{1em} if (POSSIBLY() = true) then
\hspace{2em} return(true);
\hspace{1em} $INC = INC + temp - \{\epsilon\}$;
(10) $GS[i] = GS[i] - 1$;
(11) $i = i + 1$;
\hspace{1em} end;
(12) return(false);
end.

Figure 4.3: Linear Space Algorithm for detecting POSSIBLY($\Phi$)
Algorithm POSSIBLY returns true only if $\Phi$ is true at a global state. Hence $\text{Possibly}(\Phi)$ is true.

**Theorem 6** The space complexity of Algorithm POSSIBLY in Figure 4.3 is $O(m)$ where $m$ is the total number of events in the given run.

**Proof** Observe that the algorithm does not use vector clocks to check if a global state is consistent. We store only $\text{dep}(e)$ for every event which uses a constant space. The local variable $\text{temp}$ inside the function POSSIBLY() can contain at most two events. So every time the function is invoked, only a constant amount of additional storage is used. Since the depth of recursion is equal to the total number of events $m$, the space complexity of the algorithm is $O(m)$.

### 4.4 A Parallel Algorithm

In this section, we parallelize the algorithm presented in the Section 4.2 to reduce the time taken to detect $\text{Possibly}(\Phi)$. (The linear space algorithm presented in Section 4.3 can also be parallelized similarly.) We assume that a parallel machine consisting of $t$ processors is available. Our algorithm is independent of the architecture of the parallel machine. In fact the distributed system on which the computation was performed can also be used.

To ease the description of the algorithm, we assume that the input (timestamps of all the events, and the value of variables related to $\Phi$ after every event) is available at all the processors. Later, we discuss various ways of providing the input to all the processors.

In our parallel algorithm, every global state is tested exactly by one processor. A salient feature of our algorithm is that processors do not exchange messages to decide whether they can test a particular global state or not. We
achieve this by using the same idea used in Section 4.2 to reduce the space complexity of algorithm POSSIBLY.

A global state will be tested only by a processor that tested the largest predecessor (which is unique) of the global state. If a processor, say $\mathcal{P}_r_i$, finds some idle processor, $\mathcal{P}_r_i$ will delegate one of the global states that it has to test to the idle processor.

One of the processors will also act as a processor allocator. A processor that becomes idle will register itself with the allocator. When a processor has more than one global state to test, it will request the allocator for idle processor(s). The allocator will serve the requests in the first come first serve basis. Initially, processor $\mathcal{P}_r_0$ is assigned the initial global state, and all other processors are idle.

Now we illustrate our algorithm with the example shown in Figure 4.4. The number of processors used is 3. Processor $\mathcal{P}_r_0$ is assigned the global state $< 0, 0 >$ initially. Since the global state $< 0, 0 >$ is the largest predecessor of $< 1, 0 >$ and $< 0, 1 >$ processor $\mathcal{P}_r_0$ has to test them. However, processors $\mathcal{P}_r_1$ and $\mathcal{P}_r_2$ are idle. So, $\mathcal{P}_r_0$ assigns state $< 0, 1 >$ to $\mathcal{P}_r_1$ and tests state $< 1, 0 >$. $\mathcal{P}_r_0$, after testing $< 1, 0 >$, considers the states $< 2, 0 >$ and $< 1, 1 >$. Since $< 1, 0 >$ is the largest predecessor of the states $< 2, 0 >$ and $< 1, 1 >$, it is processor $\mathcal{P}_r_0$’s responsibility to test them. At this stage, $\mathcal{P}_r_2$ is idle, so state $< 1, 1 >$ is assigned to $\mathcal{P}_r_2$. $\mathcal{P}_r_2$ tests $< 2, 0 >$.

$\mathcal{P}_r_1$ after testing $< 0, 1 >$ examines $< 1, 1 >$ and finds that $< 0, 1 >$ is not the largest predecessor of $< 1, 1 >$, so it does not test $< 1, 1 >$. $\mathcal{P}_r_2$ informs the allocator that it is idle as it does not have any other global state to test. Proceeding in a similar fashion, $\mathcal{P}_r_0$ tests states $< 2, 1 >$, $< 2, 2 >$, $< 2, 3 >$, and $< 3, 3 >$ and $\mathcal{P}_r_2$ tests $< 1, 2 >$, and $< 1, 3 >$.

A formal description of the algorithm is shown in Figure 4.5. An idle processor, say $\mathcal{P}_r_i$, begins testing when it receives a start($S$) message ($S$ is a
global state). \( P_r_i \) first tests \( \Phi \) at \( S \). If \( \Phi \) is true at \( S \), it notifies all the processors and exits. Otherwise \( P_r_i \) finds the list of all the successors of \( S \) for which \( S \) is the largest predecessor and stores them in queue \( Q_i \). Processor \( P_r_i \) then requests the allocator for any idle processor by sending \( \text{request}(x) \) message to the allocator where \( x \) is the number of processors that \( P_r_i \) needs. \( P_r_i \) then takes the first element in \( Q_i \) and repeats the steps in the loop.

In the meanwhile, if \( P_r_i \) receives \( \text{allocated}(P_r_i) \) message, \( P_r_i \) sends \( \text{start}(S') \) to \( P_r_j \) where \( S' \) is the last element in \( Q_i \). When \( Q_i \) becomes empty, \( P_r_i \) informs the allocator that is idle and waits for a \( \text{start} \) message.

4.4.1 Correctness and Analysis

The correctness of the parallel algorithm is similar to the proof in Theorem 3. We analyze the space complexity of the algorithm assuming that only one processor
On receiving \textit{start}(GS), execute the following steps.

\begin{verbatim}
S = GS;
done = false;
repeat
  count = 0;
  if \Phi is true at S, notify all the processors and exit;
  for each \( S' \) such that \( S' \) is a successor of \( S \)
    if \( \text{value}(S) = \max(\text{value}(\text{pred}(S')) \) then
      add \( S' \) to the head of \( Q_i \);
      count = count + 1;
  if \( Q_i \) is empty then
    done = true
  else
    S = head of \( Q_i \);
    send \textit{request}(count - 1) message to the allocator;
    delete the first element of \( Q_i \);
  until(done)
inform the allocator that \( P_{ri} \) is idle.
\end{verbatim}

On receiving \textit{allocated}(\( P_{rj} \)) from the allocator, execute the following steps.

\begin{verbatim}
S' = tail of \( Q_i \);
delete tail of \( Q_i \);
Send \textit{start}(S') to processor \( P_{rj} \);
\end{verbatim}

Figure 4.5: Algorithm executed by processor \( P_{ri} \)
executes the parallel algorithm.

**Theorem 7** The space complexity of the parallel algorithm is $O(mn^2)$ where $m$ is the total number of events in the given run and $n$ is the number of processes.

**Proof:** Note that $\mathcal{P}_i$ adds a global state to the head of queue and removes a global state from the head of the queue. (The queue is used as a stack.) Thus the global states at a higher level of the lattice are tested first before testing the next global state in the same level. This ensures that at most $n$ global states are stored per level. (There can be at most $n$ immediate successors for a global state.) The space required to store a global state is $O(n)$. Since there are $m$ levels in the lattice, the total space required is $O(mn^2)$.

The following situation of the algorithm merits observation: Whenever $\mathcal{P}_i$ receives an allocated($\mathcal{P}_j$) message, $\mathcal{P}_i$ sends start($S'$) to $\mathcal{P}_j$ where $S'$ is the last element in $Q_i$. The last global state in $Q_i$ is at a level earlier than the level of the global state that is at the head of the queue $Q_i$. The reasoning behind this is that $\mathcal{P}_j$ may get to test more number of global states if it starts from an earlier level than if it starts from a latter level. One can visualize the working of the parallel algorithm as though the parallel machine is proceeding in a breadth first fashion and each processor within the machine is proceeding in a depth first fashion.

We have evaluated the performance of our parallel algorithm by a simulation. Figure 4.6 shows the relationship between total time taken for the parallel machine to test all the global states and the number of processors. As the number of processors increases the time taken to test all the global states reduces. Figure 4.7 shows the speedup achieved as the number of processors is varied. The speedup achieved is close to the optimal value, for the experiments we have performed, since the number of global states is very large.
Figure 4.6: Time taken vs. No. of Processors.

Figure 4.7: Speedup vs. No. of Processors.
Providing Input

If a parallel machine is dedicated for testing Possibly($\Phi$), one way to distribute the input is to send the input to all the processors. Another approach is to store the input in a shared memory, and the processors can get the input from the shared memory as and when they require it. If the existing distributed system itself is used for testing, the input can be provided separately to all the processors or the input can be stored in a common place, e.g., in a mounted network file system or a distributed shared memory.

4.5 Increasing the granularity of the execution step

The performance of algorithms for detecting arbitrary global predicates can be improved by considering a sequence of consecutive events instead of a single event when detecting $\Phi$. The value of a local variable related to $\Phi$ may not change during every event. A consecutive sequence of states in a process in which the values of the local variables related to $\Phi$ remain unchanged can be considered as identical states with respect to the detection of $\Phi$. An interval is a maximal sequence of events such that the values of the local variables related to $\Phi$ are the same after the occurrence of every event in the sequence. A process begins a new interval if an event changes the value of its local variable related to $\Phi$.

Process $P_i$ maintains an interval clock $V_i$ consisting of $n$ components. Process $P_i$ increments the $i^{th}$ component of $V_i$ whenever it begins a new interval. When a process sends a message, it timestamps the message with the current value of its interval clock. When $P_i$ receives a message with timestamp $T$, $V_i[j]$ is set to $\max(V_i[j], T[j])$ for all $j$. The timestamp of interval $I_i$ is denoted by $TS(I_i)$. For an interval $I_i$ in $P_i$, $TS(I_i)$ is the value of the updated interval clock $V_i$ when the first event of the interval $I_i$ occurred.

We say that two intervals $I_i$ and $I_j$ are consistent if $TS(I_i)[j] \leq TS(I_j)[j]$ and $TS(I_j)[i] \leq TS(I_i)[i]$. A global interval is a collection of intervals with one
interval from every process such that the intervals are pairwise consistent. A
global interval \( GI = < I_1, \ldots, I_n > \) is consistent if \( I_i \) and \( I_j \) are consistent for all \( i, j \). The set of all global intervals forms a lattice. A node in a lattice is a global interval, and there is an edge between global intervals \( GI_i \) to \( GI_j \) if the computation can proceed from one global interval to another by executing a sequence of events (interval) in a process. Our claim is that it is sufficient if we check all the global intervals instead of all the global states to detect Possibly(\( \Phi \)) for any arbitrary predicate. We first present some definitions that are useful in proving this claim.

For an event \( e \), \( \text{mincut}(e) \) is a consistent cut \( C \) such that \( e \in C \) and for any consistent cut \( C' \) if \( e \in C' \), then \( C \subseteq C' \). The \text{wavefront} of a cut \( C \) is \( < e_1, \ldots, e_n > \) such that \( e_i \) is in \( P_i \) and for any \( e_i' \) in \( P_i \), if \( e_i \rightarrow e_i' \) then \( e_i' \not\in C \) [?]. Let \( I.first \) and \( I.last \) denote the first event and the last event of the interval \( I \), respectively.

**Theorem 8** There exists a consistent global interval at which \( \Phi \) is true if and only if there exists a consistent cut (global state) at which \( \Phi \) is true.

**Proof** (\( \Rightarrow \)) Let \( GI = < I_1, \ldots, I_n > \) be a consistent global interval at which \( \Phi \) is true. From the definition of consistent global interval, it is clear that \( I_i \) and \( I_j \) are consistent, for any \( i \) and \( j \). Consider any event \( e \) in \( P_i \) such that \( I_i.last \rightarrow e \). Then, \( e \not\in I_j.first \), since \( I_i \) and \( I_j \) are consistent. Therefore, \( e \not\in \text{mincut}(I_j.first) \).

