INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6″ x 9″ black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
NOTE TO USERS

The original manuscript received by UMI contains broken or light print. All efforts were made to acquire the highest quality manuscript from the author or school. Page(s) were microfilmed as received.

This reproduction is the best copy available

UMI
Design of Efficient Fault-Tolerant Systems on Wireless Networks

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Guohong Cao, M.S.

* * * * *

The Ohio State University

1999

Dissertation Committee:
Professor Mukesh Singhal, Adviser
Professor Anish Arora
Professor Wuchi Feng

Approved by

[Signature]
Adviser
Department of Computer and Information Science
ABSTRACT

Mobile wireless environments pose challenging problems in designing fault-tolerant systems because of the dynamics of mobility and limited bandwidth available on wireless links. Traditional fault-tolerance schemes, therefore, cannot be directly applied to these systems. However, fault tolerance is much more important in mobile computing systems since mobile computing systems are more prone to failures. In this dissertation, we investigate the design of fault tolerance at two levels: at the network level, we design efficient fault-tolerant channel allocation algorithms; at the operating system level, we study checkpointing and recovery techniques, which are attractive approaches for transparently adding fault-tolerance to distributed applications.

We develop a fault-tolerant channel acquisition approach that can tolerate communication link failures, network congestion, and node crashes. Also, we identify two guiding principles in designing channel selection algorithms. Following these principles, we propose a channel selection algorithm to reduce the call failure rate. By integrating the channel selection algorithm into our fault tolerant channel acquisition algorithm, we get a complete distributed fault-tolerant channel allocation algorithm. Simulation results show that our algorithm significantly reduces the failure rate under network congestion, communication link failures, and node failures compared to non-fault-tolerant channel allocation algorithms. Moreover, our algorithm reduces the message overhead compared to known distributed algorithms.
In checkpointing and recovery, we discover some problems in previous research and prove that no non-blocking min-process algorithm exists. Based on this result, we develop an efficient min-process algorithm and an efficient non-blocking algorithm for mobile computing systems. Our min-process algorithm only forces a minimum number of processes to take checkpoints and significantly reduces the blocking time. In our non-blocking algorithm, we propose a new concept of "mutable checkpoint", which can be saved anywhere; e.g., the main memory or local disk of mobile hosts. In this way, taking a mutable checkpoint avoids the overhead of transferring large amount of data over the wireless network to the stable storages at mobile support stations. Simulation results show that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm forces only a minimum number of processes to take their checkpoints on stable storage.
Dedicated to Jeffery and Lihong
ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my dissertation advisor, Professor Mukesh Singhal, for his support and guidance, and so many things I learned from him throughout this dissertation research.

I wish to thank Dr. Anish Arora and Dr. Wuchi Feng, members of my dissertation committee, for reading the dissertation and suggesting improvements. I would like to express my thanks to Prof. Naphtali D. Rishe, Prof. Bharat K Bhargava, and Prof. Michel Raynal for their understanding and support of my research. Thanks also go to Dr. Ravi Prakash and Dr. Xuefeng Dong, whose research results stimulated most of my dissertation work. Dr. Prakash clarified some of my questions in checkpointing; Dr. Dong gave me a lot of valuable suggestions when I setup the channel allocation simulation bed.

I also thank The Ohio State university for giving me the honor of Presidential Fellow. I would also like to thank all my colleagues in CIS for their friendship and assistance.

Last, but not least, I would like to thank my lovely wife Lihong, for her love and support, my parents for their understanding and encouragement, and my son Jeffery, for coming into the world in time to contribute so much inspiration to this dissertation.
VITA

October 14, 1968 .............................................. Born - Hebei, China

1990 ................................................................. B.S. Computer Engineering.
Xi'an Jiaotong University, China

The Ohio State University

1997-present ..................................................... Graduate Research (Teaching) Associate, Presidential Fellow.
The Ohio State University

PUBLICATIONS


**FIELDS OF STUDY**

Major Field: Computer and Information Science

Studies in:

- Software Systems Prof. Mukesh Singhal
- Networking Prof. Ten H. Lai
- Software Methodology Prof. Neelam Soundarajan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem Description and Our Results: Channel Allocation</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Overview</td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 Previous Work in Channel Selection</td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Previous Work in Channel Acquisition</td>
<td>7</td>
</tr>
<tr>
<td>1.2.4 Our Results</td>
<td>8</td>
</tr>
<tr>
<td>1.3 Problem Description and Our Results: Coordinated Checkpointing</td>
<td>9</td>
</tr>
<tr>
<td>1.3.1 Overview</td>
<td>9</td>
</tr>
<tr>
<td>1.3.2 Previous Work in Coordinated Checkpointing</td>
<td>11</td>
</tr>
<tr>
<td>1.3.3 Our Results</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Organization of the Dissertation</td>
<td>13</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Comparison of the update, the search, the efficient, and the fault-tolerant approaches</td>
<td>54</td>
</tr>
<tr>
<td>3.2 Simulation parameters for uniform traffic distribution</td>
<td>56</td>
</tr>
<tr>
<td>3.3 Simulation parameters for non-uniform traffic distribution</td>
<td>57</td>
</tr>
<tr>
<td>5.1 A comparison of system performance</td>
<td>131</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 The architecture of a mobile computing system</td>
<td>4</td>
</tr>
<tr>
<td>2.1 A hexagonal cellular model</td>
<td>16</td>
</tr>
<tr>
<td>2.2 An optimal partition</td>
<td>18</td>
</tr>
<tr>
<td>2.3 An interference partition subset is a convex hull</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Consistent and inconsistent global checkpoints</td>
<td>25</td>
</tr>
<tr>
<td>3.1 The search approach</td>
<td>29</td>
</tr>
<tr>
<td>3.2 The update approach</td>
<td>30</td>
</tr>
<tr>
<td>3.3 Impact of ( \delta ) on call failure rate under uniform distribution</td>
<td>58</td>
</tr>
<tr>
<td>3.4 Impact of ( \delta ) on call failure rate under non-uniform distribution</td>
<td>59</td>
</tr>
<tr>
<td>3.5 Comparisons of call failure rate under uniform distribution</td>
<td>60</td>
</tr>
<tr>
<td>3.6 Comparisons of call failure rate under non-uniform distribution</td>
<td>61</td>
</tr>
<tr>
<td>3.7 Comparisons of intra-handoff overhead between our algorithms and the two-step strategy</td>
<td>62</td>
</tr>
<tr>
<td>3.8 Comparisons of message complexity</td>
<td>63</td>
</tr>
<tr>
<td>3.9 Percentage of secondary channel acquisition</td>
<td>64</td>
</tr>
<tr>
<td>3.10 Comparisons of secondary channel acquisition delay</td>
<td>65</td>
</tr>
</tbody>
</table>
3.11 Comparisons of average channel acquisition delay ........................................ 66
3.12 Call failure rate under MSS or communication link failures .................. 67
3.13 Call dropping rate under network congestion ........................................... 70
4.1 Checkpointing and dependency information ............................................ 79
4.2 Inconsistent checkpoints .......................................................................... 80
4.3 The difference between causal dependency and z-dependency ................ 82
4.4 Tracing the dependency ........................................................................... 86
5.1 The first phase of our min-process checkpointing algorithm ..................... 94
5.2 Checkpointing and dependency information ............................................ 95
5.3 An example of checkpointing ................................................................. 97
5.4 Further reduce the number of checkpoints .............................................. 101
5.5 The number of checkpoints in point-to-point communication environment 123
5.6 The number of checkpoints in a group communication environment ....... 124
5.7 The number of checkpoints in a group communication environment ....... 125
A.1 The first problem ...................................................................................... 142
A.2 The second problem .................................................................................. 143
CHAPTER 1

Introduction

1.1 Motivation

The falling cost of both communication and mobile computing devices (laptop computers, hand-held computers, etc.) is making mobile computing affordable to both business users and private consumers. People on the road want to use their laptops to read Email, login on remote machines, buy or sell stocks, and so on; rescue workers at disaster sites (fires, floods, earthquakes, etc.) would like to send messages, keep records, and communicate with each other by mobile devices; a military general needs to gather information from his soldiers and send commands to them. In these applications, a system failure can cause loss of opportunity, financial loss, or even loss of human lives. It is clear that in many applications, highly dependable systems are needed.

Mobile wireless environments pose challenging problems in designing fault-tolerant systems because of the dynamics of mobility and limited bandwidth available on wireless links. Traditional fault-tolerance schemes cannot be directly applied to these systems. However, fault tolerance is much more important in mobile computing systems since mobile computing systems are more prone to failure. This is because
(i) wireless networks have high error rates and more frequent disconnections and (ii) mobile devices are more prone to failure, physical damage, or loss.

In this dissertation, we address these problems at two levels: the network level and the operating system level. At the network level, we design fault-tolerant network protocols. In particular, we design efficient fault-tolerant channel allocation algorithms. At the operating system level, we study checkpointing and recovery techniques, which are attractive approaches for transparently adding fault-tolerance to distributed applications. Even though checkpointing and recovery can tolerate both mobile devices and wireless network failures, we still study fault-tolerant network protocols, since wireless networks are not reliable and using operating system level approaches to tolerate network failures have high overhead compared to using network level approaches.

Channel allocation [3, 26, 31, 55] has received considerable attention, but none of the existing schemes takes into consideration of node failures, communication link failures, or even network congestion. Although the checkpointing and recovery approach [28, 38, 39] has been extensively studied in traditional distributed systems, the new constraints posed by mobile computing systems, such as low bandwidth of wireless channels, high search cost, and limited battery life, make traditional checkpointing algorithms unsuitable to them. In the following, we describe problems in these areas and propose our solutions.
1.2 Problem Description and Our Results: Channel Allocation

1.2.1 Overview

Cellular communication networks divide the geographical area into smaller regions, called cells [31]. As shown in Figure 1.1, in the center of each cell, there is a mobile service station (MSS), which can be a computer or a telephone switch. Inside each cell, there are many mobile hosts (MHs), which can be portable computers, cellular phones, or personal digital assistants (PDAs). An MH can move around inside a cell. It can also move from one cell to another cell, which is referred to as handoff (see Chapter 2.2 for a definition). The MSS is connected to the wired network and can also communicate with the MHs in its cell using wireless channels. An MH can have wireless communication with any others in the network, fixed or mobile, only through the MSS of the cell in which it is present. We use node to represent both MH and MSS, unless explicitly stated otherwise.

Each cell has a geometric center. To establish a communication session (or a call), an MH sends a request to the MSS in its cell. The session is supported if a wireless channel can be allocated for the communication between the MH and the MSS. Since frequency spectrum available for civilian use is limited, the frequency channels have to be reused as much as possible in order to support the increasing demand for wireless communication. However, two different cells cannot use the same channel if their geographic distance is less than a threshold called the minimum channel reuse distance ($D_{\text{min}}$) [3. 49]: otherwise, the communication sessions would interfere with each other. The interference neighbors of a cell $C_i$ are cells whose geographic distance from $C_i$ is less than $D_{\text{min}}$. 
Figure 1.1: The architecture of a mobile computing system

A channel is *available* for a cell if its use in the cell does not interfere with others. When a cell needs a channel, it acquires an available channel using a channel allocation algorithm. A channel allocation algorithm includes two parts: a channel acquisition algorithm and a channel selection algorithm. The channel acquisition algorithm is responsible for collecting information from other cells and making sure that two cells within $D_{\min}$ do not use the same channel. The channel selection algorithm is used to choose a channel from a large number of available channels in order to achieve better channel reuse. The performance of a channel acquisition algorithm is measured by the message complexity and the acquisition delay. The message complexity is measured in terms of the number of messages exchanged per channel acquisition. The acquisition delay is the time required for an MSS to allocate a channel. The performance of the
channel selection algorithm is measured by the call failure rate [3]. A call is said to have been failed if there is no channel available for use when the call is being set up or when it is being handed over to another cell due to a host mobility.

1.2.2 Previous Work in Channel Selection

There are three types of channel selection algorithms: fixed, flexible, and dynamic [37]. In the fixed strategies [46], a set of channels are permanently allocated to each cell, which is allowed to use the allocated channels and no others. In the dynamic strategies [17, 26], a cell may use any channel that will not cause channel interference. Channels are not pre-allocated to cells, but assigned on a dynamic basis. Typically, each channel is associated with a priority, and when a cell needs a channel, it picks the available channel which has the highest priority. The channel is later returned to the system when it is no longer needed by the cell. Flexible strategies [65] combine the aspects of both fixed and dynamic strategies, where each cell is allocated a fixed set of permanent channels and a number of flexible channels are set aside to be dynamically allocated to cells upon requests.

Among these three strategies, dynamic strategies have been the focus of recent research [3, 26, 49]. Thus, we only consider DCS strategies. With DCS strategies, a cell may use any channel that will not cause channel interference. Typically, each channel is associated with a priority: when a cell needs a channel, it picks the available channel which has the highest priority. Thus, various DCS strategies differ from each other in the way priorities are assigned to channels. There are three ways to assign channel priorities: static, dynamic, and hybrid. In a static-priority strategy such as the geometric strategy [3], each channel in each cell is assigned a fixed priority.
that does not change over time. In a dynamic-priority strategy such as the two-step strategy [25], the channel priority is dynamically computed. A hybrid-priority scheme [27, 73] is something in between: the channel priority is calculated as a static base-priority plus a dynamic adaptive-priority.

In the geometric strategy [3], each cell is assigned some channels as primary channels based on a priori. These primary channels are prioritized. During a channel acquisition, a cell acquires the available primary channel that has the highest priority. If none of the primary channels is available, the cell borrows a channel from its neighbors according to some fixed priority assignment approach. When a cell acquires a channel, it always acquires the channel with the highest priority. When a cell releases a channel, it always releases the channel with the lowest priority.

In the borrowing with directional channel-locking (BDCL) strategy [73], when a cell needs to borrow a channel, it borrows the channel that has the lowest priority from the “richest” interference neighbor; i.e., the cell with the most available primary channels. The motivation behind this is to reduce the chance that the lender might soon use up its primary channels and have to acquire a secondary channel.

In the Nanda-Goodman strategy [49], when a cell borrows a channel, it selects the channel which will cause a smaller number of neighbors to become interfered. When a cell releases a channel, it releases a channel which will make itself available in more interference neighbors.

The two-step strategy [25] combines the geometric strategy and the Nanda-Goodman strategy. In this approach, by using resource planing, the primary channels can be optimally utilized. At the same time, when a cell borrows a channel, it selects the
channel which will cause a minimum number of neighbors to become interfered. However, since it does not consider the "richness", the lender may soon use up its channel and borrow channels again, and then the advantage of resource planning is lost.

All these algorithms depend on a mobile switching center (MSC) to accomplish channel acquisition, which are referred to as centralized channel acquisition algorithms. More specifically, each cell notifies the MSC when it acquires or releases a channel so that the MSC knows which channels are available in each cell at any time and assigns channels to cells accordingly.

1.2.3 Previous Work in Channel Acquisition

Recently, distributed channel acquisition algorithms [26, 55] have received considerable attention because of their high reliability and scalability. In this approach, an MSS communicates with other MSSs directly to find the available channels and make sure that the channel assignment does not interfere with other cells. In general, there are two approaches in distributed channel acquisition algorithms: Search [55] and Update [26]. In the search approach [55], when a cell needs a channel, it searches all interference neighbors to find the set of currently available channels and then picks one according to the underlying channel selection strategy. In the update approach [26], a cell maintains information about available channels. When a cell needs a channel, it selects an available channel according to the underlying channel selection strategy and consults with its interference neighbors whether it can acquire the selected channel. Also, a cell informs its interference neighbors each time it acquires or releases a channel so that each cell always knows the available channels of its interference neighbors.
Both approaches require that a borrower waits for the acknowledgment from its interference neighbors. Thus, the borrower cannot borrow a channel when it cannot communicate with anyone of them. Moreover, since many communication sessions such as handoff, audio, and video have time constraints, a long communication delay due to network congestion has the same effect as a communication link failure or node (MH or MSS) failure, where the borrower may fail to borrow a channel even though there exists available channels in its neighbors. In real-life networks, under heavy traffic load, a cell has a large probability to experience an intermittent network congestion, a communication link failure, or a node failure. In these approaches [26, 55], since a cell has to consult with a large number of interference neighbors to borrow a channel, the failure rate will be much higher under heavy traffic load. Therefore, these approaches are not suitable for real-life networks.

1.2.4 Our Results

In this dissertation, we propose a fault-tolerant channel acquisition algorithm which tolerates communication link failures and node (MH or MSS) failures. In the proposed algorithm, a borrower does not need to receive a response from every interference neighbor. It only needs to receive a response from a small portion of them. Thus, as long as the borrower can communicate with a small portion of its interference neighbors, it can borrow a channel from them. Also, we identify two guiding principles in designing channel selection algorithms. Following these principles, we propose a channel selection algorithm to further improve the performance of the two-step strategy by considering the “richness” and the interference property. By integrating the channel selection algorithm into our channel acquisition algorithm, we get a
complete distributed channel allocation algorithm. By keeping the borrowed chan-
nels, the channel allocation algorithm makes use of the temporal locality and adapts
to the network traffic fluctuations: i.e., free channels are transferred to hot cells to
achieve load balancing. Detailed simulation experiments are carried out to evaluate
our proposed methodology. Simulation results show that our algorithm significantly
reduces the failure rate under network congestion, communication link failures, and
node failures compared to non-fault-tolerant channel allocation algorithms. More-
over, our fault-tolerant algorithm reduces the message overhead compared to known
distributed channel allocation algorithms, and outperforms them in terms of failure
rate under uniform as well as non-uniform traffic distribution.

1.3 Problem Description and Our Results: Coordinated Check-
pointing

1.3.1 Overview

The mobility of MHs raises some new issues [1.40] that complicate the design of
checkpointing algorithms:

- Changes in the location of an MH complicate the routing of messages. Messages
  sent by an MH to another MH may have to be rerouted because the destination
  MH has moved. Locating an MH generally increases the delay and message
  complexity.

- Due to the vulnerability of mobile computers to catastrophic failures, e.g., loss,
  theft, or physical damage, the disk storage on an MH cannot be considered
  as stable storage. A reasonable solution [1] is to utilize the stable storage at
  MSSs to store checkpoints of MHs. Thus, to take a checkpoint, an MH has to
  transfer large amount of data to its local MSS over the wireless network. Since
the wireless network has low bandwidth and MHs have relatively low computation power, a checkpointing algorithm should only force minimum number of processes to take checkpoints.

- The battery at an MH has limited life. To save energy, the MH can power down individual components during periods of low activity [30]. This strategy is referred to as the doze mode operation. An MH in the doze mode is woken up on receiving a message. Therefore, energy conservation and low bandwidth constraints require a checkpointing algorithm to minimize the number of synchronization messages.

- MHs may disconnect from the network temporarily or permanently. The disconnection of MHs should not prevent the checkpointing process.

These constraints make traditional checkpointing algorithms for distributed systems unsuitable for mobile computing systems.

Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications since it avoids the domino effect [39] and minimizes the stable storage requirement. In this approach, the state of each process in the system is periodically saved on stable storage, which is called a checkpoint of the process. To recover from a failure, the system restarts its execution from a previous consistent global checkpoint saved on stable storage. A system state is said to be consistent if it contains no orphan message: i.e., a message whose receive event is recorded in the state of the destination process, but its send event is lost [39, 64]. In order to record a consistent global checkpoint, processes must synchronize their checkpointing activities. In other words, when a process takes a checkpoint, it asks (by sending
checkpoint requests to) all relevant processes to take checkpoints. Therefore, coor-
dinated checkpointing suffers from high overhead associated with the checkpointing process.

1.3.2 Previous Work in Coordinated Checkpointing

Much of the previous work [24, 38, 39, 45] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints during checkpointing. However, these algorithms (called blocking algorithms) force all relevant processes in the system to block their computations during the checkpointing process. Checkpointing includes the time to trace the dependency tree and to save the states of processes on stable storage, which may be long. Moreover, in mobile computing systems, due to the mobility of MHs, a message may be routed several times before reaching its destination. Therefore, blocking algorithms may further degrade the performance of mobile computing systems [7, 28].

Recently, nonblocking algorithms [28, 41, 60] have received considerable attention. In these algorithms, processes need not block during checkpointing by using a checkpointing sequence number to identify inconsistent messages. However, these algorithms assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms, such as poor reliability, bottlenecks, etc. Moreover, these algorithms require all processes in the system to take checkpoints during checkpointing, even though many of the checkpoints may not be necessary. If they are modified to permit more processes to initiate checkpointing, which makes them distributed, the new algorithm suffers from another
problem: in order to keep the checkpoint sequence number updated, any time a process takes a checkpoint, it has to notify all processes in the system. If each process can initiate a checkpointing process, the network would be flooded with control messages and processes might waste their time taking unnecessary checkpoints.

The Prakash-Singhal algorithm [54] was the first algorithm to combine these two approaches. More specifically, it forces only a minimum number of processes to take checkpoints and does not block the underlying computation during checkpointing. However, we [12] showed that this algorithm may result in an inconsistency.

1.3.3 Our Results

In this dissertation, we identify problems in the Prakash-Singhal algorithm. Then, we generalize the result and prove that there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints. The proof is based on a new concept “z-dependency”, which captures the essence of coordinated checkpointing. This implies that there are three directions in designing efficient coordinated checkpointing algorithms. One extreme is to relax the non-blocking condition while keeping the min-process property. The other extreme is to relax the min-process condition while keeping the non-blocking property. Between these two extremes, we can design blocking non-min-process algorithms that significantly reduce the blocking time as well as the number of checkpoints.

In this dissertation, we present a min-process checkpointing algorithm for mobile computing systems. In this algorithm, the checkpointing information such as dependency vectors are saved at MSSs. To initiate a checkpointing, the initiator collects dependency vectors from MSSs and then it determines and notifies all the processes
which need to take checkpoints. In this way, the blocking time can be reduced from as much as $O(N \cdot T)$ ($N$ is the number of processes and $T$ is the message delay) to a negligible constant.

We also propose a non-blocking coordinated checkpointing algorithm. This algorithm is based on a new concept of "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint. Mutable checkpoints can be saved anywhere: e.g., the main memory or the local disk of MHs. In this way, taking a mutable checkpoint avoids the overhead of transferring large amount of data over the wireless network to the stable storage at MSSs. We have developed techniques to minimize the number of mutable checkpoints. Simulation results showed that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm forces only a minimum number of processes to take their checkpoints on stable storage.

1.4 Organization of the Dissertation

The dissertation is organized as follows. Chapter 2 presents the cellular network model, the resource planning model, and the model of distributed applications.

In Chapter 3, we present our results in distributed channel allocation. Firstly, we investigate the fundamental differences between two general distributed channel acquisition algorithms: the search approach and the update approach. The update approach has shorter acquisition delay and lower call failure rate, but higher message complexity. On the other hand, the search approach has lower message complexity, but longer acquisition delay and higher call failure rate. Secondly, we propose a novel distributed acquisition algorithm, which has similar message complexity as the
search approach and similar acquisition delay as the update approach. Thirdly, we present our fault-tolerant acquisition algorithm which can tolerate communication link failures and node (Mชม or MSS) failures. Fourthly, we identify two guiding principles in designing channel selection algorithms. Following these principles, we propose an efficient channel selection algorithm. We also present solutions to integrate the channel selection algorithm into our channel acquisition algorithms. Finally, we demonstrate the advantage of our approach by analytical and simulation methods.

Chapter 4 defines a new concept "z-dependency". which is more general than causal dependency and can be used to model coordinated checkpointing. Based on this concept, we prove the impossibility result in coordinated checkpointing: that is, there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints.

In Chapter 5, we propose two efficient checkpointing algorithms for mobile computing systems: the min-process algorithm and the non-blocking algorithm. Based on "mutable checkpoints", our non-blocking algorithm forces only a minimum number of processes to take their checkpoints on the stable storage. A correctness proof and performance evaluations are also provided.

Finally, Chapter 6 presents conclusions of the dissertation and discusses directions for future research.
CHAPTER 2

System Model and Definitions

In this chapter, we present the cellular network model, the resource planning model, and the model of distributed applications. The cellular network model and the resource planning model are used to study the channel allocation problem. The model of distributed applications is useful for studying checkpointing and recovery techniques.

2.1 Cellular Network Model

The geographical area is divided into hexagonal cells of radius $R$ in a cellular network [31]. A $n \times m$ cellular network has $n$ rows and $m$ columns of cells. (Figure 2.1 shows a $9 \times 9$ cellular network.) The cell at row $r$ and column $c$ is denoted by $(r, c)$ (or denoted by $C_i$ if the cell can be unequally identified by $i$).

Each cell has a geometric center. In the rectangular Cartesian coordinates system with $x$-axis pointing to the direction of row 0, the coordinates $(x, y)$ of the center of cell $(r, c)$ can be calculated as:

$$(x, y) = (r, c) \begin{pmatrix} \sqrt{3}R & 0 \\ \frac{\sqrt{3}R}{2} & \frac{3}{2}R \end{pmatrix}$$  \hspace{1cm} (2.1)
The distance between two cells $C_i$ and $C_j$, denoted by $\text{dist}(C_i, C_j)$, is defined to be the Euclidean distance between their centers. Thus, given their Cartesian coordinates $C_i = (x_i, y_i)$ and $C_j = (x_j, y_j)$, the distance between $C_i$ and $C_j$ is defined as $\text{dist}((x_i, y_i), (x_j, y_j)) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

Each cell is supported by an MSS in the center (we use cell and MSS interchangeably). The MSSs are connected to each other by a wired network. To establish a communication session (or a call), an MH has to send a request to the MSS in its cell. The session is supported if a wireless channel can be allocated for communication between the MH and the MSS. Two cells can use the same channel if the geographic distance between them is no less than a threshold $D_{\text{min}}$: otherwise, their communication sessions will interfere with each other, which is referred to as channel interference.
Definition 1  Given a cell $C_i$, the set of interference neighbors of $C_i$, denoted by $I.N_i$, is:

$$I.N_i = \{C_j | \text{dist}(C_i, C_j) < D_{min}\}.$$ 

As shown in Figure 2.1, $R$ is the cell radius. $D_{min}$ is the minimum channel reuse distance. Cell (3.3) has 30 interference neighbors when $D_{min} = 3\sqrt{3}R$. $I.N_i$ may contain less than 30 cells if $C_i$ is near the boundary of the network. If a fourth-power law attenuation is assumed [3, 46], the signal to interference ratio is given by $[S/I]_{min} = [(D_{min}/R) - 1]^4/6$. With $D_{min} = 3\sqrt{3}R$, $[S/I]_{min} \approx 17dB$, which is a reasonable value in practice.

When the distance between two cells is exactly $D_{min}$, these cells are called co-channel cells. For example, cells (0.3), (0.6), (3.0), (3.3), (3.6), (6.0), and (6.3) are co-channel cells of cell (3.3).

A node (MH or MSS) may either crash or fail to send or receive messages. Communication links may fail by crashing or by failing to deliver messages. Combinations of such failures may lead to network partition failures [22], where nodes in a partition may communicate with each other, but no communication can occur between nodes in different partitions.

2.2 Resource Planning Model

Most channel selection strategies require a priori on channel status in order to achieve better channel reuse. For instance, in the channel borrowing strategies [27, 36, 49], each cell is allocated a set of “nominal” channels beforehand: in the geometric strategy [3], each cell must know its “first-choice” channels prior to any channel
acquisition. We call the process of assigning special status to channels as *resource planning* [25, 26].

