A Hierarchical Fault-Tolerance Framework
for Mobile Intelligent Agent Systems

by
Jian Chen

B.Eng., Huazhong University of Science and Technology, 1996
M.Eng., Tsinghua University, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia
April 2002
© Jian Chen, 2002
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 24, 2002
Abstract

"Mobile Agent Systems" is an emerging technology that is becoming increasingly popular. A huge amount of research activity has been carried out in this direction, and various Mobile Agent Systems have been built by different institutions and companies.

One of the most serious challenges that Mobile Agent Systems faces is the Fault-tolerance problem, which hinders Mobile Agent Systems to be put into real applications.

To deal with the problem, a novel architecture, called Multi-Layered Dual-Monitor Architecture (MLDMA), has been developed. MLDMA not only does maintain the simplicity of centralized control mechanisms, but also has high scalability. A new checkpoint algorithm has been designed for MLDMA's hierarchical structure, and some security concerns are addressed.

After introducing a mobile agent based file-sharing application—Wavetella, more discussion follows of Fault-tolerance solutions for the application layer, and comparisons are made among different approaches.
# Table of Contents

Abstract ......................................................................................................................... ii  
Table of Contents .......................................................................................................... iii  
List of Figures ................................................................................................................ v  
List of Tables ................................................................................................................... vi  
Acknowledgements ......................................................................................................... vii  
Chapter 1 Introduction .................................................................................................. 1  
  1.1 AGENTS AND MOBILE AGENTS ........................................................................... 2  
  1.1.1 Agents ................................................................................................................. 2  
  1.1.2 The Mobile Agent Paradigm .............................................................................. 3  
  1.1.3 Major Benefits .................................................................................................... 4  
  1.2 FAULT-TOLERANCE ISSUES ................................................................................ 6  
  1.3 GOAL .................................................................................................................... 7  
  1.4 THESIS CONTRIBUTIONS ................................................................................... 8  
  1.5 THESIS ORGANIZATION .................................................................................... 9  
Chapter 2 Background .................................................................................................. 10  
  2.1 WAVE – A MOBILE INTELLIGENT SYSTEM ...................................................... 10  
  2.1.1 Basics and Components ................................................................................... 10  
  2.1.2 Language ........................................................................................................... 13  
  2.1.3 Applications ...................................................................................................... 14  
  2.2 RELATED WORK .................................................................................................. 15  
Chapter 3 Hierarchical Fault-Tolerance Architecture .................................................. 18  
  3.1 CENTRALIZED VS. DISTRIBUTED ARGUMENTS ........................................... 18  
  3.2 DESIGN ............................................................................................................... 20  
  3.2.1 Multi-Layered Dual-Monitor Architecture ...................................................... 20  
  3.2.2 Monitors ........................................................................................................... 21  
  3.2.3 Vice-Monitors ................................................................................................... 23  
  3.2.4 Hierarchical Structure ...................................................................................... 27  
  3.2.5 Initiation ............................................................................................................ 28  
  3.2.6 Load Balancing Problem ................................................................................... 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.7 Security Issues</td>
<td>31</td>
</tr>
<tr>
<td>3.3 IMPLEMENTATION</td>
<td>32</td>
</tr>
<tr>
<td>3.3.1 WAVE System Structure</td>
<td>32</td>
</tr>
<tr>
<td>3.3.2 Implementation</td>
<td>33</td>
</tr>
<tr>
<td>3.3.3 Experiments</td>
<td>34</td>
</tr>
<tr>
<td>Chapter 4 Checkpointing Algorithm for the Hierarchical Architecture...</td>
<td>37</td>
</tr>
<tr>
<td>4.1 CONSISTENT SYSTEM STATE</td>
<td>37</td>
</tr>
<tr>
<td>4.2 ALGORITHM</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1 Phase 1</td>
<td>39</td>
</tr>
<tr>
<td>4.2.2 Phase 2</td>
<td>41</td>
</tr>
<tr>
<td>4.2.3 Phase 3</td>
<td>42</td>
</tr>
<tr>
<td>4.2.4 Multiple checkpoints</td>
<td>43</td>
</tr>
<tr>
<td>4.2.5 Late Messages</td>
<td>44</td>
</tr>
<tr>
<td>4.2.6 CN control</td>
<td>45</td>
</tr>
<tr>
<td>4.3 EXTENSIONS</td>
<td>46</td>
</tr>
<tr>
<td>4.4 IMPLEMENTATION</td>
<td>46</td>
</tr>
<tr>
<td>Chapter 5 Security Issues in the Fault-Tolerance Problem</td>
<td>47</td>
</tr>
<tr>
<td>5.1 PROBLEMS</td>
<td>47</td>
</tr>
<tr>
<td>5.2 SECURITY ISSUES IN FAULT-TOLERANCE</td>
<td>48</td>
</tr>
<tr>
<td>5.3 SOLUTIONS</td>
<td>49</td>
</tr>
<tr>
<td>5.3.1 Overview and Message Data structure</td>
<td>49</td>
</tr>
<tr>
<td>5.3.2 Authentication and Key Distribution</td>
<td>51</td>
</tr>
<tr>
<td>5.3.3 Extension</td>
<td>53</td>
</tr>
<tr>
<td>Chapter 6 Wavetella</td>
<td>55</td>
</tr>
<tr>
<td>6.1 INTRODUCTION</td>
<td>55</td>
</tr>
<tr>
<td>6.2 DESIGN</td>
<td>56</td>
</tr>
<tr>
<td>6.3 IMPLEMENTATION</td>
<td>59</td>
</tr>
<tr>
<td>6.3.1 From WAVE to Java</td>
<td>59</td>
</tr>
<tr>
<td>6.3.2 From Java to WAVE</td>
<td>59</td>
</tr>
<tr>
<td>6.4 FAULT-TOLERANCE AND WAVETELLA</td>
<td>60</td>
</tr>
<tr>
<td>6.4.1 Static Network</td>
<td>61</td>
</tr>
<tr>
<td>6.4.2 Dynamic Network</td>
<td>64</td>
</tr>
<tr>
<td>6.4.3 Discussion</td>
<td>64</td>
</tr>
<tr>
<td>Chapter 7 Conclusion</td>
<td>66</td>
</tr>
<tr>
<td>7.1 DISCUSSIONS</td>
<td>66</td>
</tr>
<tr>
<td>7.2 FUTURE WORK</td>
<td>67</td>
</tr>
<tr>
<td>Reference:</td>
<td>69</td>
</tr>
<tr>
<td>Appendix A A Brief Introduction of WAVE Language</td>
<td>72</td>
</tr>
<tr>
<td>Appendix B Screenshots from Experiments</td>
<td>76</td>
</tr>
<tr>
<td>Appendix C Abbreviations</td>
<td>79</td>
</tr>
</tbody>
</table>

iv
List of Figures

Figure 1-1 Dimensions of Software Agents Research .................................. 2
Figure 1-2 Generic Mobile Agent Infrastructure ........................................... 4
Figure 2-1 Wave Agent Execution Environment .......................................... 11
Figure 2-2 wave agent structure .................................................................. 12
Figure 2-3 Example of WAVE Paradigm ....................................................... 13
Figure 3-1 Overall Structure of Multi-Layered Dual-Monitor Architecture ...... 21
Figure 3-2 Heartbeat signals in Local Monitorial Domain ............................. 24
Figure 3-3 Implementation Details ............................................................... 33
Figure 3-4 KN Topology for Experiments ..................................................... 35
Figure 4-1 Inconsistent states .................................................................... 38
Figure 4-2 Data Structure of s_checkpoint .................................................. 39
Figure 4-3 Data Structure of ConsistReq ...................................................... 42
Figure 5-1 Message Format ........................................................................ 49
Figure 5-2 Certificate Data Structure in ASN.1 DER ..................................... 52
Figure 6-1 Wavetella components ............................................................... 57
Figure 6-2 Critical Code for Catching Output ............................................. 60
Figure 6-3 Recovery Example .................................................................... 62
Figure 6-4 WAVE code for Fault-Tolerance ............................................... 63
Figure A-1 Core WAVE ............................................................................. 73
Figure B-1 Screenshots before Crash ......................................................... 77
Figure B-2 Screenshots after recovery (After Carlsberg went down) .......... 78
List of Tables

Table 2-1 Brief WAVE Syntax ......................................................... 14
Table 3-1 WAVE System Modules ...................................................... 32
Acknowledgements

First of all, I am greatly indebted to my supervisor, Dr. Son Vuong. This thesis would not have been possible without his inspiration and encouragement. I learned a lot from Dr. Vuong in my years at UBC. I am grateful to Dr. George Tskinis for being my second reader.

I would like to thank my wife, Xiaohong, for her consistent support. I hope she will be proud of me.

Thanks also go to my friends at UBC.

Jian Chen

The University of British Columbia

Apr 2002
Chapter 1

Introduction

Mobile Agents are such a promising technology that has drawn tremendous attention from researchers in the field of distributed computing. It has been exploited in some of the applications domains.

Although several commercial or research Mobile Agent Systems have been developed, they either don’t fully provide support for fault-tolerance mechanisms, or provide only a partial solution to the problem. Mobile agents usually work in a “best-effort” fashion. Users have no idea whether those intelligent units will survive during the system/network outages.

Thus, it is generally understood that mobile agent technology will only be applied to real applications if fault-tolerance problem can be conquered. In this thesis, we are going to develop a general framework of fault-tolerance for Mobile Agent Systems.
Let's start with the concept of Agent and Mobile Agent.

### 1.1 Agents and Mobile Agents

#### 1.1.1 Agents

The term “agent” originates from “agein”, which stands for “to drive or to lead” in Greek [RP97]. Today, the term “agent” usually denotes “something that acts in an environment” [PMG98], among thousands of modern definitions. In the realm of computer science, researchers focus on the software agent, which acts on behalf of people, and helping people with active services such as searching, trading and so on [Maes97].

According to Gilbert et al. [Gilbert95], research on software agents extends in three directions: **intelligence**, **agency** and **mobility**. These directions constitute the dimensions of the entire working space of present software agent research.