Thus, for any \( e \) in \( P_i \)

\[
I_i.last \rightarrow e \Rightarrow e \not\in \text{mincut}(I_j.first)
\]  \hfill (4.1)

Let \( C = \bigcup_{i=1}^{n} \text{mincut}(I_i.first) \). \( C \) is the supremum of \( n \) consistent cuts, hence \( C \) is also a consistent cut [?]. Let the wavefront of \( C \) be \( < e_1, \ldots, e_n > \). To prove that \( \Phi \) is true at \( C \), it is sufficient if we show that \( e_i \) is in \( I_i \) for all \( i \). Now for any \( i \), either \( e_i = I_i.first \), or \( I_i.first \rightarrow e_i \). If \( e_i = I_i.first \), then \( e_i \) is in \( I_i \).
Now consider the case in which $I_i.first \rightarrow e_i$. Since $e_i \in C$, there exists a $j$ such that $e_i \in mincut(I_j.first)$. From (4.1) (by taking contrapositive) it is clear that $I_j.last \not\rightarrow e_i$. This implies that $e_i \rightarrow I_i.last$, or $e_i = I_i.last$. Therefore, $e_i$ is in $I_i$. Hence $\Phi$ is true at the consistent cut $C$.

$(\Leftarrow)$ Let $<e_1, \ldots, e_n>$ be the wavefront of the cut at which $\Phi$ is true. Let $e_i$ be in an interval $I_i$ for all $i$. Consider any $i$, $j$. Since $e_i$ and $e_j$ are consistent, for any $e$ in $P_j$ such that $e_j \rightarrow e$, $e \neq e_i$. Observe that $I_i.first = e_i$, or $I_i.first \rightarrow e_i$ and $e_j = I_j.last$, or $e_j \rightarrow I_j.last$. Therefore, $e \neq e_i$ (where $e$ is an event such that $I_j.last \rightarrow e$ and $e$ is in $P_j$). Therefore, $e \neq I_i.first$. Hence, $TS(I_i)[j] \leq TS(I_j)[j]$.

Similarly, it can shown that $TS(I_j)[i] \leq TS(I_i)[i]$. Thus, $<I_1, \ldots, I_n>$ is a consistent global interval and $\Phi$ is true at this global interval.

Theorem 8 implies that it is sufficient to consider global intervals instead of global states to detect Possibly($\Phi$). When a process begins a new interval, it sends the timestamp of the interval and the value of the local variable related to $\Phi$ to the monitor process. Algorithm POSSIBLY described in Figure 4.1 and the parallel algorithm described in Figure 4.5 can use global intervals to detect Possibly($\Phi$). The successor and predecessor functions can be computed using the timestamps of the intervals. In a process, the number of intervals can be considerably less than the number of events if every event does not change the value of the local variable. Therefore, the total number of global intervals can be substantially less than the total number of global states, improving the performance of our algorithms to detect Possibly($\Phi$) both in space and time.

The linear space algorithm presented in Section 4.3 cannot use global intervals without increasing the space complexity for the following reasons. The number of intervals that directly depends on an interval can be $n$ in the worst case and hence the size of $dep(e)$ can be $n$. Also, to find whether a global interval is consistent we need an interval clock (whose size is $n$ integers).

Our concept of interval clock is an extension of weak vector clock used by
Marzullo and Neiger [2]. A process $P_i$ increments its $i^{th}$ component (terminates an interval) either when it executes an event that potentially changes $\Phi$, or when it executes a receive event through which it perceives that another process has potentially changed $\Phi$ [2]. But in our technique, a process terminates an interval only when it executes an event that changes a value of a variable related to $\Phi$. A process $P_i$ may not change the variable related to $\Phi$ but it may receive messages from a process $P_j$ that frequently changes its variable related to $\Phi$. If we use our method, process $P_i$ will contain only one interval; whereas if we use the weak vector clock, the number of intervals in $P_i$ can be more depending on the number of messages it receives from $P_j$. Hence, the number of global intervals in our case can be considerably less than the number of global states obtained by using weak vector clocks.

Figure 4.8: An example to compare interval clocks and weak vector clocks

To compare interval clocks and weak vector clocks, consider the example shown in Figure 4.8. The local variable of $P_1$ is $a$, and its value is changed by five events. The local variable of $P_2$ is $b$, and its value is changed once. If we use weak vector clocks, the number of global states that must be tested is 26; whereas the number of global intervals is 9 if we use interval clocks. The difference can be substantial if the number of processes is large.
4.6 Concluding Remarks

In this chapter, we presented space efficient algorithms to detect Possibly(Φ) for any arbitrary predicate Φ. Reduction in space is achieved by traversing the lattice in a depth first fashion and by not storing the nodes that are already visited. We have then presented a parallel algorithm to detect Possibly(Φ). Our experimental study shows that the speedup achieved is close to the optimal value. Also, we have improved the performance of our algorithm, both in space and time, by increasing the granularity of an execution step from an event to a sequence of events. If every event does not change the values of the variables that are related to the predicate Φ, the number global intervals will be considerably less than the number of global states thereby improving the performance of our algorithm.
Chapter 5

Distributed Snapshots with Causal Message Ordering

Recording a global state (taking a snapshot) is a fundamental problem in distributed systems. Since the information is distributed among the processes and there is no common clock, the processes have to coordinate to record a global state. Due to the inherent asynchrony in distributed systems, the recorded global state may not have occurred during the execution. However, the recorded global state can be useful in many applications – in detecting stable properties like “program has terminated,” “processes are deadlocked,” etc. [?], and in setting distributed breakpoints. A distributed breakpoint is a natural extension of the conventional notion of a breakpoint in a sequential program to a distributed program [?]. A distributed breakpoint is initiated by one process of a distributed system, and is used to stop the execution of all the processes such that the resulting global state is consistent.

In this chapter, we present an optimal distributed algorithm to record a global state of a distributed system with causally ordered message delivery. The message complexity of our algorithm is $O(n)$ bits where $n$ is the number of processes in the system. The algorithm is message optimal. The time complexity of our algorithm is also $O(n)$. Optimality is achieved by using some “information” already available for implementing causally ordered message delivery. We
assume that the underlying causal ordering protocol can be modified to make this information available to the snapshot algorithm. Our algorithm improves the algorithm presented by Acharya and Badrinath [?] that uses $O(n)$ control messages where each message contains $n$ integers. Several algorithms have been proposed for recording global states both in FIFO and NON-FIFO networks [?, ?, ?, ?].

In Section 5.1, we present some definitions. The snapshot algorithm is presented in Section 5.2 and the correctness of the algorithm in Section 5.3. We compare our algorithm with the algorithm of Acharya and Badrinath in Section 5.4. Section 5.5 concludes the chapter.

5.1 Preliminaries

The ordering of events in a distributed system is based on the “happened before” relation, denoted by $\rightarrow$, introduced by Lamport [?]. Causal ordering of message delivery is preserved if, for any two messages $m$ and $m'$ that have the same destination and $\text{send}(m) \rightarrow \text{send}(m')$, $\text{recv}(m) \rightarrow \text{recv}(m')$. An example of causal ordering is shown in Figure 5.1(a). Process $P_1$ first sends message $m$ to $P_3$ and then sends message $m''$ to $P_2$. Process $P_2$ sends a message $m'$ to $P_3$ after receiving message $m''$. Since $\text{send}(m) \rightarrow \text{send}(m')$, causal ordering requires $P_3$ to receive $m$ before $m'$. Note that causal ordering implies FIFO, but FIFO need not imply causal ordering. In Figure 5.1(b) the FIFO property is satisfied but causal ordering is violated. For any two messages $m$ and $m'$ sent to the same process such that $\text{send}(m) \rightarrow \text{send}(m')$, causal ordering requires that $\text{recv}(m) \rightarrow \text{recv}(m')$, whereas FIFO requires this only if $m$ and $m'$ were sent by the same process. Thus causal ordering is a stronger property than FIFO. In our model, all messages are delivered in causal order. The reader is referred to [?, ?, ?] for implementation details of causally ordered message delivery.

For definitions of consistent and complete global states, please refer to Chapter 2.
The performance of the snapshot algorithm is measured by the communication complexity. The communication complexity (worst case) is the maximum number of bits transmitted by the snapshot algorithm.

5.2 Snapshot Algorithm

Our snapshot algorithm makes use of the information already available for maintaining causal order of message delivery. For ease of exposition, we refer to the protocol that ensures causal order message delivery as the causal order protocol. The snapshot algorithm is initiated by a process by multicasting a marker message to all the processes. The causal ordering protocol at the initiator will append some information to the marker to maintain causally ordered message delivery. When the causal order protocol at a process receives a marker it saves this information appended to the marker. Subsequently, on receiving an application message, the causal order protocol checks (using the information saved) whether the sending of the marker happened before the sending of the application message. If so, it marks the application message new and delivers the message.
to the process. Otherwise it marks the application message old and delivers the message to the process. We assume that this modification can be incorporated into the protocol that maintains causal ordering.

The above mentioned modification can be done if the implementation of causal order protocol is based on vector clocks [?] or on message counting [?]. Consider the implementation based on message counting. Each process $P_i$ maintains two arrays $\text{DELIV}_i[1, \ldots, n]$ and $\text{SENT}_i[1, \ldots, n, 1, \ldots, n]$. $\text{DELIV}_i[j]$ denotes the number of messages from $P_j$ delivered to $P_i$. $\text{SENT}_i[j, k]$ is the number of messages sent by $P_j$ to $P_k$ that $P_i$ knows of. Whenever a message $m$ is sent by $P_i$, $\text{SENT}_i$ is sent with $m$. Now when the causal order protocol at $P_j$ receives the marker message, $<\text{marker}, \text{ST}>$, the causal order protocol saves the value $\text{ST}[i, j]$. On receiving a subsequent message $<m, \text{ST}_m>$, if $\text{ST}_m[i, j] \geq \text{ST}[i, j]$, then $m$ is marked new. Otherwise $m$ is marked old.

We next present the snapshot algorithm. A process that initiates the algorithm is called an initiator. The initiator may be the process in which the user wants to set a distributed breakpoint. For simplicity, we assume that only one process initiates the algorithm. (Our algorithm can be extended for multiple initiators.)

**Steps executed by the initiator**
- The initiator multicasts a marker message to all the processes and waits for a done message from each process.
- After receiving a done message from each process, the initiator multicasts a terminate message to all the processes.

**Steps executed by a process $P_i$ (including the initiator)**
- On receiving a marker message, process $P_i$ records its local state, initializes the state of all incoming channels to null, and sends a done message to the initiator.
- On receiving an application message from $P_j$, process $P_i$ checks whether the message is marked old. If so, process $P_i$ records the message as a part of the channel $C_{ji}$’s state.
• On receiving a terminate message from the initiator, process \( P_i \) terminates the snapshot algorithm.

5.3 Correctness and Complexity

Let \( \text{marker}_i \) and \( \text{terminate}_i \) denote, respectively, the copies of the marker and the terminate messages received by \( P_i \). Let \( \text{done}_i \) denote the done message sent by \( P_i \).

**Lemma 7** A message \( m \) sent by \( P_i \) to \( P_j \) after \( P_i \) records its local state is not in the global state recorded by the snapshot algorithm.

**Proof:** Recall that the marker message is multicast by the initiator to all the processes. Let \( P_i \) send a message \( m \) to \( P_j \) after \( P_i \) records its state. Clearly, \( \text{recv}(\text{marker}_i) \rightarrow \text{send}(m) \). Since \( \text{send}(\text{marker}) \rightarrow \text{recv}(\text{marker}_i) \), by transitivity, \( \text{send}(\text{marker}) \rightarrow \text{send}(m) \). Causal order message delivery ensures that \( \text{recv}(\text{marker}_j) \rightarrow \text{recv}(m) \). Therefore, \( P_j \) has recorded its state before receiving \( m \). Also, the causal order protocol marks \( m \) new. Thus, \( m \) will not be in the channel \( C_{ij} \)'s recorded state.

**Lemma 8** If a message \( m \) is sent by \( P_i \) to \( P_j \) before \( P_i \) records its local state, then \( \text{recv}(m) \) is in the recorded state of \( P_j \) or \( m \) is in the recorded state of channel \( C_{ij} \).