**Resource Planning**

1. Partition the set of all cells into a number of disjoint subsets, $G_0, G_1, \ldots, G_{k-1}$, such that any two cells in the same subset are apart by at least a distance of $D_{\text{min}}$. Accordingly, partition the set of all channels into $k$ disjoint subsets: $P_0, P_1, \ldots, P_{k-1}$.

2. The channels in $P_i (i = 0, 1, \ldots, k - 1)$ are *primary channels* of cells in $G_i$, and *secondary channels* of cells in $G_j (j \neq i)$.

3. A cell requests a secondary channel only when no primary channel is available.

![Figure 2.2: An optimal partition](image)
Channel reuse patterns are affected by the way under which cells are partitioned. As cells always acquire primary channels first, a channel is likely to be acquired in its primary cells and its reuse pattern is likely to be one of the subsets $G_i (i = 0, 1, \cdots, k - 1)$. In order to achieve the highest channel reuse, each subset $G_i$ should contain as many cells as possible. Obviously, the smaller value $k$ has, the more cells each subset may contain, and the better channel reuse may achieve.

No matter how the cells are partitioned, there is a lower bound $k_0$ such that $k \geq k_0$. The value $k_0$ is determined by the value of $D_{mn}$. For instance, with $D_{mn} = 3\sqrt{3}R. k_0 = 9$. For convenience, we say that a cell $C_i$ is a primary (secondary) cell of a channel $r$ if and only if $r$ is a primary (secondary) channel of $C_i$. Thus, the cells in $G_i$ are primary cells of the channels in $P_i$ and secondary cells of the channels in $P_j (j \neq i)$.

In [25], Dong and Lai presented the optimal partition method which divides the cells into $k_0$ subsets: $G_0, G_1, \cdots, G_{k_0-1}$. They proved that each subset is an optimal channel reuse pattern, i.e., a channel can achieve the optimal reuse if and only if it is used in all the cells of the same set. With optimal partition, channels can be acquired following the optimal reuse patterns, which is the ultimate objective of resource planning.

The optimal partition method has the following property [26]: a cell and its co-channel cells are in the same subset. Figure 2.2 shows the optimal partition, where cells are divided into nine subsets $G_A, G_B, \cdots, G_1$. Cells in $G_A = \{C_{A_i} | 0 \leq i \leq 8\}$ are co-channel cells and they are in the same subset.

**Lemma 1** In the optimal partition method, given a cell $C_i$ and a set $G_j (j = 0, 1, \cdots, k_0 - 1)$, if $C_i \notin G_j$, then $IN_i \cap G_j \neq \emptyset$. 

19
Proof. Let $C_k \in G_j$ be the closest cell to $C_i$. If $dist(C_i, C_k) \geq D_{\text{min}}$, then at least one of the cells in $\text{co-channel}(C_k)$ is closer to $C_i$ than $C_k$. Since a cell and its co-channel cells are in the same subset in the optimal partition method [26], $C_k$ is then not the closest cell to $C_i$ in $G_j$. Hence, $dist(C_i, C_k) < D_{\text{min}}$, and $IN_i \cap G_j \neq \emptyset$. □

Definition 2 For a cell $C_i \notin G_p$ and a channel $r \in P_p$, the interference primary cells of $r$ relative to $C_i$, denoted by $IP_i(r)$, are the cells which are primary cells of $r$ and interference neighbors of $C_i$; i.e., $IP_i(r) = G_p \cap IN_i$. $IP_i(r)$ is also referred to as an interference partition subset of $C_i$ relative to $r$.

![Figure 2.3: An interference partition subset is a convex hull](image)

In fact, for a cell $C_i$ and a channel $r$, $IP_i(r)$ consists of either three or four cells, i.e., the centers of the cells in $IP_i(r)$ form a triangle or a parallelogram, respectively, as shown in Figure 2.3. Needless to say, triangles and parallelograms are convex hulls.

**Theorem 1** The optimal partition method has the following two properties:

**Property 1:** $\forall C_i, C_j \in G_p : \text{distance}(C_i, C_j) \geq D_{\text{min}}$. 

20
**Property 2:** \( \forall C_i, C_j : C_i \in N_j \Rightarrow \forall r(IP_i(r) \cap IP_j(r) \neq \emptyset) \).

*Proof.* Property 1 is obvious from the definition of resource planning. Thus, we only need to prove Property 2. Property 2 can be explained as follows: assume a cell \( C_i \) is an interference neighbor of \( C_j \); if \( C_i \) and \( C_j \) request the same channel \( r \), they have at least one common cell which is an interference primary cell of \( r \). Assume there exists a channel \( r \) such that \( IP_i(r) \cap IP_j(r) = \emptyset \). Then, the distance between \( C_i \) and \( C_j \) is greater than the minimum distance between a vertex of \( IP_i(r) \) and \( IP_j(r) \), which is at least \( D_{\text{min}} \). This contradicts the fact that \( \text{dist}(C_i, C_j) < D_{\text{min}} \). \( \Box \)

Figure 2.2 shows one partition, which divides the cells into nine subsets \( G_A, G_B, \ldots, G_I \). Cells in \( G_A = \{C_{A_i} | 0 \leq i \leq 8 \} \) can use the same channel without channel interference. If two interference neighbors \( C_{G_2} \) and \( C_{D_5} \) request the same primary channel of a cell in \( G_A \), they have a common interference primary cell \( C_{A_8} \). Since the distance between any two nearest cells in a subset is exactly \( D_{\text{min}} \), it is an optimal partition in the sense that each channel is maximally reused by its neighbors.

**Handoff and intra-Handoff**

A mobile host may go across the boundary between two cells while being active. When this occurs, the necessary state information must be transferred from its previous MSS to the MSS in the new cell. This process is known as *handoff* (or *inter-handoff*) [46]. During a handoff, an MH releases its current channel to its previous MSS and is assigned a new channel by the new MSS.

To achieve better channel reuse, *intra-handoff* (or a channel switch) may be necessary [5, 26]. In an intra-handoff operation, an MH releases its current channel and is assigned a new channel within the same cell. The motivation behind intra-handoff
can be understood by an example. In Figure 2.2, suppose cell $C_{F_1}$ borrows a channel $r1$ from $A_1$ and assigns it to a mobile host $MH_f$. Cells $C_{A_1}$, $C_{A_2}$, $C_{A_4}$, and $C_{A_5}$ cannot use channel $r1$ due to interference. If a call in $C_{F_1}$ terminates and a primary channel $r2$ is released, an intra-handoff from $r1$ to $r2$ by $MH_f$ improves channel reuse, since $r1$ can be reused by four other cells $C_{A_1}$, $C_{A_2}$, $C_{A_4}$, and $C_{A_5}$. A drawback of intra-handoff is of course the overhead. Fortunately, most of the channel selection strategies do not demand many intra-handoffs [5, 26]. Thus, intra-handoff may be necessary for better channel reuse.

2.3 Model of Distributed Applications

The distributed computation we consider consists of $N$ sequential processes denoted by $P_0, P_1, P_2, \cdots, P_N$ running concurrently on fail-stop $MH$s or $MSS$s in the network. The execution of a process consists of three types of events: message send, message delivery, and internal events. Internal events represent local computations at the processes. The system does not have a global physical clock and the local clocks of the constituent nodes, where the process execute, may not be perfectly synchronized.

The order in which two events occur at two different processes cannot be determined solely on the basis of local time of occurrence. To totally order these events, we use Lamport’s scheme [44] to assign a timestamp (the sequence number and the process number) to each event. The sequence number assigned is greater than that of any message sent, received, or observed at that process. The process with lower timestamp has higher priority which is determined as follows:

1: The event with smaller sequence number has higher priority.
2: If the events have equal sequence numbers, the event with smaller process number has higher priority.

There exists a logical communication channel between each pair of processes. The links in the static network support FIFO message communication. As long as an MH is connected to an MSS, the channel between them ensures FIFO communication in both directions. Message transmission through these links takes an unpredictable, but finite amount of time. During normal operation, no messages are lost or modified in transit.

A process can send a message to either one process or a group of processes. Multicast communication can be implemented either by broadcasting the message to all the processes or through multiple point-to-point communications. The group of processes to which a process sends multicast messages is not fixed: i.e., a process \( P_i \) may send one multicast message to a group of processes \( G_1 \) and later another multicast message to a different group of processes \( G_2 \). Dynamic multicast groups are allowed because a process can do a multicast to any group of processes without having to form groups \textit{a priori}.

The messages generated by the underlying distributed application is referred to as \textit{computation messages}. Messages generated by processes for channel allocation, checkpoint advancement, and failure recovery will be referred to as \textit{system messages}. Also, when a message of either type reaches a process, the process has the ability to peek at the message contents before actually processing it. The reception/arrival of a message and its processing by the receiving process may not necessarily happen at the same time. These are two distinct events. The arrival of a message is recorded only on its processing.
**Consistent Checkpoints:** Rollback recovery achieves fault tolerance by periodically saving the state of a process during failure-free execution and restarting from a saved state upon a failure to reduce the amount of lost work. The saved process state is called a *checkpoint*. The procedure of restarting from previously checkpointed state is called *rollback-recovery*. The process of saving process (system) states is referred to as *checkpointing* or *taking a checkpoint*. In a distributed computation, processes save their local states, which are known as local checkpoints. All the local checkpoints, one from each process, collectively form a global checkpoint.

Each checkpoint taken by a process is assigned a unique sequence number. The $i^{th}$ ($i \geq 0$) checkpoint of process $P_p$ is assigned a sequence number $i$ and is denoted by $C_{p,i}$. We also assume that each process $P_p$ takes an initial checkpoint $C_{p,0}$ immediately before execution begins and ends with a *virtual* checkpoint that represents the last state attained before termination [50]. The $i^{th}$ checkpoint interval of process $P_p$ denotes all the computation performed between its $i^{th}$ and $(i+1)^{th}$ checkpoint, including the $i^{th}$ checkpoint but not the $(i+1)^{th}$ checkpoint.

The state of a distributed computing system is the collection of the individual states of all participating processes and the states of the communication channels. Figure 2.4 shows the space-time diagram of a distributed computation where messages in the computation are depicted by arrows. Intuitively, a consistent system state is one that may occur in a legal execution of a distributed computing. A more precise definition of a *consistent system state* is one in which “every message that has been received is also shown to have been sent in the state of the sender” [14]. A global checkpoint represents a system state. A consistent global checkpoint represents a consistent system state. For example, in Figure 2.4, the global checkpoint $\{C_{1,2}, C_{2,1}, C_{3,1}\}$ is a
consistent global checkpoint. The global checkpoint \( \{C_{1,2}, C_{2,1}, C_{3,2}\} \) does not represent a consistent system state, since it involves an orphan message \( m_2 \), whose receiving event is recorded in the global checkpoint but its sending event is not recorded. The global checkpoint \( \{C_{1,1}, C_{2,1}, C_{3,1}\} \) represents a consistent checkpoint even though it involves message \( m_1 \) which has been sent but not yet received. Such a message is referred to as an *in-transit* message. Koo and Toueg [39] argued that an in-transit message is indistinguishable from the situation in which the in-transit message is lost in the communication channel during normal execution. Thus, in-transit messages can be recovered by the reliable communication protocol. More detailed information about how to deal with in-transit messages can be found in [29, 72]. In the following, we use consistent global checkpoint and consistent checkpoint interchangeably if there is no confusion.

Figure 2.4: Consistent and inconsistent global checkpoints
CHAPTER 3

Distributed Channel Allocation

In this chapter, we present our results in distributed channel allocation. Firstly, we investigate fundamental differences between two general distributed channel acquisition algorithms: the search approach and the update approach. The update approach has shorter acquisition delay and lower call blocking rate, but higher message complexity. On the other hand, the search approach has lower message complexity, but longer acquisition delay and higher call blocking rate. Secondly, we propose a novel distributed acquisition algorithm, which has similar message complexity as the search approach and similar acquisition delay as the update approach. Thirdly, we present our fault-tolerant acquisition algorithm which can tolerate communication link failures and node (MH or MSS) failures. Fourthly, we identify two guiding principles in designing channel selection algorithms. Following these principles, we propose an efficient channel selection algorithm. We also present solutions to integrate the channel selection algorithm into our channel acquisition algorithms. Finally, we demonstrate the advantage of our approach by analytical and simulation results.
3.1 Search vs. Update

The fundamental requirement of any distributed channel acquisition algorithm is that a channel will not be acquired by two cells within the minimum reuse distance simultaneously. Due to the distributed nature, MSSs must communicate with each other via message passing. We use Lamport's logical clock scheme [44] to assign timestamps to events so that events occurred at different cells can be totally ordered by their timestamps.

This chapter compares the search approach and the update approach for channel acquisitions. The basic idea of the two schemes is that a cell must consult interference neighbors before it acquires a channel. It should be noted that the search approach and the update approach are general schemes which can be applied to implement any channel acquisition algorithms.

3.1.1 The Search Approach

In the search approach [55], when a cell \( C_i \) needs to borrow a channel, it changes to \textit{search mode} and sends \textit{request} messages to each cell in \( I.N_i \). When a cell \( C_j \) receives a \textit{request} from \( C_i \), if \( C_j \) is not in the search mode or \( C_j \) is in the search mode but \( C_j \)'s request has higher timestamp (lower priority) than \( C_i \)'s, \( C_j \) sends a \textit{reply} message to \( C_i \) which contains information about its used channels; otherwise, \( C_j \) defers the \textit{reply} (similar to [59]). After \( C_i \) has received all the \textit{reply} messages from each cell in \( I.N_i \), it computes the available channels and picks one, say \( r \), from them. \( C_i \) sends \textit{confirm} messages to the lenders of \( r \). If all lenders reply \textit{agree}, \( C_i \) can use \( r \); otherwise, \( C_i \) picks another available channel and repeats the process. If there is no available channel left, the call request is failed. When a channel \( r \) is borrowed, the
lender marks \( r \) as an interference channel, and it can not use \( r \) until \( r \) is returned by all borrowers. Figure 3.1 shows the formal description of the search approach.

### 3.1.2 The Update Approach

In the update approach [26], a cell maintains information about the available channels. When a cell \( C_i \) needs to borrow a channel, it picks an available channel \( r \) according to the underlying channel selection strategy and then sends a request message to each cell in \( I.N_i \). A cell that receives a request replies with a reject if either it is using \( r \) or it is also requesting for \( r \) with a smaller timestamp; otherwise, it replies with an agree. If \( C_i \) has received agree messages from all the cells in \( I.N_i \), \( C_i \) notifies them that it has successfully acquired channel \( r \); otherwise, \( C_i \) picks another available channel and repeats the process. When \( C_i \) finishes the use of the borrowed channel, it sends a release to each cell in \( I.N_i \). Figure 3.2 shows the formal description of the update approach.

### 3.1.3 A Comparison

In the search approach, a cell communicates with its interference neighbors only when it needs to borrow a channel. However, in the update approach, a cell keeps communicating with its interference neighbors in order to get the up-to-date information. Clearly, the update approach is likely to have significantly higher message complexity than the search approach does. In the search approach, a cell needs to confirm a selected channel, which doubles the acquisition delay compared to the update approach.

Many good channel selection strategies rely on a cost function to determine which channel to borrow and which channel to release. For example, when a newly available
Notation:

- $IN_i$: defined before.
- $S$: the set of all the channels in the system.
- $U_i$: the set of used channels at $C_i$. $U_i$ is initialized to empty.
- $I_i(r)$: the set of cells to which $C_i$ has sent an agree$(r)$. If $I_i(r) \neq \emptyset$, $r$ is an interference channel of $C_i$. Then, $C_i$ cannot use $r$, but it can lend it to other cells. $I_i(r)$ is initialized to empty.
- $pick(A)$: pick a channel $r$ from a set of available channels $A$ using the underlying channel selection algorithm.

(A) When a cell $C_i$ needs a channel to support a call request, it sets its search mode and sends a request message to each cell $C_j \in IN_i$.

(B) When a cell $C_i$ receives a request message from $C_j$, if it is also in the search mode and its request has smaller timestamp than $C_j$'s request, it defers its response until it clears its search mode; otherwise, it sends a reply$(U_i)$ message to $C_j$.

(C) When a cell $C_i$ receives all reply messages from cells in $IN_i$. It determines the currently available channels as follows:

$$A_i = S - U_i - \bigcup_{j \in IN_i} U_j$$

(C.1) If $A_i$ is empty, $C_i$ drops the call, clears its search mode, and processes all the deferred reply messages. If $A_i$ is not empty, let $r = pick(A_i)$. $C_i$ sends a confirm$(r)$ to the owner of channel $r$.

(D) When a cell $C_i$ receives a confirm$(r)$ from $C_j$, if it is not using $r$, it adds $C_j$ to $I_i(r)$ and responses with an agree; otherwise, it responses with a reject.

(E) If $C_i$ receives an agree message from each cell to which it has sent a confirm$(r)$, the request is successful. It makes $r$ as used channel, clears the search mode, and responds all deferred request messages.

If $C_i$ receives an reject, it deletes $r$ from $A_i$, sends an abort$(r)$ to every cell from which it has received an agree, and then goes to Step C.1.

When a cell $C_i$ receives an abort$(r)$ from $C_j$, it removes $C_j$ from its $I_i(r)$.

Figure 3.1: The search approach
Notation:

- $IN_i, S, U_i, I_i, \text{pick}(A)$: defined before.
- $U_i^j$: The set of used channels $U_j$ as known to $C_i$.

(A) When a cell $C_i$ needs a channel to support a call request, let $r = \text{pick}(S - U_i - \bigcup_{j \in I_i} U_i^j)$. $C_i$ sends a request($r$) message to each cell $C_j \in IN_i$.

(B) When a cell $C_i$ receives a request($r$) message from $C_j$, if it is using $r$ or it is also requesting $r$ and its request has smaller timestamp than $C_j$’s request, it adds $r$ to $U_i^j$ and replies an agree message; otherwise, it sends a reject to $C_j$.

(C) When a cell $C_i$ receives all reply messages from cells in $IN_i$, it makes $r$ as used channel; otherwise, it sends abort($r$) to those cells from which it has received agree message, and then goes back to step (A).

(D) When $C_i$ finishes the use of a channel $r$, it sends a release($r$) to each cell in $IN_i$. When a cell $C_i$ receives an abort($r$) or release($r$) from $C_j$, it removes $r$ from $U_i^j$.

Figure 3.2: The update approach
channel has a higher priority than a used channel does. An intra-handoff is necessary to achieve better channel reuse. In the update approach, a cell maintains all the necessary information about its interference neighbors in order to use the cost function. Thus, this approach can support many channel selection strategies. However, in the search approach, a cell collects neighbor information only after a search. But the collected information maybe outdated when the cell releases a channel. Therefore, many good channel selection strategies cannot be supported in the search approach. Moreover, the search approach [55] locks the borrowed channel during channel borrowing, which also reduces channel reuse. In the following, we propose a channel acquisition algorithm which reduces the acquisition delay and does not lock the borrowed channel.

3.2 An Efficient Channel Acquisition Algorithm

In this chapter, we propose an efficient distributed acquisition algorithm, which has similar message complexity as the search approach and similar acquisition delay as the update approach.

3.2.1 Reducing the Acquisition Delay

In the search approach, a cell has to confirm a selected channel with the lenders (interference primary cells) since a lender may assign that channel to a new call immediately after it sent a reply. One way to avoid confirm is to let the channel lenders wait until they know which channel the borrower has selected. However, this requires all interference neighbors to lock their channels for $2 \times T$ ($T$ is one way communication delay), which may not be desirable most of the time. Our solution to this problem is as follows. When a cell receives a request, it marks some channels
as reserved channels and then sends its channel information to the borrower. The borrower selects a channel using its underlying channel selection algorithm. If the selected channel is not a reserved channel, it can use the selected channel without confirming with the lenders. Otherwise, it needs to confirm with the lenders similar to the search approach. In both situations, a cell sends finish (or transfer) messages to its interference neighbors before it starts using the borrowed channel. For any interference neighbor, if a call arrives during the channel borrowing process, it assigns a reserved channel to the call. If a call arrives after a cell has used all its reserved channels, the cell cannot assign any other available channels to this call until it receives the finish (or transfer) messages.

3.2.2 The Efficient Channel Acquisition Algorithm

In the distributed channel acquisition algorithm, if a cell has an available primary channel $r$, it uses $r$ immediately unless an interference neighbor is in the search mode. If $C_i$ does not have any available primary channel, it searches neighboring cells to find the set of available channels and picks one from them. When $C_i$ borrows a channel $r$, the interference primary cells of $r$ relative to $C_i$ cannot use $r$ until $C_i$ returns $r$.

If no channel is selected from the reserved channels, five types of messages are exchanged among MSSs to borrow or return a channel: request, reply, finish (or transfer), and release. Otherwise, two additional messages are needed: confirm and abort. In the channel acquisition algorithm, several call requests may arrive when a cell is in the search mode. In this case, the cell can just pick more channels and assign them to these call requests. For simplicity, this is not explicitly presented in the
algorithm. The following is the formal description of our efficient channel acquisition algorithm.

**Notations**

- \( IN_i, IP_i(r), S, U_i, A_i, \text{pick}, I_i \): defined before.

- \( P_i \): the set of primary channels assigned to \( C_i \).

- \( \text{reserved}_i \): the set of reserved channels at \( C_i \). \( C_i \) has at most \( N_i \) reserved channels, where \( N_i \) is a system tuning parameter (see Chapter 3.6.2).

- \( \text{send}_i \): the set of cells to which \( C_i \) has sent a reply but has not received the finish (or transfer) message.

**Initialization**

For any cell \( C_i \), \( A_i \) and \( P_i \) are set to be the set of primary channels assigned to \( C_i \). \( U_i \), \( \text{reserved}_i \), and \( \text{send}_i \) are set to empty. For any channel \( r \in P_i \), \( I_i(r) \) is set to empty.

(A) When a cell \( C_i \) needs a channel to support a call request, if \( A_i = P_i - U_i - \{ r : I_i(r) \neq \emptyset \land r \in P_i \} \) is empty, it sets its search mode, and sends a request message to each cell \( C_j \in IN_i \); otherwise, it does the follows:

(A.1) \( \text{send}_i = \emptyset \): let \( r = \text{pick}(A_i) \). it adds \( r \) to \( U_i \), and then uses \( r \) to support the call. When the call is finished, it deletes \( r \) from \( U_i \).

(A.2) \( \text{send}_i \neq \emptyset \): if \( \text{reserved}_i \) is not empty, let \( r = \text{pick}(\text{reserved}_i) \). \( C_i \) adds \( r \) to \( U_i \), deletes \( r \) from \( \text{reserved}_i \), and then uses \( r \) to support the call; otherwise, it defers the call request until \( \text{send}_i \) is empty.

(B) When a cell \( C_i \) receives a request message from \( C_j \), if it is also in the search mode and its request has smaller timestamp than \( C_j \)’s request, it defers its response until it clears its search mode; otherwise, it does the follows:

(B.1) \( \text{send}_i = \emptyset \): \( C_i \) picks \( N_i \) channels and assigns them to \( \text{reserved}_i \), then it adds \( C_j \) to its \( \text{send}_i \) and sends \( \text{reply}(U_i, \text{reserved}_i) \) to \( C_j \).

(B.2) \( \text{send}_i \neq \emptyset \): \( C_i \) adds \( C_j \) to its \( \text{send}_i \) and sends \( \text{reply}(U_i, \text{reserved}_i) \) to \( C_j \).

(C) When a cell \( C_i \) receives all reply messages from cells in \( IN_i \):
(C.1) It determines the currently available channels as follows:
\[ A_i = S - U_i - \{r : I_i(r) \neq \phi \land r \in P_i \} - \bigcup_{k \in IN, U_k} \]

(C.2) If \( A_i \) is empty, \( C_i \) drops the call, clears its search mode, and processes all the deferred reply messages. If \( A_i \) is not empty, let \( r = \text{pick}(A_i) \). If there is any cell \( C_j \) such that \( C_j \in IP_i(r) \land r \in \text{reserved}_j \), \( C_i \) sends a confirm\((r)\) to \( C_j \); otherwise, the request is successful.

(C.3) In case of success, \( C_i \) clears its search mode, adds \( r \) to \( U_i \), sends transfer\((r)\) to every cell in \( IP_i(r) \), sends finish to every cell in \( IN_i \setminus IP_i(r) \), and processes all the deferred reply messages. When \( C_i \) finishes the call, it sends release\((r)\) to every cell in \( IP_i(r) \).

(D) When a cell \( C_i \) receives a confirm\((r)\) from \( C_j \), if it is not using \( r \), it deletes \( r \) from \( \text{reserved}_i \) and responds with an agree; otherwise, it responds with a reject.

(E) If \( C_i \) receives an agree message from each cell to which it has sent a confirm\((r)\), the request is successful, go to Step C.3; otherwise, \( C_i \) deletes \( r \) from \( A_i \), sends an abort\((r)\) to every cell from which it has received an agree, and then goes to Step C.2.

(F) When a cell \( C_i \) receives an abort\((r)\), release\((r)\), finish, or transfer\((r)\) from \( C_j \), it does the following:

\( \text{abort}(r) \): \( C_i \) adds \( r \) to \( \text{reserved}_i \).

\( \text{release}(r) \): \( C_i \) deletes \( C_j \) from \( I_i(r) \).

\( \text{finish} \): \( C_i \) deletes \( C_j \) from \( \text{send}_i \). If \( \text{send}_i \) is empty, it processes all deferred call requests.

\( \text{transfer}(r) \): \( C_i \) adds \( C_j \) to \( I_i(r) \) and deletes \( C_j \) from \( \text{send}_i \). If \( \text{send}_i \) is empty, it processes all deferred call requests.

3.2.3 Correctness Proofs

Theorem 2 The distributed channel acquisition algorithm ensures that a cell and its interference neighbors do not use the same channel concurrently.

Proof. Assume the contrary that two cells \( C_i \) and \( C_j \) (\( C_i \in IN_j \)) are using the same channel \( r \). Since the distance between two primary cells is at least \( D_{\text{min}} \) (Theorem 1), \( C_i \) and \( C_j \) cannot both be primary cells of \( r \). Hence, at least one of them is a secondary cell of \( r \).
Case 1: $C_i$ is a primary cell of $r$ and $C_j$ is a secondary cell of $r$. Then $C_i \in IP_j(r)$.

When $C_i$ receives its own call request, depends on the condition of $reserved_i$, there are three possibilities.

Case 1.1: $send_i = \emptyset$. $C_i$ uses $r$ to support the call request, and adds $r$ to $U_i$ (Step A.1). When $C_j$ receives $C_i$'s $reply(U_i, reserved_i)$. $r \in U_i \iff r \notin A_j$ according to Step C.1. Then $C_j$ cannot acquire $r$.

Case 1.2: $send_i \neq \emptyset \land r \in reserved_i$. $C_i$ uses $r$ to support the call request, and adds $r$ to $U_i$ (Step A.2). In order to use $r$, $C_j$ sends $confirm(r)$ to $C_i$, but $C_i$ rejects this $confirm(r)$.

Case 1.3: $send_i \neq \emptyset \land r \notin reserved_i$. There are two possibilities:

Case 1.3.1: $C_j \in send_i$. $C_i$ waits until $C_j$ acquires $r$ (Step A.2).