![Figure 1.1 Dimensions of Software Agents Research](image)

The dimension of **intelligence** regards the research in artificial intelligence. AI pioneers target at providing agents with the ability to reason, plan and learn. The second dimension of software agent research, **agency**, relates to improving the way that agents interact with other entities in the system. **Mobility**, the last
dimension in the diagram, aims at building mobile agents, enabling the mobility of data and computation, and is also the main topic of this thesis.

1.1.2 The Mobile Agent Paradigm

A lot of effort has been made to define the term “mobile agent”. Inspired by [PK98] and [Kotz97], we suggest a possible definition of mobile agent is: a mobile agent is an autonomous and identifiable computer program that can move from host to host in a heterogeneous network under its own control, and act on behalf of the user or another entity.

There are common attributes for mobile agents, as following:

- **Mobility:**
  This is the rudimental aspect of mobile agents. Differing from a stationary agent that sits still on a local system, a mobile agent can actively migrate from one host to another, based on local computation results. The path can be preset or decided on the fly.

- **Autonomy:**
  Mobile agents should be able to perform operations in an asynchronous fashion, without intervention of any kind of centralized control. Actions are based on the current environment and its own internal state.

- **Reactivity:**
  Similarly to a person in a real world, a mobile agent must be capable of interacting with other entities, such as the environment or other agents. It also has to react punctually, based upon its own logic.

In order to build a mobile agent system that fulfills all these requirements, many kinds of different system designs have been proposed. Generally speaking, they share a common layered infrastructure, as described in Figure 1.2.
In this generic system, an application creates mobile agents, and lets them running over an Agent Platform. An Agent Platform utilizes the resources and communication channels provided by OS and Networks, to support the possible actions of these mobile agents.

### 1.1.3 Major Benefits

**Reduction of Network communication**

When there is a computation requiring a huge amount of data stored in a distributed fashion, a lot of data will be transferred back and forth, if working in a traditional Client/Server paradigm. Although there will be a little bit overhead in sending an agent’s code and related states across the network, overall the network bandwidth consumption will be reduced significantly, because an agent can simply work on site.

**Enhancement of Reliability**

Because a mobile agent is a program that can act autonomously, without interaction with the user, computation can be done without the participation of
both parties. Even if the network goes down, the agent can still continue its work on the resource, and get the results back to the user later.

**Dynamic Protocols and intelligent data**

The boost of Internet leads to an increasing number of protocols and data formats for data exchanges. If a computer hasn't had a particular protocol, then it probably can't take part in the data communications following that protocol in traditional sense. Mobile agents can carry the special protocol, and “teach” the participating hosts how to work it out. Thus, protocols become dynamic, and dynamic services become possible.

**Better Support for Wireless Applications**

Due to the attributes of wireless communication, users connect or disconnect from wireless networks, deliberately or due to the unreliability of the communication. Thanks to their ability to act independently, mobile agents can still work under poor conditions. They can still run off-line until a user re-connects to the network.

**Facilitate Software Deployment**

Mobile agents can be utilized to automate the software installation and updating process. This is a great relief for service people who are in charge of these kinds of tasks, since the user’s computing environment is quite complex, and users may not be aware there is a chance to installation or update, or not know how to do so, when technical service people are not available. Mobile agents are a perfect solution for that. They can roam around, gather information about the environment, download files, and eventually set up everything for users.

Besides those merits, mobile agents are also good at distributed computing, facilitating real-time interactions, and so on. The mobile agent paradigm opens a whole new area for researchers and scientists.
1.2 Fault-Tolerance Issues

Like any new technology, research on mobile agents is just beginning, and still far from reaching maturity. In some areas, such as fault-tolerance, security, coordination and control, the current solutions are not adequate. Our work tries to take one step further in the direction of fault-tolerance.

Because mobile agents are more independent from the availability of the communication channels, mobile agent systems are more reliable than RPC systems. However, the fault-tolerance problem is still a major obstacle that keeps the mobile agent system running only in the research labs. There are still some key issues having major impacts on the fault-tolerance mechanisms of mobile agent systems [SBS00], such as:

- **Autonomy**: Once mobile agents are injected into the network, end-users do not have much control over their execution. In case of outage, mobile agents should manage themselves, and an effective tracking system should also be introduced.

- **Asynchronous Interaction**: In traditional Client/Server architecture, both ends are involved in a synchronous interaction, and it is fairly easy to detect the failures of each. But mobile agents work in an asynchronous fashion. Thus, additional work on fault detection should be included.

- **Disconnected Computing**: If communication channels suffer from temporary failure, mobile agents should still be able to carry out their tasks, and waiting for the connection to be restored.

- **Resource Control**: Sometimes, mobile agents need to migrate to other hosts, due to the failure of their current host. Migration should be undertaken in consideration of the resources that other hosts have to offer. There should be a mechanism that ensures enough resources and maintains load balancing.
• **Migration:** In order to have mobile agent processes move to other hosts, we need to be sure of what kind of state information gets carried along with the agents. This is also important for checkpointing and failure recovery.

• **Transactional Support:** When a mobile agent is executing, it is possible that both the state of the host and the internal state of the agent itself can change. In case of any failure, these changes should be kept in a consistent and effective manner, which is quite critical for the failure recovery process.

• **Adaptive Computing:** The mobile agent paradigm ought to be active when some failures occur. A complete fault-tolerance solution must also contain the necessary support for automatic upgrading, failure reconfiguration and adaptive computing.

### 1.3 Goal

Mobile agent technology will only be practical and applied in real applications if fault-tolerance problems can be handled well.

In order to achieve this, we design a whole new infrastructure which embeds the fault-tolerance abilities into mobile agent systems. The infrastructure must be general enough to capture the general attributes of most existing mobile agent systems, and be applied to any without essential changes.

Our approach is based on a hierarchical system structure, which solves both the problems of scalability and availability. A new Dual-Heartbeat detection protocol is introduced, in order to detect any failures in the system. Moreover, an improved checkpointing algorithm is designed, which helps the system make effective checkpoints. This algorithm fits well to the hierarchical structure.
WAVE [SAPA 99] was selected as a testing mobile agent platform. It is quite general, and doesn’t have any fault-tolerance mechanisms. Due to the large amount of implementation work involved in updating the whole WAVE system, only part of the whole architecture has been completed to date. The design has been, however, partially validated during the implementation process, and will be the guidance for future improvements.

1.4 Thesis Contributions

In this thesis, a novel architecture for building fault-tolerant mobile agent systems was proposed. The architecture was partially implemented on a typical mobile agent system — WAVE, and it could be easily integrated in other mobile agent system due to its generality.

The essential idea behind the solution is to utilize a hierarchical system structure to establish the basic framework for the whole fault-tolerance mechanism, and apply centralized local control to monitor and maintain the system. The solution is simple, scalable and reliable.

Besides the general design of the system structure, a new distributed checkpointing algorithm is also introduced, which is tailored to the hierarchical structure.

Some security concerns are discussed. Because of some existing similarities between both problems, the framework for fault-tolerance can be easily extended to handle security problems as well.

At the end of the thesis, a remarkable mobile agent application—Wavetella is presented in detail. Wavetella is a powerful file-sharing application based on the WAVE system, which can help users to search the file they are interested in, and transfer the needed file. At the same time, a high-level solution for fault-tolerance is proposed and discussed.
1.5 Thesis Organization

There are 7 chapters that comprise the whole thesis.

In Chapter 2, the background information of the thesis is introduced, which includes a brief overview of WAVE and a survey of related work.

Chapter 3 outlines the basic design of the system. The dual-monitored hierarchical structure is described in detail, and a new fault-detection protocol is proposed. Also some details are given about the system initiation process.

Chapter 4 is about a new checkpointing algorithm. The algorithm is designed to serve the hierarchical structure.

Chapter 5 gives some security concerns about the system, a partial solution for security problems, and a discussion of a solution to both fault-tolerance and security problems.

Chapter 6 presents an application called Wavetella, which is a file-sharing system based on WAVE. At the end of that chapter, a new application level approach for fault-tolerance is put forward.

Chapter 7 gives the conclusion and future work.
Chapter 2

Background

2.1 WAVE – A Mobile Intelligent System

WAVE [SAPA96, 99, VI96, VM96] is a novel model and technology for highly parallel distributed processing in open computer networks. It can be applied to problems over arbitrary open, heterogeneous, and unknown network environments, without any centralized control or information.

2.1.1 Basics and Components

The leading roles of the WAVE paradigm are special recursive programs called waves, which can navigate throughout the physical and virtual network in a self-replicating and splitting manner. The logical network that waves roam around is named Knowledge Network (KN), which represents any modeled world, and can
exist on a physical network, such as the Internet. KN can be constructed with arbitrary topology where nodes and links may stand for any information.

**WAVE Host**

The logical nodes of KN are distributed on physical nodes. Hosts holding one or more logical node are called **Wave Hosts (WH)**. WHs should be able to provide the wave agent execution environment, made up of the following three major components, as shown in Figure 2.1:

**Figure 2.1 Wave Agent Execution Environment**

- **WAVE Interpreter (WI):** WI is a daemon process in charge of wave program interpretation and evaluation, KN node and link maintenance, message exchanging with other WI, and so on. It is the core of the whole WAVE system.

- **OS Interface Processor (OSIP):** OSIP assists WI to arrange enough computing resources for the waves' execution.
• **Communication Processor (CP):** The CP is responsible for providing the communication channel for WI, either via TCP or UDP. WIs rely on those channels to passing wave agents back and forth.

**WAVE Agent**

WAVE agents are not as simple as mobile codes. The conceptual breakthrough is that WAVE agents are integrated with both WAVE programs and data, and programs and data are interchangeable entities. The structure of a mobile agent is shown in Figure 2.2:

![Figure 2.2: wave agent structure](image)

Besides the Data and Code components, which are relatively common to mobile agents, WAVE agents also contain a unique Control part, which supports higher-level control of the parallel execution. The Data part contains information about the environment and the wave agent’s internal state (detail are covered in the Language section). The Code part is the program to be interpreted by WI.