**Proof:** Consider a message \( m \) sent by \( P_i \) to \( P_j \) before \( P_i \) records its local state. If \( P_j \) has received \( m \) (sent by \( P_i \)) before recording the local state, then \( \text{recv}(m) \) is in the local state of \( P_j \). Assume that \( P_j \) receives \( m \) after recording its local state. We know that \( \text{send}(m) \rightarrow \text{recv}(\text{marker}_i) \). Since \( P_i \) sends a done message to the initiator after recording its state, \( \text{send}(m) \rightarrow \text{send}(\text{done}_i) \). Only after receiving done messages from all the processes the initiator multicasts terminate message.
to all the processes. Therefore, $send(m) \rightarrow send(terminate)$. Causal order message delivery ensures that $recv(m) \rightarrow recv(terminate_j)$. Message $m$ will be marked $old$ by the causal order protocol, because $send(marker) \not\rightarrow send(m)$. The snapshot algorithm at a process records an application message marked $old$ as part of the state of the channel through which it received the message. Therefore, $m$ is recorded in the state of $C_{ij}$.

The algorithm terminates since each process eventually receives the $terminate$ message from the initiator. When a process receives the $terminate$ message it has recorded its local state and the state of all its incoming channels. The causal ordering protocol can stop marking the messages as $new$ or $old$ when the snapshot algorithm terminates.

**Theorem 9** The global state recorded is consistent and complete.

**Proof:** From Lemma 7, the recorded global state is consistent, and from Lemma 8 it is complete.

**Theorem 10** The message complexity of the snapshot algorithm is $O(n)$ bits, and the algorithm is optimal with respect to message complexity.

**Proof:** The initiator sends $2n$ messages ($n$ marker and $n$ terminate messages), and all other processes send one $done$ message each. Thus the message complexity is $O(n)$ bits, since each message is of size $O(1)$ bits. For the lower bound, note that the initiator has to inform all the processes to record their respective local states. This requires at least one message bit per process. Thus the lower bound for the message complexity is $\Omega(n)$ bits. Hence the algorithm is message optimal.
5.4 Comparison

We briefly review Acharya–Badrinath algorithm before comparing it with our algorithm. Their algorithm requires that the underlying causal ordering protocol at each process \( P_i \) maintain two arrays, \( SENT_i[1, \ldots, n] \) and \( RECD_i[1, \ldots, n] \). At any instant, \( SENT_i[j] \) denotes the number of messages \( P_i \) has sent to \( P_j \) till that instant, and \( RECD_i[j] \) denotes the number of messages \( P_i \) has received from \( P_j \) till that instant. An initiator process initiates the snapshot algorithm by multicasting a token message to all the processes. After receiving the token message, process \( P_i \) records its state and replies by sending its local state and the two arrays \( SENT_i \) and \( RECD_i \) to the initiator. The initiator, after receiving replies from all the processes, computes the channels states as sequence of message-ids. A channel \( C_{ij} \)'s state is given by \( \{RECD_j[i] + 1, \ldots, SENT_i[j]\} \).

The Acharya–Badrinath algorithm is two-phased, one for sending token, and the other for sending reply. Our algorithm is three-phased, one for sending marker, second for sending done, and the third for sending terminate. However, in the Acharya–Badrinath algorithm, the global state is assembled at the initiator and the state of a channel is recorded as the sequence of message-ids of the in-transit messages. Since the initiator does not know the contents of those messages, their algorithm relies on either the senders keeping a log of the messages sent, or the receivers storing the messages as they arrive [?]. The initiator has to send another \( O(n) \) control messages to inform the processes about the states of their incoming channels. Thus, their algorithm is also three-phased if the contents of the channels’ states have to be known. In our algorithm, a process records the states of all of its incoming channels similar to Chandy–Lamport algorithm [?]. A process need not store a message that does not belong to a recorded channel state. Also, the computation performed by the initiator in our algorithm is much less compared to their algorithm where the initiator computes \( O(n^2) \) comparisons) the states of all the channels.
The Acharya–Badrinath algorithm uses $O(n)$ messages, but the reply messages are long. (The size of a message can be $O(n \log n)$ bits assuming that the number of messages sent by a process is polynomial in $n$. The message complexity of our algorithm is $O(n)$ bits. The size of messages we use is always $O(1)$ bits. We achieve low message complexity and reduce the computation overhead performed at the initiator by relying on the ability of causal ordering protocol to mark the application messages as new or old. This may require some modification to the “code” of the causal ordering protocol. Acharya–Badrinath algorithm does not require this modification, but it requires that the underlying causal ordering protocol maintain SENT and RECD arrays. These are the only differences between our assumptions and their assumptions.

5.5 Concluding Remarks

We have presented an algorithm to record a global state of a distributed system. The algorithm makes use of the information already available for maintaining causally ordered message delivery. Every process locally records its state and the states of all of its incoming channels. The message complexity of the algorithm is $O(n)$ bits. The algorithm can be extended to situations where there are multiple initiators. When a process initiates the process of recording a snapshot, the initiator stamps the marker with its id. The additional steps performed by the causal order protocol are more involved. Now an application message may be marked new with respect to one initiator, but it need not be marked new with respect to another initiator. So the messages that are marked new are also stamped with initiators’ ids. When a process receives a new message stamped with initiators ids, the message is not recorded as part of the channel state in the recordings that correspond to the stamped ids. All other steps of the algorithm are similar to that of the single initiator.
Chapter 6

Causally Ordered Message Delivery in Mobile Systems

There is a growing trend in using mobile computing environment for several applications, and it is important that the mobile systems are provided adequate support both at the system level and at the communication level. In this chapter, we consider the problem of providing a particular kind of communication support, namely, causally ordered message delivery to mobile hosts. Causal ordering is useful in several applications like management of replicated data, resource allocation, monitoring a distributed system, USENET etc., [?, ?]. Causal ordering is best suited for applications that involve human interactions from several locations [?] – such applications are typical in mobile systems. Some of the major applications of distributed mobile systems include providing information, stock trading, teleconferencing, etc. It is undesirable for a user to receive messages out of causal order. In this chapter, we present three algorithms for causal ordering in mobile systems.

6.1 Motivation

Any of the existing algorithms for causal ordering in static hosts can be executed by every mobile host, and all the relevant data structures can be stored in the
mobile hosts. This solution will work. But while designing algorithms for mobile systems, the following factors must be taken into account.

F1. The capability of mobile hosts may be very limited in terms of the disk space and processing power [?, ?]. The amount of computation performed by a mobile host should be low.

F2. The available bandwidth is low and the cost of message transmission is high in the wireless medium compared to the wired medium [?]. So the communication overhead in the wireless medium should be minimal.

F3. The number of mobile hosts (MHs) may not be known in advance and it may be substantially larger than the number of mobile support stations (MSSs). Hence it is desirable to design algorithms such that the overhead (communication and computation) does not increase with the number of mobile hosts.

F4. A mobile host may often disconnect itself from the network and reconnect at a later time. Algorithms designed for mobile systems should be able to easily handle the effect of host disconnections and connections.

If mobile hosts are made to execute the traditional causal ordering algorithms (by storing the relevant data structures in the mobile hosts), none of the above factors (F1–F4) can be satisfied. Keeping these factors in mind, we design three algorithms for causal ordering in mobile systems.

The first algorithm stores the data structures of MHs relevant to causal ordering in the MSSs, and the algorithm is executed by the MSSs on behalf of the MHs. However, the message overhead (length of the header*) is proportional to

*header of a message is the extra information sent with the message to maintain causal ordering.
the square of the number of mobile hosts. Thus, factor F3 is not satisfied. Also, the algorithm does not handle hosts disconnections very well and is not easily scalable.

Algorithm 2 eliminates the problems of Algorithm 1. The size of the message header is proportional to the square of the number of MSSs. Since the size of the header does not vary with the number of mobile hosts, the algorithm is scalable (with respect to the number of the mobile hosts) and host disconnections/connections do not pose any problem. But there may be some “inhibition” in delivering the messages to the mobile hosts. Our experimental results suggest that delay due to inhibition is less than the delay involved in transmitting and processing the long header (of each message) used in Algorithm 1. Also, the load placed on the MSSs is very small compared to Algorithm 1.

Algorithm 3 is a hybrid algorithm and is a trade-off between Algorithm 1 and Algorithm 2. Every MSS is partitioned into $k$ logical MSSs to reduce the delay due to “inhibition” in delivering the messages to MHs. However, $k$ cannot be large as this will increase the size of the header and hence the message overhead. A summary of our results is shown in Table 6.1 where $n_h$ is the number of MHs, $n_s$ is the number of MSSs, and $k$ is the number of logical MSSs per MSS. Note that in general $n_h \gg n_s$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Size of message header</th>
<th>“handoff” Complexity</th>
<th>“inhibition”</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1</td>
<td>$O(n_h^2)$ integers</td>
<td>$O(1)$ messages</td>
<td>No</td>
</tr>
<tr>
<td>Algorithm 2</td>
<td>$O(n_s^2)$ integers</td>
<td>$O(n_s)$ messages</td>
<td>Yes</td>
</tr>
<tr>
<td>Algorithm 3</td>
<td>$O(k \times n_s^2)$ integers</td>
<td>$O(k \times n_s)$ messages</td>
<td>decreases with $k$</td>
</tr>
</tbody>
</table>

Table 6.1: Comparison of the three causal ordering algorithms.
6.2 Mobility Model

A distributed mobile system consists of a set of mobile hosts and static hosts. A mobile host (MH) is a host whose geographical location can change with time while retaining its connectivity to the network. A static host is a host whose location does not change during the computation. A static host can also be a mobile support station (MSS). An MSS has the necessary infrastructures to support the mobile hosts. For simplicity, we assume that the system consists of only MSSs and MHs. A static host can be considered as an MH that does not move.

The geographical area within which an MSS supports MHs is called a cell. Communication between MHs and MSSs is through a wireless channel. An MH can communicate directly with an MSS only if the MH is located in the cell of the MSS. A mobile host may belong to at most one cell at any time. Mobile hosts communicate with other hosts through their MSSs.

MSSs are connected among themselves using wired channels. We assume that a logical channel exists between every pair of MSSs. These logical channels need not be FIFO channels; whereas the wireless channels are FIFO. Both wired and wireless channels are reliable and take an arbitrary, but finite amount of time to deliver messages. The MSSs and the wired channels constitute the static network which is similar to the model described in Chapter 2.

An example of a distributed mobile system is shown in Figure 6.1. Hosts $s_1$, $s_2$ and $s_3$ are three MSSs connected by wired channels. Hosts $h_1$, $h_2$ and $h_3$ are three mobile hosts. Initially, $h_1$ and $h_3$ are in the cell of $s_1$, and $h_2$ is in the cell of $s_3$.

A mobile host communicates with other mobile hosts through its MSS. Consider Figure 6.1. To send a message to MH $h_2$, MH $h_1$ first sends the message

---

$\dagger$ The MSS of an MH is the MSS in whose cell the MH is located.
to its MSS $s_1$. MSS $s_1$ then sends the message to MSS $s_3$, in whose cell $h_2$ is located, and requests $s_3$ to deliver the message to $h_2$. Similarly, a message for $h_1$ is first sent to MSS $s_1$, and then $s_1$ delivers the message to $h_1$. Even if $h_1$ and $h_2$ are in the same cell, they communicate through their common MSS.

A mobile host can migrate from one cell to another cell at any time. Every MSS periodically broadcasts a beacon [?]. Let MH $h_1$ move from the cell of MSS $s_1$ to the cell of $s_2$ as shown in Figure 6.1. $h_1$ discovers that it is in the cell of $s_2$ after receiving the beacon broadcast by $s_2$. MH $h_1$ informs MSS $s_2$ of $h_1$’s id and the id of its previous MSS $s_1$. A handoff procedure is then executed between $s_2$ and $s_1$. $s_2$ informs $s_1$ about $h_1$’s migration and gets the relevant information associated with $h_1$ from MSS $s_1$.