Case 1.3.1: $C_j \notin send_i$. If $C_j$'s request arrives before $send_i = \emptyset$, it is similar to Case 1.3.1; otherwise, it is similar to Case 1.1.

Case 2: $C_j$ is a primary cell of $r$ and $C_i$ is a secondary cell of $r$. Similar to Case 1.

Case 3: both $C_i$ and $C_j$ are secondary cells of $r$. In order to borrow channel $r$, $C_i$ and $C_j$ must have received each other's $reply$ message. Without loss of generality, we assume that $C_i$'s $request$ has smaller timestamp than $C_j$'s $request$. Then, $C_j$ receives $C_i$'s $reply$ after $C_i$ has borrowed channel $r$ and added $r$ to $U_i$. When $C_j$ receives $C_i$'s $reply(U_i, reserved_i)$. $r \in U_i \iff r \notin A_j$ according to Step C.1. Then, $C_j$ cannot acquire $r$.

Theorem 3 The distributed channel acquisition algorithm is deadlock free.

Proof. New channel requests originating concurrently in different cells get totally ordered by their timestamps. An MSS in search mode sends $reply$ messages to all requests with a lower timestamp and defers others. As the same ordering of channel requests is seen by all the MSSs, there is no circular deferring of replys among the MSSs.
Since the communication link is reliable, the MSS whose request has the highest priority can always receive all reply messages from its interference neighbors and determine whether to confirm the selected channel. An MSS receiving a confirm responds immediately either by an agree or a reject message. Then, the MSS whose request has the highest priority can always decide whether it can successfully borrow a channel or not. Then, it processes the deferred reply and sends finish or transfer messages. The MSS deferring its own call request can always receive finish or transfer message, empty send, and then process the deferred call request. □

3.3 A Fault-Tolerant Channel Acquisition Algorithm

All the previous distributed channel acquisition algorithms [8, 26, 55] are not fault-tolerant, since channels cannot be borrowed if any interference neighbor fails. In this chapter, we propose a fault-tolerant channel acquisition algorithm [10] which tolerates communication link failures and node (MH or MSS) failures. In the proposed algorithm, a borrower does not need to receive a response from every interference neighbor. It only needs to receive a response from a small portion of them. Thus, as long as the borrower can communicate with a small portion of its interference neighbors, it can borrow a channel from them.

3.3.1 The Fault-Tolerant Channel Acquisition Algorithm

In our distributed fault-tolerant channel acquisition algorithm, when a cell \( C_i \) needs a channel, if it has an available primary channel \( r \), it marks \( r \) as an used channel and uses \( r \) immediately; otherwise, \( C_i \) changes to search mode and sends a request message to each cell in \( I.N_i \). When a cell \( C_j \) receives the request from \( C_i \), if \( C_j \) is not in search mode or \( C_j \) is in search mode but \( C_j \)'s request has higher timestamp
than C,\textsc{'s}. C, sends a reply message which appends the information about its used channels to C,; otherwise, C, defers the reply (similar to [59]). When C, has received reply messages from cells in $IN_i$ or timeout, it computes the available channels and picks one according to the underlying channel selection algorithm. The borrower C, has to confirm the selected channel with the lenders, since a lender may assign that channel to a new call immediately after it sends out a reply message. If each lender responds with an agree message, C, can use the borrowed channel; otherwise, it picks another available channel and confirms again. If there is no available channel left, the call request is failed. When a channel $r$ is borrowed, the lender marks $r$ as an interference channel, and it can not use $r$ until $r$ is returned by all borrowers.

In our fault-tolerant channel acquisition algorithm, two types of control messages are used to acquire the information on available channels: request and reply. If two or more cells in each other's interference neighbors request for the same channel concurrently, a conflict arises. If there is no conflict, three types of messages are exchanged among MSSs to confirm or return a channel: confirm, agree, and release. If there is a conflict, two additional messages are needed: abort and conditional agree. The following is the formal description of our fault-tolerant channel acquisition algorithm.

**Notations:**

- $IN_i$, $IP_i(r)$, $P_i$, $A_i$, $pick$, $I_i$: defined before.
- $B_i$: a set of channels that $C_i$ can borrow.
- $S_i$: a set of channels that $C_i$ attaches with its reply.
- $CI_i(r,j)$: a set of cells. which saves the state of $I_i(r)$ when $C_i$ receives $C_j$'s request message.

(A) When a cell $C_i$ needs a channel to support a call request, let $temp_i = P_i - U_i - \{r | r \in P_i \land I_i(r) \neq \phi\}$, if $temp_i$ is empty, $C_i$ sets a timer and sends a request message to every cell $C_j \in IN_i$; otherwise, it picks a channel $r \in temp_i$ and adds $r$ to $U_i$. When the call is finished, it deletes $r$ from $U_i$.  

37
(B) When a cell $C_i$ receives a request from $C_j$, it sends $\text{reply}(P_i - U_i - \{r \mid r \in P_i \cap (I_i(r) \cap IN_j \neq \emptyset)\})$ to $C_j$. For any $r \in P_i$, $CI_i(r,j) = I_i(r)$:

(C) When a cell $C_i$ receives all $\text{reply}(S_j)$ messages or time out, it sets a new timer and does the following:

(C.1) for each $r$ in the system, $B_i = B_i \cup r$ if
- $C_i$ is not using $r$.
- for any $C_k \in IP_i(r)$. $C_i$ has received a $\text{reply}(S_k)$ from $C_k$.
- for any $C_j \in IP_i(r)$. $r \in S_j$

(C.2) if $B_i$ is not empty, $C_i$ selects a channel $r \in B_i$ using the underlying channel selection strategy and sends $\text{confirm}(r)$ messages to all cells in $IP_i(r)$; otherwise, it drops the call.

(D) When a cell $C_i$ receives a $\text{confirm}(r)$ from $C_j$, its response depends on the current status of $r$:

(D.1) If $r \in P_i - U_i - \{r \mid r \in P_i \cap I_i(r) \neq \emptyset\}$. $C_i$ replies with an $\text{agree}(r)$ and adds $C_j$ to $I_i(r)$.

(D.2) If $r \in U_i$, $C_i$ replies with a $\text{reject}(r)$.

(D.3) If $I_i(r) \neq \emptyset$. let $\text{temp}_i = I_i(r) - CI_i(r,j)$. $C_i$ does the following:
- if $\text{temp}_i \cap IN_j = \emptyset$
  then $C_i$ replies with an $\text{agree}(r)$ and adds $C_j$ to $I_i(r)$;
- else if $\forall C_k \in (\text{temp}_i \cap IN_j)$, $C_j$'s request has a smaller timestamp than $C_k$'s request
  then $C_i$ replies with a $\text{conditional\-agree}(\text{temp}_i \cap IN_j, r)$ and adds $C_j$ to $I_i(r)$;
  else $C_i$ replies with a $\text{reject}(r)$.

(E) If $C_i$ receives a response to its $\text{confirm}(r)$ from every cell in $IP_i(r)$ and the following two conditions are satisfied, then the request is successful:

(E.1) Every response is either an agree or conditional-agree.

(E.2) For each $\text{conditional\-agree}(S,r)$ and for each $C_j \in S$. $C_i$ has received an $\text{agree}(r)$ from some $C_k \in (IP_i(r) \cap IP_j(r))$.

Otherwise, the request is failed. In case of success, $C_i$ uses $r$ to support the call and sends $\text{release}(r)$ to every cell in $IP_i(r)$ when the call terminates; in case of failure or time out, $C_i$ sends an $\text{abort}(r)$ to those cells in $IP_i(r)$ from which it has received an agree or conditional-agree message, deletes $r$ from $B_i$, and then goes to Step (C.2).
Outdated reply, agree, and conditional agree messages are discarded by comparing timestamps. When a cell $C_i$ receives a release($r$) or abort($r$) from $C_j$, it deletes $C_j$ from $I_i(r)$.

**Conflict Resolution:** In our algorithm, control messages are timestamped using Lamport's clock [44] to determine the priority of requests. The solution to conflicts is shown in Step D.3. By maintaining $I_i(r)$ and $CI_i(r,j)$, a cell $C_i$ never grants concurrent requests for the same channel from cells within $D_{min}$. Hence, no two cells requesting the same channel within $D_{min}$ receive agree messages from the same interference primary cell.

Besides conflict resolution, the adoption of conditional agree messages can also avoid wasting available channels as follows. In Figure 2.2, assume $C_{H_4}$ and $C_{F_4}$ concurrently request a channel, and $C_{H_4}$'s request has smaller timestamp. If there is one common available channel $r$ in $C_{A_4}$, $C_{A_5}$, $C_{A_7}$, and $C_{A_8}$, then $C_{A_4}$, $C_{A_5}$, $C_{A_7}$, and $C_{A_8}$ are interference primary cells of $r$ relative to $C_{H_4}$ and $C_{F_4}$, i.e., $IP_{H_4}(r) = IP_{F_4}(r) = \{C_{A_4}, C_{A_5}, C_{A_7}, C_{A_8}\}$. If $C_{H_4}$'s request arrives at $C_{A_4}$, $C_{A_5}$, and $C_{A_7}$ earlier than $C_{F_4}$'s request, but arrives at $C_{A_8}$ later than $C_{F_4}$'s request, $C_{A_4}$, $C_{A_5}$, $C_{A_7}$, or $C_{A_8}$ cannot send agree messages to both $C_{H_4}$ and $C_{F_4}$ due to the possibility of an interference. Then, without conditional agree messages, both $C_{H_4}$ and $C_{F_4}$ cannot use channel $r$. With conditional agree messages, $C_{H_4}$ gets agree messages from $C_{A_4}$, $C_{A_5}$, and $C_{A_7}$, and a conditional agree from $C_{A_8}$, then it can acquire $r$. However, $C_{F_4}$ cannot acquire $r$ since $C_{A_4}$, $C_{A_5}$, and $C_{A_7}$ reject its requests.

**Fault-Tolerance:** In the proposed algorithm, a borrower does not need to receive a response from every interference neighbor. It only needs to receive a response from each cell in an interference partition subset as long as there is one common available channel among them. Based on Theorem 1, any two interference cells have at least...
one common interference primary cell of a channel. Also, the common interference primary cell never replies an agree message to more than one cell requesting the same channel. Thus, channel interference is avoided.

Since the number of cells in an interference partition subset is far less than the number of interference neighbors, our algorithm tolerates node and communication link failures. For example, in a typical cellular network model with \( D_{\text{min}} = 3\sqrt{3}R \), the number of interference neighbors of a cell is 30, and the number of interference primary neighbors of a cell is 3 or 4. Then, in the best case, a cell can still borrow channels even though it cannot communicate with as many as \((30 - 3) = 27\) (i.e., \(27/30 = 90\%\)) of its interference neighbors. In the worst case, even though a cell cannot communicate with as many as \(((30/4)\] = 7\) (i.e., \(7/30 = 23\%\)) of its interference neighbors, it can still communicate with the remaining \(30 - 7 = 23\) cells, which includes all cells (at most 4) of an interference partition subset. If there are common available channels among cells in this interference partition subset, the cell can borrow these available channels.

**Outdated Messages:** Due to communication link failure or network congestion, messages such as reply, agree, and conditional-agree may arrive at a cell after the cell has terminated the channel acquisition process. We call these messages outdated messages. Outdated messages must be identified and discarded; otherwise, two cells may interfere with each other. In order to identify outdated messages, when a cell receives a message, such as reply, agree or conditional-agree, it compares the timestamp of the received message with the timestamp of its own request message. If the received message has a smaller timestamp than its own request, it is an outdated
messages. Also, if a cell is not in the process of channel acquisition, all received \textit{reply}, \textit{agree}, and \textit{conditional agree} are outdated messages.

**The Timer:** Timers are used in our algorithm to deal with MSS or communication link failures. The selection of the timeout period affects the system performance. If the timeout period is too long, a handoff may be dropped due to the long delay. If the timeout period is too short, there may be less opportunity for the channel selection algorithm to choose a channel. The timeout period also depends on the application. For example, a handoff request can tolerate much less delay than a new call request.

Suppose the time limit to borrow a channel is $T_{\text{limit}}$. For simplicity, we set the timer to $T_{\text{limit}}/2$. Then, under network congestion, link failures or MSS failures, the borrower waits $T_{\text{limit}}/2$ time interval for \textit{reply} messages, and another $T_{\text{limit}}/2$ time interval for \textit{confirm} responses. We do not set a different timeout period for confirm responses. The reason is as follows. A borrower only sends \textit{confirm} messages to cells from which it has received \textit{reply} messages. Since the probability that a failure occurs during this $T_{\text{limit}}/2$ time interval is very low: it has a large probability to receive an \textit{agree} message from them.

There are other possible approaches. For example, there can be a timer for each round of confirm so that if a conflict or failure occurs during the confirming process, the borrower can select another channel. For simplicity, we use one time period in our algorithm.

**Real-Time Communication:** Our algorithm can \textit{partially} support real-time communication. Even when some cells suffer from network congestion, a cell can still borrow channels from cells that are not experiencing network congestion. Also, a delay deadline is guaranteed by the timer. However, if a cell suffering from network
congestion needs to borrow a channel, it is very likely to experience a long delay. Even in this situation, our algorithm still has shorter delay compared to non-fault-tolerant algorithms since our algorithm does not need to wait for the response messages from the cells suffering from network congestion.

3.3.2 Correctness Proofs

**Theorem 4** The distributed channel acquisition algorithm ensures that a cell and its interference neighbors do not use the same channel concurrently.

*Proof.* Assume to the contrary that two cells $C_i$ and $C_j$ ($C_i \in I.N_j$) are using the same channel $r$. Since the distance between two primary cells is at least $D_{min}$ (Theorem 1), $C_i$ and $C_j$ cannot both be primary cells of $r$. Hence, at least one of them is a secondary cell of channel $r$.

**Case 1:** $C_i$ is a primary cell of $r$ and $C_j$ is a secondary cell of $r$. Then $C_i \in IP_j(r)$.

There are two possibilities:

**Case 1.1:** $C_i$ receives $C_j$'s request after its own request. To use channel $r$, $C_i$ adds $r$ to $U_i$ (Step A). When $C_j$ receives $C_i$'s reply($P_i - U_i - \{r \mid r \in P_i \land (I_i(r) \cap I.N_j \neq \emptyset)\}$), $r \in U_i \implies r \notin B_j$ according to Step C.1. Then $C_j$ cannot acquire $r$.

**Case 1.2:** $C_i$ receives $C_j$'s request before its own request. $r$ is an interference channel when $C_i$ starts the request (Step D.1). then $C_i$ will not acquire $r$.

**Case 2:** $C_j$ is a primary cell of $r$ and $C_i$ is a secondary cell of $r$. Similar to Case 1.

**Case 3:** Both $C_i$ and $C_j$ are secondary cells of $r$. According to Theorem 1, $IP_i(r) \cap IP_j(r) \neq \emptyset$. Without loss of generality, we assume that $C_i$'s request has a smaller timestamp than $C_j$'s request. If $C_i$ finally acquires $r$, then it must have received an agree message from a neighboring primary cell $C_k \in IP_i(r) \cap IP_j(r)$.

There are two possibilities depending on when $C_k$ receives $C_j$'s request:
Case 3.1: \( C_k \) receives \( C_j \)'s request after it sends agree to \( C_i \). According to Step D.1 or D.3, \( C_k \) adds \( C_i \) to \( I_k(r) \). When \( C_k \) receives \( C_j \)'s request, \( C_i \in I_k(r) \implies I_k(r) \cap I.N_j \neq \emptyset \). Then, when \( C_j \) receives \( reply(B_k) \) from \( C_k \), \( r \notin B_k \) according to Step B. \( r \notin B_k \implies r \notin B_j \) according to Step C.1. Thus, \( C_j \) cannot acquire \( r \).

Case 3.2: \( C_k \) receives \( C_j \)'s request before it sends agree to \( C_i \). Then \( C_i \notin CI_k(r,j) \). When \( C_k \) receives \( C_j \)'s confirm, if \( C_i \notin I_k(r) \), \( C_j \in I_k(r) \implies temp_i \cap I.N_i \neq \emptyset \), and then \( C_i \) can not get an agree from \( C_k \) (Step D.3). If \( C_i \in I_k(r) \), we have \( C_i \notin CI_k(r,j) \cap C_i \in I_k(r) \implies C_i \in (temp_k \cap I.N_j) \). Since \( C_j \)'s request has larger timestamp than \( C_i \)'s request, \( S_k \) responses with a reject to \( C_j \)'s confirm (Step D.3). Hence, \( C_j \) cannot acquire \( r \).

Theorem 5  The channel acquisition algorithm is deadlock free.

Proof. In the channel acquisition algorithm, an MSS receiving a request responds immediately by a reply. An MSS receiving a confirm also responds immediately by an agree, a conditional agree, or a reject. A borrowing MSS uses a timer to make sure that it will not wait forever. Hence, our algorithm is deadlock free. □

3.3.3 Failure Recovery

Even though our channel acquisition algorithm is fault-tolerant, quick recovery can significantly reduce the failure rate. Hence, we briefly describe how to recover from failures.

**MH Failures:** When an MH fails in the middle of a communication session, the session is terminated. Hence, the channel that was being used for the communication session is no longer in use. The corresponding MSS detects the failure of the MH in
its cell and deletes the channel from its \( U \). Thus, as far as channel allocation is concerned, failure of an \( MH \) is conceptually handled in the same way as the completion of a communication session.

**MSS Failures:** We assume that MSS failures are fail-stop in nature. When an MSS, say \( C_i \) fails, all the communication sessions between \( C_i \) and \( MHs \) in its cell are terminated. Hence, no channel is in use inside \( C_i \) after \( C_i \) fails before it recovers. When \( C_i \) fails, its neighbors may still send request, confirm, release, or abort message to \( C_i \). Since a cell does not need to receive responses from all its interference neighbors, other cells can still borrow channels that are not \( C_i \)'s primary channels.

When \( C_i \) recovers from a failure, it needs to reconstruct its \( U \) and \( I \) as follows. \( C_i \) clears \( U \) and \( I \), and then it broadcasts cell_recovery to all its interference neighbors. Any cell receiving cell_recovery replies with the channels that it borrowed from \( C_i \). Based on these replies, \( C_i \) reconstructs its \( I \). For example, adds \( C_j \) to \( I_i(r) \) if \( C_j \) borrowed channel \( r \) from \( C_i \). \( C_i \) can use sliding window protocol [63] to make sure that every interference neighbor responds to its cell_recovery message.

**Communication Link Failures:** When there is a communication link failure, the underlying protocol at the network layer should route messages from some other links. Since the loss of control messages such as release and abort may significantly reduce system performance, sliding window protocol should be used to guarantee that the messages such as abort and release are not lost. In case of a network partition, a failure recovery procedure is used to recover from the loss of these two messages. During the recovery, an MSS, say \( C_i \), broadcasts link_recovery to all its interference neighbors. A cell \( C_j \) receiving a link_recovery from \( C_i \) replies all the channels that
$C_j$ borrowed from $C_i$. When $C_i$ receives these messages, it reconstructs its $I_i$ similar to the MSS failure recovery.

Due to communication link failure or network congestion, messages such as reply, agree, and conditional agree may arrive at a cell after the cell has terminated the channel acquisition process. How to deal with these outdated messages has been discussed in Chapter 3.3.1.

### 3.4 A Channel Selection Algorithm

Similar to the geometric strategy [3] and the update approach [26], our channel selection algorithm makes use of the resource planning model defined in Chapter 2.2. The primary channels for each cell are prioritized. During a channel acquisition, a cell acquires the available primary channel that has the highest priority. If none of the primary channels is available, the cell borrows a channel from its neighbors according to some priority assignment approach. Before presenting our priority assignment strategy, we make two observations and identify two guiding principles in designing channel selection algorithms.

**Observation 1:** In Figure 2.2, after $C_{H_1}$ borrows a channel $r_1$ from $C_{A_4}$ ($r_1$ is a primary channel of $C_{A_4}$), $C_{A_1}$, $C_{A_2}$, $C_{A_4}$, and $C_{A_5}$ cannot use $r_1$ due to channel interference. Thus, the borrow of $r_1$ interferes with four cells. Suppose $C_{F_1}$ borrows $r_2$ from $C_{A_4}$. Later, $C_{H_4}$ runs out of channels. If $C_{H_4}$ borrows a channel $r_3$ from $C_{A_4}$, the borrow of $r_3$ interferes with four cells: $C_{A_4}$, $C_{A_5}$, $C_{A_7}$, and $C_{A_8}$. However, if $C_{H_4}$ borrows channel $r_2$ from $C_{A_4}$, the borrow of $r_2$ only interferes with two new cells: $C_{A_7}$ and $C_{A_8}$. ($C_{A_4}$ and $C_{A_5}$ cannot use $r_2$ since $C_{F_1}$ borrowed $r_2$.) Later, if $C_{F_4}$ wants to borrow a channel from $C_{A_4}$, it cannot borrow $r_1$ and $r_2$ due to channel interference.

45
but the borrow of any other channel may interfere with four cells. Note that if $C_{H_4}$ borrows $r1$ instead of $r2$ from $C_{A_4}$. $C_{F_4}$ can borrow $r2$ and only interferes two new cells $C_{A_7}$ and $C_{A_6}$. Based on this observation, we identify the following principle.

**Principle 1:** *When a cell borrows a channel, it should select a channel which causes a smaller number of lenders to become interfered. Also, if possible, the selected channel should be the same channel borrowed by its co-channel cells.*

**Observation 2:** In Figure 2.2, suppose $C_{H_1}$ borrows channel $r1$ from $C_{A_4}$. As a result, $C_{A_1}$, $C_{A_2}$, $C_{A_3}$, and $C_{A_5}$ cannot use $r1$. Suppose $C_{A_4}$ runs out of primary channels just after it lends channel $r1$ to $C_{H_1}$. Then, it needs to borrow a channel from other cells which may interfere with four cells. Therefore, $C_{H_1}$ should only borrow channels from the “richest” interference neighbor; i.e., the cell with the most available primary channels. The motivation behind this is to reduce the chance that the lender might soon use up its primary channels and have to acquire a secondary channel. Based on this observation, we identify the following principle.

**Principle 2:** *A cell should try to borrow a channel from the “richest” interference neighbor.*

We propose a priority assignment strategy to integrate these two principles. Let the cells be partitioned into $k$ disjoint optimal reuse patterns $G_0, G_1, \ldots, G_{k-1}$ as defined in Chapter 2.2. Without loss of generality, we assume that there are a total of $k \cdot N$ channels numbered $0, 1, \ldots, k \cdot N - 1$ which are evenly divided into $k$ subsets: $P_0, P_1, \ldots, P_{k-1}$ (This assumption is not essential and is made only for ease of presentation). A cell $C_i$ is assigned $N$ channels numbered $min_i, min_i + 1, \ldots, min_i + N - 1$. 

46
Note that $\forall i, j (C_i \in G_k \land C_j \in G_k \implies min_i = min_j)$. To present the priority assignment strategy, we introduce the following notations.

- $A_i$: the set of currently known available channels at $C_i$.
- $PC(r)$: the set of primary cells of $r$.
- $CO_i(r)$: the number of co-channel cells of $C_i$ that are borrowing channel $r$.
- $I_i(r)$: the set of cells to which $C_i$ has lent channel $r$.
- $\delta$: when a cell needs to borrow a channel, if possible, it should not borrow channels from those cells whose available channels are less than the threshold $\delta$. $\delta$ is a system tuning factor (see Chapter 3.6.2).

**Definition 3** Given a cell $C_i \notin PC(r)$, the “richness” of channel $r$ relative to $C_i$, denoted by $RH_i(r)$, is measured as the minimum number of primary channels which are available in the interference primary cells of $r$ relative to $C_i$:

$$RH_i(r) = \min \{|A_j| : C_j \in PC(r) \cap I.N_i\}$$

**Definition 4** Given a cell $C_i \notin PC(r)$, before the borrow of channel $r$, the number of lenders which have lent $r$ to some cells other than $C_i$, denoted by $BI_i(r)$, is defined as:

$$BI_i(r) = |\{C_j : C_j \in PC(r) \cap I.N_i \land I_j(r) \neq o\}|$$

Based on these definitions, the priority of a channel $r$ relative to cell $C_i$ is defined as follows:

$$P_i(r) = \begin{cases} 
  m - (r - min_i) & \text{if } C_i \in PC(r) \\
  (r - min_k) + o \ast (CO_i(r) + BI_i(r)) \ast (RH_i(r) - \delta) & \text{if } C_i \notin PC(r) \land C_k \in PC(r)
\end{cases}$$

(3.1)

where $m \gg o \geq N$, e.g., $m = 11 \ast o \ast o \ast o = N + 1$.

From Equation 3.1, the primary channels in a cell have the highest priority since $m$ is a significant large number. For secondary channels, the priority is determined by
Principle 1 and Principle 2: if two channels have the same interference property and “richness”, the channel with higher number has higher priority. Initially, we do not consider the relative importance of Principle 1 and Principle 2: this strategy could be extended by changing the relative importance of these two principles. We found (by simulation) that a lender should not lend any channel to others when its available channels are lower than $S$ if it is possible: i.e., the borrower can borrow channels from other cells.

### 3.5 Complete Channel Allocation Algorithms

Most of the existing DCS strategies [20, 21, 27, 34, 36, 37, 51, 65] need up-to-date information to calculate channel priority. This can be easily implemented in centralized algorithms, since an MSC monitors every release and acquisition of channels and then it has the up-to-date information. However, in a distributed channel allocation algorithm, due to unpredictable message delay, obtaining the instantaneous global state information is practically impossible. Thus, we can only obtain the approximately up-to-date information by increasing message overhead. For example, in the update approach [26], a cell sends update notification messages to its interference neighbors each time it acquires or releases a channel so that each cell always knows the available channels of its interference neighbors. In order to combine the channel selection algorithm with our distributed channel acquisition algorithms (the efficient channel acquisition algorithm and the fault-tolerant algorithm) and do not significantly increase the message overhead, we make some modifications to our algorithms.
3.5.1 Reducing the Update Message Overhead

To make use of locality, a cell does not return the borrowed channel immediately after its use. Instead, it keeps the borrowed channel so that these channels can be used when the borrower runs out of channel again. Thus, there are two kinds of borrowed channels: using-borrowed channel and available-borrowed channel. Using-borrowed channel are counted as used channels. Available-borrowed channels are counted as available channels and can be used again without contacting with interference neighbors. When a cell knows that its lender's available channels are less than a threshold $\eta$ (determined later), it releases the available-borrowed channels from that lender. It should perform an intra-handoff to release the using-borrowed channels from that lender if it is possible (it is not possible when there is no other available channel). Whenever a communication session (or a call) in a cell is over, it checks if its available channels are larger than $\eta'$: if so, it releases some borrowed channels.

There are two approaches to reduce the update notification message overhead. In Approach 1, we modify Step A of our channel acquisition algorithm as follows: when a cell acquires or releases a primary channel, it notifies all cells which have borrowed channels from it. Then, a cell keeps the up-to-date information for calculating the channel priority of its interference neighbors. This approach significantly reduces the update notification message overhead compared to the update approach, since the number of borrowers is very small compared to the number of interference neighbors. In Approach 2, a cell only notifies the cells that have borrowed channels from it when its available channels are less than $\eta'' (\eta'' > \eta)$.
The disadvantage of Approach 2 is that the borrower may not know the up-to-date information. The advantage of Approach 2 is low message overhead and low intra-handoff overhead. Knowing the up-to-date information is only helpful when releasing the borrowed channels. Since we want to make use of locality by keeping borrowed channels, and a borrowed channel will be released when its lender’s available channels are lower than \( r_j \), it may not be necessary to know the up-to-date information of the lender considering the high message overhead. Thus, we implemented Approach 2 in our algorithm.