Figure 2.3 demonstrates how the WAVE model works in a given scenario. There are eight logical nodes sitting on top of four wave hosts. The physical network connection is identified with solid line links, while dashed lines represent logical links, which construct the KN together with logical nodes. A wave agent is injected from node a, and then replicates and migrates itself to neighboring nodes (node b and c). This process carries on recursively, until a goal node (for example,
node e) or a goal condition is met. Then the result carries back to the original
inject nodes (node a).

![Diagram of Wave Network](image)

**Figure 2.3 Example of WAVE Paradigm**

2.1.2 Language

As mentioned, the WAVE language possesses some unique characteristics,
compared with other conventional programming languages, such as dynamic KN
creation and processing, self-navigating and self-modifying code, conciseness and
compactness of the language structure, and so on. All of these merits are rooted in
two conceptual differences. One is that interchangeable data and programs lead to
a uniform representation. The other is the dynamic generation of programs by the
contents of nodes and links, which could make it extremely flexible for open computer systems.

A simple representation of WAVE syntax is the following:

\[
\text{wave} \rightarrow \{\text{move},\}\]

\[
\text{move} \rightarrow \text{unit} \{\text{act unit}\} | [\text{rule}] (\text{wave})
\]

\[
\text{.....}
\]

Table 2.1: Brief WAVE Syntax

"wave" is made of moves, which are delimited by periods as sequential parts or by commas as parallel parts. "move" can be basic operations, or acts, or another wave controlled by rules. For detail information about the WAVE language, please check appendix A, or consult other resources, such as [SAPA 99].

2.1.3 Applications

The WAVE system has been implemented in C on UNIX and LINUX, by P.M. Borst at the University of Surrey [Borst95]. The system works on top of TCP or UDP/IP, and also includes a graphical interface based on TCL/TK.

Up to now, numerous applications have been developed in WAVE, including solutions for traditional graph and network theory problems, modeling and control of mobile telecommunication networks, simulation of communication protocols such as mobile-IP protocol, congestion control and routing protocols, and more [SCBW94, SAPA99, VM96, VI96, GLV01]. As well, WAVE is also good at integrating distributed, heterogeneous databases, simulations of road, air traffic and cooperating satellite system, and so forth.

As a new model and technology for highly parallel, distributed, and cooperative navigation, transformation, and processing of open systems, WAVE can handle
problems with more complex models of computation. However WAVE still suffers from fragileness. There is no mechanism that keeps wave agents roaming around the KN in case of node or link failures. There is even no available function that can detect such failures. A system cannot support real applications without certain Fault-Tolerance abilities. Thus, it is critical to find a solution of that problem for WAVE, and other mobile agent systems.

2.2 Related Work

In recent years, research has addressed the significance of Fault-tolerance problem in the mobile agents area, and much effort has been made to embed Fault-tolerance functions or structures into existing mobile agent systems.

The Fault-tolerance model for the Concordia system [Walsh98] is based on a transactional message queuing mechanism and a two-phase commit protocol. It does ensure the reliable network transmission of mobile agents over unreliable network connections, but it is only half way of being completion, because the mobile agent hosts are not well protected.

While in [Roth98], a protocol was presented to provide the exactly-once property into the migration of mobile agents. The basic idea behind this protocol is the replication of mobile agents on other nodes. If an active “working agent” fails, an election occurs, and a new “working agent” continues to run as a substitute.

A similar three-phase commit protocol is introduced in [Silva98]. An interesting point is the introduction of distributed storage of recovery information. However the protocol is very expensive and slow.

In [DRHAS98], a centralized-control mechanism is put forward for the fault-tolerance problem. A checkpoint manager (CM) is setup to manage fault-tolerance issues. The CM is assumed to be reliable, but that cannot overcome the problem of “Single Point of Failure”.

15
There is also some other research on centralized-control mechanisms. In [GBD00], an efficient checkpointing algorithm is proposed. The algorithm is non-blocking, and a lot of consideration has been made to reduce the communication overhead.

Generally speaking, these fault-tolerance techniques address well the problem of protecting mobile agents from host or link failures. The states of mobile agents are well kept and can be restored after a crash. But there is another side of the problem: how can we keep the network topology intact if something negative happens to the nodes and links?

Let’s take WAVE as an example. From the user’s point of view, it is a virtual Knowledge Network, an excellent abstraction that frees the user from any need to understand the underlying physical computing device. Since the distribution of the logical nodes is simply arbitrary, it is absolutely possible to keep the topology intact even if some physical nodes fail.

Previous research often neglects this aspect of fault-tolerance. Actually, protecting the logical network is also important. If parallel computing takes place over the WAVE system, the logical nodes may also contain some relevant partial information as environmental variables. It is critical to maintain those results, and migrate those nodes to other physical hosts if the current host crashes. Besides that, the node itself can be converted to a WAVE program later, and missing a node probably means the loss of some flexibility of the WAVE system.

Further more, since the mobile agent system is complete new paradigm and mobile agents could act actively, it may be possible to let mobile agents maintain a system by themselves. It is quite similar to policing in the real world. Also, it can be better if fault-tolerance mechanisms are available at both the system and the application level, in a mutually complementary style. It may then be possible to create a timely and efficient fault-tolerant solution.

From existing works, we know that good fault-tolerance architecture should be able to capture the consistent state of the system, to back up those states in a
reliable manner, to detect any failure and act promptly. These requirements inspire us to explore new fault-tolerant system architecture, instead of a simply patching to existing mobile agent systems.
Chapter 3

Hierarchical Fault-Tolerance Architecture

3.1 Centralized vs. Distributed Arguments

By examining the general survey of Fault-Tolerance techniques for Mobile Agent Systems in the previous sections, we can actually divide those approaches into two groups: Centralized Control and Distributed Control, based on whether there are any special coordinators to supervise the Fault-Tolerance mechanisms. A coordinator could be a separate process, or be embedded into the daemon process.

It is easy to see the advantages of the Centralized Control Mechanism — simplicity. We only need to setup a process on a certain node to monitor the whole mobile agent system. That monitoring process constantly dispatches some detective information units (as simple as a single protocol packet or as complex as
a mobile agent), to tell whether everything is still fine. If something goes wrong, then the monitoring process launches the recovery procedures, to restore the mobile computing environment, even crashed processes, from information that has already been backed up in stable storage. The whole procedure is simple, and causes little network traffic overhead.

The weakness and bottleneck of the Centralized Control Mechanism is also obvious — its scalability problem. If the mobile agent system spans continents, it is unimaginable to setup a monitoring process running at any single spot. The first problem is the delay in communication. It is difficult for the Monitor to tell whether an agent process is crashed, or if the reply messages are delayed, and it will not achieve good performance on Fault-Tolerance. The second problem is the workload of the Monitor. If there are thousands, or even millions of agent processes running on a huge number of nodes, no matter how powerful a host is, it is almost impossible to push such a burden on a single host or server, to handle enormous network traffic and enormous memory to maintain the information of the whole environment. The third problem is also vital to the Centralized Control Mechanism – Single Point of Failure. If the host, which the Monitor process is running on, crashes, then the whole system turns into a chaotic state. Although some Mobile Agent Systems [GBD00] have already taken that into account, and utilize some election algorithms to select a new central Monitor after the failure, it is still hard to realize that when the system runs over wide-area. It is always difficult to elect a president in a country as large as the United States.

The Distributed Control Mechanisms are good in terms of scalability. They don’t require a centralized monitor to supervise the Fault-Tolerance process. Their fatal weakness is complexity, which often leads to some very complicated implementations. Besides that, Distributed Control Mechanisms also bring up a high volume of network traffic. Usually, they require $O(n^2)$ communication messages, where $n$ is the number of processes in the system. This because every process in the system in equal, and each must know something about the whole
system or at least, all neighbors. The total number of messages increases dramatically as the number of participating nodes grows.

3.2 Design

3.2.1 Multi-Layered Dual-Monitor Architecture

In order to keep the scalability of the system and avoid the “Single Point of Failure” problem as well, we designed a new framework to handle the Fault-Tolerance Problem in Mobile Agent Systems, called: Multi-Layered Dual-Monitor Architecture (MLDMA).

Generally, we follow the idea of the Centralized Control Mechanism. We still keep the idea of having a superior process to monitor the system, but instead of having only one such process, we have included a group of Monitor processes, which acts in a distributed fashion. A hierarchical structure is introduced to coordinate these Monitor processes.
3.2.2 Monitors

Monitors not only inspect the status of the system, but also provide stable backup storage for other processes. All the processes in the system have a data structure to hold information about the Monitor. The Monitorial data structure is composed of the following:

- Information about the Monitor, which includes the address of the Monitor and other authentication information for security purposes,
- Similar information about the Vice-Monitor, which will be covered in the next section,

- The layer-number of the node, which is quite useful for the hierarchical structure.

- Other information the future use.

The classic method for monitoring the system is that the Monitor regularly sends out some probing messages to all the daemon processes in its domain, and expects echo messages from those processes. We modify the scheme a little to a new monitorial protocol, which is called **Heartbeat Monitoring Protocol (HMP)**. The Monitor no longer sends probing messages to all daemon processes, which we think unnecessary. Instead, each daemon process just checks the Monitor address information in the Monitorial data structure, and sends out a heartbeat message to the Monitor to show its availability. The heartbeat message may be just a small data packet, or a large data packet containing backup information of the process. Packet type is identified in the Header of the packet. When the Monitor receives the heartbeat signal, it knows that related process is still “healthy”. If the heartbeat message contains further information about the process backup, then the Monitor extracts the information from the message and puts it into a special-purpose local data structure. The special-purpose local data structure holds information about the subsystem that the monitor supervises.

For each process, the Monitor holds a timer for its heartbeat signal. A timeout is triggered if the heartbeat for a certain process hasn’t reached the Monitor for a long time. Then the Monitor sends a probing packet to that process, asking it to confirm. Those probing packets are sent three times, until the Monitor makes sure the process is not available anymore. Then the Monitor informs all the processes in its domain to rollback to a previous consistent state, either by using local backup files or the one on the Monitor. At the same time, the Monitor detects whether the host, which the crashed process was running on, is still available or not. If it is, then a new process is restored on that host, and the running
environment is restored to the previous checkpoint state. If the host is crashed as well, there is no way for us to keep the process on the same host, in which case a process migration is necessary. The Monitor decides, on which host the restored process will be placed, based on the workload information of each host. Such information is also kept in the special-purpose local data structure. If there is only one such process crashed, the monitor can easily restore the process on the host with the least heavy-workload host, and then update relevant information, such as TCP/UDP connections, of other processes. If there is more than one process crashes, then Monitor tries to distribute those processes on all hosts in its domain, and uses a load-balancing algorithm to make the distribution.