\footnote{Due to the mobility of $h_2$, MSS $s_1$ may not always know the location of $h_2$. However, there are several protocols [? , ? , ?] that ensure message delivery to mobile hosts.}
A mobile host can disconnect itself from the network by sending a \textit{disconnect} message to its current MSS and can reconnect at a later time by sending a \textit{connect} message. If an MSS receives a message for any of the disconnected mobile hosts, the message can be stored and delivered to mobile host after it reconnects or the message can be dropped, depending on the application.

\section{Simulation Model}

We have evaluate the performance our algorithms for mobile system by simulation. The model used for our simulations is described below.

Our simulation model is similar to that of [?]. The simulation is event driven and it is run on a Sparc 10 station. The events are send events, receive events, and handoff events. The bandwidth of a wired channel is assumed to be 100 Mbits/sec, and the propagation delay in a wired channel is 7 ms. For a wireless channel, the bandwidth and propagation delay are assumed to be 1 Mbits/sec and 500 \(\mu\)s, respectively.

Initially, the cells of the mobile hosts are assigned randomly. The time interval between two send events in a mobile host is an exponentially distributed random variable with a mean of \(t_s\) seconds. The amount of time a mobile host spends in a cell is also an exponentially distributed random variable with a mean of \(t_h\) seconds. The values of \(t_s\) and \(t_h\) are varied \((0.1, 1.0, 10 \text{ secs})\) to consider different scenario of communication and mobility. The destination of a message and the cell to which an MH switches are determined randomly. The size of each message is varied (exponential distribution) between 2 Kbytes and 4 Kbytes. The value of every point in the graph is an average of 1000 experiments performed.
6.3 Preliminaries

Causal ordering was first proposed for the Isis system [?]. There are several algorithms that implement causal ordering for distributed systems with static hosts [?, ?, ?]. The algorithm by Birman and Joseph appends, to every message, the history of the communications that happened before the sending of the message [?]. The size of the message header can become unbounded. However, the channels need not be reliable. The algorithm by Raynal, Schiper and Toueg, referred henceforth as the RST algorithm, is based on message counting and assumes the channels to be reliable [?]. The RST algorithm, which we will discuss subsequently, appends $N^2$ integers to every message, where $N$ is the number of hosts in the system. The algorithm by Schiper et al. [?] uses vector clocks and is somewhat similar to the RST algorithm. In this chapter we extend the RST algorithm to mobile systems.

6.3.1 RST Algorithm

The RST algorithm for causal ordering maintains two arrays, $\text{DELIV}_i[N]$ and $\text{SENT}_i[N, N]$, for each host $P_i$. $\text{DELIV}_i[j]$ denotes the total number of messages received by $P_i$ from $P_j$. $\text{SENT}_i[k, j]$ indicates $P_i$'s knowledge about the number of messages $P_k$ has sent to $P_j$. The following steps are executed at $P_i$ to ensure causal ordering.

Whenever $P_i$ sends message $M$ to $P_j$, $P_i$ appends its current value of $\text{SENT}_i$ to $M$. ($P_i$ sends $(M, \text{SENT}_i)$ to $P_j$.) Observe that $\text{SENT}_i$ contains information about the message that were sent before $M$ was sent. $P_i$ then increments $\text{SENT}_i[i, j]$ by 1.

On receiving a message containing $(M, ST)$ from $P_j$, the causal ordering algorithm at $P_i$ first checks if $\text{DELIV}_i[k] \geq ST[k, i]$ for all $k$. ($P_i$ checks whether all the messages, sent to $P_i$, that are causally dependent on $M$ have been delivered.) If so, the message $M$ is delivered to the application, $\text{DELIV}_i[j]$ is
incremented by 1, \( \text{SENT}_i[j, i] \) is set to \( ST[j, i] + 1 \), and finally \( \text{SENT}_i[i, j] \) is set to \( \max(ST[i, j], \text{SENT}_i[i, j]) \) for all \( i, j \). If not, \( M \) is buffered till \( \text{DELIV}_i[k] \geq ST[k, i] \) for all \( k \).

Consider the sample execution consisting of three hosts shown in Figure 6.2. Host \( P_1 \) sends message \( (m_1, \text{SENT}_1) \) to \( P_3 \) and increments \( \text{SENT}_1[1, 3] \) by

\[
\begin{align*}
P_3 & \quad \text{[0 0 0]} \\
\text{[0 0 1]} & \quad \text{(m3, c)} \\
0 0 0 & \quad \text{a= 0 0 0} \\
0 1 0 & \quad \text{b= 0 0 0} \\
\text{[0 1 0]} & \quad \text{(m1, a)} \\
\text{[0 0 1]} & \quad \text{(m2, b)} \\
\text{[0 0 0]} & \\
\end{align*}
\]

Figure 6.2: A sample execution

1. \( P_1 \) then sends message \( (m_2, \text{SENT}_1) \) to \( P_2 \). On receiving \( (m_2, ST_{m2}) \), \( P_2 \) updates \( \text{SENT}_2 \) based on \( ST_{m2} \). After updating \( \text{SENT}_2 \), \( P_2 \) gains the knowledge that \( P_1 \) has sent one message to \( P_3 \) before sending \( m_2 \) to \( P_2 \) (the value of \( ST_{m2}[1, 3] \) is 1). \( P_2 \) transfers this knowledge to \( P_3 \) when it sends \( (m_3, \text{SENT}_3) \) to \( P_3 \). Assume that \( m_3 \) arrives at \( P_3 \) before \( m_1 \). Now \( m_3 \) will not be delivered to the application, because \( \text{DELIV}_3[1] \nless ST_{m3}[1, 3] \). Message \( m_3 \) will be delivered only after \( m_1 \) is delivered.

6.3.2 Reliable Message Delivery

Before we explain our algorithms, we describe a simple protocol executed by an MH and its MSS to ensure reliable message delivery (without duplication) when the MH moves from one cell to another cell.
Let an MH $h_i$ be in the cell of MSS $s_j$. Every message sent by $h_i$ is first sent to $s_j$ and then $s_j$ sends the message to the destination. Similarly every message received by $h_i$ is received through MSS $s_j$. Let $\text{MH}_i\text{RSEQNO}_i$ denote the number of messages received by MH $h_i$ from MSS $s_j$. Let $\text{MH}_i\text{SEQNO}_i$ denote the number of messages sent by $h_i$ that have been received by $s_j$. $\text{MH}_i\text{RSEQNO}_i$ is maintained at MH $h_i$ and $\text{MH}_i\text{SEQNO}_i$ is maintained at the MSS of MH $h_i$. Messages sent by $s_j$ to $h_i$ are numbered sequentially in increasing order and are stored in a FIFO queue, $\text{PEND}_i\text{ACK}_i$, within $s_j$. MH $h_i$ sends an ack after receiving a message from $s_j$ and increments $\text{MH}_i\text{RSEQNO}_i$. After receiving an ack for a message it sent to $h_i$, MSS $s_j$ deletes that message from $\text{PEND}_i\text{ACK}_i$. Messages sent by $h_i$ to $s_j$ are also numbered by $h_i$ sequentially in the increasing order and are stored locally in MH $h_i$ till the messages are acknowledged. After receiving a message from $h_i$, MSS $s_i$ sends an ack to $h_i$ and increments $\text{MH}_i\text{SEQNO}_i$. Next the algorithms for causal ordering are described.

### 6.4 Algorithm 1

In this section, the RST algorithm is extended to mobile systems. Algorithm 1 consists of two modules: static module and handoff module. 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.

#### 6.4.1 Static Module

For each MH $h_i$, we maintain two arrays $\text{MH}_i\text{DELIV}_i[n_h]$ and $\text{MH}_i\text{SENT}_i[n_h, n_h]$, where $n_h$ is the number of mobile hosts in the mobile computing environment. $\text{MH}_i\text{DELIV}_i[j]$ denotes the total number of messages received by $h_i$ from $h_j$. $\text{MH}_i\text{SENT}_i[k, j]$ indicates $h_i$’s knowledge about the number of messages $h_k$ has sent to $h_j$. Assume that MH $h_i$ is in the cell of MSS $s_j$. To reduce the communication and computation overhead of MH $h_i$, these arrays are stored in MSS $s_j$. 
Since the messages from (to) \( h_i \) go through MSS \( s_j \), the causal ordering algorithm is executed by MSS \( s_j \). A message \( M \) sent to MH \( h_i \) becomes deliverable to \( h_i \) if the reception of \( M \) by \( h_i \) does not violate causal ordering. All messages for \( h_i \), received by \( s_j \), which are not yet deliverable are stored in the queue MH\_PENDING at MSS \( s_j \).

Initially, all the entries in the arrays MH\_DELIV and MH\_SENT are set to 0 and MH\_PENDING is empty. To send a message \( m \) to another MH \( h_j \), \( h_i \) first sends the message \( m \) to its MSS \( s_j \). \( s_j \) tags MH\_SENT with \( m \), and sends \((m, MH\_SENT_i)\) to the MSS of \( h_j \). There are several protocols [?, ?, ?] that ensure message delivery to mobile hosts. Any of these protocols can be used.

MSS \( s_j \), on receiving a message \((m, ST_m)\) meant for \( h_i \) from MH \( h_j \), first checks whether \( m \) is deliverable. Message \( m \) is deliverable if \( MH\_DELIV[k] \geq ST_m[i, k] \) for all \( k \). If so, \( s_j \) transmits \( m \) to \( h_i \); increments MH\_DELIV\( j \) and updates MH\_SENT\( i \). Message \( m \) is also queued by \( s_j \) in PEND\_ACK\( i \). Message \( m \) will be deleted from PEND\_ACK\( i \) after receiving an ack for \( m \) from MH \( h_i \). If \( m \) is not deliverable, \( m \) is stored in MH\_PENDING till \( m \) becomes deliverable. Whenever a message \( m \) is delivered to \( h_i \), \( s_j \) checks MH\_PENDING for any message that may be deliverable after \( m \) was delivered. A formal description of the module is given in Figure 6.3.

### 6.4.2 Handoff Module

Let \( h_i \) move from the cell of MSS \( s_j \) to the cell of MSS \( s_k \). The handoff module is then executed by \( s_j \) and \( s_k \). After entering the cell of \( s_k \), MH \( h_i \) sends the message \texttt{hello}(\( h_i, s_j, MH\_RSEQNO_i \)) to \( s_k \). Also, \( h_i \) retransmits the messages (to \( s_k \)) for which it did not receive ack from its previous MSS \( s_j \). MSS \( s_k \) then informs \( s_j \) that \( h_i \) has switched from MSS \( s_j \) to MSS \( s_k \) by sending a \texttt{handoff\_begin}(\( h_i \)) message to \( s_j \). After receiving \texttt{handoff\_begin}(\( h_i \)), \( s_j \) transfers MH\_DELIV\( i \), MH\_SENT\( i \), MH\_PENDING\( i \), and PEND\_ACK\( i \) to MSS \( s_k \) and fi-
Let MH \( h_i \) be in the cell of MSS \( s_j \).

1. On receiving message \( m \) from \( h_i \) to be sent to MH \( h_j \), MSS \( s_j \) executes the following steps.

   (a) Send \((m, \text{MH\_SENT}_i)\) to \( h_j \).

   (b) \( \text{MH\_SENT}_i[i, j] = \text{MH\_SENT}_i[i, j] + 1. \)

   (c) \( \text{MH\_SEQNO}_i = \text{MH\_SEQNO}_i + 1. \)

   (d) Send an ack to \( h_i \).

2. MSS \( s_j \), on receiving a message \( m \) for \( h_i \) from \( h_j \), executes the following steps.

   (a) If \( m \) is not deliverable to \( h_i \), queue \((m, ST_m)\) in \( \text{MH\_PENDING}_i \).

   (b) If \( m \) is deliverable, then

   i. Transmit \( m \) to \( h_i \) and queue \( m \) in \( \text{PEND\_ACK}_i \).

   ii. \( \text{MH\_DELIV}_i[j] = \text{MH\_DELIV}_i[j] + 1. \)

   iii. \( \text{MH\_SENT}_i[j, i] = ST_m[j, i] + 1. \)

   iv. \( \text{MH\_SENT}_i[k, l] = max(\text{MH\_SENT}_i[k, l], ST_m[k, l]), \) for all \( k, l \)

   v. If any message \((m, ST_m)\) in \( \text{MH\_PENDING}_i \) becomes deliverable, goto step 2(b)i.