To make use of locality, \( r_j' \) should be as large as possible. However, keeping too many borrowed channels may increase the failure rate since other interference cells cannot use them. Certainly, we do not want to make use of locality at the expenses of increasing failure rate. Thus, we choose \( r_j' \) to be a small value. \( r_j \) should be as small as possible so that the borrower can keep the borrowed channel. However, if \( r_j \) is too small, the lender may run out of channel. \( r_j'' \) should be as small as possible to reduce the update notification message overhead. However, a larger value can help the borrower get more up-to-date information. Based on the above considerations and our simulation results, we choose \( r_j \) and \( r_j' \) to be 5% of the number of primary channels. \( r_j'' \) to be 10% of the number of primary channels. Since this is not our major concern, we will not further investigate how to choose the value of these parameters.

Note that our channel acquisition algorithms (the efficient channel acquisition algorithm and the fault-tolerant algorithm) are independent of the channel selection algorithm being used. We can use the same channel selection algorithm as that of the update approach or any other newly developed channel selection algorithm. Certainly, if we choose the channel selection algorithm used in the update approach or the search
approach, the update notification message overhead will be avoided. Note that the update notification messages are not required by the channel selection algorithm used in the update approach, but it is necessary for the channel acquisition algorithm used in the update approach.

3.5.2 Reducing the Overhead of Intra-handoff

In Figure 2.2, suppose cell $C_{A_1}$ has two primary channels $r1$ and $r2$. $C_{A_1}$ is using $r2$, while cells $C_{A_2}$, $C_{A_4}$, and $C_{A_5}$ are using $r1$. Even though $r1$ is available in $C_{A_1}$ and $r2$ is available in cells $C_{A_2}$, $C_{A_4}$, and $C_{A_5}$, neither $r1$ nor $r2$ can be borrowed by $C_{H_1}$. If an intra-handoff is performed (i.e., $C_{A_1}$ releases $r2$ and uses $r1$), $C_{H_1}$ can borrow $r2$. Thus, when a cell has several available primary channels, it acquires the highest priority channel and releases the lowest priority channel. If a newly available primary channel has higher priority than some used primary channels, an intra-handoff is performed.

Since intra-handoffs increase system overhead, we use the following approach to reduce the number of intra-handoffs. If an intra-handoff is between two channels whose channel sequence numbers are smaller than a threshold $\theta$, this intra-handoff can be avoided. The reason is as follows. According to our channel priority assignment strategy, a cell uses small sequence number channels and lends high sequence number channels to other cells. For a cell $C_i$, if both intra-handoff channels have small sequence numbers, $C_i$ is more likely to have a large number of available channels, and it has a low probability for other cells to select the intra-handoff channels to borrow.
In our algorithm, for a cell $C_i$, the threshold $\theta$ is set to be $\min + \frac{\sqrt{N}}{2}$. Certainly, a fine grain tuning may further reduce the number of intra-handoffs, but it may also increase the failure rate.

3.6 Performance Evaluation

By integrating the proposed channel selection algorithm and the proposed efficient channel acquisition algorithm, we get an efficient channel allocation algorithm. By integrating the proposed channel selection algorithm and the proposed fault-tolerant channel acquisition algorithm, we get a fault-tolerant channel allocation algorithm.

We evaluate the performance of the proposed channel allocation algorithms under two environments: without failure (of MSSs or communication links) and with failures. Without considering failures, we study the performance of the proposed fault-tolerant channel allocation algorithm, efficient channel allocation algorithm, the search approach [35], and the update approach [26]. We analyze the message complexity and acquisition delay of each approach. Also, we implemented these algorithms and evaluate their performance under both uniform and non-uniform traffic distributions.

When considering failures, we compare the performance of two channel allocation algorithms: one is the proposed fault-tolerant channel allocation algorithm, and the other is a non-fault-tolerant channel allocation algorithm which is a trivial modification of the proposed algorithm, where a borrower needs to consult with all interference neighbors. To avoid deadlocks in the non-fault-tolerant algorithm, a call request fails if the borrower can not get all responses within a time limit. We do
not compare our algorithm with other algorithms under failure environment since all
known distributed channel allocation algorithms do not provide fault tolerance.

3.6.1 Performance Analysis and Comparison

We analyze the performance of our efficient channel allocation algorithm and our
fault-tolerant channel allocation algorithm, and compare them to the search approach
[55] and the update approach [26]. Let \( n_p \) denote the number of interference primary
neighbors of a cell. For our efficient channel allocation algorithm, the number of
messages per primary channel acquisition and the primary channel acquisition delay
are both 0. If there is no need to confirm with the lenders, the average secondary
channel acquisition delay is \( 2 \times T \) and the number of messages per secondary channel
acquisition is \( 3n + 12 + n_p \), which includes \( n + 6 \) (there are six co-channel cells)
number of request and reply messages. \( (n - n_p) \) number of finish. and \( n_p \) number
of transfer and release messages. If a cell confirms \( m \) \( (m \geq 1) \) times before it
acquires a channel, the number of messages per secondary channel acquisition is
\( 3n + 12 + n_p + 2 \times m \times n_p + (m - 1) \times n_p \), where \( 2 \times m \times n_p \) indicates \( m \) rounds of
maximum \( n_p \) number of confirm and agree (or reject) messages. and \( (m - 1) \times n_p \)
indicates the maximum number of abort messages for the \( m - 1 \) times failed confirm.

For our fault-tolerant channel allocation algorithm, the number of messages per
primary channel acquisition and the primary channel acquisition delay are both 0. If
there is no need to confirm with the lenders, the average secondary channel acquisition
delay is \( 2 \times T \) and the number of messages per secondary channel acquisition is \( 2n +
12 + 3 \times n_p \), which includes \( n + 6 \) (there are six co-channel cells) number of request
and reply messages. \( n_p \) number of confirm, agree, and release messages. If a cell
confirms \( m \) \( (m \geq 1) \) times before it acquires a channel, the number of messages per
secondary channel acquisition is \(2n + 12 + 3 \times n_p + 2 \times m \times n_p + (m - 1) \times n_p\), where \(2 \times m \times n_p\) indicates \(m\) rounds of maximum \(n_p\) number of confirm and agree (or reject) messages, and \((m - 1) \times n_p\) indicates the maximum number of abort messages for the \(m - 1\) times failed confirm.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Messages Complexity</th>
<th>Acquisition Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search approach</td>
<td>(\alpha_1 \times (2 \times n + 3 \times n_p \times m_1 + 2 \times n_p)))</td>
<td>(2 \times T \times (2 + m_1) + T_1)</td>
</tr>
<tr>
<td>Update approach</td>
<td>(2 \times n + \alpha_2 \times (3 \times n_p \times m_2 + 2 \times n_p))</td>
<td>(2 \times T \times (1 + m_2) + T_2)</td>
</tr>
<tr>
<td>Efficient app.</td>
<td>(\leq \alpha_3 \times (3 \times n + 12 + 3 \times n_p \times m_3 + n_p) + n_u)</td>
<td>(2 \times T \times (1 + m_3) + T_3)</td>
</tr>
<tr>
<td>Fault-tolerant</td>
<td>(\leq \alpha_3 \times (2 \times n + 12 + 3 \times n_p \times m_4 + 2 \times n_p) + n_u)</td>
<td>(2 \times T \times (2 + m_4) + T_4)</td>
</tr>
</tbody>
</table>

Table 3.1: Comparison of the update, the search, the efficient, and the fault-tolerant approaches

Let \(n_u\) denote the number of update messages needed in our efficient and fault-tolerant algorithms. \(\alpha(\alpha_1, \alpha_2, \alpha_3, \alpha_4)\) denote the percentage of secondary channel acquisition. \(m(m_1, m_2, m_3, m_4)\) denote conflict rates. and \(T'(T'_1, T'_2, T'_3, T'_4)\) denote the extra deferred delay due to a conflict. Our efficient channel allocation algorithm and our fault-tolerant channel allocation algorithm have the same channel selection strategy. Without considering failure and message delay variation: i.e., the timer of the fault-tolerant algorithm can be set long, both algorithms have the same number of update messages, the same percentage of secondary channel acquisition, and the same call failure rate. Table 3.1 lists the average number of messages per channel acquisition and the average secondary channel acquisition delay in the search approach, the update approach, our efficient approach, and our fault-tolerant approach.
In a typical cellular network model with $D_{min} = 3\sqrt{3}R$, we have $n = 30$ and $n_p = 3$ or 4. Normally (according to the simulation), $m(m_1, m_2, m_3, m_4)$ and $T'(T'_1, T'_2, T'_3, T'_4)$ are both very small compared to $T$. Also, $\alpha_3 < \alpha_2 < \alpha_1$. From Table 3.1, our efficient channel allocation algorithm almost cuts the secondary channel acquisition delay to half compared to the search approach. When the channel request load is low, most of the time, it is not necessary for a cell to borrow channels from others; thus, both $\alpha$ and $n_u$ are near 0 under low channel request load. When the channel request load increases, more cells run out of primary channels and have to make more secondary channel acquisitions, then the value of $\alpha$ and $n_u$ increases. Normally (according to the simulation results), even when the channel request load increases to 100%, $\alpha$ is still less than 0.3 and $n_u$ is far smaller than $n$. (Note that $n_u$ is 0 if we use the geometric strategy or the update approach as the underlying channel selection algorithm). Thus, both our efficient channel allocation algorithm and our fault-tolerant channel allocation algorithm reduce the message complexity compared to the update approach whose message complexity is always larger than $2 \times n$.

In Table 3.1, we do not differentiate the delay $T$ used in different approaches. Actually, the delay $T$ should be the maximal delay $T_{max}$ in the search, the update, and our efficient approach, since a cell needs to wait for reply messages from all interference neighbors in these three approaches. The delay $T$ used in our fault-tolerant channel allocation algorithm is the minimum of $T_{max}$ and $T_{timer}$, where $T_{timer}$ is the value of the timer used in our fault-tolerant channel allocation algorithm. Thus, even though our fault-tolerant algorithm has a long delay from table 3.1, it may have much shorter delay under network congestion, and the delay is bounded.
3.6.2 Simulation Parameters

The simulated cellular network is a wrapped-around layout with $12 \times 12$ cells. The total number of channels in the system is 396. With $D_{\text{min}} = 3\sqrt{3}R$, each cell is assigned $396/9 = 44$ channels. Under normal condition (no network congestion), the average one-way communication delay between two MSSs is 2 milliseconds, which covers the transmission delay, the propagation delay, and the message processing time.

Under uniform traffic distribution (shown in Table 3.2), traffic in each cell is characterized by the mean arrival time, the mean service time, and the mean inter-handoff time, all assumed to be negative exponentially distributed.

Non-uniform traffic distribution is modeled by a two-state Markov Modulated Possion Process, where a cell can be in one of two states: hot state or normal state. As shown in Table 3.3, a cell spends most of its time in the normal state. A cell in the normal state is characterized by low arrival rate and high inter-handoff rate. On the contrary, a cell in the hot state is characterized by high arrival rate and low inter-handoff rate to capture more arriving new users and prevailing stationary users. Each cell can dwell in either state for an exponentially distributed time independent of one another.

<table>
<thead>
<tr>
<th>Mean arrival rate in a cell</th>
<th>$\lambda$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean inter-handoff rate in a normal cell</td>
<td>$1/80$s</td>
</tr>
<tr>
<td>Mean service time per communication session</td>
<td>$180$s</td>
</tr>
</tbody>
</table>

Table 3.2: Simulation parameters for uniform traffic distribution
Mean arrival rate in a normal cell | \( \lambda \)  
---|---  
Mean arrival rate in a hot cell | \( 3\lambda \)  
Mean inter-handoff rate in a normal cell | \( 1/80s \)  
Mean inter-handoff rate in a hot cell | \( 1/180s \)  
Mean rate of change from normal state to hot state | \( 1/1800s \)  
Mean rate of change from hot state to normal state | \( 1/180s \)  
Mean service time per communication session | \( 180s \)  

Table 3.3: Simulation parameters for non-uniform traffic distribution

**The Value of \( N_i \):** We assume \( N_i \) in our efficient channel allocation algorithm is 1. If \( N_i \) is 0, all the interference neighbors of a borrower have to lock their channels for \( 2 \times T \), which is not affordable. With \( N_i = 1 \), when a lender receives a new call from its own cell during its locking period (from it sends out a *reply* to receive a *finish* or *transfer*), it can assign the *reserved* channel to the new call, but it needs to wait if it receives another new call during the locking period. Under 100% channel request load, our simulation shows that the probability for any interference neighbor of a borrower receiving two calls during the locking period is as low as 0.00003, which is negligible. With \( N_i > 1 \), the probability of receiving two calls during the locking period can be further reduced. However, the probability for the borrower to select a *reserved* channel becomes larger. When a cell selects a *reserved* channel, it needs an extra round of *confirm* messages and doubles the acquisition delay. For example, with \( N_i = 2 \), the extra delay due to *confirm* at the borrower side is nearly doubled compared to \( N_i = 1 \) from our simulation. Since 0.00003 is already a very small number, further reducing the probability of waiting at the lender side by increasing \( N_i \) cannot compensate for the increased delay at the borrower side. Thus, we only
consider \( N_i = 1 \) in our simulation. Similarly, a lender only needs \( 2 \cdot T \) to ask the borrowers to return the borrowed channels, and the lender has a very low probability (0.00003) of receiving another call during the \( 2 \cdot T \). Hence, we choose \( \eta = 1 \) in our simulation.

![Figure 3.3: Impact of \( \delta \) on call failure rate under uniform distribution](image)

**The Value of \( \delta \):** We use simulation to determine the value of parameter \( \delta \). We consider \( \delta \) to be 0.5% (\( \delta = 2 \)), 10% (\( \delta = 5 \)), 20% (\( \delta = 9 \)) of the primary channels. From Figure 3.3 and 3.4, we can see that \( \delta = 2 \) has the best performance, and \( \delta = 0 \) has the worst performance. However, the difference is not too much. Moreover, \( \delta = 2 \) performs worse than others under some situations, e.g., \( \delta = 5 \) performs better when the arrival rate is 650 under non-uniform distribution. When \( \delta = 0 \), even the borrower only has one channel left, it may still lend it to other cells, and then the lender is likely to run out of channel and borrow channels again. This can be solved by increasing
\[ \delta = 0 \]
\[ \delta = 5 \]
\[ \delta = 9 \]

\[ 0.001 \times 440, 460, 480, 500, 520, 540, 560, 580, 600, 620, 640, 660 \]

No. of call arrivals per hour per cell (non-uniform)

Figure 3.4: Impact of \( \delta \) on call failure rate under non-uniform distribution

\( \delta \) to 2. However, further increasing \( \delta \) performs worse since Principle 1 may not be adequately considered. In the following, we assume \( \delta = 2 \).

3.6.3 Simulation Results (Without Failure)

For each arrival rate, the mean value of a measured data is obtained by collecting a large number of samples such that the confidence interval is reasonably small. In most cases, the 95\% confidence interval for the measured data is less than 10\% of the sample mean.

The performance of the channel allocation algorithm is measured by the call failure rate \([3]\) \[ R_f = R_b + (1 - R_b) \times R_d \] where \( R_b \) is the call blocking rate and \( R_d \) is the call dropping rate. The call blocking rate is the percentage of new calls that are blocked due to insufficient resources, and the call dropping rate is the percentage of on-going calls that are dropped during inter-cell handoffs due to insufficient resources.
A Comparison of Call Failure Rate: As explained earlier, our efficient algorithm and our fault-tolerant algorithm have the same channel selection strategy. Without considering failure and message delay variation: i.e., the timer of the fault-tolerant algorithm can be set long, both algorithms should have the same number of call failure rate. Thus, we will not explicitly differentiate our efficient approach from our fault-tolerant approach when comparing the call failure rate and the percentage of secondary channel acquisition.

The call failure rate of our algorithm is compared with the geometric strategy, the search approach, the update approach, and the two-step strategy. Since the geometric strategy, the update approach, the two-step strategy, and our algorithm are all based on the optimal resource planning model, the call blocking rate in these four approaches
Our algorithm -
Update approach -
Geometric approach -
Search approach -
Two-step approach -

Figure 3.6: Comparisons of call failure rate under non-uniform distribution

does not have too much difference, with our algorithm slightly outperforming the other three since only our algorithm follows both Principle 1 and Principle 2. In the geometric strategy, neither Principle 1 nor Principle 2 is followed. The Update approach only partially follows Principle 2, which makes it slightly outperform the geometric strategy. In the two-step strategy, only Principle 1 is considered, which makes it outperform the update approach. Compared to the search approach, our algorithm significantly reduces the call blocking rate. This is due to the fact that the search approach is a simple channel borrowing approach without considering any channel reuse. Moreover, the search approach locks the borrowed channel during channel borrowing, which also reduces channel reuse.

Note that under non-uniform traffic distribution, the two-step strategy slightly outperforms our algorithm when the traffic load becomes very high (650 calls per
hour per cell). However, the two-step strategy is a centralized algorithm, and then it has poor reliability and scalability. Also, most of time, it has higher call blocking rate than ours. From Figure 3.7, we can see that our algorithm has less intra-handoffs than the two-step strategy. In the two-step strategy, when a primary channel is released, there will be an intra-handoff if a call is using a borrowed channel. In our approach, since a borrowed channel can be temporarily saved locally, this kind of intra-handoff is not necessary. According to our simulation parameters, a call on average experiences three inter-handoffs and each inter-handoff has a channel release which is more likely associated with an intra-handoff. Thus, there are about three intra-handoffs for each call. Both the geometric strategy and the update approach have similar intra-handoff overhead to the two-step strategy. The search approach does not have intra-handoff overhead, which is one of the key reasons why it has a high call blocking rate.

**Message Complexity per Channel Acquisition:** As shown in Figure 3.8, the number of messages per channel acquisition in the update approach [26] is never
lower than \( 2 \times n = 60 \) (\( n \) is the number of interference neighbors), since a cell has to communicate with its interference neighbors whenever it acquires or releases a channel. In the search approach [55], our efficient approach, and our fault-tolerant approach, a cell only communicates with its interference neighbors when it needs to borrow a channel. From Figure 3.8, the message complexity of the search approach, our efficient approach, and our fault-tolerant approach increases from near 0 to about 20 as the channel request load increases. This can be explained by the fact that most of the call requests can be satisfied by the primary channel acquisition under low channel request load. As channel request load increases, more cells run out of primary channels and have to make more secondary channel acquisitions.

As shown in Figure 3.8, although including update notification messages, the proposed algorithms still has lower message complexity than the search approach under uniform traffic distribution. This can be explained by the fact that these algorithms have different secondary channel acquisition percentage. From Figure
Figure 3.9: Percentage of secondary channel acquisition

3.9. we can see that the search approach has higher secondary channel acquisition rate than our algorithm does. since the search approach does not consider channel reuse; that is, a cell just randomly borrows a channel from its neighbors. In our algorithm, a cell only borrows a channel from the "richest" interference neighbors, which reduces the chance that the lender might soon use up its primary channels and have to acquire a secondary channel. Also, keeping the borrowed channels reduces the number of channel borrowing. Note that acquiring an available-borrowed channel does not counted as a secondary channel acquisition. since in this case, the borrower does not need to contact with its interference neighbors.

Under non-uniform traffic distribution, only some cells are in the hot state, and most of the borrowers are hot cells (cells in the hot state). In our approach, when a cell finishes using the borrowed channel, it keeps the channel. Then, free channels are transferred to these hot cells, and hence, new communication sessions in the hot cells can be supported without borrowing channels again. Under uniform traffic
distribution. When the traffic load is high, most cells run out of channel; when the traffic load is low, most of them have free channels. Thus, the advantage of keeping channel under uniform traffic distribution is not that significant compared to that under non-uniform traffic distribution. This explains why our approach has much lower secondary channel acquisition percentage than other approaches under non-uniform traffic distribution compared to uniform traffic distribution.

Under uniform traffic distribution, when the traffic load becomes very high: e.g., there are 850 call arrivals per hour per cell. It is more likely that the lenders have less than \( n \) available channels, and hence the borrowers cannot keep the borrowed channel. As a result, the secondary channel acquisition percentage in our approach increases much faster compared to other approaches at this point. Certainly, it is still lower than the secondary channel acquisition percentage in other approaches.

![Graph](image)

**Figure 3.10**: Comparisons of secondary channel acquisition delay
Acquisition Delay per Channel Transfer: As explained in Chapter 3.6.1, the acquisition delay of our fault-tolerant approach depends on the $T_{\text{max}}$ and $T_{\text{turn}}$. To simplify the experiment, we only compare our efficient approach with the search approach and the update approach. As shown in Figure 3.10, the search approach has the highest secondary channel acquisition delay since it needs to confirm every borrowed channel. Both our efficient approach and the update approach almost reduce the acquisition delay by half as compared to the search approach. In our efficient approach, a cell may select a reserved channel, which results in an extra round of confirm messages and doubles the acquisition delay. From Figure 3.10, the chance of selecting a reserved channel is less than 20%. If there is a time constraint, our efficient approach can be modified so that another channel is selected if a reserved channel is chosen. However, this may increase the call failure rate.
Since the primary channel acquisition delay is 0, and cells in our efficient approach borrow channels less frequently than the search approach and the update approach (see Figure 3.9), our efficient approach has the lowest average acquisition delay among these approaches (see Figure 3.11).

3.6.4 Simulation Results (With Failures)

In this chapter, we compare the performance of the fault-tolerant and non-fault-tolerant channel allocation algorithms under non-uniform traffic distribution.

![Graphs showing call failure rate under various conditions](image)

Figure 3.12: Call failure rate under MSS or communication link failures
Call Failure Rate Under MSS Failures or Network Partitioning: If an MSS fails, every call requests in that cell fails. Even though the call failure rate in the failed cell is 100%, other cells may still have a very low call failure rate. To reflect the performance of operational cells, the call failure rate is calculated without considering the failed cells when there are MSS failures. Note that a MSS failure does not necessarily mean that the MSS crashes. A failed MSS may still be able to support its MHs, but it cannot communicate with other MSSs, and then it cannot borrow (lend) channels from (to) other MSSs.

Figure 3.12 compares the call failure rate of the fault-tolerant and non-fault-tolerant algorithms under four conditions: no failure, one MSS failure, three MSS failures, and seven MSS failures. As shown in Figure 3.12, the call failure rate of the non-fault-tolerant channel allocation algorithm is significantly higher (about 100 times when there are 400 call arrivals per hour per cell) than that of the fault-tolerant channel allocation algorithm. This can be explained as follows. In the fault-tolerant channel allocation algorithm, a cell can still borrow channels even if it cannot communicate with some interference neighbors, but in the non-fault-tolerant channel allocation algorithm, a cell cannot borrow any channel from its neighbors when it cannot communicate with any of its interference neighbors. With $D_{min} = 3\sqrt{3}R$, the number of interference neighbors of a cell is 30, and the number of interference primary neighbors of a cell is 3 or 4. Then, a cell can only use its primary channels when it cannot communicate with any of its 30 interference neighbors. Thus, the non-fault-tolerant algorithm has a high failure rate under MSS failures. For the fault-tolerant algorithm, in the best case, a cell can still borrow channels even though
it cannot communicate with as many as \((30 - 3) = 27\) (i.e., \(27/30 = 90\%\)) of its interference neighbors. In the worst case, even though a cell cannot communicate with as many as \(\lfloor(30/4)\rfloor = 7\) (i.e., \(7/30 = 23\%\)) of its interference neighbors, it can still communicate with the remaining \(30 - 7 = 23\) cells, which includes all cells (at most 4) of an interference partition subset. If there are common available channels among cells in this interference partition subset, the cell can borrow these available channels.

From Figure 3.12, we can see that the call failure rate with \(MSS\) failures is higher than the call failure rate without \(MSS\) failures in the fault-tolerant channel allocation algorithm. This can be explained by the fact that the failed \(MSS\) may have borrowed some channels and these channels cannot be used by the lenders until the failed \(MSS\) is recovered. (The figure shows the call failure rate without considering recovery.)

The fault-tolerant channel allocation algorithm exhibits the useful property of \textit{graceful degradation}, which is highly desirable in fault-tolerant distributed systems. In a failure-free environment, the algorithm has low call failure rate. As \(MSS\) failures occur and increase, the number of available channels decreases and the call failure rate increases.

\textbf{Failure Rate Under Network Congestion:} Network congestion depends on a large number of parameters, e.g., the number of arriving messages, the queue length, the service rate, etc. Since an \(MSS\) has other functionalities besides handling channel borrowing, it is very difficult to model. To simplify the model, we assume that a network congestion only occurs in hot cells. It has an exponentially distributed time with mean time of 60s. We assume that the communication delay is 4 ms during a network congestion. The maximum tolerable delay of an inter-cell handoff is much less.
than that of a call request. We assume the maximum tolerable delay of an inter-cell handoff is 10 ms.

Since the non-fault-tolerant algorithm needs to wait for every interference neighbor, it needs more than $4 \times 4 = 16 ms$ to borrow a channel from its neighbors, which is longer than the maximum tolerable delay 10 ms. Thus, the handoff requests are dropped during network congestion. However, in our algorithm, a cell can still borrow a channel from other neighbors which does not suffer from network congestion. In this case, it needs only $2 \times 4 = 8 ms < 10 ms$ to borrow a channel. This explains why the fault-tolerant algorithm has lower handoff dropping rate than the non-fault-tolerant algorithm does as shown in Figure 3.13.

From Figure 3.13, we can also see that the fault-tolerant channel allocation algorithm under network congestion has very higher dropping rate compared to the dropping rate without network congestion. This can be explained as follows. Network congestion occurs when a cell is in the hot state, which is reflected by low
inter-handoff rate. Suppose $C_i$ is suffering from network congestion. It is more likely that most of $C_i$'s neighbors are still in the normal state, which has high inter-handoff rate. In other words, there are many more $MHS$ coming from $C_i$'s neighbors than those leaving $C_i$. Since $C_i$ is experiencing network congestion, any communication with outside cells has long delay, which results in high dropping rate. Certainly, it is still much lower than the dropping rate in the non-fault-tolerant algorithm. Note that cells in both the fault-tolerant and non-fault-tolerant algorithms keep the borrowed channels for some time. This reduces the dropping rate when cells are in the hot state, but when a cell changes from the normal state to the hot state, the dropping rate is still high. Based on this reasoning, if a cell begins to borrow channels when the number of its available channels is below some threshold (a system tuning factor) instead of waiting for running out of channel, the dropping rate of our algorithm may be further reduced. If a different model (i.e., network congestion occurs randomly) is used, the dropping rate of the fault-tolerant channel allocation algorithm will be much lower, since the congested cell may not be in the hot state. However, the dropping rate of the non-fault-tolerant channel allocation algorithm will not change since the borrower needs to communicate with its congested interference neighbors even though they are in the hot states.