3.2.3 Vice-Monitors

Traditional Centralized Control Mechanisms experience difficulty when handling Single Point of Failure. If the centralized control unit, such as a Monitor in our case, crashes for some reason, it is very difficult for the whole system to come back to a stable state. To deal with this problem, we introduce another control unit, Vice-Monitor, to give a hand to the Monitor.

The Vice-Monitor is not actually less powerful than the Monitor. It is a backup for the Monitor and has more influence in some aspects. In order to be a good backup for the Monitor, the Vice-Monitor also holds a copy of all important information about the system from the Monitor.

To make the Monitor and the Vice-Monitor cooperate with each other, we conceived another control protocol, the Heartbeat Co-Monitoring Protocol (HCMP). The Monitor sends another kind of heartbeat signal to the Vice-Monitor, to identify its availability. Heartbeat signals can be a very simple data packet, or can become quite large. If the information about the system stored in the Monitor is changed or updated, the Monitor transfers the data to the Vice-Monitor. That can also be regarded as a large heartbeat signal. On receiving the
heartbeat signal from the Monitor, the Vice-Monitor replies with a heartbeat signal to report its existence. Figure 2 demonstrates how heartbeat signals work in the local Monitorial Domain.

![Figure 3.2: Heartbeat signals in Local Monitorial Domain](image)

On both the Monitor and the Vice-Monitor's side, a timer also is setup. If the Vice-Monitor fails to receive a heartbeat signal from the Monitor on time, it sends a probing message to ask the Monitor to conform its existence. After sending it out N (default value is 3) times, the Vice-Monitor takes for granted that the Monitor is no longer available, and then, overthrows the old power and takes over control. First of all, the Vice-Monitor broadcasts to all the processes in its domain, which is the same as the domain of the Monitor, to inform them of the unavailability of the Monitor. All the processes that receive this message check whether the message is from the Vice-Monitor, based on the information stored in the local Monitorial data structure. If the message from the Vice-Monitor is confirmed, each process modifies its own local Monitorial data structure, making the Vice the formal one. Then the Vice-Monitor checks whether the host, which the monitor was running on, is still available or not. If it is still working well, then
the Vice-Monitor starts another Vice-Monitor process there, broadcasts to all the processes about the new Vice-Monitor, and then becomes a Monitor.

Things become a little more complex if the host that the old Monitor resides on crashes. There may be some other process running on it as well. The Vice-Monitor then has to find out whether there are other victims. If that is the case, all the victim processes have to be restored on some other hosts, according to some kind of load-balancing algorithms.

The next challenge is how to determine the next new Vice-Monitor, and where its new home should be? A selection process must be carried out by the old Vice-Monitor (becoming a Monitor later). Several selection criteria have to be taken into account:

- Whether the Monitor can reach the Vice-Monitor easily? It is more preferable that those two monitors be connected by a fast and reliable data network. It is easier for the Monitor to backup system data to the Vice one, and also easier for the Vice-Monitor to detect outage of the Monitor.

- Whether the Vice-Monitor is powerful enough? The Vice-Monitor should have a large and reliable storage, and better computing capability. The current Vice-Monitor becomes the Monitor in the future. It handles things quickly.

- Some other criteria for future use.

In order to measure how “close” the hosts are, we invented a new conception—
**General Network Distance (GND).** Our observation is that fast transfer links are often in the same subnet, or in the small Local Area Network. The host in those networks usually has a similar IP address. Our assumption is that the closer two hosts are, the more similar their IP addresses are. If the address of host A is \(a_1.a_2.a_3.a_4\), and that of host B is \(b_1.b_2.b_3.b_4\), then the General Network Distance between A and B is this:

\[
GND = |a_1-b_1|*2^{24} + |a_2-b_2| * 2^{16} + |a_3-b_3| * 2^8 + |a_4-b_4|.
\]
The formula is not that accurate, because IP addresses are not assigned geographically. Even if $a_1$ is quite "close" to $b_1$, but their actual locations may be thousands of miles away. Nevertheless, the proposed scheme provides some advice for choosing hosts, and we can at least tell hosts are close if the GND is small enough, say, less than $2^{16}$, which means $a_1 = b_1$, and $a_2 = b_2$.

After the new Monitor starts the new Vice-Monitor process on an ideal host, it notifies the Super-Monitor about the change. (More information about the Super-Monitor can be found in the section about Hierarchical Structure). Then the Fault-Recovery process is complete.

The procedure is quite similar when the Monitor doesn’t receive any reply heartbeat for 3 times. Then the Monitor broadcasts to the public the tragedy of the Vice-Monitor. A new Vice-Monitor is entitled, and the selection procedure and possible process migrations are the same as in the previous scenario. After that, the Monitor tells all other processes in the domain to modify their information about the Vice-Monitor. Similarly, a notification also has to be sent to the Super-Monitor of the higher layer.

For the sake of better performance, we have made some other optimizations in the process.

Some centralized approaches can also deal with the failure of the Central Control Unit. Usually, there is an election process to select the next host for the "Monitor". No matter how good the election process is, it takes some time, and may lead to a chaotic state. Having taken that into consideration, we decide to have the selection process running earlier. The Vice-Monitor, the next generation leader, comes to its place soon after the Monitor seizes power. Then if the Monitor passes away, the Vice-Monitor can get control in no time, and also, some inconsistent results caused by the distributed election can be avoided.

Another optimization for fast recovery is on the Vice-Monitor process. When the Vice-Monitor process starts, it starts collecting performance statistics for each host in its domain. Currently, we just take two metrics: the echo time, and the
time to execute a little benchmark program. In the implementation, we only let the Vice-Monitor send some simple packets to each host, and measure the interval between sending messages and receiving echoes. In the future, we plan to inject a little benchmark program into each node, and see how well they do on the program; then we can have a better idea of the computing capability of each host.

Taking those figures into account, now the Vice-Monitor knows exactly how good and how “close” the other hosts are. When the Vice-Monitor becomes the Monitor, it can name the next qualified successor quickly.

3.2.4 Hierarchical Structure

Another problem that the Centralized Control Approaches face is the scalability issue. If the Mobile Agent System is running over the Internet, and some nodes may be continents away, delay and other network problems arise inevitably. Only one Monitor to govern such a wide area is not adequate.

To settle the problem, we group the processes by the General Network Distance or round-trip-time among those hosts. The processes that belong to the same group should be “close” enough, and manageable by the Monitor. The processes managed by a specific Monitor are called the “domain” of that Monitor.

There are a lot of Monitors in the system. The new problems are how to handle all those Monitors, how to let them cooperate with each other, and keep consistency.

Our solution is to group further and build a hierarchical structure. We divide the Monitors into groups, and put a Super-Monitor on top of each group. If we still have a lot of Super-Monitors, we can have Super-Monitors grouped, and assigned a Super-Super-Monitor to manage a Super-Monitor group. We repeat these grouping operation again and again, and finally, we see a hierarchical tree-like management structure coming into being. It looks like a single-backboned Internet Architecture. (Now it is time for a multi-backboned Internet, maybe we can further extend our managerial model to a multi-backbone-like structure.)
As you may recall, we keep a variable in the process's local managerial data structure to record layer numbers of the node. That variable identifies the roles of the process in the whole system. We assign the layer number 0 for the leave processes in a hierarchical tree structure, and layer number 1 for the Monitors that directly manage the leave processes, and so on. If a process's layer number is \( n \), then the maximum layer number in its domain is \( n-1 \). Control Messages are sent out with the layer number of the sending process.

With the layer number information in the control messages, we can make a authorization-ranking scheme: the message with a higher layer number has more priority over ones with lower numbers. This scheme provides more flexibility to control the entire system. The Super-Monitor overrules the messages of the Monitors that are in its domain, if it has foreseen some global benefits can be gained by doing so. Even the Monitor and the Vice-Monitor crash together, the Super-Monitor can still do some recovery work to the local Monitorial domain, and start a pair of new Monitor and Vice-Monitor processes there. How much recovery work can be done depends on how much information is backed up from the Monitor to the Super-Monitor. The processes in that domain react according to the control messages sent by the Super-Monitor, because the layer numbers in the messages are larger.

### 3.2.5 Initiation

But, how can we set up the groups and construct the hierarchical structure?

Grouping all the processes in the system is surprisingly similar to the Clustering problem in the research on Data Mining [KR90, NH94], which focuses on getting insight into the nature of grouping or structure of a data set. Many existing algorithms target at this problem, and they can be classified into two categories: Hierarchical and Partitioning Algorithms.
Hierarchical Algorithms are either agglomerative or divisive. Agglomerative methods divide each item into a separate group first, and then keep on choosing two groups and merging them into one. In contrast, divisive methods begin by putting all items into one group, and then keep on splitting them into two.

On the other hand, the Partitioning method allows the user to set the number of the groups, say k, in advance; then it tries to find the best k groups.

However, there are some specific characteristics in our process-grouping problem, leading to the development of our own algorithm to group the process and setting up the hierarchical structure. The processes and the hosts that processes are running on are not identical to each other. We need to take the "power" of the host, such as network connection and computational capacity, into account. It is similar to a classic clustering problem, but with a "weight" on each node. When we do the grouping, we try to have some powerful hosts in a group, to room the Monitor processes.