Figure 6.3: Static Module of Algorithm 1
nally sends message \textit{handoff\_over}(h_i; \textit{MH\_SEQNO}_i) to \(s_k\). After sending the \textit{handoff\_over} message, if \(s_j\) receives any messages sent by \(h_i\) (when \(h_i\) was in \(s_j\)'s cell), MSS \(s_j\) drops them. (This scenario is possible because the wireless channels take an arbitrary amount of time to deliver messages.)

On receiving these data structures, \(s_k\) first transmits all messages in \textit{PEND\_ACK}_i with sequence number greater than \textit{MH\_RSEQNO}_i in FIFO order. Also, \(s_k\) forwards the messages (to their destinations) retransmitted by \(h_i\) with sequence number greater than \textit{MH\_SEQNO}_i. The handoff procedure is then terminated at \(s_k\). A description of the handoff module appears in Figure 6.4. If MH \(h_i\) switches to some other cell before the handoff is completed, the current handoff is completed before a new handoff begins.

\textbf{Theorem 11} Algorithm 1 ensures causally ordered message delivery.

\textbf{Proof:} Note that in Algorithm 1 the MSSs executes the RST algorithm for causal ordering on behalf of MHs. If the mobile hosts do not move, the correctness of the algorithm follows from the correctness of the RST algorithm and from the fact that the wireless channel between an MH and an MSS is FIFO. If an MH \(h_i\) switches its cell, then, during the handoff, all the messages in \textit{PEND\_ACK}_i are first sent to \(h_i\) in the FIFO order. Also, \(h_i\) retransmits the messages for which it has not got acks from its previous MSS before switching. The data structures \textit{MH\_DELIV}_i, \textit{MH\_SENT}_i and \textit{MH\_PENDING}_i are transferred to the new MSS of \(h_i\) during the handoff. The new MSS starts executing the causal ordering algorithm only after the handoff terminates (i.e., only after receiving the data structures required to maintain causal ordering).

\section*{6.4.3 Analysis}

For every message sent by MH \(h_i\), the MSS (in whose cell \(h_i\) resides) sends \textit{MH\_SENT}_i with the message. Hence, the size of the header for every message
Let MH $h_i$ switch from the cell of MSS $s_j$ to the cell of MSS $s_k$.

1. Steps executed by MSS $s_k$.

   (a) On receiving message $\text{hello}(h_i, s_j, \text{MH}\_\text{RSEQNO}_i)$ from $h_i$, send message $\text{handoff\_begin}(h_i)$ to MSS $s_j$.

   (b) Receive $\text{MH\_SENT}_i$, $\text{MH\_PENDING}_i$, $\text{PEND\_ACK}_i$ from $s_j$.

   (c) After receiving $\text{handoff\_over}(h_i, \text{MH\_SSEQNO}_i)$, transmit all the messages in $\text{PEND\_ACK}_i$ with number greater than $\text{MH\_RSEQNO}_i$ in FIFO order to $h_i$.

   (d) Send messages sent by $h_i$ with number greater than $\text{MH\_SSEQNO}_i$ to their destinations.

2. Steps executed by MSS $s_j$.

   (a) On receiving $\text{handoff\_begin}(h_i)$ from $s_k$, transfer $\text{MH\_DELIV}_i$, $\text{MH\_SENT}_i$, $\text{MH\_PENDING}_i$, $\text{PEND\_ACK}_i$ to $s_k$, and then send message $\text{handoff\_over}(h_i, \text{MH\_SSEQNO}_i)$ to $s_k$.

Figure 6.4: Handoff Module of Algorithm 1
sent over the static network is $O(n_h^2)$ integers. The handoff module uses $O(1)$ messages of size $O(n_h^2)$ numbers when MH $h_i$ switches its cell.

Now, consider the factors F1–F4 discussed in Section 6.1. Since Algorithm 1 is executed at MSSs, factors F1 and F2 are satisfied. The overhead in the wireless medium is kept minimal. But factors F3–F4 are not satisfied. An overhead of $O(n_h^2)$ integers over the static network is costly if $n_h$ is very large. Also, due to disconnections and connections, $n_h$ varies. So during disconnections, some of the entries in the arrays \texttt{MH\_DELIV}, and \texttt{MH\_SENT} may be useless. The arrays need not be static, but maintaining dynamic arrays can become complicated if the MH disconnections and connections are frequent. In addition, the processing time for updating the matrix \texttt{MH\_SENT} will be substantial for large $n_h$, and the nontrivial processing time increases the delay in delivering a message. These are reflected well in our experimental study shown in Figure 6.5. If the processing delay in updating \texttt{MH\_SENT} is ignored, the average delay experienced by a message is less than the average delay when processing time is taken into account. Algorithm 2, which is presented next, eliminates these disadvantages.

### 6.5 Algorithm 2

In Algorithm 1, messages are tagged with complete information to explicitly maintain causal ordering among the mobile hosts. In Algorithm 2, messages are tagged with sufficient information just to maintain causal ordering among the MSSs. Since the wireless channel between an MSS and an MH in its cell is FIFO, maintaining causal ordering at the static network level is sufficient if the MHs do not move. To ensure that causal ordering is not violated after an MH moves, we incorporate some steps into the handoff procedure.
Figure 6.5: Average Message delay using Algorithm 1 with and without processing delay
6.5.1 Static Module

The static module is similar to the static module of Algorithm 1, but for some of the data structures. For each MSS $s_i$, we maintain arrays $\text{MSS\_DELIV}_i[n_s]$, $\text{MSS\_SENT}_i[n_s, n_s]$, and a buffer $\text{MSS\_PENDING}_i$. (This is unlike in Algorithm 1 where we maintain these data structures for every mobile host.) Observe that the sizes of the arrays $\text{MSS\_DELIV}_i[n_s]$ and $\text{MSS\_SENT}_i[n_s, n_s]$ vary with $n_s$, the number of MSSs. The value of $\text{MSS\_DELIV}_i[j]$ indicates the number of messages (whose destination can be different MHs) received from MSS $s_j$ by MSS $s_i$. $\text{MSS\_SENT}_i[k, j]$ denotes the number of messages sent by MSS $s_k$ (not necessarily delivered) to MSS $s_j$ that $s_i$ knows of. Every MSS knows (need not be exact) about the location of the MHs. Initially, we assume that the initial locations of MHs are known to all MSSs. We show how this knowledge gets updated in the next section. In other aspects, the static module is similar to the static module of Algorithm 1 and is described in Figure 6.6.

6.5.2 Handoff Module

The handoff module is more involved when compared to the handoff module of Algorithm 1. Since causal ordering is explicitly maintained only at the MSSs level, some measures have to be taken during handoff to maintain causal ordering after an MH moves.

Before we describe the handoff module, we illustrate the problem at hand with an example. Consider mobile hosts $h_1$, $h_2$, and $h_3$. Assume that $h_1$, $h_2$ and $h_3$ are in the cells of MSSs $s_1$, $s_2$ and $s_3$ respectively. Let $h_3$ send a message $m_1$ to $h_1$ ($m_1$ will be sent to MSS $s_1$) and then send a message $m_2$ to $h_2$. Before receiving $m_1$, let $h_1$ switch to the cell of $s_2$. Now, MH $h_2$, after receiving $m_2$ from $h_3$, sends a message $m_3$ for $h_1$ to $s_2$. If $s_2$ delivers $m_3$ to $h_1$, causal ordering will be violated because $h_1$ has not yet received $m_1$. Also, $s_2$ cannot infer with the knowledge it has gained so far whether there are any in-transit messages for $h_1$. 
Let MHs $h_k$ and $h_l$ be in the cells of MSSs $s_i$ and $s_j$ respectively.

1. On receiving message $m$ from $h_k$ to be sent to the MSS of $h_l$, MSS $s_i$ executes the following step.
   
   (a) Send $(m, \text{MSS}_{i;i})$ to the MSS of $s_j$.
   
   (b) $\text{MSS}_{i;i}[i, j] = \text{MSS}_{i;i}[i, j] + 1$.
   
   (c) $\text{MH}_{i;i.\text{SEQNO}} = \text{MH}_{i;i.\text{SEQNO}} + 1$.
   
   (d) Send an ack to $h_k$.

2. MSS $s_i$, on receiving a message for $h_k$ from $s_j$ executes the following steps.
   
   (a) If $m$ is not deliverable to $h_k$, queue $(m, ST_m)$ in $\text{MH}_{i;i.\text{PENDING}}$.
   
   (b) If $m$ is deliverable, then
      
      i. Transmit $m$ to $h_k$ and queue $m$ in $\text{PEND}_{i;i.\text{ACK}}$.
      
      ii. $\text{MSS}_{i,j;i}[j] = \text{MSS}_{i,j;i}[j] + 1$.
      
      iii. $\text{MSS}_{i,j;i}[i, i] = ST_m[i, i] + 1$.
      
      iv. $\text{MSS}_{i,j;i}[k, l] = max(\text{MH}_{i,j;i}[k, l], ST_m[k, l])$, for all $k, l$
      
      v. If any message $(m, ST_m)$ in $\text{MH}_{i;i.\text{PENDING}}$ becomes deliverable, goto step 2(b)i.

Figure 6.6: Static Module of Algorithm 2
sent to $s_1$. However, if $s_2$ delivers $m_3$ after ascertaining that all the messages for $h_1$ sent to $s_1$ have been delivered, causal ordering will not be violated. Now, we describe the handoff module.

Assume that a mobile host $h_k$ switches from the cell of MSS $s_i$ to the cell of MSS $s_j$. After switching, MH $h_k$ sends $\text{hello}(h_k, s_i, \text{MH}_\text{RSEQNO}_i)$ message to $s_j$. The message $\text{hello}(h_k, s_i, \text{MH}_\text{RSEQNO}_i)$ to $s_j$ indicates that $h_k$ has switched from the cell of $s_i$ to $s_j$. On receiving this message, $s_j$ sends the message $\text{handoff}_\text{begin}(h_k)$ to $s_j$, and then broadcasts the message $\text{notify}(h_k, s_i, s_j)$ to all the MSSs. The message $\text{notify}(h_k, s_i, s_j)$ signifies that MH $h_k$ has switched from MSS $s_i$ to MSS $s_j$. An MSS $s$, on receiving $\text{notify}(h_k, s_i, s_j)$ message, updates its local knowledge about the location of MH $h_k$ and sends a $\text{last}(h_k)$ message to $s_i$. After receiving $\text{notify}(h_k, s_i, s_j)$, MSS $s$ will send messages meant for MH $h_k$ only to $s_j$ (the new MSS of $h_k$) and not to $s_i$ (the previous MSS of $h_k$).

MSS $s_i$, after receiving the message $\text{handoff}_\text{begin}(h_k)$ from $s_j$ sends message $\text{enable}(h_k, \text{PEND}\_\text{ACK}_k, \text{MH}_\text{SEQNO}_i)$ to $s_j$ and waits for $\text{last}(h_k)$ messages from all the MSSs. Meanwhile, if any message received by $s_i$ for $h_k$ becomes deliverable to $h_k$, $s_i$ marks it as old and forwards it to $s_j$.

Once $s_j$ receives the message $\text{enable}(h_k, \text{PEND}\_\text{ACK}_k, \text{MH}_\text{SEQNO}_i)$ message, it starts sending the application messages sent by $h_k$ with sequence number greater than $\text{MH}_\text{SEQNO}_i$ to their destinations. Also, $s_j$ delivers all the messages in $\text{PEND}\_\text{ACK}_k$ in the FIFO order to MH $h_k$. Duplication is avoided by delivering only those messages with sequence number greater than $\text{MH}_\text{RSEQNO}_i$. $s_j$ also delivers all the messages for MH $h_k$ that are marked old to $h_k$ in the order in which the messages arrived. Any messages for $h_k$ that are not marked old will be queued in $\text{MSS}_\text{PENDING}_j$.