### 3.7 Distinctive Properties of Channel Allocation

In this chapter, we compare the problem of channel allocation to other well studied problems such as the mutual exclusion problem, the k-mutual exclusion problem, the quorum-based mutual exclusion problem, and the drinking philosopher problem.
**Channel Allocation vs. Mutual Exclusion:** In the context of a cell and its neighbors, the use of a particular channel to support a communication session is equivalent to a critical section execution by the cell in which the channel is being used. Several neighboring cells may be concurrently trying to choose channels to support sessions in their region. This can lead to conflicts because the number of communication channels is limited. The resolution of such conflicts is similar to the mutual exclusion problem [44, 59].

However, the channel allocation problem is more general than the mutual exclusion problem. Firstly, a cell may be supporting multiple communication sessions from different mobile hosts, in its region, each session using a different communication channel. This is equivalent to a cell being in multiple, distinct critical sections concurrently. Secondly, existing mutual exclusion algorithms for distributed systems [44, 59, 61] assume that a node specifies the identity of the resource it wants to access in a critical section. Depending on the availability of that resource, appropriate decisions can be made. However, in distributed channel allocation, a cell asks for any channel as long as there is no co-channel interference. Due to the non-specificity of the request and because neighboring mobile service stations make channel allocation decisions independently based on locally available information, the decision process becomes more difficult.

Moreover, existing distributed mutual exclusion algorithms do not impose any upper bound on the time from the instant a node issues a request for the resource to the instant the node is granted that resource. These algorithms are not suitable for the channel allocation problem that requires the decisions to be made quickly, in real time. So, a conservative approach that makes the channel allocation decisions quickly
needs to be adopted. Such an approach may drop calls/communication requests that a more general but time consuming approach would have supported. This is a trade-off that has to be accepted.

**Channel Allocation vs. k-Mutual Exclusion:** In k-mutual exclusion [23, 58], a fixed $k \geq 1$ nodes are allowed to simultaneously execute in the critical section. k-mutual exclusion is similar to channel allocation, since we can imagine that the critical section consists of $k$ available channels. However, there is a distinct difference between them: in channel allocation, the acquisition of one available channel may cause interference with other cells. The difference makes token-based k-mutual exclusion algorithms [23] unsuitable, since even though a MSS catches a channel (token), it can not use the channel unless it makes sure that it does not cause channel interference to its neighbors. Moreover, token-based k-mutual exclusion algorithms may have long delay.

Another major difference of Channel allocation and k-Mutual exclusion is due to *channel reuse*. In the channel allocation problem, as long as two cells are far away from the minimum reuse distance, they can use the same channel. However, in the k-mutual exclusion problem, as long as $k$ nodes are executing in the critical section, other nodes must wait no matter how far away they are.

**Channel Allocation vs. Quorum Based ME:** To obtain mutually exclusive access to a resource in the network, a node, say $P_i$, is required to receive permissions from a quorum of $P_i$ in the system. If all nodes in the quorum of $P_i$ grant permissions to $P_i$, $P_i$ is allowed to access the resource. Fault tolerance can be obtained by using fault-tolerant quorums [2, 15, 42, 57] which are obtained by adding redundancy to quorums. However, in channel allocation, since the requirement of channel reuse, it
may not be possible to construct fault-tolerant quorums. Also, there is an upper bound on the time required to get a channel. Thus, a totally different approach must be used (See our approach in 3.3).

In quorum-based mutual exclusion, a node only sends requests to nodes in its quorum. When a quorum is small, the message complexity is small. In our fault-tolerant approach, requests are sent to MSSs (interference neighbors) in many quorums, since a cell does not maintain the information of available channels. As a result, our fault-tolerant approach can achieve much higher resiliency; i.e., as much as 90%. Also, quorum-based mutual exclusion does not consider channel reuse: i.e., only one node can access the critical section at one time.

**Channel Allocation vs. Drinking Philosopher:** The Drinking Philosopher problem can be described as follows. A philosopher is in one of three states: thinking, thirsty, and drinking. A thirsty philosopher needs a nonempty set of bottles that he wishes to drink from. He may need different sets of bottles for different drinking sessions. On holding all needed bottles, a thinking philosopher starts drinking. A thirsty philosopher remains thirsty until he gets all bottles that he needs to drink. On entering the drinking state, a philosopher remains in that state for a finite period after which he becomes thinking.

In the drinking philosopher problem, a philosopher needs a nonempty set of bottles that he wishes to drink from. In other words, a philosopher specifies the identities of the resources it wants to access in a critical section. Depending on the availability of those resources, appropriate decisions can be made. However, in channel allocation, a cell asks for any channel as long as there is no co-channel interference. Due to the non-specificity of the request and because neighboring cells make channel allocation
decisions independently based on locally available information, the decision process becomes more difficult.

Moreover, existing drinking philosopher solutions do not impose any upper bound on the time from the instant a philosopher issues a request for the resource to the instant the philosopher is granted that resource. These algorithms are not suitable for the channel allocation problem that requires the decisions to be made quickly in real-time. So, a conservative approach that make the channel allocation decisions quickly needs to be adopted. Such an approach may drop calls that a more general but time consuming approach would have supported. This is a trade-off that has to be accepted. Also, as discussed before, channel reuse cannot be modeled in the drinking philosopher problem.
CHAPTER 4

Coordinated Checkpointing: Proof of Impossibility

Much of the previous work [38, 39] in coordinated checkpointing has focused on minimizing the number of synchronization messages and the number of checkpoints. However, these algorithms (called blocking algorithms) force all relevant processes in the system to block their computations during checkpointing. Checkpointing includes the time to trace the dependency tree and to save the states of processes on stable storage, which may be long. Therefore, blocking algorithms may dramatically reduce the performance of these systems [28].

Recently, nonblocking algorithms [28, 41] have received considerable attention. In these algorithms, processes need not block during checkpointing by using a checkpointing sequence number to identify inconsistent messages. However, these algorithms assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms, such as poor reliability, bottlenecks, etc. Moreover, these algorithms require all processes in the system to take checkpoints during checkpointing, even though many of the checkpoints may not be necessary.

The Prakash-Singhal algorithm [54] is the first algorithm to combine these two approaches. More specifically, it forces only a minimum number of processes to take
checkpoints and does not block the underlying computation during checkpointing. However, we found some problems in their algorithm (see Appendix A). In this chapter, we prove a more general result: "there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints." Before presenting the proof, we explain how to reduce the number of checkpoints and how to make a checkpointing algorithm non-blocking.

4.1 Minimizing Dependency Information and the Number of Checkpoints

Most of the existing coordinated checkpointing algorithms [24, 39, 45] rely on the two-phase protocol and save two kinds of checkpoints on stable storage: tentative and permanent. In the first phase, the initiator takes a tentative checkpoint and forces all relevant processes to take tentative checkpoints. Each process informs the initiator whether it succeeded in taking a tentative checkpoint and blocks until released by the initiator in phase two. A process may refuse to take a checkpoint depending on its underlying computation. After the initiator has received positive acknowledgments from all relevant processes, the algorithm enters the second phase. If the initiator learns that all processes have successfully taken tentative checkpoints, it asks them to make their tentative checkpoints permanent; otherwise, it asks them to discard their tentative checkpoints. A process, on receiving the message from the initiator, acts accordingly.

To reduce the number of checkpoints, a process keeps track of the process dependency, and hence, the initiator only forces dependent processes to take checkpoints. For any process, after it takes a checkpoint, it recursively forces all dependent processes to take checkpoints. The Koo-Toueg algorithm [39] uses this scheme, and it
has been proved [39] that this algorithm forces only a minimum number of processes
to take checkpoints.

In order to reduce the overhead of recording the dependency relationship among
processes, we use a similar approach as the Prakash-Singhal algorithm [34], where
each process $P_i$ maintains a boolean vector $R_i$, which has $n$ bits. $R_i[j] = 1$ represents
that $P_i$ depends on $P_j$. At $P_i$, the vector is initialized to 0 except $P_i[i]$.

When process $P_i$ sends a computation message $m$ to $P_j$, it appends $R_i$ to $m$.
After receiving $m$, $P_j$ includes the dependencies indicated in $R_i$ into its own $R_j$ as
follows: $R_j[k] = R_j[k] \lor m.R[k]$, where $1 \leq k \leq n$. $\lor$ is the bitwise inclusive OR
operator. Thus, if a sender $P_i$ of a message depends on a process $P_k$ before sending
the computation message, the receiver $P_j$ also depends on $P_k$ through transitivity. In
Figure 4.1, $P_2$ depends on $P_1$ because of $m1$. When $P_3$ receives $m2$, $P_3$ (transitively)
depends on $P1$.

The dependency information can be used to reduce the number of processes that
must participate in the checkpointing process and the number of messages required to
synchronize the checkpointing activity. In Figure 4.1, the vertical line $S_1$ represents
the global consistent checkpoints at the beginning of the computation. Later, when
$P_3$ initiates a new checkpointing process, it only sends checkpoint requests to $P_1$ and
$P_2$. As a result, only $P_1$, $P_2$, and $P_3$ take new checkpoints. $P_4$ and $P_5$ continues their
computation without taking new checkpoints.

4.2 Basic Ideas Behind Non-blocking Algorithms

A non-blocking checkpointing algorithm does not require any process to suspend
its underlying computation. When processes do not suspend their computations, it is
possible for a process to receive a computation message from another process which is already running in a new checkpoint interval. If this situation is not properly dealt with, it may result in an inconsistency. For example, in Figure 4.2. \( P_2 \) initiates a checkpointing process. After sending checkpoint requests to \( P_1 \) and \( P_3 \), \( P_2 \) continues its computation. \( P_1 \) receives the checkpoint request and takes a new checkpoint, then it sends \( m_1 \) to \( P_3 \). Suppose \( P_3 \) processes \( m_1 \) before it receives the checkpoint request from \( P_2 \). When \( P_3 \) receives the checkpoint request from \( P_2 \), it takes a checkpoint (see Figure 4.2). In this case, \( m_1 \) becomes an orphan.

Most non-blocking algorithms [28, 60] use a checkpoint sequence number (csn) to avoid inconsistencies. In these algorithms, a process is forced to take a checkpoint if it receives a computation message whose csn is greater than its local csn. For example, in Figure 4.2. \( P_1 \) increases its csn after it takes a checkpoint and appends the new
Figure 4.2: Inconsistent checkpoints

csn to m1. When P3 receives m1, it takes a checkpoint before processing m1 since the csn appended to m1 is larger than its local csn.

This scheme works only when every process in the computation can receive each checkpoint request and then increases its own csn. Since the Prakash-Singhal algorithm [54] forces only a subset of processes to take checkpoints, the csn of some processes may be out-of-date and may not be able to avoid inconsistencies. The Prakash-Singhal algorithm attempts to solve this problem by having each process maintain an array to save the csn, where csn[i][j] is the expected csn of Pi. Note that Pi's csn[i][j] may be different from Pj's csn[j][i] if there has been no communication between Pi and Pj for several checkpoint intervals. By using csn and the initiator identification number, they claim that their non-blocking algorithm can avoid inconsistencies and minimize the number of checkpoints during checkpointing. However, we found two problems in their algorithm (see Appendix A). Next, we prove a more general result: “there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints.”
4.3 Proof of Impossibility

Before presenting the proof, we define a new concept called "z-dependency", which is more general than causal dependency and can be used to model coordinated checkpointing.

4.3.1 Causal Dependency vs. z-dependency

**Definition 5** If a process \( P_p \) sends a message to a process \( P_q \) during its \( i \)th checkpoint interval and \( P_q \) receives the message during its \( j \)th checkpoint interval, then \( P_q \) z-depends on \( P_p \) during \( P_p \)'s \( i \)th checkpoint interval and \( P_q \)'s \( j \)th checkpoint interval, denoted as \( P_p \prec_j^i P_q \).

**Definition 6** If \( P_p \prec_j^i P_q \), and \( P_q \prec_k^l P_r \), then \( P_r \) transitively z-depends on \( P_p \) during \( P_r \)'s \( k \)th checkpoint interval and \( P_p \)'s \( i \)th checkpoint interval. denoted as \( P_p \prec_k^{i+l} P_r \) (we simply call it "\( P_r \) transitively z-depends on \( P_p \)" if there is no confusion).

**Proposition 1**

\[
P_p \prec_j^i P_q \implies P_p \prec_j^i P_q \\
P_p \prec_j^i P_q \land P_q \prec_k^l P_r \implies P_p \prec_k^{i+l} P_r
\]

The definition of "z-dependency" here is different from the concept of "causal dependency" used in the literature. We illustrate the difference between causal dependency and z-dependency in Figure 4.3. Since \( P_2 \) sends \( m1 \) before it receives \( m2 \), there is no causal dependency between \( P_1 \) and \( P_3 \) due to these messages. However, these messages do establish a z-dependency between \( P_3 \) and \( P_1 \): \( P_3 \prec_j^{k-1} P_2 \land P_2 \prec_j^{l-1} P_1 \implies P_3 \prec_{i-1}^{k-1} P_1 \).

**Definition 7** A min-process checkpointing algorithm is an algorithm satisfying the following condition: when a process \( P_p \) initiates a new checkpointing process and
Figure 4.3: The difference between causal dependency and \( z \)-dependency

takes a checkpoint \( C_{p,i} \). A process \( P_q \) takes a checkpoint \( C_{q,j} \) associated with \( C_{p,i} \) if and only if \( P_q \stackrel{\text{causal}}{\preceq}^{j-1}_{i-1} P_p \).

In coordinated checkpointing, to avoid an inconsistency, the initiator forces all dependent processes to take checkpoints. For any process, after it takes a checkpoint, it recursively forces all dependent processes to take checkpoints. The Koo-Toueg algorithm \cite{39} uses this scheme, and it has been proved \cite{39} that this algorithm forces only a minimum number of processes to take checkpoints. In the following, we prove that the Koo-Toueg algorithm is a min-process algorithm and a min-process algorithm forces only a minimum number of processes to take checkpoints. To simplify the proof, we use \( \text{"}P_p \mapsto^i P_q\text{"} \) to represent the following: \( P_q \) causally depends on \( P_p \) when \( P_q \) is in the \( i \)th checkpoint interval and \( P_p \) is in the \( j \)th checkpoint interval.

\textbf{Proposition 2} \hspace{1cm} P_p \mapsto^i P_q \implies P_p \stackrel{\text{causal}}{\preceq}^j_i P_q

\hspace{1cm} P_p \preceq^i_j P_q \implies P_p \mapsto^j_i P_q

82
4.3.2 Proof of Impossibility

Based on the concept of “z-dependency”, we prove the impossibility result in this chapter.

Lemma 2 An algorithm forces only a minimum number of processes to take checkpoints if and only if it is a min-process algorithm.

**Proof.** It has been proved [39] that the Koo-Toueg algorithm forces only a minimum number of processes to take checkpoints; thus, we only need to prove the following: in [39], when a process $P_p$ initiates a new checkpointing process and takes a checkpoint $C_{p,i}$, a process $P_q$ takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$ if and only if $P_q \prec_{i-1} P_p$.

**Necessity:** In [39], when a process $P_p$ initiates a new checkpoint $C_{p,i}$, it recursively asks all dependent processes to take checkpoints. For example, $P_p$ asks $P_{k_m}$ to take a checkpoint. $P_{k_m}$ asks $P_{k_{m-1}}$ to take a checkpoint, and so on. If a process $P_q$ takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$, there must be a sequence:

$$P_q \vdash_{j-1} P_{k_1} \wedge P_{k_1} \vdash_{s_{k_1}} P_{k_2} \wedge \cdots \wedge P_{k_m} \vdash_{s_{k_m - 1}} P_{k_{m-1}} \wedge P_{k_{m-1}} \vdash_{s_{k_{m-1}}} P_{k_{m-2}} \wedge \cdots \wedge P_{k_1} \vdash_{s_{k_1}} P_{k_0} \vdash_{i-1} P_p$$

$$\Rightarrow P_q \prec_{s_{k_1}} P_{k_1} \wedge P_{k_1} \prec_{s_{k_2}} P_{k_2} \wedge \cdots \wedge P_{k_{m-1}} \prec_{s_{k_{m-1}}} P_{k_{m-2}} \wedge P_{k_{m-2}} \prec_{s_{k_m}} P_{k_{m-1}} \wedge P_{k_{m-1}} \prec_{i-1} P_p$$

$$\Rightarrow P_q \prec_{i-1} P_p$$

**OR**

$$P_q \vdash_{i-1} P_p$$

$$\Rightarrow P_q \prec_{i-1} P_p$$

$$\Rightarrow P_q \prec_{i-1} P_p$$

**Sufficiency:** If $P_q \prec_{i-1} P_p$ when $P_p$ initiates a new checkpoint $C_{p,i}$, $P_q$ takes a checkpoint $C_{q,j}$ associated with $C_{p,i}$; otherwise, if $P_q \vdash_{i-1} P_p$, there must be a sequence:
Then, when \( P_p \) initiates a new checkpoint \( C_{p,i} \), \( P_p \) asks \( P_{k_1} \) to take a checkpoint, \( P_{k_1} \) asks \( P_{k_2} \) to take a checkpoint, and so on. In the end, \( P_{k_m} \) asks \( P_q \) to take a checkpoint. Then, \( P_q \) takes a checkpoint \( C_{q,j} \) associated with \( C_{p,i} \). □

**Definition 8** A non-blocking min-process algorithm is a min-process checkpointing algorithm which does not block the underlying computation during checkpointing.

**Lemma 3** In a non-blocking min-process algorithm. assume \( P_p \) initiates a new checkpointing process and takes a checkpoint \( C_{p,i} \). If a process \( P_r \) sends a message \( m \) to \( P_q \) after it takes a new checkpoint associated with \( C_{p,i} \), then \( P_q \) takes a checkpoint \( C_{q,j} \) before processing \( m \) if and only if \( P_q \prec_{i-1} P_p \).

**Proof.** From the definition of min-process. \( P_q \) takes a checkpoint \( C_{q,j} \) if and only if \( P_q \prec_{i-1} P_p \). Thus. we only need to show that \( P_q \) takes \( C_{q,j} \) before processing \( m \). It is easy to see. if \( P_q \) takes \( C_{q,j} \) after processing \( m \). \( m \) becomes an orphan (as in Figure 4.2). □

From Lemma 3. in a non-blocking min-process algorithm. when a process receives a message \( m \). it must know if the initiator of a new checkpointing process transitively \( z \)-depends on it.

**Lemma 4** In a non-blocking min-process algorithm. there is not enough information at the receiver of a message to decide whether the initiator of a new checkpointing process transitively \( z \)-depends on the receiver.

**Proof.** The proof is by construction (using a counter-example). In Figure 4.4. assume messages \( m_6 \) and \( m_7 \) do not exist. \( P_1 \) initiates a checkpointing process. When \( P_4 \)
receives $m_4$. There is a z-dependency as follows.

$$P_2 \preceq_0 P_4 \land P_4 \preceq_0 P_1 \implies P_2 \preceq_0 P_1.$$  

However, $P_2$ does not know this when it receives $m_5$. There are two possible approaches for $P_2$ to get the z-dependency information:

**Approach 1** (Tracing the in-coming messages): In this approach, $P_2$ gets the new z-dependency information from $P_1$. Then, $P_1$ has to know the z-dependency information before it sends $m_5$ and appends the z-dependency information to $m_5$. In Figure 4.4, $P_1$ cannot get the new z-dependency information ($P_2 \preceq_0 P_1$) unless $P_4$ notifies $P_1$ of the new z-dependency information when $P_4$ receives $m_4$. There are two ways for $P_4$ to notify $P_1$ of the new z-dependency information: first is to broadcast the z-dependency information (not illustrated in the figure); the other is to send the z-dependency information by an extra message $m_6$ to $P_3$, which in turn notifies $P_1$ by $m_7$. Both of them dramatically increase message overhead. Since the algorithm does not block the underlying computation, it is possible that $P_1$ receives $m_7$ after it sends out $m_5$ (as shown in the figure). Thus, $P_2$ still cannot get the z-dependency information when it receives $m_5$.

**Approach 2** (Tracing the out-going messages): In this approach, since $P_2$ sends message $m_3$ to $P_3$, $P_2$ hopes to get the new z-dependency information from $P_3$. Then, $P_3$ has to know the new z-dependency information and it would like to send an extra message (not shown in the figure) to notify $P_2$. Similarly, $P_5$ needs to get the new z-dependency information from $P_4$, which comes from $P_3$, and finally from $P_1$. Certainly, this requires much more extra messages than Approach 1. Similar to Approach 1, $P_2$ still cannot get the z-dependency information in time since the computation is in progress.
**Theorem 6** No non-blocking min-process algorithm exists.

**Proof.** From Lemma 3. in a non-blocking min-process algorithm, a receiver has to know if the initiator of a new checkpointing process transitively z-depends on the receiver, which is impossible from Lemma 4. Therefore, no non-blocking min-process algorithm exists. □

**Corollary 1** there does not exist a non-blocking algorithm that forces only a minimum number of processes to take their checkpoints.

**Proof.** The proof directly follows from Lemma 2 and Theorem 6. □

**4.3.3 Related Work**

Netzer and Xu [50] introduced the concept of “zigzag” paths to define the necessary and sufficient conditions for a set of local checkpoints to lie on a consistent global
checkpoint. Our definition of “z-dependency” captures the essence of zigzag paths. If an initiator forces all its transitively z-dependent processes to take checkpoints, the resulting checkpoints are consistent, and no zigzag path exists among them. If the resulting checkpoints are consistent, there is no zigzag path among them, and all processes on which the initiator transitively z-depends have taken checkpoints. However, there is a distinct difference between a zigzag path and z-dependency. A zigzag path is used to evaluate whether the existing checkpoints are consistent; thus, it is mainly used to find a consistent global checkpoint in an uncoordinated checkpointing algorithm. It has almost no use in a coordinated checkpointing algorithm since a consistent global checkpoint is guaranteed by the synchronization messages.

The z-dependency is proposed for coordinated checkpointing and it reflects the whole synchronization process of coordinated checkpointing, e.g., in the proof of Lemma 1. z-dependency is used to model the checkpointing process. Based on z-dependency, we found and proved the impossibility result. It is impossible to prove the result only based on zigzag paths.

Wang [69, 70] considered the problem of constructing the maximum and the minimum consistent global checkpoints that contain a target set of checkpoints. In his work, a graph called rollback dependency graph (or “R-graph”) can be used to quickly find zigzag paths. Based on R-graph, Wang developed efficient algorithms to calculate the maximum and the minimum consistent global checkpoints for both general non-deterministic executions and piecewise deterministic executions. Manivannan et al. [48] generalized this problem. They proposed an algorithm to enumerate all such consistent global checkpoints that contain a target set of checkpoints and showed how the minimum and the maximum checkpoints are special cases. Their work [48] is based on
a concept called "Z-cone", which is a generalization of zigzag paths. All these works [48, 69, 70] are based on zigzag paths. Similar to [50], these works [48, 69, 70] are not useful in coordinated checkpointing since checkpoints are guaranteed to be consistent in coordinated checkpointing. Thus, the notions of zig-zag paths and Z-cone help find consistent global checkpoints in uncoordinated checkpointing, while the notion of z-dependency helps understand the nature of coordinated checkpointing.
CHAPTER 5

Low-Cost Checkpointing Algorithms for Mobile Computing Systems

From Theorem 6, no non-blocking min-process algorithm exists. This implies that there are three directions in designing efficient coordinated checkpointing algorithms. On one extreme, we can propose a non-blocking algorithm, which relaxes the min-process condition while minimizing the number of tentative checkpoints. The other extreme is to relax the non-blocking condition while keeping the min-process property; that is, we can design a min-process algorithm which tries to minimize the blocking time. Between these two extremes, we can also design blocking non-min-process algorithms that significantly reduce the blocking time as well as the number of checkpoints. In this Chapter, we propose a min-process algorithm and a non-blocking algorithm for mobile computing systems. Correctness proofs and performance evaluation are also provided.

5.1 Handling Mobility and Disconnections

Changes in the location of an MH complicate the routing of messages. Messages sent by an MH to another MH may have to be rerouted because the destination MH disconnected from the old MSS and is now connected to a new MSS. Many
routing protocols for the network layer have been proposed [33, 66] to handle MH mobility.

A MH becomes disconnected whenever it moves outside the range of all the cells, or whenever the user turns off the network interface. We consider two different types of disconnection. An voluntary disconnection allows the protocol to exchange a few messages with the stable storage just before the MH become isolated. Examples of this type of disconnection include situations in which the user calls a logout command, or the communication layers inform the protocol when the MH is about to move outside the range of the cells (when the wireless signal becomes weaker). An involuntary disconnection corresponds to the opposite case, in which the protocol is not able to exchange any messages with stable storage. This may happen due to battery failure, processor failure, or network failure. Involuntary disconnections are different from voluntary disconnection and are discussed in Chapter 5.3.7.

An MH may get disconnected from the network for an arbitrary period of time. At the application level, the checkpointing algorithm may generate a request for the disconnected MH to take a checkpoint. Delaying a response to such a request until the MH reconnects at some MSS, may significantly increase the completion time of the checkpointing algorithm. So, we propose the following solution to deal with disconnections.

Note that only local events can take place at an MH during the disconnect interval. No message send or receive event occurs during this interval. Hence, no new dependencies with respect to other processes are created during this interval. The dependency relation of an MH with the rest of the system, as reflected by its local
checkpoint, is the same no matter when the local checkpoint is taken during the dis­
connect interval. Thus, we require an $MH$ to take a checkpoint before it disconnects.

**Handling Disconnection and Reconnection:** Suppose a mobile host $MH_i$ wants
to disconnect from its local $MSS_p$. $MH_i$ takes a local checkpoint and transfers its
local checkpoint to $MSS_p$ as $disconnect\_checkpoint_i$. $MH_i$ also keeps the local check­
point in its local disk so that it can recover transient failures locally without asking for
a global recovery. If $MH_i$ is asked to take a checkpoint during the disconnect inter­
val. $MSS_p$ converts $disconnect\_checkpoint_i$ into $MH_i$'s new checkpoint. and uses the
message dependency information of $MH_i$ to propagate the checkpoint request. $MH_i$
also sends $disconnect(sn)$ message to $MSS_p$ on the $MH$-to-$MSS$ channel supplying
the sequence number $sn$ of the last message received on the $MSS$-to-$MH$ channel.

On the receipt of $MH_i$'s $disconnect(sn)$. $MSS_p$ knows the last message that $MH_i$
received from it and buffers all computation messages received until the end of the
disconnect interval.

Later. suppose $MH_i$ reconnects at an $MSS$. say $MSS_q$. If $MH_i$ knows the ident­
ty of its last $MSS$. say $MSS_p$, it sends a $reconnect(MH_i, MSS_q)$ message to $MSS_p$
through $MSS_q$. If $MH_i$ lost the identity of its last $MSS$ for some reason. then $MH_i$'s
reconnect request is broadcasted over the network. On receiving the reconnect re­
quest. $MSS_p$ transfers all the support information (the checkpoint. dependency vec­
tor. buffered messages. etc.) of $MH_i$ to $MSS_q$. and deletes all information related
to the disconnection. Then. $MSS_q$ forwards all the support information to $MH_i$.

When the data sent by $MSS_p$ arrives at $MH_i$. $MH_i$ processes the buffered messages.
If $MSS_p$ has taken a checkpoint for $MH_i$. $MH_i$ clears its message dependency in­
formation before processing the buffered messages. With this. the reconnect routine
terminates and the relocated mobile host $MH_i$ resumes normal communication with other $MH$s (or MSSs) in the system.