When the system starts, the daemon process on each host executes a benchmark program, and then broadcasts the result to other processes. At the same time, the "Distance" between each host is measured, either by the round-trip-time of the echo message, or by the General Network Distance. From the benchmarks, we first select the most powerful host, and start a Monitor process there. Of course, the processes that run on the same host are included in the same group. Then we try to find the host closest to the first host in terms of the "Distance". If the "Distance" is below a certain threshold, then the processes on those two hosts can join into a larger group. The threshold can be specified at the user's request, or 1/3 of the longest "Distance" by default. The algorithm continues to find the next closest host to either host in the group. If the distance to both hosts is less than the threshold, that host also joins the group. This round of the grouping process stops when the next closest Host is too far away from one of the group member. Then we exclude those grouped hosts, and launch another round of grouping processes with the rest of the hosts. This procedure can repeat again and again, till all the
processes have been grouped. When all the groups are formed, the Monitor processes can further create the Vice-Monitor for its group.

Similarly, we can use this grouping algorithm to group the groups and then setup the Super-Monitor. After certain rounds of grouping, the hierarchical structure is established.

The grouping algorithm we are using is not the best. Usually, a Partitioning Clustering Algorithm can achieve better results, but we need to specify the number of outcome groups, which is difficult to decide at the very beginning. In order to get better performance, we can use our approach first to get an ideal group number, and then use a Partitioning Clustering Algorithm, such as CLARANS in [NH94] to get an ideal grouping. If each group has a host powerful enough to be Monitor, then the outcome is terrific. If not, we use the group results from the first step.

3.2.6 Load Balancing Problem

A mobile agent system usually is not static. From time to time, new daemon processes join in, or old members leave the system. In those situations, how to make the system work in a Load-balanced state becomes an important issue.

This problem is different from its traditional load-balancing counterpart in distributed computing, which is about moving computing processes (or mobile agents) to the appropriate place. We are not trying to move the daemon processes in the Mobile Agent Systems. Instead, we just want to put the daemon processes into suitable groups, and avoid over-running the Monitor and Vice-Monitor.

There are generally two categories of load-balancing approaches, named state-based and model-based, respectively. State-based approaches balance the load according to a current snapshot of the system state, which is both hard and expensive to obtain. Better balancing results can be achieved, while model-based
approaches rely on a predefined model to predict the system state, which can be inaccurate.

Since the load-balancing problem is such a huge topic, it alone could be a master or Ph.D. thesis. We only design a very simple model-based algorithm for our system.

Each Monitor process keeps the number of processes in its domain, called the **busy factor**. When a new process wants to join the whole system, it sends out a broadcasting message to all the processes, monitoring processes or daemon processes. When a Monitor receives a probing message, it sends an invitation message with the busy factor to that new process. The newcomer then compares the first three invitations, and joins the Monitor with the smallest busy factor.

We omitted leaving processes, and just assumed the Monitors are “glad” to see their workload is decreased.

If a Monitor process cannot handle that many processes in its domain anymore, a splitting procedure is launched. The Vice-Monitor becomes another Monitor, and both Monitors send out invitations to members, to let them decide which group to join.

Basically, it is very difficult to make a well-designed algorithm for load-balancing problems here. That would be a good enhancement feature for future work.

### 3.2.7 Security Issues

Another aspect that we need to elaborate further on for our Architecture is the Security issue. Our current approach needs to be strong enough for malicious attacks. If a malicious process broadcasts to all other processes about a “crash” of the Monitor, then the Monitor will be forced out of management, and if the Vice-Monitor can’t handle all the unexpected heartbeat signals, it will lead to certain instability. We cannot simply let each process check the IP of the message sender, since the IP is very easy to fake.
A detailed discussion of security issues can be found in Chapter 5, and it is actually very interesting to see the Fault-Tolerance problem and the Security problem meet here, two major problems in the research of Mobile Agent Systems.

### 3.3 Implementation

The Hierarchical Fault-Tolerance architecture has been partially implemented on WAVE – the novel mobile agent system.

#### 3.3.1 WAVE System Structure

Wave source code is about 70,000 lines, written in C programming language. Generally speaking, it is composed of the following modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>libcp</td>
<td>Communication Processor: communication routines (TCP/UDP protocol)</td>
</tr>
<tr>
<td>libdp</td>
<td>Data Processor: data maintenance and data processing operations</td>
</tr>
<tr>
<td>libnp</td>
<td>Knowledge Network Processor: nodes, links and nodal variables</td>
</tr>
<tr>
<td>libsm</td>
<td>Storage Manager: global structure definitions, #defines, allocation and deallocation routines for all module</td>
</tr>
<tr>
<td>libtp</td>
<td>Track Processor: maintenance and processing of tracks</td>
</tr>
<tr>
<td>libutil</td>
<td>Utilities: external code which is linked to the interpreter</td>
</tr>
<tr>
<td>libwa</td>
<td>Wave Analyzer: wave string maintenance, WAVE language parser</td>
</tr>
<tr>
<td>libwgr</td>
<td>Wave Graphics: TCL/TK-based graphic subsystem</td>
</tr>
<tr>
<td>libwk</td>
<td>Wave Kernel: main function, initialization, top-level loop</td>
</tr>
<tr>
<td>libwtkgr</td>
<td>Wave Tk-Grapics: platform-dependent sym-links to TCL/TK-headers and libraries (please add proper links here for porting)</td>
</tr>
<tr>
<td>libft</td>
<td>Fault-tolerance Processor: Faults Detector (Monitor), Recovery routines, that module is newly introduced by us.</td>
</tr>
</tbody>
</table>

*Table 3.1: WAVE System Modules*
3.3.2 Implementation

The major modules that we have to play with are: libsm, libwk and libcp. These modules are the cores of the whole source code. Besides, we also introduce a new module, libft, to setup the Monitors and perform other routine checks.

As illustrated in Figure 3.3, we have some extra processes, such as Ft_client, Ft_monitor and Ft_monitor2, running beside the WAVE kernel process. Those processes simply behave as we have explained in the previous section, constantly probing the availability of the entities that they need to watch, and performing the fault-recovery procedures when failures are detected in the system.

Implementation is one of the major parts of this work, but it is deeply coupled with detailed information in the source code, and hard to elaborate here. For
further information, please check the new source code with Fault-Tolerance features.

Because of the time constraints, we didn’t finish all features of our hierarchical fault-tolerance architecture, which are presented in the previous sections. Only the bottom level of the structure is built. The outline is clearly defined, and we believe it can be done in the future.

3.3.3 Experiments

The testing was set up on 5 Linux boxes in Graduate Lab 306. Those boxes were:

- **Giraf (142.103.14.59)**
- **Falken (142.103.14.57)**
- **Koff (142.103.14.60)**
- **Albani (142.103.14.55)**
- **Carlsberg (142.103.14.56)**

All these boxes run Linux 7.1, kernel version 2.4.2.

We run the Monitor process on Giraf, and the Vice-Monitor on Falken. The other three acted as ordinary hosts.

**The knowledge Network:**

We injected the following WAVE code to create the Knowledge Network:

\[
\text{CR(}@\#e1.\text{Fe1=A.T=C.2}\#e2.\text{Fe2=A.T=C.2}\#e4.\text{Fe4=A.T=C.2}\#e6.\text{Fe6=A.T=C.}\backslash
\text{2}\#e5.\text{Fe5=A.T=C.2}\#e3.\text{Fe3=A.T=C.2}\#Fe1)\]

The topology of the network is shown in Figure 3.4. There are 6 logical nodes, distributed on 3 WAVE hosts. Each host roomed 2 nodes.
Test Case 1:

We simulated the crash of a host by using "wcleanup" on one of the ordinary hosts. Then the entire Wave Interpreter was shut down on that host. After that, the Monitor process detected failure, and recreated the crashed logic nodes on the other two "healthy" hosts. We can show the logic network is still intact after the disturbance, by injecting a wave program to show the logic nodes. At the same time, since we had developed a graphical application to display the topology of the KN, we applied that application to show that the "lost" logical nodes had been created on some other WAVE hosts.

Test Case 2:

The simulation of the crash of the Monitor process was done by using "wcleanup" on Giraf. When the Monitor Process passed away, the Vice-Monitor (Falken) noticed right away, and took over control. All the heartbeat signals were
redirected to Falken, and the system came back to a normal state. Again, we used the WAVE graphical display utility to show the KN was still intact.

For more details of the experiments, please refer to some of the screenshots in the Appendix B.
Chapter 4

Checkpointing Algorithm for the Hierarchical Architecture

4.1 Consistent system state

There are different definitions of consistent system state over different communication channels, due to the different kinds of communication quality provided. We do not try to make our algorithm suitable for both reliable and unreliable channels. Because message loss is allowed in unreliable channels, higher-level application needs to be involved to determine the actual state of the system, while it is much easier and simpler to handle consistency over reliable channels.
With reliable channels, we can define a consistent system state as in [GBD00]: a consistent system state is one in which every message that has been sent is also shown to be received in the state of the receiver.

There are two kinds of inconsistency that the checkpointing algorithm needs to be dealt with. One is received-not-sent inconsistency (Figure 4.1.a), and the other is sent-not-received inconsistency (Figure 4.1.b).

![Figure 4.1: Inconsistent states](image)

The line across the processes represents the checkpointing process. Because nodes are not synchronized, the checkpointing line is not straight. In 4.1.a, message m2 was sent after the checkpointing line, and was not registered by P1. But it is registered by P2, when received. Then a received-not-sent inconsistency problem arises. Similarly, m3 caused a sent-not-received problem in Figure 4.1.b. Figure 4.1.c is a perfect example of a consistent system snapshot.

### 4.2 Algorithm

Since we have already got a hierarchical, administrative structure, we can easily assign the Monitors some roles as Checkpoint coordinators, whose task is to make checkpoints and perform some other related checkpoint maintenance functions.

The basic procedure of this checkpointing algorithm is this:

**Phase 1)** The Monitors first broadcast to all local daemon processes, asking them to make a tentative checkpoint, then local daemon processes report their states to the Monitors.
Phase 2) From the state's information from daemon processes, the Monitors decide whether the system is in a consistent state. If consistency is already achieved, then the Monitors broadcast again, asking the local daemon processes to make the tentative checkpoints permanent.

Phase 3) If the system's state remains inconsistent, then the Monitors wait for further updates, or raise the issues to the higher-layer Monitors.

4.2.1 Phase 1

It is quite easy for the Monitors to make some initial messages for each daemon process, but how can they construct feedback information of each daemon process, which should be able to represent their states?