MSS $s_i$ (the previous MSS of $h_k$), after receiving $\text{last}(h_k)$ from all the MSSs sends the message $\text{handoff}_\text{over}(h_k)$ to MSS $s_j$. Observe that, no messages for $h_k$ sent to $s_i$ will be in transit after $s_i$ receives $\text{last}(h_k)$ from all the MSSs. The
handoff terminates at \( s_j \) after \( \text{handoff\_over}(h_k) \) is received by \( s_j \). If \( s_j \) receives the message \( \text{handoff\_begin}(h_k) \) from some other MSS before the current handoff terminates (this can happen if \( h_k \) switches it cell), \( s_j \) will respond to the message only after the handoff terminates. A description of the handoff module is shown in Figure 6.7.

We next prove the correctness of Algorithm 2.

**Correctness Argument**

Let \( mh\_send(m) \) be the event corresponding to sending of message \( m \) by a mobile host. Let \( mh\_recv(m) \) be the event corresponding to receiving of message \( m \) by a mobile host. Let \( mss\_send(m) \) denote the event in an MSS corresponding to the sending of message \( m \) (sent by an MH) to another MSS, and \( mss\_recv(m) \) denote the event in an MSS corresponding to the receipt of \( m \) sent by another MSS (in which event \( mss\_send(m) \) happened) to be delivered to an MH.

**Lemma 9** Let \( m_1 \) and \( m_2 \) be two messages sent to MH \( h_k \). If \( mh\_send(m_1) \rightarrow mh\_send(m_2) \), then \( mss\_send(m_1) \rightarrow mss\_send(m_2) \).

**Proof:** First assume that messages \( m_1 \) and \( m_2 \) are sent by two different MHs. Since \( mh\_send(m_1) \rightarrow mh\_send(m_2) \), there exists a sequence of events such that \( mh\_send(m_1) \rightarrow mss\_send(m_1) \ldots \rightarrow mh\_recv(m) \ldots \rightarrow mh\_send(m_2) \). Since \( mh\_send(m_2) \rightarrow mss\_send(m_2) \), it follows that \( mss\_send(m_1) \rightarrow mss\_send(m_2) \). Now consider the case in which \( m_1 \) and \( m_2 \) are sent by the same mobile host. Assume that the send events \( mss\_send(m_1) \) and \( mss\_send(m_2) \) happened in two different MSSs. (The mobile host may have switched cells in between \( mh\_send(m_1) \) and \( mh\_send(m_2) \).) Otherwise, the result is obvious. Let the mobile host switch from MSS \( s_1 \) to MSS \( s_2 \). MSS \( s_2 \) will send \( m_2 \) only after it receives an \( \text{enable} \) message from MSS \( s_1 \) as part of handoff. MSS \( s_1 \) would have sent \( m_1 \) before sending the \( \text{enable} \) message. Hence, \( mss\_send(m_1) \rightarrow mss\_send(m_2) \).
Assume that MH $h_k$ switched from the cell of $s_i$ to $s_j$.

1. Steps executed by $s_j$.

   (a) After receiving $\text{hello}(h_k, s_i, \text{MH}_{-}\text{RSEQNO}_k)$ from MH $h_k$, send $\text{handoff}_\text{begin}(h_k)$ to MSS $s_i$, and then send $\text{notify}(h_k, s_i, s_j)$ to all the MSSs.

   (b) On receiving $\text{enable}(h_k, \text{PEND\_ACK}_k, \text{MH\_SEQNO}_k)$, deliver messages in $\text{PEND\_ACK}_k$ in FIFO order, and start sending messages sent by MH $h_k$ to their destinations. Use $\text{MH\_SEQNO}_k$ and $\text{MH\_RSEQNO}_k$ to avoid duplication of messages.

   (c) Deliver any messages for $h_k$ from $s_i$ that are marked $\text{old}$ to $h_k$. Queue all other messages for $h_k$ in $\text{MH\_PENDING}_j$.

   (d) Terminate on receiving $\text{handoff\_over}(h_k)$ from $s_i$.

2. Steps executed by MSS $s_i$

   (a) Send $\text{enable}(h_k, \text{PEND\_ACK}_k, \text{MH\_SEQNO}_k)$ message after receiving $\text{handoff}_\text{begin}(h_k)$ message from $s_j$. After sending $\text{enable}$ message drop any message received from $h_k$.

   (b) If any message for $h_k$ becomes deliverable, mark it as $\text{old}$ and forward it to $s_j$.

   (c) After receiving $\text{last}(h_k)$ from all MSSs, send $\text{handoff\_over}(h_k)$ to $s_j$.

3. Steps executed by all the MSSs.

   (a) On receiving $\text{notify}(h_k, s_i, s_j)$, update the local knowledge about $h_k$’s location to $s_j$ and send $\text{last}(h_k)$ to $s_i$.

Figure 6.7: Handoff Module of Algorithm 2
Theorem 12 Let $m_1$ and $m_2$ be two messages sent to $MH_h$. If $mh_{send}(m_1) \rightarrow mh_{send}(m_2)$, then $mh_{recv}(m_1) \rightarrow mh_{recv}(m_2)$.

Proof: From Lemma 9, $mss_{send}(m_1) \rightarrow mss_{send}(m_2)$. If $m_1$ and $m_2$ are sent to the same MSS, then from the correctness of the RST algorithm it follows that $mss_{recv}(m_1) \rightarrow mss_{recv}(m_2)$. Now, $m_1$ and $m_2$ are delivered to $MH_h$ directly or through handoff (if $h_k$ switches cells). In either case $mh_{recv}(m_1) \rightarrow mh_{recv}(m_2)$. Now consider the case in which $m_1$ and $m_2$ are sent to two different MSSs. Let $m_1$ was sent to MSS $s_i$ and $m_2$ was sent to MSS $s_j$. This can happen only if $MH_h$ switched from the cell of MSS $s_i$ to the cell of $s_j$. Now assume that $m_1$ was not received by $MH_h$ when it was in the cell of $s_i$. (Otherwise, the result is fairly obvious.) Observe that MSS $s_j$ will not deliver $m_2$ to $h_k$ till the handoff procedure is completed. MSS $s_i$ sends $handoff_{over}(h_k)$ message to $s_j$ only after it receives $last(h_k)$ from each MSS. By the time MSS $s_i$ receives $last(h_k)$ from all the MSSs, $s_i$ would have forwarded $m_1$ to $s_j$ or it would have transferred $m_1$ as part of $PEND_ACK_k$. In either case, $s_j$ will deliver $m_1$ before the handoff is terminated. Message $m_2$ will be delivered after the handoff is terminated. Hence, $mh_{recv}(m_1) \rightarrow mh_{recv}(m_2)$.

6.5.3 Analysis

Since the size of $mss_{SENT}$ is $n_h^2$, the size of each message header over the wired network is $O(n_h^2)$ integers. The overhead does not depend on $n_h$, the number of MHs. Clearly, factors F3–F4 are satisfied. MH connections/disconnections do not affect the size of the arrays $mss_{DELIV}$ and $mss_{SENT}$. During handoff, a notify message has to be sent to all the MSSs, and all the MSSs send $last$ messages. Hence, the handoff module uses $O(n_s)$ messages. The storage requirement of Algorithm 2 and the load placed on the MSSs are less than that of Algorithm 1.

Though the handoff module is involved, it does not affect the performance compared to Algorithm 1 due to the following reasons. (i) $MH_h$ does not wait
Figure 6.8: Comparison of Algorithm 1 and Algorithm 2 with respect to message delay with \( n_x = 10 \)

for the handoff module to terminate to receive messages. It continues to receive old messages. (ii) Messages sent by \( h_k \) for other MHs are sent by \( s_j \) (the new MSS of \( h_k \)) immediately after \( s_j \) receives an enable message.

The drawback of Algorithm 2 is the possibility of a message being temporarily “inhibited” during delivery to an MH. There is an inhibition in delivering a message to an MH if it is queued in \texttt{MSS\_PENDING} even though the delivery of the message does not violate causal ordering. Messages may be inhibited because causal ordering is explicitly implemented among the MSSs. Reception of a message may violate causal ordering from an MSS’s point of view; whereas its delivery to an MH may not violate causal ordering from the MH’s point of view. However, this delay is less than the delay introduced by Algorithm 1 in transmitting and processing the header of each message. The average delay in delivering a message in Algorithm 2 is considerably less than the delay in Algo-
rithm 1 when \( n_h \) increases, as shown in Figure 6.8. When \( n_h \) is less than 30 the message header in both the algorithms are comparable in size. The message delay in Algorithm 2 is more than that of Algorithm 1 due to the inhibition inherent in Algorithm 2. However, as \( n_h \) increase the delay due to processing the message header in Algorithm 1 dominates.

6.6 Algorithm 3

This Algorithm reduces the delay in delivering the messages to MH due to inhibition, the drawback of Algorithm 2, without much increase in the message overhead. The algorithm achieves this by partitioning every physical MSS into \( k \) logical MSSs.

If an MH enters the cell of an MSS, the MH will be allocated to one of the logical MSS depending on the load in each logical MSS of the MSS. The MHs will communicate with the other MHs through their logical MSSs. Every logical MSS maintains two arrays \( \text{MSS\_DELIV}[k \times n_s] \) and \( \text{MSS\_SENT}[k \times n_s, k \times n_s] \) and a queue \( \text{MSS\_PENDING} \). The algorithm is same as Algorithm 2 except for the fact that causal ordering is explicitly maintained among the logical MSSs. The size of the message header is \( O(k^2 \times n_s^2) \).

Messages to MHs that belong to different logical MSSs will not inhibit each other though they may be in the same cell. Thus, as \( k \) increases, the unnecessary delay in delivering the message to MH decreases. However, \( k \) cannot be large, since the size of the message header will increase and, as a result, the time to process the message header will become a dominating factor. In Figure 6.9, the average message delay initially decreases when \( k \) increases. But when \( k \) becomes large the average message delay increases.
Figure 6.9: Message delays for various values of $k$ with $n_s = 10$. 

6.7 Concluding Remarks

In this chapter, we have considered the problem of causally ordered message delivery to mobile hosts. A direct implementation of an existing algorithm can incur a large message overhead. Algorithm 2 reduces the overhead and it is scalable since the overhead of a message does not vary with the number of mobile hosts. The message overhead can be further reduced by sending the SENT arrays incrementally, in a manner similar to the scheme proposed by Singhal and Kshemakalyani for efficient implementation of vector clocks [7]. Algorithm 3 reduces the delay due to inhibition, inherent in Algorithm 2, in delivering the messages to MHs by partitioning every MSS into $k$ logical MSSs. The value of $k$ should not be large as a large value of $k$ increases the message overhead.

Our algorithms for causal ordering in mobile systems are based on the RST algorithm for static hosts. Other algorithms can also be modified to work
in mobile systems. Only the static modules of our algorithms are based on the RST algorithm. The handoff modules are fairly independent of the RST algorithm. If one wants to handle host mobility in an existing distributed system that supports causal ordering, handoff module of Algorithm 2 can be used with some modification to the existing algorithm that provides causal ordering at the static network level.
Chapter 7

Tolerating Mobile Support Station Failures

When a mobile support station fails, vital state information of the live mobile hosts stored in the support station is lost. The mobile hosts are forced to wait till the support station recovers. This leads to new problems in providing a highly fault-tolerant mobile computing environment. As mobile computing environments become widely popular, it is important to design mobile computing environments where “uninterrupted” service is available in spite of support station failures.

Conventional recovery schemes [?, ?, ?, ?, ?, ?] can be used to recover a failed mobile support station. However, the (actual) down time for support stations may be long and hence the time for recovery can be long. It is undesirable to make the live mobile hosts (whose partial states are stored in the failed support stations) wait for a long period of time. In many critical applications, we need a fault-tolerant scheme that does not block mobile hosts while their support stations recover from failures.

In this chapter, we propose two schemes to tolerate the simultaneous failures of up to \( k \) mobile support stations. In both schemes, the information stored at a mobile support station is replicated at several “secondary” support stations. The two schemes differ in the way in which replication is achieved. When a mobile support station fails, the mobile hosts under its coverage can switch to one of their secondary support stations and continue their operations. To switch to one of its secondary support station, a mobile host may move. If movement of
a mobile host is impossible or undesirable, then the network may be designed to
cover each mobile host by at most $k + 1$ support stations. Thus the “affected”
mobile hosts need not wait for the failed support stations to recover. We also
present several methods for selecting the set of secondary support stations for a
mobile host and discuss methods to cope with mobile host failures.