5.2 A Min-Process Checkpointing Algorithm

5.2.1 The Min-Process Checkpointing Algorithm

**Data Structures at MSSs:** In mobile computing systems, all communications to and from an $MH$ pass through its local MSS. Therefore, when an MSS receives an application message to be forwarded to a local $MH$, it first updates the dependency information that it maintains for the $MH$ and then forwards it to the $MH$. The dependency information is recorded by boolean vector $R_i$ for process $P_i$. The vector has $n$ bits, representing $n$ processes. When $R_i[j]$ is set to 1, it represents that $P_i$ depends on $P_j$. For every $P_i$, $R_i$ is initialized to 0 except $R_i[i]$, which is initialized to 1. When a process $P_i$ running on an $MH$, say $MH_p$, receives a message from a process $P_j$, $MH_p$'s local MSS sets $R_i[j]$ to 1.

**The First Phase of The Algorithm:** When a process running on an $MH$ initiates a checkpointing, it sends a checkpoint request to its local MSS, which will be the proxy MSS (If the initiator runs on an MSS, then the MSS is the proxy MSS). The proxy MSS sends $R_{request}$ messages to all MSSs in the system (denoted by $S_{mss}$) to ask for dependency vectors. In response to the $R_{request}$, each MSS returns the dependency vectors that it maintains for processes running on the $MH$s in its cell. Having received all the dependency vectors, the proxy MSS constructs an $N \times N$ dependency matrix with one row per process, represented by the dependency vector of the process. Based on the dependency matrix, the proxy MSS can locally calculate all the processes on which the initiator transitively depends. This is essentially the
same as finding the transitive closure of the initiator in the dependency tree which is constructed using the dependency vectors. Then, it can be transformed to a matrix multiplication [16]. After the proxy \( MSS \) finds all the processes that need to take checkpoints, it adds them to the set \( S_{forced} \) and broadcasts \( S_{forced} \) to all \( MSS \)s. which are waiting for the result. When an \( MSS \) receives \( S_{forced} \), it checks if any processes in \( S_{forced} \) are also in its cell. If so, the \( MSS \) sends checkpoint \emph{request} messages to them. Any process receiving a checkpoint \emph{request} takes a checkpoint and sends a \emph{response} to its local \( MSS \). After an \( MSS \) has received all \emph{response} messages from the processes to which it sent checkpoint \emph{request} messages, it sends a \emph{response} to the proxy \( MSS \). Figure 5.1 shows the first phase of our min-process checkpointing algorithm.

\textbf{The Second Phase of The Algorithm:} After the proxy \( MSS \) has received a \emph{response} from every \( MSS \), the algorithm enters the second phase. If the proxy \( MSS \) learns that all processes have successfully taken tentative checkpoints, it asks them to make their tentative checkpoints permanent. Otherwise, it asks them to discard their tentative checkpoints. A process, on receiving the message from the proxy \( MSS \), acts accordingly (Techniques to reduce discarded checkpoints can be found in [53]).

\textbf{An Example:} In Figure 5.2, \( D_i \) denotes the dependency vector of process \( P_i \). When \( P_i \) initiates a checkpointing, the proxy \( MSS \) constructs the dependency matrix \( D \) and calculates \( D_1 \times D = (1 \ 1 \ 1 \ 0 \ 0) \). Since \( (1 \ 1 \ 1 \ 0 \ 0) \times D = (1 \ 1 \ 1 \ 0 \ 0) \), based on the procedure \emph{Calculate}, \( S_{forced} = \{ P_1, P_2, P_3 \} \). Thus, \( P_1 \) asks \( P_2 \) and \( P_3 \) to take checkpoints.
Algorithm executed at the proxy MSS:
for \( \forall i(MSS_i \in S_{mss}) \), send a \( R \).request message to \( MSS_i \);
Upon receiving all \( R \) vectors from each \( MSS_i \) do
  construct matrix \( D \); calculate\( (D) \):

Algorithm executed at an MSS, say \( MSS_k \):
Upon receiving \( R \).request from the proxy MSS:
  for \( \forall i(\text{Location}(P_i) \in \text{Cell}_k) \), send \( R_i \) to the proxy MSS:
Upon receiving \( S_{forced} \) from the proxy MSS:
  for \( \forall i(\text{Location}(P_i) \in \text{Cell}_k \land P_i \in S_{forced}) \), send a
  checkpoint request to \( P_i \);
  continue its computation:
Upon receiving response messages from all processes to which it sends request messages:
  send a response to the proxy MSS:

Algorithm executed at any process \( P_i \):
Upon receiving a request from \( MSS_j \):
  take a checkpoint. send a response to \( MSS_j \):

Calculate \( (D : N \times N) \)
/* \( D_i \) denotes the dependency vector of process \( P_i \). Assume
\( P_j \) is the initiator. */
\( A = D_j \); \( D_j = D_j \times D \)
While \( A \neq D_j \) do \{ \( A = D_j \); \( D_j = D_j \times D \) \}
\( S_{forced} = \phi \);
for \( \forall i(D_j[i] = 1) \), \( S_{forced} = S_{forced} \cup P_i \);

Figure 5.1: The first phase of our min-process checkpointing algorithm
5.2.2 Proof of Correctness

**Lemma 5** A process takes a checkpoint if and only if the initiator transitively depends on it.

*Proof.* In the proposed algorithm, the proxy MSS uses the procedure `calculate` to find out the transitive closure of the initiator. During the execution of `calculate`, no new dependency relation is formed since MSSs are blocked. Therefore, a process $P_i$ belongs to $S_{forced}$ if and only if the initiator transitively depends on $P_i$. Since an MSS only sends a checkpoint request to a process in $S_{forced}$, a process $P_i$ takes a checkpoint only if the initiator transitively depends on $P_i$. Thus, we only need to show that a process receives a checkpoint and takes a checkpoint if the initiator transitively depends on it.
If $P_i$ is running on a MSSp, a checkpoint request is sent to $P_i$ when $S_{forced}$ reaches MSSp. If $P_i$ is running on an MH, say $MH_j$, which is in MSSp's cell, then there are three possibilities when $S_{forced}$ reaches MSSp:

**Case 1:** $MH_j$ is still connected to MSSp: the request is forwarded to $MH_j$, then to $P_i$.

**Case 2:** $MH_j$ is disconnected from the network: MSSp takes a checkpoint on the behalf of $P_i$ by converting disconnect.checkpoint, into $P_i$'s new checkpoint.

**Case 3:** $MH_j$ has moved to MSSq (handoff): MSSp forwards the request to MSSq, which forwards it to $MH_j$, then to $P_i$ by the underlying network.

Thus, if the initiator transitively depends on $P_i$, $P_i$ receives a checkpoint request and takes a checkpoint.

**Theorem 7** The algorithm creates a consistent global checkpoint.

**Proof.** The proof is by contradiction. Assume there is a pair of processes $P_p$ and $P_q$ such that at least one message $m$ has been sent from $P_q$ after $P_q$'s last checkpoint $C_{q,i}$ and has been received by $P_p$ before $P_p$'s last checkpoint $C_p$. We also assume $C_{p,i}$ is associated with the initiator $P_r$'s checkpoint $C_{r,k}$. Then, based on Lemma 3:

- $P_p$ takes a checkpoint $\Rightarrow P_p < k_{i-1} P_r$.
- $P_p$ receives $m$ from $P_q$ $\Rightarrow P_q < j_{i-1} P_p$.
- $P_q < j_{i-1} P_p \land P_p < k_{i-1} P_r \Rightarrow P_q < j_{k-1} P_r$.
- $P_q < j_{k-1} P_r \Rightarrow P_q$ takes a checkpoint.

Thus, the sending of $m$ is recorded at $P_q$. A contradiction. □

96
5.3 A Non-blocking Checkpointing Algorithm

In this chapter, we present a non-blocking checkpointing algorithm, which mini­
mizes the number of checkpoints saved on stable storage.

5.3.1 Basic Ideas of Mutable Checkpoints

Basic Scheme

A simple non-blocking scheme for checkpointing is as follows: when a process $P_i$ sends a message, it piggybacks the current value of $csn_i[i]$ ($csn$ is explained in Chapter 4.2). When a process $P_j$ receives a message $m$ from $P_i$, $P_j$ processes the message if $m.csn \leq csn_j[i]$; otherwise, $P_j$ takes a checkpoint, updates its $csn$ ($csn_j[i] = m.csn$), and then processes the message. This method may result in a large number of checkpoints. Moreover, it may lead to an avalanche effect, in which processes in the system recursively ask others to take checkpoints.

![Diagram of checkpointing](image)

Figure 5.3: An example of checkpointing
For example, in Figure 5.3, to initiate a checkpointing process, $P_2$ takes its own checkpoint and sends checkpoint requests to $P_1$, $P_3$ and $P_4$. When $P_2$'s request reaches $P_4$, $P_4$ takes a checkpoint. Then, $P_4$ sends message $m3$ to $P_3$. When $m3$ arrives at $P_3$, $P_3$ takes a checkpoint before processing it because $m3.csn > csn_3[4]$. For the same reason, $P_1$ takes a checkpoint before processing $m2$.

$P_0$ has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m1$ to $P_1$. $P_1$ takes checkpoint $C_{1,2}$ before processing $m1$ because $P_0$ has taken a checkpoint with checkpoint sequence number larger than $P_1$ expected. Then, $P_1$ requires $P_3$ to take another checkpoint (not shown in the figure) due to $m2$ and $P_3$ in turn asks $P_4$ to take another checkpoint (not shown in the figure) due to $m3$. If $P_4$ had received messages from other processes after it sent $m3$, then those processes would have been forced to take checkpoints. This chain may never end.

We reduce the number of checkpoints based on the following observation. In Figure 5.3, if $m4$ does not exist, it is not necessary for $P_1$ to take $C_{1,2}$ since checkpoint $C_{1,1}$ is consistent with the rest of checkpoints. Based on this observation, we get the following revised scheme.

When a process $P_j$ receives a message $m$ from $P_i$, $P_j$ only takes a checkpoint when $m.csn > csn_j[i]$ and $P_j$ has sent at least one message in the current checkpoint interval.

In Figure 5.3, if $m4$ does not exist, $C_{1,2}$ is not necessary according to the revised scheme. However, if $m4$ exists, the revised scheme still results in a large number of checkpoints and may result in an avalanche effect.

**Enhanced Scheme**
We now present the basic idea of our scheme that eliminates avalanche effects during checkpointing. From Figure 5.3, we make two observations:

**Observation 1:** It is not necessary to take checkpoint $C_{1,2}$ even though $m_4$ exists, since $P_1$ will not receive a checkpoint request associated with $C_{0,1}$. Note that $m_4$ will not become an orphan even though it does not take checkpoint $C_{1,2}$.

**Observation 2:** From Chapter 4.3.2, $P_1$ does not have enough information to know if it will receive a checkpoint request associated with $C_{0,1}$ when $P_1$ receives $m_1$.

These observations imply that $C_{1,2}$ is unnecessary but still unavoidable. Thus, there are two kinds of checkpoints in response to computation messages. In Figure 5.3, $C_{1,1}$ is different from $C_{1,2}$. $C_{1,1}$ is a checkpoint associated with initiator $P_2$, and $P_1$ will receive a checkpoint request for this checkpointing initiated by $P_2$. $C_{1,2}$ is a checkpoint associated with initiator $P_0$, but $P_1$ will not receive a checkpoint request for this checkpointing initiated by $P_0$ in the future. To avoid inconsistency, $P_1$ should keep $C_{1,1}$ when it receives $P_2$'s request. However, $P_1$ can discard $C_{1,2}$ after the checkpointing initiated by $P_0$ terminates ($C_{0,1}$ becomes permanent checkpoint) since at that time, $P_1$ is sure that it will not receive any checkpoint request associated with $P_0$'s initiation. Moreover, if $P_0$ has finished its checkpointing process before it sends $m_1$, $P_1$ does not need to take checkpoint $C_{1,2}$.

We introduce a new concept, called *mutable checkpoint*, to reflect the essence of checkpoints (like $C_{1,1}, C_{1,2}$) triggered by computation messages. A mutable checkpoint is neither a tentative checkpoint nor a permanent checkpoint, but it can be turned into a tentative checkpoint. When a process takes a mutable checkpoint, it does not send checkpoint requests to other processes and it does not need to save the checkpoint on stable storage. It can save the mutable checkpoint anywhere: e.g..
in the main memory or the local disk of MHSs. Suppose a process \( P_1 \) has taken a mutable checkpoint. When \( P_1 \) receives a checkpoint request, it transfers the mutable checkpoint to the stable storage and forces all dependent processes to take tentative checkpoints. In this way, \( P_1 \) turns its mutable checkpoint into a tentative checkpoint. If \( P_1 \) does not receive a checkpoint request after the checkpointing activity terminates (implementation details will be discussed in the next chapter), it discards the mutable checkpoint.

In Figure 5.3, when \( m_2 \) arrives at \( P_1 \), \( P_1 \) takes a mutable checkpoint \( C_{1,1} \) before processing it because \( m_2.csn > csn_{1}[3] \). \( C_{1,1} \) is turned into a tentative checkpoint when \( P_1 \) receives the checkpoint request sent by \( P_2 \). If \( P_0 \) has finished its checkpointing activity before it sends \( m_1 \), \( P_1 \) does not need to take a mutable checkpoint \( C_{1,2} \). Otherwise, \( P_1 \) takes a mutable checkpoint \( C_{1,2} \), which will be discarded when \( P_0 \)'s checkpointing terminates. Since \( C_{1,2} \) is a mutable checkpoint, it does not force \( P_3 \) to take a new checkpoint. Thus, our scheme avoids the avalanche effect and significantly reduces the checkpointing overhead. If there is no ambiguity, we simply refer to a tentative or permanent checkpoint as a checkpoint.

**Further reduction in the number of checkpoints**

In the above scheme, a process may receive unnecessary checkpoint requests and may take unnecessary checkpoints. As shown in Figure 5.4, \( P_2 \) initiates a checkpointing process by taking a checkpoint \( C_{2,1} \) and forces \( P_1 \) to take a checkpoint \( C_{1,1} \) (due to \( m_2 \)). Later, to initiate a checkpointing process, \( P_3 \) takes a checkpoint \( C_{3,1} \) and sends a request to \( P_2 \) due to \( m_1 \). When \( P_2 \) receives the request, it takes a checkpoint \( C_{2,2} \) and forces \( P_1 \) to take a checkpoint \( C_{1,2} \). However, \( C_{2,2} \) and \( C_{1,2} \) are not necessary since \( m_1 \) is not an orphan even though \( C_{1,2} \) and \( C_{2,2} \) do not exist.
These unnecessary checkpoints can be avoided by the following method. When a process $P_i$ sends a checkpoint request to $P_j$, it attaches $csn_i[j]$ to the request. On receiving the request, $P_j$ compares the attached $csn_i[j]$ ($req\_csn$) with its own $csn_j[j]$. If $csn_j[j] > req\_csn$ (i.e., $P_j$ has recorded the sending of the message which creates the dependency between $P_i$ and $P_j$), $P_j$ does not need to take a checkpoint; otherwise, it takes a checkpoint. In Figure 5.4, when $P_3$ sends a request to $P_2$, it attaches $csn_3[2] = 0$ to the request. When $P_2$ receives the request, $csn_2[2]$ has been increased to 1 due to $C_{2,1}$. Thus, $P_2$ ignores this request and does not take checkpoint $C_{2,2}$, and subsequently $P_2$ does not force $P_1$ to take checkpoint $C_{1,2}$.

5.3.2 Data Structures

The following data structures are used in our non-blocking algorithm:

$R_i$: an array of $n$ bits at process $P_i$. $R_i[j] = 1$ represents that $P_i$ receives a computation message from $P_j$ in the current checkpoint interval. Explained in Chapter 4.1.
\textit{csn}_i: an array of \( n \) checkpoint sequence numbers (\textit{csn}) at each process \( P_i \). \( \textit{csn}_i[j] \) represents the checkpoint sequence number of \( P_j \) that \( P_i \) knows. In other words, \( P_i \) expects to receive a message from \( P_j \) with checkpoint sequence number \( \textit{csn}_i[j] \).

\textit{weight}: a non-negative variable of type real with maximum value of 1. It is used to detect the termination of the checkpointing as in [32].

\textit{trigger}_i: a tuple \( (\textit{pid}, \textit{inum}) \) maintained by each process \( P_i \). \textit{pid} indicates the checkpoint initiator that triggered the latest checkpointing process. \textit{inum} indicates the \textit{csn} at process \( \textit{pid} \) when it took its own local checkpoint on initiating the checkpointing.

\textit{sent}_i: a boolean, which is set to 1 if \( P_i \) has sent a message in the current checkpoint interval.

\textit{cp\_state}_i: a boolean, which is set to 1 if \( P_i \) is during the checkpointing process.

\textit{old\_csn}: a variable used to save the \textit{csn} of the current tentative (permanent) checkpoint.

\( CP_i \): a record maintained by each process \( P_i \). Each record has the following fields:

- \textit{mutable}: the mutable checkpoint of \( P_i \).
- \textit{R}: \( P_i \)'s own boolean vector before it takes the current mutable checkpoint.
- \textit{trigger}: the \textit{trigger} which is associated with the current mutable checkpoint.
- \textit{sent}: \( P_i \)'s own \textit{sent} before it takes the current mutable checkpoint.

\textit{csn} is initialized to an array of 0's at all processes. The trigger tuple at process \( P_i \) is initialized to \( (i, 0) \). The \textit{weight} and \textit{cp\_state} at a process is initialized to 0. When a process \( P_i \) sends a computation message, it appends its \( \textit{csn}_i[i] \) to the message. Also, \( P_i \) checks if \( \textit{cp\_state}_i \) is equal to 1. If so, it appends its trigger to the computation message.

When a process \( P_j \) receives a checkpoint request from \( P_i \), we say "\( P_j \) inherits a request from \( P_i \)" if only if \( \textit{old\_csn}_j \leq \textit{req\_csn} \) (\textit{req\_csn} is appended with the request)
and $P_j$ takes a tentative checkpoint. In this definition, we use $old\_csn_j$ instead of $csn_j[j]$ used in Chapter 5.3.1, since $csn_j[j]$ is also increased when taking a mutable checkpoint, but we need to compare $req\_csn$ with the $csn$ of the current tentative (permanent) checkpoint.

### 5.3.3 The Non-blocking Checkpointing Algorithm

In this chapter, we present our non-blocking checkpointing algorithm. To clearly present the algorithm, we assume that at any time, at most one checkpointing is in progress. In Chapter 5.3.6, we extend the algorithm for concurrent invocations.

**Checkpointing Initiation:** Any process can initiate a checkpointing process. When a process $P_i$ initiates a checkpointing process, it takes a local checkpoint, increments its $csn_i[i]$, sets $weight_i$ to 1, sets $cp\_state_i$ to 1, and stores its own identifier and the new $csn_i[i]$ in its trigger. Then, it sends a checkpoint request to each process $P_j$ such that $R^i[j]=\emptyset$ and resumes its computation. Each request carries the trigger of the initiator, $R^i$, and a portion of the weight of the initiator, whose weight is decreased by an equal amount.

**Reception of a Checkpoint Request:** When a process $P_i$ receives a request from $P_j$, it first compares $req\_csn$ with its $old\_csn$ to see if it needs to inherit the request. If $P_i$ does not need to inherit the request, it sends the appended $weight$ to the initiator and then exits. Otherwise, it updates its $csn$ and $cp\_state$, and compares $P_j\_trigger (msg\_trigger)$ with $P_i\_trigger (own\_trigger)$. If $msg\_trigger = own\_trigger$ (implying that $P_i$ has already taken a checkpoint for this checkpointing), $P_i$ checks if there is a mutable checkpoint which has a trigger identical to $msg\_trigger$. If not, $P_i$ sends the appended $weight$ to the initiator; otherwise, $P_i$ saves the mutable checkpoint on
stable storage (the mutable checkpoint is turned into a tentative checkpoint), and then propagates the request. If \( P_i \) propagates the request to all processes on which it depends, it may result in a large number of redundant system messages, since some processes on which \( P_i \) depends may have received the request from other processes. The Koo-Toueg algorithm [39] uses this approach, and its system message overhead can be as large as \( O(N^2) \), where \( N \) is the number of processes in the system. On the other hand, only propagating the request to processes on which \( P_i \) depends, but \( P_j \) (the sender) does not, may not work, since a process receives a request does not necessarily mean that it inherits the request. We solve this problem by attaching some information (\( csn \) and \( R \) which are saved in \( MR \)) to the request. Then, \( P_i \) only propagates the request to each process \( P_k \) on which \( P_i \) depends, but \( P_k \) may have not inherited the request: that is, if \( P_i \) knows (by \( MR \)) some other process has sent the request to \( P_k \) with \( req.csn \geq csn_i[k] \) (\( req.csn \) is appended with the request and saved in \( MR[k].csn \)). it does not need to send the request to \( P_k \); otherwise, it has to send the request since \( P_k \) may inherit the request from \( P_i \). Also, \( P_i \) appends the initiator's trigger and a portion of the received weight to all those requests. At last, \( P_i \) sends a reply to the initiator with the weight equal to the remaining weight and resumes its underlying computation. If \( msg.trigger \neq own.trigger \), \( P_i \) takes a tentative checkpoint, increases its \( csn_i[i] \), and propagates the request as above. Then, \( P_i \) clears \( R_i \) and \( sent_i \), sends a reply to the initiator with the remaining weight, and then resumes its underlying computation.

**Computation Messages Received During Checkpointing:** When \( P_i \) receives a computation message from \( P_j \), \( P_i \) compares \( m.csn \) with its local \( csn_i[j] \). If \( m.csn \leq csn_i[j] \), the message is processed and no checkpoint is taken. Otherwise, it implies
that $P_j$ has taken a checkpoint before sending $m$. $P_i$ updates its $csn_i[j]$ to $m.csn$ and checks if the following conditions are satisfied:

- **Condition 1:** $P_j$ is in checkpointing process before sending $m$.
- **Condition 2:** $P_i$ has sent a message since last checkpoint.
- **Condition 3:** $P_i$ has not taken a checkpoint associated with the initiator (in the msg_trigger).

If all of them are satisfied, $P_i$ takes a mutable checkpoint and updates its data structures, such as $csn, CP, R, cp.state,$ and $sent$. If only Condition 1 is satisfied, $P_i$ only increases $csn_i[i]$ and sets $cp.state_i$ to 1.

**Termination and Garbage Collection:** The initiator adds weights received in all reply messages to its own weight. When its weight becomes equal to 1, it concludes that all processes involved in the checkpointing have taken their tentative checkpoints. Then, it broadcasts commit messages to all processes in the system. If a process has taken a tentative checkpoint, on receiving the commit message, it makes its tentative checkpoint permanent and clears $cp.state$. Other processes also clear their $cp.state$ and discard mutable checkpoints if there is any. Note that when a process discards its mutable checkpoints, it updates its $R$ and $sent$.

Instead of broadcasting commit messages to all processes, in [11], the initiator only sends commit messages to those processes from which it has received reply messages. However, to clear $cp.state$, each process needs to maintain a history of the processes to which it has sent messages when its $cp.state$ is equal to 1. Also, it notifies them to clear their $cp.state$. There is a tradeoff between these two approaches. If there are many communications among processes during last checkpoint interval, the broadcast approach is better. On the other hand, if there are only a limited number
of message exchanges during last checkpoint interval, the update approach [11] is better. To obtain the advantages of both approaches, the initiator can use a counter to save the number of processes that have taken checkpoints. If the counter is larger than a value (a system tuning parameter), the broadcast approach is used; otherwise, the update approach is used. Since this paper concentrates on reducing the overhead of saving checkpoints, we simply use the broadcast approach.

A formal description of our non-blocking checkpointing algorithm is given below:

**Actions taken when** $P_i$ **sends a computation message to** $P_j$:

- if $cp.state_i = 1$
  - then send($P_i$, message, $csn_i[i]$, own_trigger); sent_i := 1:
  - else send($P_i$, message, $csn_i[i]$, NULL); sent_i := 1:

**Actions for the initiator** $P_j$:

- increment($csn_j[j]$); own_trigger := ($P_j$, $csn_j[j]$); $cp.state_j := 1$:
- for $k := 0$ to $N$ do $MR[k]$.$csn := 0$: $MR[k]$.$R := 0$:
  - prop_cp($R_j$, $MR$, $P_j$, own_trigger, 1.0):
- take a local checkpoint (on stable storage): $old.csn_j := csn_j[j]$: sent_j := 0: reset $R_j$:

**Actions at process** $P_i$, **on receiving a checkpoint request from** $P_j$:

- receive($P_j$, request, $MR$, recv_csn, msg_trigger, req_csn, recv_weight):
  - $csn_i[j] := recv_csn$:
  - if $old.csn_i > req.csn$
    - then send($P_i$, reply, recv_weight) to the initiator: return:
    - $cp.state_i := 1$:
  - if $msg.trigger = own.trigger$
    - then if $CP_i.trigger = msg.trigger$
      - then prop_cp($CP_i$, $R$, $MR$, $P_i$, msg_trigger, recv_weight):
        - save $CP_i.mutable$ on stable storage: $old.csn_i := csn_i[i]$: $CP_i := NULL$:
        - send($P_i$, reply, weight) to the initiator:
      - else send($P_i$, reply, recv_weight) to the initiator:
    - else increment($csn_i[i]$); own_trigger := msg_trigger:
prop.cp(R_i, MR, P_i, msg_trigger, recv_weight):

- take a local checkpoint (on stable storage): \( old\_csn_i := csn_i[i] \);
- send(\( P_i, reply, weight_i \)) to the initiator:
  \( sent_i := 0 \); reset \( R_i \);

**Actions at process \( P_i \), on receiving a computation message from \( P_j \):**

- receive(\( P_j, m, recv\_csn, msg\_trigger \)):
  - if \( recv\_csn \leq csn_i[j] \)
    - then \( R_i[j] := 1 \); process the message:
  - else if \( csn_i[msg\_trigger.pid] = msg\_trigger.inum \)
    - then \( csn_i[j] := recv\_csn; R_i[j] := 1 \); process the message:
  - else \( csn_i[j] := recv\_csn; R_i[j] := 1 \); process the message:

- if \( msg\_trigger \neq NULL \land sent_i = 1 \land msg\_trigger \neq own\_trigger \)
  - then take a local checkpoint, save it in \( CP_i.mutable \):
    \( CP_i.trigger := msg\_trigger; CP_i.R := R_i; CP_i.sent := sent_i; \)
    \( sent_i := 0 \); reset \( R_i \);
  - if \( msg\_trigger \neq NULL \land cp\_state_i = 0 \)
    - then \( cp\_state_i := 1 \); increment \( (csn_i[i]) \); \( own\_trigger := msg\_trigger \):

**prop.cp(R_i, MR, P_i, msg_trigger, recv_weight)**

- \( weight_i := recv\_weight_i \);
- for \( k := 0 \) to \( N \) do \( temp[k].csn := max(MR[k].csn, csn_i[k]) \); \( temp[k].R := max(MR[k].R, R_i[k]) \);
- for any \( P_k \) such that \( (R_i[k] = 1) \land max(MR[k].csn, csn_i[k]) \neq MR[k].csn \)
  - \( weight_i := weight_i/2 \); send(\( P_i, request, temp, csn_i[i], msg\_trigger, csn_i[k], weight_i \));

**Actions in the second phase for the initiator \( P_i \):**

- receive(\( P_j, reply, recv\_weight \)); \( weight_i := weight_i + recv\_weight \);
- if \( weight_i = 1 \)
  - then \( cp\_state_i := 0 \); broadcast(\( commit, msg\_trigger \));

**Actions at other process \( P_j \) on receiving a broadcast message:**

- receive(\( commit, msg\_trigger \)); \( csn_j[msg\_trigger.pid] = msg\_trigger.inum \);
  \( cp\_state_j := 0 \);
- if \( CP_j.trigger = msg\_trigger \land CP_j \neq NULL \)
  - then \( sent_j := sent_j \cup CP_j.sent; R_j := R_j \cup CP_j.R; CP_j := NULL \);
if there is a tentative checkpoint associated with msg_trigger, make it permanent:

5.3.4 An Example

The basic idea of the algorithm can be better understood by the example in Figure 5.3. To initiate a checkpointing process, $P_2$ takes its own checkpoint and sends checkpoint requests to $P_1$, $P_3$ and $P_4$, because $R_2[1] = 1$, $R_2[3] = 1$, and $R_2[4] = 1$. When $P_2$'s request reaches $P_4$, $P_4$ takes a checkpoint. Then, $P_4$ sends message $m_3$ to $P_3$. When $m_3$ arrives at $P_3$, $P_3$ takes a mutable checkpoint before processing it because $m_3.csn > csn_3[4]$ and $P_3$ has sent a message during the current checkpoint interval. For the same reason, $P_1$ takes a mutable checkpoint before processing $m_2$.