In order to deal with this problem, a new data structure is introduced, which is named s_checkpoint, detailed information is displayed in Figure 4.2.

```c
struct s_checkpoint {
    int CN; // Checkpoint Number
    int srDelta_local; // Local Send/Receive difference
    int foreignMsgSent; // Counter for Inter-domain sent messages
    int foreignMsgRcvd; // Counter for Inter-domain received messages
    long foreignDomainIDList[foreignDomainNumber]; // List of foreign domainID, the id of those domains who took part in the inter-domain communications.
}
```

Figure 4.2: Data Structure of s_checkpoint

**Checkpoint Number (CN)** is incremented monotonically and used to identify the checkpoints. Each daemon process keeps a current CN, which is unchanged, until the Monitor of its domain triggers a new round of the checkpointing process. CN is also included in every message sent out by the daemon process, and we can tell whether a message is late or not by checking its CN.

Another reason to bring up CN is to make the algorithm non-blocking. If certain conditions are satisfied, every single domain can make its own checkpoints
asynchronously, and all the checkpoints with minimal CN represent a consistent overall system state. For example, the CN of domain A is 7, CN of domain B is 6, and CN of domain C is 8. Then, checkpoints with CN equal to 6 on each daemon processes constitute a consistent description for the system. If the checkpointing process in domain B is blocked for some reason, domains A and C can still further make their own checkpoints, if certain conditions are met. That is quite valuable to improve the scalability of the checkpoint algorithm.

Another element we add to every message, sent by the daemon processes, is "domainID". Each Monitor has a domainID assigned, which can uniquely identify the domain that it supervises. The idea behind "domainID" is to facilitate the whole checkpointing process. Each daemon process can easily differentiate the incoming message, and make different statistical work according to the domainID. The Monitor is in charge of maintenance work. If a new daemon process is added to a domain, or daemon processes are re-grouped, the Monitor informs new members about the domainID. At the current stage, the IP address of the Monitor is used as the domainID.

srDelta_local is also applied to count the difference between the messages sent and received. It is incremented when a message is sent, and decremented when a message is received. Instead of counting every message, srDelta_local only works on local communications, which require senders and receivers to be located in the same domain; that is, they share the same Monitor. A daemon process can easily tell the “nationality” of the message by checking the domainID. If it is the same as a daemon process, then the message is a local one.

foreignMsgSent and foreignMsgRcvd are introduced to count how many messages are involved in the inter-domain communications. When a message is sent or received, the daemon process checkes whether the other party is in the same local domain. If it is, then daemon process increments or decrements the srDelta_local accordingly. Otherwise, the message is an inter-domain one, and the process increments the foreignMsgSent or foreignMsgRcvd, according to
whether the message is incoming or outgoing. At the same time, a record of the domainID of the other end is kept in \texttt{foreignDomainIDList}, \texttt{foreignMsgSent} and \texttt{foreignMsgRcvd} work together with \texttt{foreignDomainIDList}, which is a list of foreign message domainIDs kept between checkpoints. These provide further information to the Monitors, assisting them to make decisions on whether the system, which is over several domains, is in a consistent state. These two members will be re-visited in the next section.

After the introduction of each element of feedback information, there are details of phase 1:

- First, the Monitor takes a tentative checkpoint, and broadcast an initial message that contains a new CN to all the daemon processes in its domain.
- When a daemon process receives an initial message, it retrieves the CN inside, and then compares it with its own CN. If the local CN is smaller, then the daemon process makes a tentative checkpoint, increments the local CN, and replies to the Monitor with a \texttt{ready} message, which contains \texttt{s_checkpoint} data.

4.2.2 Phase 2

When the Monitor process receives a ready message, it retrieves \texttt{s_checkpoint} first. If all the \texttt{s_checkpoint} information from each daemon process is received, the Monitor first sums up \texttt{srDelta_local}. If the result is zero, then we can tell the local domain is in a \texttt{self-consistent} state. If \texttt{foreignMsgSent} and \texttt{foreignMsgRcvd} are both zero at the time, then the state of the local domain is already consistent, independent of other domains. The Monitor broadcasts \texttt{Commit} messages to all the daemon process of its domain. When a process receives a Commit message, it changes the tentative checkpoint to a permanent checkpoint. At this point, previous permanent checkpoints may still be useful for
the system recovery. We cannot determine whether some of the checkpoints can be discarded at this point, so we leave this question for future discussion.

Besides the Commit messages, the Monitor also report a ConsistReq Message to the Super-Monitor, if there is one on top of it. The ConsistReq Message helps the Super-Monitor make global decisions.

### 4.2.3 Phase 3

The algorithm is relatively complex if either foreignMsgSent or foreignMsgRcvd is not zero. That means some inter-domain communications have happened before taking the checkpoint. In order to examine whether the system is in a consistent state, we need to take a higher view of the domains. The hierarchical structure will certainly be helpful at this time.

When a Monitor finds out that there have been some inter-domain communications, it summarizes the s_checkpoint data collected, and makes a list of DomainIDs, the ID of those domains who took part in the previous inter-domain co-operations. After that, the Monitor inquires for consistency to the higher-layer Super-Monitor, with some additional information in ConsistReq, such as the following:

```c
struct ConsistReq {
    int CN;          // Checkpoint Number
    int DomainID;    // DomainID of the (Super)Monitor
    int foreignMsgSent; // Counter for Inter-domain sent
                         // messages
    int foreignMsgRcvd; // Counter for Inter-domain
                         // received messages
    long foreignDomainIDList[foreignDomainNumber];
                         // List of foreign domainID, the id of those domains
                         // who took part in the inter-domain communications.
};
```

**Figure 4.3: Data Structure of ConsistReq**

When the Super-Monitor receives a ConsistReq message, it first checks whether the domains involved are in its "super" domain. If the Super-Monitor supervises
all the related domains, it waits until other ConsistReq Messages from other sub-domains are received. Then it examines whether the super domain is in a "self-consistent" state, by counting the messages sent and received by those sub-domains, to see if the numbers are equal. If a desirable result is achieved, that is, the super domain is consistent independent of other domains, SuperMonitor broadcasts the commit message to all the member sub-domains, and the sub-domain Monitors further inform all the daemon processes to make the tentative checkpoints permanent.

If a consistent state cannot be determined by the SuperMonitor, it summarizes the ConsistReq messages within its domain, and makes another ConsistReq message of its own. The data structure of this ConsistReq is the same as that of the lower layer. This ConsistReq message is sent to the SuperMonitor of even higher levels. Those SuperSuperMonitors try to decide whether the SuperSuperdomain is in a consistent state by checking the Super-ConsistReq message. A ConsistReq message is thrown further to the next level if no decision can be made. In the worst-case scenario, the root Monitor will have to take part in the process, but either way, the decision can be made at a certain level of the hierarchy.

At the very end of phase 3, the most recent checkpoints are duplicated, and the new copies are sent to the Monitor process of the local domain. The concern behind here is that the crash of a daemon process usually comes along with the crash of the host. And if the host is out of order, then it can be impossible to retrieve checkpoint information for the daemon process that had run on that host before. So, it is quite reasonable to backup the checkpoint to some other storage to enhance availability. At the same time, we don’t need to utilize the shared file system for storing the checkpoint, thus greatly improving scalability.

### 4.2.4 Multiple checkpoints

It is possible that there may be more than one tentative checkpoint kept on the same mobile agent host at the same time. If the daemon process hasn’t received
the “Commit” message for a certain checkpoint from the Monitor, it has to keep it, even though a new round of checkpoint-taking processes may be triggered. Thanks to the reliable communication channels, that can ensure us, the message sent will be received. Eventually, a tentative checkpoint can become permanent.

When a daemon process receives a “commit” message, it changes all the tentative checkpoints, in which CN is no more than the CN in the “commit” message, to permanent checkpoints.

If there is more than one checkpoint on a mobile agent host, the Monitor of that domain keeps asking the SuperMonitor whether the CN of the last checkpoint should be kept, together with the greatest CN in its domain. This message may be reconstructed and forwarded again and again, until it reaches the root Monitor. The root Monitor then checks the minimal number of CNs, and replies with the result. The reply message eventually reaches each daemon process. Upon receiving this message, a daemon process can simply remove the old checkpoints, in which CN is less than the CN in the replied message.

At the same time, a limit is set for the number of checkpoints that can be kept on a host at the same time. If that limit is reached, then the oldest checkpoint is removed.

4.2.5 Late Messages

Sometimes, after the daemon process makes a tentative checkpoint and replies to the Monitor with a Ready message, it receives some additional messages, whose CN is less than the current CN. These messages are called “Late Messages”; they are still on their way to the destination when the daemon process starts packing up the information. Of course, the checkpoint with that CN is still a tentative one, because the numbers of messages sent and received are not balanced yet.

To handle late messages, the daemon process can simply append the late message to all the checkpoints with a CN no less than the CN in the late message. After
that, an "Update" message is sent to the Monitor, which has the same data structure as s_checkpoint, but changes the CN to the CN of the late messages, and all other statistical fields are modified accordingly.

When the monitor receives the Update message, it re-judges whether the system state is already consistent because of the additional adjustment. If it is, then a Commit message will be broadcast to all the members. If not, the monitor also has to generate an Update message, with the old CN and updated information, and send the message to the higher-level Monitor. This process may repeat, until all the related Monitors are informed, or a consistent state has been achieved.

4.2.6 CN control

When the system starts, the root Monitor broadcasts an "initial" message, to set the initial CN and the time-interval to take checkpoints. The initial CN is usually zero, and the time-interval can be quite a large number. Thus, it is still possible to let the Monitors of all sub-domains have some kind of flexibility to determine the time-interval in their domains. In this way, the system can possibly have different granularities of the checkpointing process, according to the workload attributes in different domain, and also be adjustable. The lowest-level Monitors keep a timer according to the time-intervals set by higher-level authorities, and trigger the checkpointing process by itself. After a certain period, the root Monitor broadcasts synchronization messages to keep every process in the same pace.