The organization of the chapter is as follows. Two schemes to tolerate
mobile support station failures are presented in Section 7.1. In Section 7.2, the
performance of both the schemes are analyzed through simulation. Various re-
lated issues are discussed in Section ??, and Section ?? concludes the chapter.

7.1 Schemes for tolerating MSS failures

7.1.1 Preliminaries

Due to the energy and resource limitations, a part (or all) of the data structures
and state information of a mobile host is stored in its MSS [?, ?]. Let $st_{\text{info}}(h)$
denote the state information of mobile host $h$ stored in $h$’s MSS. The information
stored in $st_{\text{info}}(h)$ depends on the application, and the actual details are not re-
quired to explain our schemes. For example, $st_{\text{info}}(h)$ for Algorithm 1 described
in chapter 6 will be the arrays $\text{MH\_SENT}$, $\text{MH\_DELIV}$ and $\text{MH\_PENDING}$.

Updating $st_{\text{info}}(h)$ is easy for an MSS (in whose cell $h$ resides), since each
message sent by/to a mobile host $h$ is sent through the MSS.

When an MSS $s$ fails, the state information (of the mobile hosts in $s$’s
cell) maintained by $s$ is lost. The mobile hosts in $s$’s cell cannot continue to
operate until the state information is available. Thus they are forced to wait till
$s$ recovers. We overcome this problem by maintaining $st_{\text{info}}(h)$ at several MSSs.

With every mobile host $h$, we associate a set of secondary MSSs denoted
by $\text{sec\_mss}(h)$. Methods to select secondary MSSs are described in Section 7.1.4.
In this section, we assume that $\text{sec\_mss}(h)$ is available for each mobile host $h$. 
The cardinality of $sec\_mss(h)$ must be at least $k$ to tolerate $k$ MSSs failures. The primary MSS of a mobile host $h$, denoted as $pri\_mss(h)$, is the MSS in whose cell $h$ resides. For each $h$, $sec\_mss(h)$ represents the MSSs where $st\_info(h)$ is replicated. Let $sender(m)$ and $dest(m)$ denote, respectively, the sender and the destination of message $m$. We also assume that mobile support stations obey the fail-stop failure model [?] and there is a mechanism by which the failure of an MSS can be detected by its neighboring MSSs and the mobile hosts in its cell.

In the following subsections, we present two schemes to tolerate MSS failures.

The first scheme takes a pessimistic approach. Before updating $st\_info(h)$, $pri\_mss(h)$ ensures that copies of $st\_info(h)$ at all the secondary MSSs of $h$ are updated. The copies of $st\_info(h)$ at $pri\_mss(h)$ and at all MSSs in $sec\_mss(h)$ are consistent. However, there is a delay in delivering messages to MSS/mobile hosts. But when $pri\_mss(h)$ fails, $h$ can switch to any one of the MSSs in $sec\_mss(h)$ and continue its computation with no delay.

The second scheme takes an optimistic approach, and $st\_info(h)$ is updated asynchronously at $sec\_mss(h)$. There is no delay in delivering the messages to MSS/mobile hosts. However, a recovery protocol must be run when $h$ switches to one of the MSSs in $sec\_mss(h)$ after $pri\_mss(h)$ fails.

7.1.2 Pessimistic Replication

Normal operation
During the normal operation, when MSS $s_1$ sends a message (whose destination is mobile host $h$) to $pri\_mss(h)$, MSS $s_1$ stores the message in a buffer. After receiving an acknowledgement from $pri\_mss(h)$, MSS $s_1$ deletes the message from its buffer. If $s_1$ receives a message notifying the failure of $pri\_mss(h)$, $s_1$ will find the new primary MSS of $h$ and retransmit the message to that MSS. Similarly, when a mobile host sends a message to its primary MSS (to be delivered to another
mobile host), it will wait for an acknowledgement from its primary MSS. Before receiving an acknowledgement, if the mobile host receives a message notifying the failure of its primary MSS, it establishes connection with another MSS, and retransmits the message. This ensures that messages are not lost when an MSS fails.

As stated earlier, every message sent by mobile host $h$ is first sent to $pri_{mss}(h)$. Similarly, every message sent to mobile host $h$ is first received by $pri_{mss}(h)$. Let $m$ be a message such that either $sender(m) = h$ or $dest(m) = h^*$. $st_{info}(h)$ may be updated when $pri_{mss}(h)$ receives $m$. $pri_{mss}(h)$ sends the message $copy(h, m)$ to all the MSSs in $sec_{mss}(h)$ if $st_{info}(h)$ must be updated. If $pri_{mss}(h)$ does not need to update $st_{info}(h)$ on receiving $m$, then no $copy(h, m)$ message is sent. We assume that MSSs are able to identify messages that require an update to $st_{info}(h)$. On receipt of the $copy(h, m)$ message, every MSS in $sec_{mss}(h)$ updates $st_{info}(h)$ stored locally and sends an acknowledgement to $pri_{mss}(h)$. Each MSS of $sec_{mss}(h)$ updates its copy of $st_{info}(h)$ as though mobile host $h$ is in its cell. After receiving acknowledgements from all the MSSs in $sec_{mss}(h)$, $pri_{mss}(h)$ updates $st_{info}(h)$, delivers the message to $dest(m)$, and sends an acknowledgement to $sender(m)$. A description of the scheme is shown in Figure 7.1.

**Handoff procedure**

A mobile host may move from the cell of its primary MSS to the cell of another MSS at any time. Let mobile host $h$ initially be in the cell of MSS $s_1$. Assume that $h$ moves to the cell of another MSS, say $s_2$. Mobile host $h$ establishes communication with $s_2$ by using the *beacon* protocol [?], and then $h$ sends the *id* of its “old” MSS ($s_1$) to $s_2$. A handoff procedure is executed between $s_1$ and $s_2$.

---

*Our replication schemes do not distinguish between the cases when $sender(m) = h$ and when $dest(m) = h$, since we are not concerned with what is updated in $st_{info}(h)$; what is updated in $st_{info}(h)$ may be different in both the cases and it depends on the application.*
Steps executed by $\textit{pri}_{\text{mss}}(h)$

1. Let $m$ be a message such that either $\textit{sender}(m) = h$ or $\textit{dest}(m) = h$. On receiving message $m$, $\textit{pri}_{\text{mss}}(h)$ checks whether an update to $\textit{st}_{\text{info}}(h)$ is required.

2. If so, $\textit{pri}_{\text{mss}}(h)$ multicasts the message $\textit{copy}(h, m)$ to all the MSSs in $\textit{sec}_{\text{mss}}(h)$, and stores $m$ in a buffer.

3. Wait for acknowledgements from all the MSSs in $\textit{sec}_{\text{mss}}(h)$. If a failure notification of an MSS in $\textit{sec}_{\text{mss}}(h)$ arrives before the acknowledgement, then $\textit{pri}_{\text{mss}}(h)$ will not wait for an acknowledgement from that failed MSS.

4. Update $\textit{st}_{\text{info}}(h)$. Send $m$ to $\textit{dest}(m)$ and delete $m$ from the buffer. After receiving an acknowledgement from $\textit{dest}(m)$ for the receipt of $m$, $\textit{pri}_{\text{mss}}(h)$ sends an acknowledgement to $\textit{sender}(m)$.

Steps executed by MSS $s$ in $\textit{sec}_{\text{mss}}(h)$

i. On receiving a message $\textit{copy}(h, m)$ from MSS $s_1$, update $\textit{st}_{\text{info}}(h)$ stored locally and send an acknowledgement to $s_1$.

ii. If mobile host $h$ establishes connection with $s$ after $\textit{pri}_{\text{mss}}(h)$ fails, set $\textit{pri}_{\text{mss}}(h)$ to $s$. Send $\textit{st}_{\text{info}}(h)$ to the new MSS in $\textit{sec}_{\text{mss}}(h)$. Request MSSs that are no longer in $\textit{sec}_{\text{mss}}(h)$ to delete $\textit{st}_{\text{info}}(h)$.

Figure 7.1: Pessimistic Replication
MSS $s_2$ notifies $s_1$ that $h$ has switched cells by sending $\text{handoff\_begin}(h)$ message to $s_1$. MSS $s_1$ waits for all the pending acknowledgements from $\text{sec\_mss}(h)$, updates $\text{st\_info}(h)$, and then sends $\text{st\_info}(h)$ to $s_2$. MSS $s_1$ also sends any pending messages (to be delivered to mobile host $h$) to $s_2$. MSS $s_1$ then sends $\text{handoff\_completed}(h)$ message to $s_2$. However, if $s_2$ belongs to $\text{sec\_mss}(h)$, then $\text{st\_info}(h)$ need not be sent to $s_2$, so $s_1$ just sends $\text{handoff\_completed}(h)$ to $s_2$. This is an added advantage of our fault tolerant scheme. If the set $\text{sec\_mss}(h)$ changes, MSS $s_2$ sends $\text{st\_info}(h)$ to all the new MSSs in $\text{sec\_mss}(h)$ and notifies MSS $s_1$ and the old MSSs in $\text{sec\_mss}(h)$ to delete $\text{st\_info}(h)$ from their storage.

**Operation after failure**

When $\text{pri\_mss}(h)$ fails, $h$ switches to another MSS, say $s$, and notifies MSS $s$ about the failure of $\text{pri\_mss}(h)$. MH $h$ also sends the set $\text{sec\_mss}(h)$ to MSS $s$. If $s \in \text{sec\_mss}(h)$, then $h$ can continue its operation since a copy of $\text{st\_info}(h)$ is stored in MSS $s$. If $s \notin \text{sec\_mss}(h)$, then $s$ acquires a copy of $\text{st\_info}(h)$ from one of the MSSs in $\text{sec\_mss}(h)$. MSS $s$ then becomes the primary MSS of $h$.

The set $\text{sec\_mss}(h)$ may change as a result of mobile host $h$ switching to $s$. All the “new” MSSs in $\text{sec\_mss}(h)$ get a copy of $\text{st\_info}(h)$ from $s$, and all the “old” MSSs in $\text{sec\_mss}(h)$, which no longer belong to $\text{sec\_mss}(h)$, delete $\text{st\_info}(h)$. MSS $s$, the new primary MSS of $h$, may receive $\text{copy}(h, m)$ sent by the old primary MSS of $h$ before failure of old primary MSS. In this case, MSS $s$ will send $m$ to all the MSSs in $\text{sec\_mss}(h)$ and wait for acknowledgements. On receiving acknowledgements from all the secondary MSSs of $h$, $s$ will process $m$ and deliver it to $\text{sender}(m)$.

Now consider the failure of an MSS during handoff. Let MH $h$ switch from the cell of $s_1$ to $s_2$, and let $s_2$ fail during handoff. Then MH $h$ will switch to another MSS in $\text{sec\_mss}(h)$ and initiate a new handoff procedure as though the handoff was not initiated with $s_2$. The previous MSS of MH $h$ will still be $s_1$ and not $s_2$. If MSS $s_1$ has also failed, then the new MSS of $h$ will acquire $\text{st\_info}(h)$
from one of the secondary MSSs of $h$.

**Theorem 13** If pessimistic replication strategy is used, MH $h$ can switch to one of the MSSs in sec.mss($h$) and continue its computation when pri.mss($h$) fails.

**Proof:** Consider the effects of the failure of pri.mss($h$) at various stages of the scheme shown in Figure 7.1. Let $s$ be an MSS to which $h$ has switched after pri.mss($h$) fails. Assume that $s \in$ sec.mss($h$). (If $s \notin$ sec.mss($h$), then $s$ acquires a copy of $st_{\text{info}}(h)$ from one of the secondary MSSs of MH $h$.)

If a failure occurs during step 1, then message $m$ has not been sent to sec.mss($h$) and $st_{\text{info}}(h)$ has not been updated. Mobile host $h$ will switch to $s$, and since an acknowledgement for $m$ is not yet sent to the sender of $m$, sender($m$) will retransmit the message.