$P_0$ has not communicated with other processes before it takes a local checkpoint. Later, it sends a message $m_1$ to $P_1$. If $P_0$ has finished its checkpointing process before it sends $m_1$, $P_1$ does not need to take the checkpoint $C_{1,2}$. Otherwise, $P_1$ takes a mutable checkpoint $C_{1,2}$ before processing $m_1$.

When $P_1$ receives the checkpoint request from $P_2$, since $C_{1,1}$ is a mutable checkpoint associated with $P_2$, $P_1$ turns $C_{1,1}$ into a tentative checkpoint by saving it on stable storage. Similarly, $P_3$ converts $C_{3,1}$ to a tentative checkpoint when it receives the checkpoint request from $P_2$. Finally, the checkpointing initiated by $P_2$ terminates when checkpoints $C_{1,1}, C_{2,1}, C_{3,1}$, and $C_{4,1}$ are made permanent. $P_1$ discards $C_{1,2}$ when it makes checkpoint $C_{1,1}$ permanent or receives $P_0$'s commit, whichever is earlier.

5.3.5 Relaxing the Reliable Channel Assumption

In order to relax the reliable channel assumption, we have to deal with message loss and duplicate problems. There are three system messages in our non-blocking
algorithm: request, reply, and commit. In the following, we present techniques to tolerate the loss and duplicate problems of these system messages.

**The request and reply Messages:** We change the implementation of termination detection as follows. The initiator always expects to receive reply messages from the processes to which it sends checkpoint request message directly or indirectly. In our non-blocking algorithm, a process \( P_i \) sends a reply to the initiator only when it receives a request from some process, say \( P_j \). When \( P_i \) sends a reply to the initiator, it appends the id of the process from which it received the checkpoint request, so that the initiator can remove the expectation about receiving that request. If \( P_i \) has taken a tentative checkpoint and propagated request messages to some other process, say \( P_k \) and \( P_i \). \( P_i \) also appends this new information to the reply. When the initiator receives this reply from \( P_i \), it removes its expectation about receiving \( P_i \)'s reply; then, it adds expectations to receive reply messages from \( P_k \) and \( P_i \), and these reply messages are associated with \( P_i \)'s request. If the request from \( P_i \) to \( P_k \), or the reply from \( P_k \) to the initiator is lost, the initiator timeouts and sends a request to \( P_k \). After that, it removes the expectation about \( P_k \)'s reply associated with \( P_i \) and adds expectation to receive \( P_k \)'s reply associated with its own request. After several timeouts, the initiator assumes that there is a failure, and then it aborts the checkpointing process and starts the recovery process. A process may receive duplicated request or reply messages. For duplicated requests, the receiver will not take a checkpoint, since it has updated its trigger on receiving the previous request, it just sends a reply to the initiator. For duplicated (or late) reply messages, since the initiator does not expect the duplicated (or late) reply messages, it simply discards it.
The initiator maintains an array \( V \), which saves the processes from which it expects to receive reply messages. Each entry of \( V \) saves the information about the expected reply messages. For example, \( V[i] = \{j, k\} \) means that the initiator expects to receive two reply messages from \( P_i \), one reply is due to the request sent from \( P_j \) to \( P_i \), the other is due to the request from \( P_k \) to \( P_i \).

**The commit and abort Messages:** If a process didn't receive commit or abort after a timeout period, it asks the initiator if the initiator has sent out commit or abort and then acts accordingly. If the process has missed a commit, and it has sent a computation message to another process which has already received a commit for the same checkpoint initiation long time ago, the receiver knows that the sender has missed a commit by looking at the \( msg\_trigger \). Thus, it notifies the sender to commit. Duplicated commit message can be recognized by comparing \( msg\_trigger \) to its own \( trigger \).

### 5.3.6 Multiple Concurrent Initiations

The simplest way to handle concurrent checkpoint initiations is to use the techniques in [39]. When a process \( P_i \) receives a checkpoint request from \( P_j \) while executing the checkpoint algorithm, \( P_i \) ignores \( P_j \)'s checkpoint request or defers the request until it finishes its current checkpointing. If \( P_i \)'s checkpoint request is ignored by a process, \( P_i \) has to abort its checkpointing efforts, which results in poor performance. A more efficient technique to handle concurrent checkpoint initiations can be found in [53]. As multiple concurrent checkpoint initiation is orthogonal to our discussion, we only briefly mention the main features of [53]. When a process receives its first request for checkpointing initiated by another process, it takes a local checkpoint and
propagates the request. All local checkpoints taken by the participating processes for a ion collectively form a global checkpoint. The state information collected by each independent checkpointing is combined. The combination is driven by the fact that the union of consistent global checkpoints is also a consistent global checkpoint. The checkpoint thus generated is more recent than each of the checkpoints collected independently, and also more recent than that collected by [62]. Therefore, the amount of computation lost during rollback, after process failures, is minimized.

5.3.7 Handling Failures During Checkpointing

Since MHs are more prone to failure, there is a possibility that during the checkpointing activity, an MH fails and all processes running on it also fail. We assume that if a process fails, some processes that try to communicate with it get to know of the failure. If the failed process is not the checkpointing initiator, the simplest way to deal with failures is to use abort messages similar to [39, 54]. More specifically, the process detecting the failures notifies the initiator, which broadcasts abort messages to all process participating in the current checkpointing. Processes discard their checkpoints (tentative or mutable) and restore some variables such as sent, old_csn, R, etc. on receiving the abort messages. If the failed process is the coordinator and the failure occurred before the process sent out commit or abort messages, on restarting after failure it broadcasts an abort corresponding to its checkpoint initiation. If it had failed after broadcasting a commit or abort, then it does not do anything more for that checkpoint initiation.

The above approach may not have good performance since the whole checkpointing aborts even when only one process participating in the checkpointing fails. We would
like to use a more efficient approach proposed by Kim and Park [38]. In their approach, processes can commit their tentative checkpoints when none of the processes on which they depend fails. Then, the consistent recovery line is advanced for those processes that committed their checkpoints. Certainly, the initiator and other processes which depends on the failure process have to abort their checkpointing and discard their tentative (mutable) checkpoints as in [39]. In this way, the checkpoint-commitment decision can be made locally, so that the total abort of the checkpointing is avoided. That is, when a process involved in a checkpointing coordination fails, the processes not affected by the failed one can make their decisions, while the protocols in [39] aborts the whole checkpointing activity.

In mobile computing, since a wireless channel is more likely to suffer from intermittent errors, failure detection in wireless networks should be different from that in static networks. Since failure detection and failure recovery are orthogonal to our discussion, more information on how to deal with process failures can be found in [40, 47, 54].

### 5.3.8 Correctness Proofs

In Chapter 5.3.2, $R_t$ represents all dependency relations in the current checkpointing period. Due to the introduction of mutable checkpoints, $R_t$ may represent the dependency relations after the last mutable checkpoint. To simplify the proof, in the following, $R_t$ means the first parameter of subroutine $prop_{cp}$ in our non-blocking algorithm. More specifically, $R_t$ should be $CP_t.R$ if there is a mutable checkpoint.

**Theorem 8** The algorithm creates a consistent global checkpoint.
Proof. We prove this by contradiction. Assume that the global state of the system is inconsistent at a time instance. Then, there must be a pair of processes \( P_i \) and \( P_j \) such that at least one message \( m \) has been sent from \( P_j \) after \( P_j \)'s last checkpoint and has been received by \( P_i \) before \( P_i \)'s last checkpoint. Since \( R_i[j] = 1 \) at \( P_i \) at the time of taking its checkpoint, \( P_i \) sends a checkpoint request to \( P_j \), or a process \( P_k \) has sent the request to \( P_j \) if \( MR[j].cn > csn_i[j] \). Thus, at least one checkpoint request is sent to \( P_j \). If \( P_j \) runs on an \( MSS \), the underlying network routes the request to it. If \( P_j \) runs on an \( MH_i \), which is in \( MSS_p \)'s cell, there are three possibilities when the request reaches \( MSS_p \):

**Case 1:** If \( MH_i \) is still connected to \( MSS_p \), the request is forwarded to \( MH_i \), then to \( P_j \).

**Case 2:** \( MH_i \) has moved to \( MSS_q \) (handoff). \( MSS_p \) forwards the request to \( MSS_q \), which forwards it to \( MH_i \), and then to \( P_j \) by the underlying routing protocol.

**Case 3:** \( MH_i \) is disconnected from the network. \( MSS_p \) takes a checkpoint on behalf of \( P_j \) by converting \( disconnect.checkpoint_j \) into \( P_j \)'s new checkpoint (as in Chapter 5.1). Since \( P_j \) cannot send any message after disconnection, it must have sent \( m \) before its disconnection. Thus, the sending of \( m \) is recorded in \( disconnect.checkpoint_j \). A contradiction.

In Case 1 and Case 2, when \( P_j \) receives the request, if \( req.cn < old.cn_j \), no matter the request comes from \( P_i \) or \( P_k \) (if the request comes from \( P_k \): \( (csn_i[j] \leq MR[j].cn) \land (MR[j].cn = req.cn) \land (req.cn < old.cn_j) \implies csn_i[j] < old.cn_j) \). \( P_j \) has already taken a checkpoint after sending \( m \). Thus, the sending of \( m \) is recorded at \( P_j \). If \( req.cn \geq old.cn_j \) (the request may come from \( P_i \) or \( P_k \)), there are two possibilities:
Case 1: $own.trigger \neq msg.trigger$. There are two possibilities for $P_j$ to take a checkpoint:

**Case 1.1:** The checkpoint is taken after the sending of $m$. Then:

- $send(m)$ at $P_j \rightarrow 1$ receive($m$) at $P_i$
- receive($m$) at $P_i \rightarrow$ checkpoint taken at $P_i$
- checkpoint taken at $P_i \rightarrow request$ sent by $P_i$ to $P_j$
- request sent by $P_i$ to $P_j \rightarrow$ checkpoint taken at $P_j$

Using the transitivity property of $\rightarrow$, we have: $send(m)$ at $P_j \rightarrow$ checkpoint taken at $P_j$. Thus, the sending of $m$ is recorded at $P_j$.

**Case 1.2:** The checkpoint is taken before the sending of $m$. As a result, $P_j$ increase $csn_j[j]$ before it sends $m$ to $P_i$, and then $m.csn > csn_i[j]$. There are two possible situations:

**Case 1.2.1:** $P_j$ has finished its checkpointing process (the last checkpoint) before it sends $m$. Hence, $P_i$ does not need to take a checkpoint when it receives $m$ and then the reception of $m$ is not recorded in the last checkpoint of $P_i$.

**Case 1.2.1:** $P_j$ has not finished its checkpointing process before it sends $m$. If $P_i$ does not need to take a mutable checkpoint before processing $m$, the reception of $m$ cannot be recorded in the last checkpoint of $P_i$. If $P_i$ takes a mutable checkpoint before processing $m$, when $P_i$ receives the request for this checkpoint initiation, $P_i$ turns the mutable checkpoint into a tentative checkpoint. Certainly, the reception of $m$ is still not recorded in the last checkpoint of $P_i$. □

Case 2: $own.trigger = msg.trigger$. In this case, $P_j$ has taken a mutable checkpoint or a tentative checkpoint. There are two possibilities:

**Case 2.1:** The checkpoint is taken after the sending of $m$. If the checkpoint is a mutable checkpoint, on receipt of the request, it is changed to a tentative checkpoint. Thus, the sending of $m$ is recorded.

---

$1 \rightarrow$ is the "happened before" relation described in [44]

\[114\]
Case 2.2: The checkpoint is taken before the sending of m. Similar to Case 1.2, we get contradictions.

Lemma 6 Every process inherits at most one checkpoint request to take a checkpoint.

Proof. After a process $P_i$ inherits a checkpoint request, it changes its own\_trigger to the trigger attached with the request and takes a checkpoint (or make a mutable checkpoint permanent). Later, when it receives other checkpoint requests corresponding to this checkpoint initiation, we already have own\_trigger = msg\_trigger. Thus, $P_i$ cannot take a mutable checkpoint: i.e., $CP_i.trigger \neq own.trigger$. Thus, $P_i$ cannot take a checkpoint on receipt of other requests corresponding to the same checkpoint initiation; that is, it does not inherit any request other than the first one.

In order to prove that our non-blocking checkpointing algorithm terminates, we introduce the following notations:

- $W(request)$: the weight carried by a request message.
- $W(reply)$: the weight carried by a reply message.
- $W(P_{init})$: the weight at the initiator.
- $W(P_{other})$: the weight at a process other than the initiator.

Lemma 7 During a checkpointing process, the following invariant holds:

$$W(P_{init}) + \sum_{\forall request} W(request) + \sum_{\forall reply} W(reply) + \sum_{\forall P_{other}} W(P_{other}) = 1$$

Proof. When $P_{init}$ initiates a checkpointing process, $W(P_{init}) = 1$. No weight is associated with other processes, and no request or reply messages are in transit. Hence, the
invariant holds. During the checkpointing process, the initiator sends out a portion of its weight in each outgoing request message. Therefore,

\[ \sum_{\text{request}} W(\text{request}) + W(P_{\text{init}}) = 1 \]

When a process \( P_i \) receives a checkpoint request, there are two possibilities:

**Case 1:** If \( P_i \) needs to take a tentative checkpoint or turn a mutable checkpoint into a tentative checkpoint, a part of the received weight is propagated to other processes in request messages, and the rest of the weight is sent to the initiator in a reply.

**Case 2:** If \( P_i \) does not need to take a tentative checkpoint or turn a mutable checkpoint into a tentative checkpoint, the entire received weight is sent back to the initiator in a reply.

Therefore, no portion of the weight in a request is retained by \( P_i \). At any instant of time during the checkpointing process, request and reply messages may be in transit, and some non-initiator processes may have non-zero weights. However, no extra weight is created or deleted at any non-initiator process. Thus, the invariant holds.

**Theorem 9** The proposed checkpointing algorithm terminates within a finite time.

**Proof.** In our algorithm, a process only propagate request messages when it inherits a request. Based on Lemma 6, every process inherits at most one request to take a checkpoint, and then each process propagates the received request at most once. Since the number of processes in the system is finite, the number of request messages generated are finite. As message propagation delay is bounded, within a finite time after the checkpoint initiation, no new request messages will be generated and all
such messages generated in the past have been delivered by the receivers. After this point of time, say $T$, the following assertion is true:

$$\sum_{\forall \text{request}} W(\text{request}) = 0$$  \hspace{1cm} (5.1)

On the receipt of a request, a non-initiator process immediately sends out the weight received in the request on the outgoing request messages or reply messages. Thus, within a finite time after $T$, the weight of all non-initiator processes becomes zero. As there are no more request messages in the system, non-initiator processes cannot acquire any weight in the future. After this point of time, say $T' > T$, the following assertion is true:

$$\sum_{\forall \text{other}} W(\text{other}) = 0$$  \hspace{1cm} (5.2)

As message propagation delay is finite, all reply messages will be received by the initiator within a finite time after $T'$. As there are no more request messages, no new reply will be generated. Hence, after time, say $T'' > T'$, the following assertion is true:

$$\sum_{\forall \text{reply}} W(\text{reply}) = 0$$  \hspace{1cm} (5.3)

Based on Lemma 7:

$$W(\text{init}) + \sum_{\forall \text{request}} W(\text{request}) + \sum_{\forall \text{reply}} W(\text{reply}) + \sum_{\forall \text{other}} W(\text{other}) = 1$$

After time $T''$, since $T'' > T' > T$, assertions (1), (2), and (3) are all true. Thus, $W(\text{init}) = 1$. At this point, the initiator sends commit messages to the processes that took checkpoints. A non-initiator process receives the commit message within a finite time. Therefore, the checkpointing algorithm terminates within a finite time. □
We now show that the number of processes that take new tentative (permanent) checkpoints during the execution of our algorithm is minimal. Based on Lemma 6, a process takes at most one checkpoint corresponding to a checkpointing process. Let \( P = \{P_0, P_1, \ldots, P_k\} \) be the set of processes that take new checkpoints during the execution of our algorithm, where \( P_0 \) is the initiator. Let \( C(P) = \{C(P_0), C(P_1), \ldots, C(P_k)\} \) be the new checkpoints taken by processes in \( P \).

When a process receives a checkpoint request, it asks all processes on which it depends to take checkpoints. The process receiving the request should take a checkpoint as soon as possible, since the longer it waits, the more processes will have dependency relation with it, and then the more processes need to take checkpoints. If the initiator knows all processes on which it depends, it can send checkpoint requests to them at once and then save the time of tracing the dependency tree. Some techniques [11, 54] exist to approximate this approach. However, it increases run time overhead since extra information has to be appended with the computation messages. Since the message delay is far less than the time between two checkpoint intervals, we do not consider the extra checkpoints resulting from the checkpoint request delay. Our algorithm can also use the techniques in [11, 54], but, as we discussed, that increases run time overhead and it is not valuable.

We define an alternate set of checkpoints: \( C'(P) = \{C'(P_0), C'(P_1), \ldots, C'(P_k)\} \) where \( C'(P_0) = C(P_0) \). and \( C'(P_i) (1 \leq i \leq k) \) is either \( C(P_i) \) or the checkpoint \( P_i \) had taken before executing our algorithm. If \( C'(P_i) \) is a new checkpoint, as we discussed, it should be taken as soon as possible, and then it is equal to \( C(P_i) \) without considering the checkpoint request delay.

**Theorem 10** \( C'(P) \) is consistent if and only if \( C'(P) = C(P) \).
Proof. The if part directly comes from Theorem 8. We now prove the only if part. The execution of our algorithm imposes a "P_i inherits a request from P_j" (defined in Chapter 5.3.2) relation on the set of processes. Since this relation is non-circular (based on Lemma 6) and there is only one initiator, it can be represented as a tree T: the root of T is the initiator, and P_j is a child of P_i if and only if P_j inherits a request from P_i. If P_j ∈ T, it must take a new checkpoint during the execution of the algorithm: hence P_j ∈ P. If P_j ∈ P, either P_j is the initiator or it inherits a request: hence P_j ∈ T. Therefore, P_j ∈ T if and only if P_j ∈ P.

Our proof is by contradiction. Suppose C'(P) ≠ C(P) and C'(P) is consistent. Let P_j ∈ P such that C'(P_j) ≠ C(P_j). Note that P_j ≠ P_0, and there exists a path from P_0 to P_k in T. Since C'(P_0) = C(P_0), there is an edge (P_i, P_j) on this path such that C'(P_i) = C(P_i) ∧ C'(P_j) ≠ C(P_j). Let m be the last message P_i receives from P_j. Since P_j inherits P_i's request, we have req.csn ≥ old.csn_j (req.csn is appended with the request) and the receipt of m is recorded in C'(P_i) (or C'(P_i)). Also, the sending of m is recorded in C(P_j). If C'(P_j) ≠ C'(P_j), C'(P_j) is the checkpoint P_j had before executing the algorithm, and then the sending of m is not recorded in C'(P_j). Thus, C'(P) is not a consistent set of checkpoints. A contradiction. □

5.4 Performance Evaluation

A mutable checkpoint is redundant if it will not be turned into a tentative checkpoint. In this chapter, to evaluate the performance of our non-blocking algorithm, we first use simulations to measure the number of redundant mutable checkpoints taken during a checkpointing period, which is the duration of a checkpointing process. from
the initiation to the termination. Then, we compare our non-blocking algorithm and our min-process algorithm with other algorithms in the literature.

5.4.1 Simulation Model

A system with $N$ MHs connected through a wireless LAN is simulated. Each MH has one process running on it and $N$ is equal to 16. The wireless LAN has a bandwidth of 2Mbps, which follows the IEEE 802.11 standard [19]. The length of each computation message is 1KB; Thus, the transmission delay of each computation message is $8 \times 1/2 = 4ms$. The length of each system message is 50Bytes. Thus, the transmission delay of each system message is $0.05 \times 8/2 = 0.2ms$. The size of a checkpoint is $1MB$ [28]. We can use incremental checkpointing [28] to reduce the amount of data that must be written on stable storage: that is, only the pages of the address space that have been modified since the previous checkpoint are transferred to the MSS. As a result, we assume that only $500KB$ are transmitted over the wireless link in order to take a tentative checkpoint, which needs $0.5 \times 8/2 = 2s$ (disk access time is not counted). In today's technology, Pentium 266MHz laptops with 64MB memory are available and will become popular soon. Thus, we assume mutable checkpoints are saved in the main memory. Since processor speed is much faster, the main memory is the bottleneck. For typical Pentium technology, 64bit wide memory bus with 83MHz bus speed is used. Thus, it needs about $\frac{125}{83 \times 8} = 3ms$ to save a mutable checkpoint. (If memory block copy is supported, the time can be further reduced.) A checkpoint is scheduled at each process with an interval of 900 seconds. If a process takes a checkpoint before its scheduled checkpoint time, the
next checkpoint will be scheduled 900s after that time. For simplicity, concurrent initiation, handoff and failures are not considered.

Each process sends out computation messages with the time interval following an exponential distribution. The message receiving pattern is considered in two computation environments: point-to-point communication and group communication. In the point-to-point communication, the destination of each message is uniformly distributed among all processes. In the group communication, processes are arranged into four groups and each group has a group leader. For intra-group communication, the destination of each message is a uniformly distributed random variable among all group members. Only group leaders can have inter-group communication, where the destination of each message is a uniformly distributed random variable among all group leaders.

5.4.2 Simulation Results

Since saving checkpoints takes a long time, in order not to block the process's execution, we use pre-copying [35, 67]: that is, the pages are copied to a separate area in the main memory and are then written from there to the stable storage. This is similar to saving a mutable checkpoint first and then turning it into a tentative checkpoint. Thus, we do not measure the number of mutable checkpoints that will be turned into tentative checkpoints. We measure the number of tentative checkpoints and the number of redundant mutable checkpoints for each checkpoint initiation under various message sending rate. The mean value of a measured parameter is obtained by collecting a large number of samples such that the confidence interval is reasonably
small. In most cases, the 95% confidence interval for the measured data is less than 10% of the sample mean.

**Point-to-point communication**

As shown in Figure 5.5, the number of tentative checkpoints for each checkpoint initiation increases as the message sending rate increases. Since the message receiving event is uniformly distributed, a process is more likely to receive a message from other processes when the message sending rate increases. Thus, it is more likely to have a dependency relationship with the initiator and thus is more likely to take a tentative checkpoint.

In Figure 5.5, when the message sending rate increases, the number of redundant mutable checkpoints for each checkpoint initiation increases at first and then decreases, and it is always less than 4% of the number of tentative checkpoints. This can be explained as follows. A process takes a mutable checkpoint only when it receives a computation message before it receives the checkpoint request during the checkpointing period. It takes a tentative checkpoint if it has received messages that created dependency relationship with the initiator during the checkpoint interval. Since the checkpointing period (at most 2 * 16 = 32s long) is much less than the checkpoint interval (900s), in general, a process takes much fewer redundant mutable checkpoints than tentative checkpoints. If the message sending rate is low, processes have low probability to send messages, and they have low probability to receive messages during the checkpointing period. Thus, they have low probability to take mutable checkpoints. If the message sending rate is high, it is more likely for a process to receive a message and take a mutable checkpoint during the checkpointing period. The mutable checkpoint is also more likely to be turned into a tentative
checkpoint. and then it is not a redundant mutable checkpoint. Also, according to our algorithm, the initiator quickly propagates the checkpoint request: thus, a process is less likely to receive a computation message before the checkpoint request during the checkpoint period, and then it is less likely to take a mutable checkpoint.

![Graph of checkpoint numbers](image)

**Figure 5.5:** The number of checkpoints in point-to-point communication environment

**Group communication**

Figure 5.6 shows the number of checkpoints in a group communication environment. Besides changing the intra-group message sending rate, a group leader also changes its inter-group message sending rate. In the left graph of Figure 5.6, for a group leader, the intra-group message sending rate is 1000 times more than the inter-group message sending rate; while in the right graph of Figure 5.6, the intra-group
message sending rate is 10000 times more. As can be seen, with group communication, the number of tentative checkpoints and redundant mutable checkpoints in the right graph is less than that in the left graph, and they are smaller than that in the point-to-point communication. In a group communication, when a process initiates a checkpointing process, processes in other groups have low probability of receiving messages from any process in the initiator’s group. Thus, they are less likely to have dependency relationship with the initiator; that is, they have low probability of taking tentative checkpoints or redundant mutable checkpoints.

Figure 5.6: The number of checkpoints in a group communication environment
5.4.3 Comparison With Other Algorithms

The following notations are used to compare our min-process algorithm and our non-blocking algorithm with other algorithms.

Notations

$C_{air}$: cost of sending a message from one process to another process. These processes are running on different $MH$s.

$C'_{air}$: cost of sending a message from an $MH$ to its $MSS$ (or vice versa). Note that $C_{air}$ includes the cost of finding the location of the $MH$ where the destination process in running. In general [33], the local $MSS$ of the source $MH$ is unaware of the current location of the target $MH$, and will have to “search” the network, i.e., query all $MSS$s. to discover the MSS that is local to the target $MH$. Thus $C_{air} >> C'_{air}$.

$C_{broad}$: cost of broadcasting a message to all processes which are running on $MH$s.
$C_{\text{broadcast}}$: cost of broadcasting a message to all MSSs. Note that $C_{\text{broadcast}} \gg C_{\text{broadcast}}'.

$C_{\text{wired}}$: cost of sending a message between any two MSSs. Note that $C_{\text{wired}} \gg C_{\text{air}}$.

$T_{\text{disk}}$: delay incurred in saving a checkpoint on stable storage in an MSS.

$T_{\text{data}}$: delay incurred in transferring a checkpoint from an MH to its MSS.

$T_{\text{msg}}$: delay incurred by system messages during a checkpointing process.

$T_{\text{wired}}$: average message delay in the static network.

$T_{\text{ch}}$: the time of a checkpointing period. $T_{\text{ch}} = T_{\text{msg}} + T_{\text{data}} + T_{\text{disk}}$.