Currently, we only use an integer to represent the checkpoint number because it works fine in our small areas. It is quite possible to change the CN to a "dot-notation" like data structure, just as we represent the IP addresses. The Monitors of each level can append theirs own checkpoint numbers to the Global CN, which makes the CN looks like, for example, "100.22.4". In the previous example, "100" is given by the root Monitor, "22" is issued by the SuperMonitor, and "4"
by Monitor. Then it becomes much easier for us to make some kind of local rollback to certain sub-domains.

4.3 Extensions

Because the domains are quite dynamic, sometimes a new process is added, or sometimes the processes is re-grouped because of load balancing or some other event. Much change can be made, such as the change of domainID for a daemon process, because it causes some of the inter-domain messages to change into intra-domain ones, and vice versa. Then, there is a lot of work to do to modify the existing information, which has already been distributed throughout the whole system. That cannot be handled here in this Thesis, and we are looking forward to working on that in the future.

4.4 Implementation

Due to time-constraints as mentioned, the checkpointing algorithm is not implemented in the WAVE system. We expect to explore it in the near future.
Chapter 5

Security Issues in the Fault-Tolerance Problem

5.1 Problems

Though the hierarchical structure itself is already strong enough to handle "natural" outages, such as the failures of hosts or Monitors, it is still quite fragile to malicious attacks. Therefore, we still need to add security measures to keep everything all right.

Without security mechanisms, for example, a malicious host can disguise itself as a Vice-Monitor, and then broadcasts to the local domain news of a "crash" of the Monitor. Suddenly, the Vice-Monitor receives all the heart-beat signals from all other hosts. It then gets confused, because it perceives that the Monitor is still "healthy", and can't deal with all the coming signals. At the same time, the
Monitor will lose contact completely. After a time-out, it then perceives that every host in the domain has crashed, and tries to reconstruct everything. Everything is then chaotic, which leads to unexpected results.

### 5.2 Security Issues in Fault-Tolerance

Security is now becoming another significant area in the research of Mobile Agents System. The essential problems [CHK95, Chess98, Gray98] in this area are the following:

- How to protect mobile agent hosts from attacks of malicious agents,
- How to protect agents from malicious hosts,
- How to keep communication among principals secure.

These questions require us to provide a complete solution for encryption, authentication, authorization, integrity protection, and so on, in order to build a practical mobile agent system.

However, our problems set, which is to target protecting the Fault-Tolerance infrastructure from malicious attacks, which is actually a fraction of the whole security realm of Mobile Agent Systems. We can also combine our fault-tolerance architecture with some other security solutions later, however, in order to make an even more general solution for both problems. First, we talk about the stand-alone solution for our system, and extend it in the coming sections.

Here are the security issues in our subset:

- **Authentication:**
  Each entity in the system should be able to find out whether messages, such as heart-beat signals or “Monitor-crashed” messages, come from the right source.
• **Message Integrity:**
  
  We need to make sure that the messages are intact and reliable during the communication process, keeping third parties from altering the content of messages.

  In our case, we are not particularly interested in encrypting all messages sent back and forth between Monitors and other processes. We only need to make sure the data is sent from the right sender, and received by the right receiver, without being tampered with. We can, perhaps, add message encryption in the future, but it brings too much overhead if we apply it to every message.

### 5.3 Solutions

#### 5.3.1 Overview and Message Data structure

The solution can be divided into several parts such as data structure, authentication, key distribution and so on. We introduce our data structure design first, and then other branches.

Figure 5.1 is the basic design for Messages.

<table>
<thead>
<tr>
<th>BEGINNING OF THE MESSAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Header; including mode (<em>ENCRYPTED</em>, <em>SIGNED</em>, <em>NONE</em>)</td>
</tr>
<tr>
<td>Initialization vector for encryption algorithm</td>
</tr>
<tr>
<td>Certificate of sender</td>
</tr>
<tr>
<td>..... (And other Certificates from upper layer)</td>
</tr>
<tr>
<td>Message integrity code</td>
</tr>
<tr>
<td>Per-message key, encrypted with recipient’s public key (if ENCRYPTED)</td>
</tr>
<tr>
<td>Message body</td>
</tr>
<tr>
<td>End of Message</td>
</tr>
</tbody>
</table>

*Figure 5.1 Message Format*
The Message Header is at the very beginning of the message, where information is specified about security operations that have applied to the message. Currently, there are three types of messages:

- **NONE**: the message is totally unprotected.
- **SIGNED**: the message is protected by the sender’s digital signature and Message integrity code, to identify the sender and keep the message intact.
- **ENCRYPTED**: the message is not only SIGNED, but also encrypted with an encryption algorithm, such as DES.

The “NONE” message is used in a system when we are fairly confident that the network and computing environment, which the mobile agent system sits on, is secure. This can occur when we launch the system in a small LAN, or when we have already applied other security mechanisms to prevent malicious attacks. There is no security overhead in the NONE message. Furthermore, there are only “header”, “body”, and “end of message” fields in the message.

The “SIGNED” message is used for moderate protection. In this case, it doesn’t matter whether the message is eavesdropped on by a third party. We use “SIGNED” when we only need to assure the receiver about the source of the message and its integrity. The SIGNED message has two more fields than the NONE message, *Certificates* and *Message Integrity Code*. These unexplored fields are kept for ENCRYPTED messages.

In case of increased security concerns on the system in the future, the “ENCRYPTED” message is “reserved” in the design. If a message is encrypted, then an initialization vector is required for the encryption algorithm. The Per-message key is also provided, encrypted with the recipient’s public key. If the message has multiple recipients, it includes a group of Per-message keys for each recipient.
In order to generate the Message Integrity Code, a checksum is computed first with some cryptographic checksum algorithms, such as MD5. Then the checksum should be encrypted with the sender's private key to generate the final Integrity Code. An example follows:

\[
\text{Integrity Code} = E(\text{MD5(message)}, K_{\text{SenderPrivate}})
\]

### 5.3.2 Authentication and Key Distribution

Since we already have a hierarchical structure in our fault-tolerance infrastructure, we can easily develop a scheme to handle the authentication and key distribution problem.

The naming scheme is simple. Each daemon process can simply have the same name with the host that it runs on. In most cases, a host only has one daemon process running, so we can apply DNS directly to solve the naming problem.

At the same time, Monitors are assigned a new role as a Certificate Authorization (CA). They then take over digital certificate signing work in their own domain. The certificates of the Monitors are issued by Super-Monitor of higher levels. The root Monitor works just like an Internet Policy Registration Authority (IPRA). Then, the X.509 [X.509] standard, which is the dominant commercial standard, fits well into our original hierarchical structure.

When a new daemon process joins in a local domain, it generates its own public and private key pairs, and registers the public key and its identity to the local Monitor. After that, the Monitor can issue its certificate to others. The Monitors also register themselves to the higher-level Monitors; root Monitor issues the certificate for itself.

For certificate revocation, we simply follow the classical Certificate Revocation List (CRL) solution, putting all the revoked certificates on the CRL.
The data structure of the certificate, which is encoded with ASN.1 DER, is shown in Figure 5.2. Basically, the format of the certificate follows the format of the certificate in X.509. The motivation for this design is extensibility. Our system can be easily modified to be a X.509 compatible system. We are not going to explain each field in the data structure here. They are quite obvious and self-explain. For detailed information, please consult [X.509].
5.3.3 Extension

Currently, a lot of solutions for security problems of Mobile Agent Systems have been put forward. These include the MAC-based (Message Authentication Code) security model for Aglet [KLO97], a relatively complete security model in SOMA [CMS 99], and Wave Secure System (WSS) in [Peng01].

Compared with these approaches, our partial solution looks relatively simple, because we don't need to take authorization problems into consideration, and there are fewer different roles in fault-tolerance. This solution, however, is quite general and open. It can apply to a system without any security protection, and provide basic security mechanisms that keep fault-tolerance architecture safe from attack. At the same time, it can be easily extended, or combined with other security models, to achieve better protection and performance.

WSS, for example, is a comprehensive solution for WAVE. It provides a Wave Certification Infrastructure to handle the authentication problem, and has an AA Resource Manager which takes over local resource management, according to the Roles of the mobile agents. Furthermore, a Code Integrity Check mechanism is introduced to protect the integrity of the agent's code. It is a quite complete solution, and has been partially implemented.

We can directly apply WSS to our system, and make it a defensive shell that wraps around the kernel system. The only needed is to fix the message type to NONE, which removes security measures and any overhead at the same time.

In the meantime, we can also take advantage of the hierarchical structure that already built, combined with the WSS solution, and further extend our Fault-tolerance solution into a "multi-talent" solution for both problems.

In this case, since everyone is included in the hierarchical structure, the problem of inter-domain is eliminated, and the overhead of issuing Passports and Visas is crossed out. Since we already have an X.509-compatible mechanism, it is fairly
convenient to further extend it to the Internet domain, making use of the existing resources for certification and other services.
Chapter 6

Wavetella

6.1 Introduction

The first “Wavetella” was created in a course project in the fall, 2000. The term, “Wavetella”, is a combination of “WAVE”, the Mobile Agent System, and “Gnutella”[GTA], which is a widely used file sharing application. The purpose of Wavetella is to construct a distributed file-sharing system on top of a Mobile Agent System.

When browsing World Wide Web pages, we usually need a couple of search engines, such as Yahoo or Google. These search engines play important roles on the Internet, as information portals. They are quite useful when to locate some websites of interest.

Like Web-browsing, File-sharing is not as simple as FTP. File-sharing applications such as Gnutella and Napster[NPSTR], are so popular because these
applications are integrated with file-lookup services at client’s end, which facilitates the user locating the desired files.

There are some “common interests” between Gnutella and Mobile Agent Systems. For example, in WAVE, there is no centralized control from a logical point of view. We have added some semi-centralized mechanisms for better performance in Fault-Tolerance, and at the same time, Gnutella emphasizes on decentralization. In addition, when a user submits a query from a client application, the request spreads all over the GnutellaNet, the name of Gnutella network. The spreading is in a “flooding” fashion, which is quite easily implemented in WAVE, because agents can easily replicate themselves, and jumping back and forth among logical nodes. Thus, it is quite reasonable to link them together, and apply mobile agent technology to the WAVE version of Gnutella: Wavetella.