If pri.mss($h$) fails during step 2 or step 3, sec.mss($h$) may or may not have received the message $\text{copy}(h, m)$. If MSS $s$ has not received the message $\text{copy}(h, m)$, the scenario is the same as the previous one. If MSS $s$ has received the message $\text{copy}(h, m)$, then, when the sender of $m$ retransmits $m$, $s$ will send $m$ to dest($m$) without updating $st_{\text{info}}(h)$.

If MSS $s$ fails during step 4, the effects are similar to the effects of failure during step 2, but for the difference where dest($m$) may receive duplicate copies of $m$ when sender of $m$ retransmits $m$. If dest($m$) has already received $m$, then $m$ will be discarded.

If an MSS fails while executing a handoff procedure for MH $h$, then MH $h$ will switch to another MSS and initiate a new handoff.

Thus, when an MSS failure occurs, mobile hosts in its cell can switch to one of their secondary MSSs and continue their computation without any blocking.
7.1.3 Optimistic Replication

Normal operation
The primary MSS of $h$ updates $st_{\text{info}}(h)$ on receiving message $m$ sent to $h$, or on receiving message $m$ from $h$. In this scheme, $pri_{\text{mss}}(h)$ sends message $copy(h, m)$ to all the MSSs in $sec_{\text{mss}}(h)$ similar to pessimistic scheme, but does not wait for acknowledgements from $sec_{\text{mss}}(h)$. Message $m$ is immediately processed and $st_{\text{info}}(h)$ is updated locally by $pri_{\text{mss}}(h)$, and then $m$ is delivered to $dest(m)$. Hence, the delay in delivering $m$ is less in this scheme than the delay in the pessimistic scheme.

A secondary MSS stores the $copy(h, m)$ message it receives in a log (volatile or stable) without processing the copy message immediately. The secondary MSS processes the messages in the log for MH $h$ and updates $st_{\text{info}}(h)$ occasionally (whenever the secondary MSS is idle or after a predetermined time interval). A secondary MSS does not send acknowledgements to the $pri_{\text{mss}}(h)$ for $copy$ messages, unlike in pessimistic scheme.

Handoff Procedure
The handoff procedure for this scheme is similar to the handoff procedure for pessimistic scheme except for one difference. When MH $h$ switches to MSS $s$ in $sec_{\text{mss}}(h)$, the MSS $s$ may not have the recent value of $st_{\text{info}}(h)$. MSS $s$ can get $st_{\text{info}}(h)$ from the previous MSS of $h$, or it can process the messages stored (locally) in the log and update $st_{\text{info}}(h)$, depending on the load of MSS $s$. All other operations are the same as in the handoff procedure described in the pessimistic scheme.

Operation after failure
$pri_{\text{mss}}(h)$ processes a message for mobile host $h$ and updates $st_{\text{info}}(h)$ immediately after sending the message $copy(h, m)$ to all the MSSs in $sec_{\text{mss}}(h)$. Hence $st_{\text{info}}(h)$ at $pri_{\text{mss}}(h)$ may get updated while the message $copy(h, m)$ are in transit to the MSSs in $sec_{\text{mss}}(h)$. This may lead to $st_{\text{info}}(h)$ being inconsistent
among pri_mss(h) and the MSSs in sec_mss(h). In case of an MSS failure, this inconsistency must be resolved.

Let us consider the failure of an MSS, say s1. Let mobile host h be in the cell of s1 and let h switch to the cell of MSS s2 ∈ sec_mss(h). s2 is the new primary MSS of h. Let m be the last message that was processed by s1 for h. Consider the case where s1 sends message copy(h, m) to sec_mss(h), processes m, updates st_info(h), delivers m to dest(m), and then fails. Now assume that MH h switches to the cell of MSS s2. In this scenario, h has the most recent state; it is possible that s2 has not yet received the message m, or even some messages prior to m. Hence, s2 has an outdated copy of st_info(h), i.e. s2 has a chronologically earlier state for h.

To make the copy of st_info(h) in MSS s2 consistent with the MH h, a recovery procedure is executed by s2. The recovery procedure executed by s2 consists of two steps:

1. Processing all the messages stored in the log for h, and
2. Processing all the in-transit messages from s1.

While executing step 1 is simple, for step 2, s2 has to ensure that it has received all the in-transit messages of the asynchronous channels. We ensure that s2 has received all the in-transit messages from s1 by using a communication primitive return flush [?].

The definition of return flush from [?] is as follows. If host p sends a return flush message to host q that is known to have failed, the communication network will return the message to p after delivering all other messages from q to p.

MSS s2 sends a return flush to s1 and starts processing the messages stored in the log for h. After processing the stored messages, it processes the in-transit
messages received and updates $st_{\text{info}}(h)$. Once MSS $s_2$ receives the return flush from the network, $st_{\text{info}}(h)$ is consistent with the state of MH $h$. The recovery procedure is complete, and MH $h$ can continue with the computation.

All the other MSSs in $sec_{\text{mss}}(h)$ may not have the updated version of $st_{\text{info}}(h)$. MSS $s_2$ sends $st_{\text{info}}(h)$ to them. The other MSSs in $sec_{\text{mss}}(h)$ receive the latest version of $st_{\text{info}}(h)$ and clear the messages stored in log for MH $h$. If the set $sec_{\text{mss}}(h)$ changes, MSS $s_2$ sends a copy of $st_{\text{info}}(h)$ to the new MSSs in $sec_{\text{mss}}(h)$ and requests the old MSSs in $sec_{\text{mss}}(h)$ to delete their copy of $st_{\text{info}}(h)$ and the messages stored in their log for MH $h$.

If MSS $s_1$ fails while executing a handoff procedure for MH $h$ (with the previous MSS of $h$), then MH $h$ will switch to another MSS as though it never switched to $s_1$.

**Theorem 14** If optimistic replication strategy is used, MH $h$ can switch to another MSSs and proceed with its computation whenever pri_{\text{mss}}(h) fails.

**Proof:** Let MH $h$ switch to MSS $s_2$ from MSS $s_1$ as a result of $s_1$’s failure. Without loss of generality, assume that $s_2 \in sec_{\text{mss}}(h)$. (Otherwise $s_2$ requests one of the secondary MSSs of $h$ to execute the recovery procedure and gets the $st_{\text{info}}(h)$ from that MSS.) Before $h$ continues its operation (after switching to MSS $s_2$), MSS $s_2$ processes all the messages in its log for MH $h$ in the received order and updates its copy of $st_{\text{info}}(h)$. $s_2$ also sends a return flush message to $s_1$. When MSS $s_2$ receives the return flush message from the network, it would have received (processed) a copy of all the messages that caused an update on $st_{\text{info}}(h)$ at $s_1$. Thus the copy of $st_{\text{info}}(h)$ at MSS $s_2$ becomes consistent with $st_{\text{info}}(h)$ at $s_1$ (prior to $s_1$’s failure), and hence $h$ can continue its computation from the cell of $s_2$.

So far, we have assumed that $sec_{\text{mss}}(h)$ is available for each MSS $h$. Next, two strategies are presented for selecting secondary MSSs for a mobile host.
7.1.4 Selection Strategies for Secondary MSSs

Different strategies for selecting secondary MSSs may be designed based on various factors such as the mobility pattern of mobile hosts, topology of the network, frequency of mobile hosts’ migration, etc. Since our schemes for fault tolerance do not depend on how secondary MSSs are selected for a mobile host, different strategies for selecting secondary MSSs can be used for different mobile hosts. Also, either the pessimistic or the optimistic replication scheme can be used with any of the strategies for selecting secondary MSSs.

Strategy 1

If the movement of the mobile host \( h \) is within a certain locality, then it is sufficient to consider those MSSs within that locality as candidates for the secondary MSSs of \( h \). The MSS within which the mobile host \( h \) resides is \( \text{pri}_\text{mss}(h) \) and all the other MSSs from the locality constitute the secondary MSSs. When \( \text{pri}_\text{mss}(h) \) fails, mobile host \( h \) switches to MSS \( s \) in \( \text{sec}_\text{mss}(h) \), marks \( s \) as \( \text{pri}_\text{mss}(h) \), and proceeds with the computation. An advantage of this strategy is that if \( h \) migrates to another cell during the normal operation, additional handoff steps need not be executed. Additional handoff steps are not needed since there is a copy of \( \text{st}_\text{info}(h) \) in its new MSS. Instead, the previous \( \text{pri}_\text{mss}(h) \) is marked as a secondary MSS, and its new MSS is marked as \( \text{pri}_\text{mss}(h) \). This strategy is effective and efficient if the movement of \( h \) is constrained to a subset of cells, e.g., some mobile hosts may move more frequently within one area, and occasionally move to other areas.

Strategy 2

Let \( \text{neighbors} \) of an MSS \( s \) be a set of MSSs whose cells are adjacent to the cell of \( s \) or overlapping with the cell of \( s \). For example, in Figure 7.2, the (geographical) neighbors of MSS \( s_2 \) are \( s_1 \), \( s_3 \), and \( s_4 \). For a mobile host \( h \) in the cell of \( s \), the neighbors of \( s \) constitute \( \text{sec}_\text{mss}(h) \). This way of maintaining \( \text{sec}_\text{mss}(h) \) is based on the expectation that when a mobile host leaves MSS \( s \), it must move to one
of the neighbors of $s$. If $\text{pri}_\text{mss}(h)$ fails, the mobile host $h$ switches to one of the neighbors of $\text{pri}_\text{mss}(h)$ and continue its operation. When $h$ moves from an MSS, say $s_1$, to another MSS, say $s_2$, during the normal operation, $\text{sec}_\text{mss}(h)$ may change since the neighbors of $s_1$ and $s_2$ may not be the same. A copy of $\text{st}_\text{info}(h)$ has to be sent to all the new secondary MSSs of $h$. If $s_2$ belongs to $\text{sec}_\text{mss}(h)$, then additional handoff steps need not be executed.

If the cell of a secondary MSS of $h$ overlaps with $\text{pri}_\text{mss}(h)$, then physical movement of $h$ is not necessary to switch to the secondary MSS. If physical movement (to switch when an MSS fails) is undesirable, then the network has to built in such a way that every cell overlaps with $k$ other cells. But, this may increase the cost of building the network for large values of $k$.

The advantage of this strategy over strategy 1 is that the mobility pattern of a mobile host need not be known. However, $\text{sec}_\text{mss}(h)$ may change when $\text{pri}_\text{mss}(h)$ changes making the handoff procedure costly. In this strategy, the set

Figure 7.2: Neighbors of $s_2 = \{s_1, s_3, s_4\}$
of secondary MSSs for a mobile host entirely depends on its primary MSS and
the network topology. In Strategy 1, the set of secondary MSSs entirely depends
on the mobility pattern of a mobile host and not on the MSS.

7.2 Analysis

In this section, we evaluate the performance of the replication schemes. In both
the schemes, for every message, \( k \) additional messages are sent to replicate the
state of an MH. In the pessimistic scheme, the application messages are delayed
by a primary MSS as it waits for acknowledgements from secondary MSSs. In
the optimistic scheme, there is a delay in recovering from an MSS failure. (A
secondary MSS has to send and receive return flush and process the messages in
its log.) We performed a detailed simulation study to measure these delays and
the storage required to buffer the messages.

7.2.1 Simulation Results

Throughout the simulation, the number of MSSs was fixed at 10. The number
of MHs was varied from 25 to 250. The values of \( k \) varies from 2 to 5. The
value of every point in the graphs is an average of the results of 1000 experiments
performed.

Figure ?? shows the delay incurred in waiting for the acknowledgements
from the secondary MSSs (pessimistic scheme). The delay increases as the num-
ber of MHs increases, because the load per MSS increases. Also, when \( k \) increases,
the delay increases due to the increase in the number of acknowledgements for
which a primary MSS has to wait.

In the pessimistic scheme, a primary MSS has to buffer each message till
it receives acknowledgements from the secondary MSSs. Figure ?? shows the
amount of storage required per MSS (on an average) to buffer the messages as