$N_{\text{min}}, N, N_{\text{mss}}, N_{\text{muta}}, N_{\text{dep}}, N_{\text{min}}$ is the number of processes that need to take checkpoints using the Koo-Toueg algorithm [39]. $N$ is the total number of processes in the system. $N_{\text{mss}}$ is the total number of MSSs in the system. $N_{\text{muta}}$ is the number of redundant mutable checkpoints during a checkpointing process. $N_{\text{dep}}$ is the average number of processes on which a process depends.

Note that $1 \leq N_{\text{dep}} \leq N - 1$.

We use five parameters to evaluate the performance of a checkpointing algorithm: the number of checkpoints required during a checkpointing process, the blocking time (in the worst case), the system message overhead, whether the algorithm is distributed or not, and the output commit delay, which is the delay incurred before the system commits to the outside world. The outside world consists of everything with which processes can communicate that does not participate in the system's rollback-recovery, such as the user's workstation display, or even the file system if no special support is available for rolling back the contents of files. Messages sent to the outside world must be delayed until the system can guarantee that the message will never be "unsent" as a result of processes rolling back to recover from any possible future failure. If the sender is forced to roll back to a state before the message was sent, recovery of a consistent system state may be impossible, since the outside world cannot in general be rolled back. Once the system can meet this guarantee, the message may
be committed by releasing it to the outside world. Generally, if a process needs output commit, it initiates a checkpointing process. Thus, the output commit delay equals the duration of the checkpointing process.

**Performance of Our Min-process Algorithm**

*The blocking time:* After an MSS has sent all its local dependent vectors to the proxy MSS, it blocks (cannot forward messages) until it receives $S_{forced}$ from the proxy MSS. Therefore, the blocking time is $2T_{wired}$.

*The synchronization message overhead:* The message overhead includes the following. The request and reply messages from the initiator to its proxy MSS: $2C'_{air}$. The proxy MSS broadcasts $R_{request}$, $S_{forced}$, and $make\_permanent$ messages to all MSSs: $3C'_{broadcast}$. MSSs send dependency vectors and response messages to the proxy MSS: $2N_{mss} \times C_{wired}$. MSSs send checkpoint request and $make\_permanent$ messages to necessary MHs and receive response messages from them. It is $3N_{min} \times C'_{air}$. Therefore, the total message overhead is $2C'_{air} + 3C'_{broadcast} + 2N_{mss} \times C_{wired} + 3N_{min} \times C'_{air}$.

*The number of checkpoints:* Similar to Koo-Toueg algorithm [39], our algorithm forces only a minimum number of processes to take checkpoints.

*The output commit delay:* Compared to saving a checkpoint on the stable storage, the delay incurred by a system message is negligible. Also, saving checkpoints on the stable storage is the bottleneck. Thus, the delay incurred by saving checkpoint on stable storage is almost the same as the output commit delay. The output commit delay of our min-process algorithm is approximately $N_{min} \times T_{ch}$.

**Performance of Our Non-blocking Algorithm**

It is easy to see that our algorithm is distributed and the blocking time is 0.
The number of tentative checkpoints: Based on the result of Theorem 10, our algorithm forces only a minimum number of processes to save checkpoints on stable storage.

The output commit delay: From the simulation results, the number of redundant mutable checkpoints is less than 4% of the number of tentative checkpoints. Based on our simulation parameters, the delay of taking a tentative checkpoint is almost 1000 times longer than that of taking a mutable checkpoint. Thus, the output commit delay of our algorithm is approximately $N_{\text{min}} \cdot T_{ch}$. Note that for some applications, the checkpoint size is pretty small and the wireless network may have high bandwidth in the future, but at that time, the memory bus bandwidth also becomes larger. Moreover, at that time, the disk access delay, which is difficult to reduce, may dominate the checkpointing time. Thus, the delay of taking a tentative checkpoint is still significantly longer than that of taking a mutable checkpoint.

The system message overhead: In the first phase, a process taking a tentative checkpoint needs two system messages: request and reply. A process may receive more than one request for the same checkpoint initiation from different processes. However, we have used some techniques to reduce this kind of situation to occur. Thus, the system message overhead is approximately $2 \cdot N_{\text{min}} \cdot C_{\text{air}}$ in the first phase. In the second phase, we hope to get the advantages of the update approach and the broadcast approach by system tuning. Thus, the system message overhead is approximately $\min(N_{\text{min}} \cdot C_{\text{air}}, C_{\text{broad}})$ in the second phase.

Comparison to Other Algorithms:

Table 5.1 compares our min-process algorithm and our non-blocking algorithm with
two representative approaches for coordinated checkpointing. The Koo-Toueg algorithm [39] has the lowest overhead (based on our five parameters) among the blocking algorithms [4, 18, 24, 38, 39, 45, 56] in literature. The algorithm in [28] has the lowest overhead (based on our five parameters) among the proposed non-blocking algorithms [14, 28, 43, 60] in literature. We do not compare our algorithms with the Prakash-Singhal algorithm since it may result in inconsistencies, and there is no easy solution to fix it without increasing overhead.

**Comparison to the Koo-Toueg algorithm:** Since mutable checkpoints can be saved in the memory, the overhead of taking a mutable checkpoint is negligible compared to taking a real checkpoint, which needs to transmit the checkpoint across the wireless network and save it in the stable disk. As a result, our non-blocking algorithm and our min-process algorithm have similar performance in terms of the number of checkpoints and the output commit delay.

In terms of message complexity, our non-blocking algorithm reduce the message overhead from $3 \cdot N_{\text{min}} \cdot V_{\text{dep}} \cdot C_{\text{air}} (1 \leq V_{\text{dep}} \leq V - 1)$ to $2 \cdot N_{\text{min}} \cdot C_{\text{air}} + \min(N_{\text{min}} \cdot C_{\text{air}}, C_{\text{broad}})$. When $N_{\text{min}} = V$, the message reduction can be from $O(V^2)$ to $O(V)$. The message complexity of our min-process is $2C'_{\text{air}} + 3C'_{\text{broad}} + 2N_{\text{mss}} \cdot C_{\text{wired}} + 3N_{\text{min}} \cdot C'_{\text{air}}$. Since $C'_{\text{air}} \gg C'_{\text{air}}$, our min-process algorithm has much low message overhead compared to our non-blocking algorithm when $N_{\text{min}} > N_{\text{mss}}$. However, if $N_{\text{min}} \ll N_{\text{mss}}$, our non-blocking algorithm will have much lower message overhead since the min-process algorithm always include the $\text{MSS}$ communication overhead of $2N_{\text{mss}} \cdot C_{\text{wired}}$. 

129
In terms of blocking time, our non-blocking algorithm reduces the blocking time from $N_{\min} \cdot T_{ch}$ to 0. In the worst case, $N_{\min} = N$. Consider our simulation parameters: $N = 16$ and $T_{ch} = 2$s. the blocking time will be $32$s: i.e., all processes cannot do anything for half a minute in the Koo-Toueg algorithm, which significantly reduces system performance. Since $2 \cdot T_{wired} \ll T_{ch}$, the blocking time in our min-process algorithm is negligible compared to the Koo-Toueg algorithm.

Comparison to the Elnozahy etc. algorithm: Compared to [28], our non-blocking algorithm and our min-process algorithm force only a minimum number of processes to take their checkpoints on stable storage. Note that there may be many applications running in the system: some of them have higher reliability requirement and others do not. In a heterogeneous environment, some MHs may be more prone to failures than others. Moreover, different processes may run at their own speed and they may only communicate with a group of processes. As a result, some processes may need to take checkpoints more frequently than others. However, the algorithm in [28] forces all processes in the system to take checkpoints for each checkpoint initiation. Thus, our algorithms significantly reduces the message overhead and checkpointing overhead compared to [28]. Furthermore, in case of output commit, our algorithms have much shorter delay compared to [28] since our algorithms require fewer processes to take checkpoints before committing to the outside world. Seems like that our non-blocking algorithm needs more system messages than [28]. However, the algorithm in [28] is a centralized algorithm and there is no easy way to make it distributed without significantly increasing message overhead. Since some processes may be in the doze mode, broadcast may waste their energy and processor power. More importantly, the
system message is relatively small, and then the overhead of system messages is much smaller compared to the overhead of saving checkpoints on stable storage.

### 5.5 Related Work

The first coordinated checkpointing algorithm was presented in [4]. However, it assumes that all communications between processes are atomic, which is too restrictive. The Koo-Toueg algorithm [39] relaxes this assumption. In this algorithm, only those processes that have communicated with the checkpoint initiator either directly or indirectly since the last checkpoint need to take new checkpoints. Thus, it reduces the number of synchronization messages and the number of checkpoints. Later, Leu and Bhargava [45] presented an algorithm, which is resilient to multiple process failures, and does not assume that the channel is FIFO, which is necessary in [39]. However, these two algorithms [39, 45] assume a complex scheme (such as slide window) to deal with the message loss problem and do not consider lost messages in checkpointing and recovery. Deng and Park [24] proposed an algorithm to address both orphan message and lost messages.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Checkpoints</th>
<th>Blocking time</th>
<th>Output commit</th>
<th>Messages</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koo-Toueg [39]</td>
<td>$N_{\min}$</td>
<td>$N_{\min} \cdot T_{ch} = N_{\min} \cdot (T_{disk} + T_{data} + T_{msg})$</td>
<td>$N_{\min} \cdot T_{ch}$</td>
<td>$3 \cdot N_{\min} \cdot N_{dep} \cdot C_{air}$</td>
<td>Yes</td>
</tr>
<tr>
<td>Elnozahy [28]</td>
<td>$N$</td>
<td>$N \cdot T_{ch}$</td>
<td>$N \cdot T_{ch}$</td>
<td>$2 \cdot C_{bread} = N \cdot C_{air}$</td>
<td>No</td>
</tr>
<tr>
<td>Min-process</td>
<td>$N_{\min}$</td>
<td>$2 \cdot T_{wired}$</td>
<td>$N_{\min} \cdot T_{ch}$</td>
<td>$2C_{air} + 3C_{bread} + 2N_{msg}$</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-blocking</td>
<td>$N_{\min} + N_{mota}$</td>
<td>$0$</td>
<td>$(N_{\min} + N_{mota}) \cdot 2 \cdot N_{\min} \cdot C_{air} = min (N_{\min} \cdot C_{air}, C_{bread})$</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: A comparison of system performance
In Koo and Toueg’s algorithm [39], if any of the involved processes is not able to or not willing to take a checkpoint, the entire checkpointing process is aborted. Kim and Park [38] proposed an improved scheme that allows the new checkpoints in some subtrees to be committed while the others are aborted.

To further reduce the system messages needed to synchronize the checkpointing, loosely synchronous clocks [18, 56] are used. More specifically, loosely-synchronized checkpoint clocks can trigger the local checkpointing actions of all participating processes at approximately the same time without the need of broadcasting the checkpoint request by the initiator. However, a process taking a checkpoint needs to wait for a period that equals to the sum of the maximum deviation between clocks and the maximum time to detect a failure in another process in the system.

All the above coordinated checkpointing algorithms [4, 18, 24, 38, 39, 45, 56] require processes to be blocked during checkpointing. Checkpointing includes the time to trace the dependency tree and to save the state of processes on stable storage, which may be long. Therefore, blocking algorithms may dramatically reduce the performance of the system [7, 28].

The Chandy-Lamport algorithm [14] is the earliest non-blocking algorithm for coordinated checkpointing. However, in their algorithm, system messages (markers) are sent along all channels in the network during checkpointing. This leads to a message complexity of $O(N^2)$. Moreover, it requires all processes to take checkpoints and the channel must be FIFO. To relax the FIFO assumption. Lai and Yang [43] proposed another algorithm. In their algorithm, when a process takes a checkpoint, it piggybacks a checkpoint request (a flag) to the messages it sends out from each channel. The receiver checks the piggy-backed message flag to see if there is a need to
take a checkpoint before processing the message. If so, it takes a checkpoint before processing the message to avoid an inconsistency. To record the channel information, each process needs to maintain the entire message history on each channel as part of the local checkpoint. Thus, the space requirements of the algorithm may be large. Moreover, it requires all processes to take checkpoints, even though many of them are unnecessary.

The Elnozahy-Johnson-Zwaenepoel algorithm [28] uses the checkpoint sequence number to identify orphan messages, thus avoiding the need for processes to be blocked during checkpointing. However, this approach requires the initiator to communicate with all processes in the computation. The algorithm proposed by Silva and Silva [60] uses a similar idea as [28] except that the processes which did not communicate with others during the previous checkpoint interval do not need to take new checkpoints. Both algorithms [28, 60] assume that a distinguished initiator decides when to take a checkpoint. Therefore, they suffer from the disadvantages of centralized algorithms, such as one-site failure, traffic bottle-neck, etc. Moreover, their algorithms require almost all processes to take checkpoints, even though many of them are unnecessary. If they are modified to permit more processes to initiate checkpointing, which makes them distributed, the new algorithm suffers from another problem: in order to keep the checkpoint sequence number updated, any time a process takes a checkpoint, it has to notify all processes in the system. If each process can initiate a checkpointing process, the network would be flooded with control messages and processes might waste their time taking unnecessary checkpoints.
All the above algorithms follow two approaches to reduce the overhead associated with coordinated checkpointing algorithms: one is to minimize the number of synchronization messages and the number of checkpoints [4, 18, 24, 38, 39, 45, 56]; the other is to make checkpointing non-blocking [14, 28, 43, 60]. These two approaches were orthogonal in previous years until the Prakash-Singhal algorithm [54] combined them. However, their algorithm has some problems and may result in an inconsistency [9].

Acharya and Badrinath [1] were the first to present an checkpointing algorithm for mobile computing system. In their uncoordinated checkpointing algorithm, an MH takes a local checkpoint whenever a message reception is preceded by a message sent at that MH. If the send and receive of messages are interleaved, the number of local checkpoints will be equal to half of the number of computation messages, which may degrade the system performance.

For other uncoordinated checkpointing algorithms, as described in [6, 64], every process may accumulate multiple local checkpoints and logs on stable storage during normal operation. A checkpoint can be discarded if it is determined that it will no longer be needed for recovery. For this purpose, processes have to periodically broadcast the status of their logs on stable storage. The number of local checkpoints depends on the frequency with which such checkpoints are taken, and is an algorithm tuning parameter. An uncoordinated checkpointing approach is not suitable for mobile computing for a number of reasons. If the frequency of local checkpointing is high, each process will have multiple checkpoints requiring a large amount of stable storage, which also introduces a lot of communication overhead in mobile computing.
system. The stable storage and communication overheads can be reduced by taking local checkpoints less frequently. However, this will increase the recovery time as greater rollback and reply will be needed. Even though some algorithms [47, 71] were proposed to reduce the number of checkpoints to be saved on stable storage, to ensure correctness, a process still needs to keep much more checkpoints in uncoordinated checkpointing algorithms than that in coordinated checkpointing algorithms. In the coordinated checkpointing algorithm presented in this paper, most of the time, each process needs to store only one permanent checkpoint on stable storage, and at most two checkpoints: a permanent and a tentative (or mutable) checkpoint only for the duration of the checkpointing. Generally speaking, uncoordinated checkpointing approaches suffer from the complexities of finding a consistent recovery line after the failure, the susceptibility to the domino effect, the high stable storage overhead of saving multiple checkpoints of each process, and the overhead of garbage collection. Thus, our coordinated checkpointing algorithm has many advantages over uncoordinated checkpointing algorithms.
CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

This dissertation addressed two problems in designing efficient fault-tolerant systems on wireless networks: channel allocation and coordinated checkpointing.

*Channel allocation:* We proposed a fault-tolerant channel acquisition algorithm which tolerates communication link failures and node ($MH$ or $MSS$) failures. In the proposed algorithm, a borrower does not need to receive a response from every interference neighbor. It only needs to receive a response from a small portion of them. Thus, as long as the borrower can communicate with a small portion of its interference neighbors, it can borrow a channel from them. Also, we identified two guiding principles in designing channel selection algorithms. Following these principles, we proposed a channel selection algorithm to further improve the performance of the two-step strategy by considering the “richness” and the interference property. By integrating the channel selection algorithm into our channel acquisition algorithm, we get a complete distributed channel allocation algorithm. By keeping the borrowed channels, the channel allocation algorithm makes use of the temporal locality and adapts to the network traffic fluctuations; i.e., free channels are transferred to hot cells to achieve
load balancing. Detailed simulation experiments were carried out to evaluate our proposed methodology. Simulation results showed that our algorithm significantly reduces the failure rate under network congestion, communication link failures, and node failures compared to non-fault-tolerant channel allocation algorithms. Moreover, our fault-tolerant channel allocation algorithm reduces the message overhead compared to known distributed channel allocation algorithms, and outperforms them in terms of failure rate under uniform as well as non-uniform traffic distributions.

We investigated the fundamental differences between the search approach and the update approach, and proposed a novel distributed acquisition algorithm, which has a message complexity similar to the search approach and an acquisition delay similar to the update approach. We also identified the distinctive properties of the channel allocation problem by comparing it to the mutual exclusion problem, the k-mutual exclusion problem, the quorum-based mutual exclusion problem, and the drinking drinking philosopher problem.

*Coordinated checkpointing:* We identified problems in the Prakash-Singhal algorithm. We generalized the result and proved that there does not exist a non-blocking algorithm which forces only a minimum number of processes to take their checkpoints. The proof is based on a new concept of "z-dependency", which captures the essence of coordinated checkpointing. We also proposed a min-process algorithm and a non-blocking algorithm for mobile computing systems. In min-process algorithm, the checkpointing information such as dependency vectors are saved at an MSS. To initiate a checkpointing, the initiator collects dependency vectors from MSSs and then it determines and notifies all the processes that need to take checkpoints. In this way, the blocking time can be reduced from as much as $O(N \times T)$ to a negligible
constant. In our non-blocking algorithm, we proposed a new concept of "mutable checkpoint", which is neither a tentative checkpoint nor a permanent checkpoint. Mutable checkpoints can be saved anywhere: e.g., the main memory or the local disk of MHs. In this way, taking a mutable checkpoint avoids the overhead of transferring large amount of data over the wireless network to the stable storage at MSSs. We developed techniques to minimize the number of mutable checkpoints. Simulation results showed that the overhead of taking mutable checkpoints is negligible. Based on mutable checkpoints, our non-blocking algorithm forces only a minimum number of processes to take their checkpoints on the stable storage.

6.2 Future Research Directions

While proposing solutions to several important problems, this dissertation also sets the stage for future work in several areas of mobile computing.

We proposed two distributed channel acquisition algorithms: the efficient channel acquisition algorithm and the fault-tolerant channel acquisition algorithm. These two algorithms have similar message complexity. The efficient channel acquisition algorithm has low acquisition delay, but it is not fault-tolerant. It would be interesting to study how to further reduce the acquisition delay of our fault-tolerant channel allocation algorithm. Simulation results show that our channel selection algorithm performs worse in some situations. How to further improve the performance of channel selection algorithms and how to systematically setup those parameters still remains to be solved. In our simulation model, we only consider voice traffic. How to efficiently support data and multimedia traffic will be our future research.
Our existing work shows that no min-process non-blocking algorithm exists, and we have already proposed a non-blocking checkpointing algorithm and a min-process checkpointing algorithm. It is difficult to say which approach is better since different approaches are suitable for different applications. Thus, identifying suitable applications for each approach may be interesting. Based on our impossibility proof, there are three research directions. We have done some work along two extreme directions: min-process and non-blocking. Between these two extremes, we can design blocking, non-min-process algorithms that significantly reduce the blocking time as well as the number of checkpoints.

In this dissertation, we concentrated on the checkpointing part. It may be interesting to combine the two-level recovery scheme [68] and our checkpointing approach. In mobile computing systems, some failures are permanent, and then it is necessary to save checkpoints on stable storages in MSSs. However, transient failures can be recovered locally to avoid the overhead of transferring checkpoints from (to) the MSSs. We plan to design a two-level checkpointing and recovery algorithm for mobile computing systems to reduce performance overhead: that is, one level is used to recover permanent failures, and the second is used to recover transient failures.

In order to track the location of MHs, a database is used to store location information. Prakash and Singhal [52] proposed schemes to make the location database fault-tolerant using quorum-based approaches, where the location information of an MH is replicated at several places. Due to built-in redundancy, such an approach is fault-tolerant. To track down an MH, searching a nearby database improves performance. To implement a quorum-based system, mutual exclusion must be ensured. Since the first quorum-based mutual exclusion algorithm was proposed in 1985, it has
been an open problem to reduce the delay in processing users' requests from $2T$ to $T$ ($T$ is the one-way communication delay). No one has been able to solve it until we [13] proposed an optimal scheme which has a delay of $T$. With our scheme, the waiting time of a request is nearly reduced to half and the system throughput is almost doubled. In future research, we plan to investigate the possibility of extending our quorum-based mutual exclusion algorithm to fault-tolerant location management.

In this dissertation, we assumed MSSs are fixed and are connected by high-speed network. This model may not be suitable for some applications such as emergency response missions or defense applications. An emergency response mission like personnel rescue during floods or earthquakes would require fast installation of the MSSs and the high bandwidth wireline network connecting them. However, laying down cables and installing the MSSs would take a long time. Similarly, in a battlefield, there may not be enough time available to install the MSSs and the fixed network. The MSSs due to their immobility are highly vulnerable and will become inviting targets for the enemy.

Future research would be the design of mobile computing systems with mobile MSSs, or ad hoc networks where no MSS exists. There are some realistic scenarios where the mobile MSSs model would be applicable. In a battlefield, soldiers may move along with a tank, where the soldiers are equivalent to MHs and the tanks are equivalent to MSSs. Tanks are moving and they are connected by wireless networks.

Several solutions for mobile computing systems use the MSSs as proxies for resource poor MHs. When MSSs themselves become mobile, and consequently resource poor, these solutions may no longer be effective. New solutions may be needed.
APPENDIX A

Problems of the Prakash-Singhal Algorithm

The coordinated checkpointing algorithm proposed by Prakash and Singhal [54] may result in inconsistencies. In the following, we identify two problems in this algorithm and discuss some possible solutions.

The First Problem

Prakash and Singhal [54] claim that their algorithm can take lazy checkpoints. In Section 4.4 of their paper, they gave the following example: as in Figure A.1, \( P_3 \) sends \( m_9 \) to \( P_4 \) after \( P_3 \) finishes taking its checkpoint. When \( P_4 \) receives \( m_9 \), it takes a new checkpoint (lazy checkpoint). Then, \( P_4 \) asks \( P_5 \) to take a checkpoint and \( P_5 \) asks \( P_6 \) to take a checkpoint. \( P_1, P_2, \) or \( P_3 \) does not send checkpoint request to \( P_1, P_5, \) and \( P_6 \), since \( P_1, P_2, \) and \( P_3 \) do not depend on \( P_4, P_5, \) or \( P_6 \). The checkpointing algorithm terminates when the weight of \( P_2 \) is equal to 1. Because the computation message \( m_9 \) does not carry any weight, \( P_4, P_5, \) and \( P_6 \) send responses to the initiator with weight 0. Therefore, the termination of the algorithm is independent of their responses. Suppose \( P_2 \) finds out that its weight becomes equal to 1 before \( P_6 \) takes its checkpoint, but after \( P_2 \) has received the responses from \( P_3 \) and \( P_5 \). Then, \( P_2 \) sends commit messages to \( P_4 \) and \( P_5 \), but not \( P_6 \). As a result, \( P_4 \) and \( P_5 \) make
their tentative checkpoints permanent. If \( P_6 \) fails before it finishes taking its local checkpoint, it results in an inconsistency.

![Diagram](image)

**Figure A.1: The first problem**

**A solution to the first problem:** As presented in our non-blocking algorithm, we solve this problem by using mutable checkpoints. The process taking a mutable checkpoint does not propagate the checkpoint request until it receives the checkpoint request associated with this checkpoint initiation. In this example, \( P_4 \) only takes a mutable checkpoint, it does not require \( P_5 \) and \( P_6 \) to take checkpoints. The mutable checkpoint will be discarded since no process makes it permanent. Note that our algorithm only needs to propagate the checkpoint request once, but the Prakash-Singhal algorithm propagates the checkpoint request twice. However, lazy checkpointing is eliminated. Certainly, a process can always propagate lazy checkpoints by initiating another round of checkpointing. For example, after \( P_4 \) takes a lazy checkpoint, it initiates (as an
initiator) another checkpointing process and makes them permanent after $P_5$ and $P_6$ have taken their checkpoints.

**The Second Problem**

![Diagram of the second problem]

Figure A.2: The second problem

We illustrate the second scenario using the example in Figure A.2. $P_2$ initiates a checkpointing process by taking its own checkpoint and sends checkpoint request messages to $P_1$, $P_3$, and $P_4$. When $P_2$'s request reaches $P_4$, $P_4$ takes a checkpoint and sends message $m3$ to $P_3$. When $m3$ arrives at $P_3$, $P_3$ takes a checkpoint before processing the message, because $m3$ is the first message received by $P_3$ such that $msg\_trigger.pid \neq own\_trigger.pid$. $P_3$ sets $rfirst$ to 1. For the same reason, $P_1$ takes a checkpoint and sets $rfirst$ to 1 before processing $m2$. 
\( P_0 \) has not communicated with other processes before it takes a local checkpoint. Later, it sends a message \( m_1 \) to \( P_1 \). Because \( P_0 \) has taken a checkpoint, its checkpoint sequence number is larger than \( P_1 \) expected. However, \( m_1 \) is not the first computation message received by \( P_1 \) with a larger checkpoint sequence number than expected \((r_{\text{first}} = 1)\). Therefore, no checkpoint is taken.

Suppose \( P_1 \) initiates a checkpointing process after it receives \( m_5 \). \( P_1 \) takes a checkpoint, and then it sends checkpoint request messages to \( P_2 \), \( P_3 \) and \( P_4 \). When \( P_1 \)'s request reaches \( P_4 \), \( P_4 \) takes a checkpoint. Suppose \( P_4 \) sends a message \( m_4 \) to \( P_3 \) after it finishes its checkpoint. Because \( m_4 \) is not the first computation message received by \( P_3 \) with a larger checkpoint sequence number than expected \((r_{\text{first}} \text{ is already 1})\), \( P_3 \) does not take a checkpoint when it receives \( m_4 \). Later, when \( P_3 \) receives the checkpoint request from \( P_1 \), it takes a checkpoint. As a result, \( m_4 \) becomes an orphan.

**Possible solutions to the second problem:** The problem arises due to the variable \( r_{\text{first}} \). In Figure A.2, one possible solution is to let \( P_3 \) clear \( r_{\text{first}} \) before it receives \( m_4 \). In this case, \( P_3 \) will take a checkpoint before processing \( m_4 \). However, things are not that simple. In order to avoid the avalanche effect or unnecessary checkpoints, there is a need to keep the value of \( r_{\text{first}} \) as long as possible. On the other hand, to avoid an inconsistency, there is a need to clear \( r_{\text{first}} \) as soon as possible. Thus, we have a contradiction.

Another possible solution can be illustrated by Figure A.2. When \( P_4 \) sends \( m_4 \), it appends the dependence information to \( m_4 \). When \( P_3 \) receives \( m_4 \), it knows that the initiator transitively depends on it, so \( P_3 \) takes a checkpoint before processing
However, we can easily construct a counter-example, such as that in Lemma 4. Therefore, to solve the second problem, we have to introduce extra checkpoints.