6.2 Design

The basic mechanisms of Wavetella are not very complex. User Interface, Query Interface, File Information Database and File Transfer Programs are four major components, which constitute the whole application. The structure is displayed in Figure 6.1.
The **User Interface (UI)**, first of all, should take a user's request as input and display the result or progress of the related operations. At the same time, **UI** also plays an important role as a translator in charge of interacting with WAVE Interpreter. When a query is submitted, **UI** generates a WAVE agent according to the query, and injects the agent into the WAVE Interpreter. The query agent carrying the criteria specified by the user repeatedly replicates itself, and a group of this kind of agent is created and spread all over the Knowledge Network (KN).

When a query agent travels to a Wavetella host, i.e. the host has Wavetella and WAVE installed. It will try to launch query operations on **Database** through **Query Interface (QI)**. **Database** keeps the information of the shared files on the local Wavetella host, such as filename, size, location, and so forth. **QI**, therefore, should be able to convert the agent's requests into the appropriate operations. If the database is a text file, **QI** reads through the file; or if the database is a
relational one, QI generates corresponding SQL statements, and sends them through relevant interfaces (ODBC, JDBC). If the target file is kept on the host, QI injects another message agent into the Knowledge Network. This message agent carries the file information, which includes Wavetella host name or address, file name, location, size and so on, back to the original Wavetella host.

The UI at the user end collects the information brought back by the message agents, and displays them to the user. If the user selects a file to download, the UI again generates a file-transfer-request agent, and injects it to the KN. The request agent only needs to return to the destination Wavetella host, and start the file transfer process. When the request agent arrives at its destination, it starts a separate file transfer client process, which tries to contact the corresponding process at the other end and upload the expected file for the user. The File transfer client process is launched through WAVE Interpreter.

When the request agent is created, a file transfer server process is launched at the user’s end, and waits for the incoming file. When the connection is established, the file transfer begins. After the file transmission, the server and client process stops and exits.

This system is not a 100% pure Mobile Agent System solution. Some of the programs, such as UI, are implemented in Java. This is because: although Mobile Agent Systems are very powerful, they are not a panacea, and not strong at creating GUI interfaces. Even if it were feasible to create armies of agents to carry binary/ascii file data back and forth, it would hardly a graceful and efficient solution, compared to the mature client/server routines in C or Java. Since we have a group of programming languages ready, which excel in these aspects, we can apply them directly. The question then becomes how to let a mobile agent system, say WAVE, not only keep its strength in “mobile” and “intelligent” aspects, but also extend itself to interaction with other programming languages.
6.3 Implementation

The major challenge in implementation is how to make WAVE and Java programs interact with each other. That is the key to improving the overall GUI appearance and file transfer efficiency.

6.3.1 From WAVE to Java

It is relatively easy to call Java programs from WAVE. The WAVE language already provides a special operation to access other applications, which is called “External Call”.

The syntax of External Call is “unit1?unit2”, in which “?” is the operator. “unit1” has two functions. First, it stores the arguments to be passed to the external program. Second, it takes the results of the external program after its termination. “unit2” is a string which specifies one or more UNIX commands.

We can’t generate a separate Java executable file, and the syntax to run a Java program is like “java class_name”. Therefore we need to wrap it into a shell script file, and let the WAVE program call this script file, in order to start Java programs.

6.3.2 From Java to WAVE

It is quite straightforward to create and inject a WAVE agent in Java. Java provides a “Runtime” class [JDK13], which can make external calls to other programs by using its “exec()” member functions. The problem is how, after injection, to get the information back, which has been carried along with WAVE agents.

WAVE doesn’t provide any kind of inter-process communication mechanisms. The only way to output is through standard output or writing to a file. Actually,
we can capture the output by redirecting the output to a new OutputStream Object. The critical part of the code looks like the following:

```java
Process procExecWave = Runtime.getRuntime().exec("winject");
OutputStream os = procExecWave.getOutputStream();
OutputStreamWriter writer = new OutputStreamWriter(os);
BufferedWriter m_streamOut = new BufferedWriter(writer);
```

Figure 6.2: Critical Code for Catching Output

Then we can check "m_streamOut", which is a BufferedWriter object containing the output information of the "winject" process. Applying it to Wavetella, a user can get the file information carried by message agents from the Java GUI interface.

### 6.4 Fault-Tolerance and Wavetella

We have discussed Fault-Tolerance aspects of Mobile Agent System at length. As disciples of J.H. Saltazer's [SRC84] famous "End-To-End" arguments, we consider how to raise the Fault-Tolerance functions up to the applications level at the very beginning phrase of the Wavetella design and implementation, in order to have more flexibility and better performance.

One merit of the WAVE system is that it only displays the logical Knowledge Network to the user. Users only need to interact with those logical nodes or links in the topologies, without any conception of the underlying physical layer. This facilitates enhancing the Fault-Tolerance abilities of Mobile Agent applications, such as Wavetella.
6.4.1 Static Network

Let's first assume the Knowledge Network is static, that is, once it is created, it will not change, even a little bit. First, we need to have a "probing agent", which travels through all the nodes and links of the system, and make a template for the topology. The template is kept in a Monitoring host, which is in charge of the Fault-Tolerance functionality.

From time to time, the Monitor injects a "maintenance agent", which carries the network template. Whenever it travels to a node, it replicates itself, as many as outgoing links attached to the node, and moved along the links to the next node. The nodes and links that have already been visited is marked to reduce the redundancy of the navigating processes. If a maintenance agent travels to a node that all the links are marked, then it stops navigating, and releases itself. If everything is all right, a maintenance agent does nothing else except travel.

If some parts of the topology are damaged, say some nodes and/or links fail, the maintenance agent finds out which nodes and links are missing during the navigation. Thanks to the highly dynamic nature of WAVE, a mobile agent can reconstruct nodes and links on the fly, based on the information kept in the template.

Let's assume that we have a Knowledge Network like the one in Figure 6.3, and link e, f and node E are failed at the moment, the maintenance agent first detects that link e is lost, and reconstructs link e to Node E, following the template information. This operation can't be completed, however because Node E fails as well. Then, the agent creates link e and node E at the same time, and hops to the next node E afterwards. Again, on node E, the agent notices that link f is also out of sight. It tries to recreate the link f to Node A. Unlike the previous time, node A is still "healthy". The maintenance agent only revives the link f, without the introduction of another node A.
One benefit of this solution is simplicity. Because the physical layer is completely hidden from users, we aren’t concerned with the location of the new node, such as node E in the previous example. Wavetella can be easily extended to have some kind of Fault-Tolerance abilities, if a Monitor is introduced, to constantly inject maintenance agents to the Knowledge network. The core code for that agent is surprisingly concise and simple, as shown in figure 6.4. The code can also be accessed in [SAPA 99].

Ftree_copying =
{
Any ## Any
INDIVISIBLE (Node_mark == None. Node_mark = 1).
SEQUENCE (
  ^Ftree_copying,
  ( Ftext = 'Flink' & LINK & 'Fnode' & CONTENT & 'Fcomputer' &
    ENTRY % &
    'Fstep' & Ntext % &
  OR_SEQUENTIAL (
    ( Ntext /= NONE. Ftext = '(' & Ftext & ') %
    STAY


Interestingly: even if only one node survives some serious outage, the whole topology can still be recovered back to its hey-days. Logically speaking, the user can still see the whole topology intact.
6.4.2 Dynamic Network

However, there is also some bad news for this higher-level Fault-Tolerance approach. First, it cannot handle dynamic network topology well. If a user wants to remove a link or a node, the maintenance agent will probably recreate those entities improperly. This kind of problem is not fatal to the whole system. Fortunately, we can easily deal with it by adding some "patches."

Whenever an action commits, which is associated with creating or deleting components in the Knowledge Network, the system catches the operation, and reports the new changes to the Monitor process. The information is then carried by a special "reporting" agent that finds its way out to the Monitor. When the Monitor receives the report, it changes the topology information stored, and starts dispatching new "maintenance" agents. The first "maintenance" agent is a little bit different from the others. It is also responsible for deleting the nodes or links that have been improperly restored with the old topology.

On the other hand, the Monitor can also play an active part in detecting new changes in the network. A timer, which triggers the Monitor to send off "probing" agents for updated topology, helps in this case.

6.4.3 Discussion

In our opinion, however, the application-level solution remains imperfect? Actually, there are more requirements to solve the Fault-Tolerance problem, and it is not as simple as just maintaining network topology. Sometimes the system must keep the application running without restarting at the very beginning, even when some of the participating nodes fail. This requires very fine granularities for the checkpointing algorithms and the commit/rollback mechanisms. Thus, it is very important to trace the internal state of the WAVE system, in order to provide better performance in Fault-Tolerance.
For a WAVE application, such as Wavetella, there is no sufficient method to detect the internal system state, though some of the factors are detectable. We may be able to enhance the WAVE system by adding some APIs, but that is uncertain. However, the application-level solution is still a quick and efficient patch to improve the performance.

"End-to-End" arguments are just a guideline for system design. Generally speaking, they do make the system simple, flexible and easy to implement. I think it may lead to a perfect solution, but currently, it is a pity that this is not the case with Wavetella.
Chapter 7

Conclusion

7.1 Discussions

This thesis proposes a novel solution for solving fault-tolerant problems in Mobile Agent Systems. The fundamental part of the solution is a new hierarchical fault-tolerance architecture, which is simple, scalable and reliable.

As well as the architecture design, some other related issues are also discussed, such as checkpointing algorithms and security concerns.

Due to the originality of the architecture, a new distributed checkpointing algorithm is designed, which captures system status effectively in an asynchronous fashion.
In order to complete our solution, some security issues are emphasized and related security mechanisms are introduced to preserve the system from malicious attacks.

We have done some implementation work on WAVE – a popular mobile agent system, and achieve good results through experiments, which ensure the proposed solutions are feasible and promising for future enhancement.

Wavetella, a scalable mobile agent-based file-sharing application, is another interesting part of this work. Wavetella is implemented on top of the WAVE system, and is able to facilitate user to searches and transfer files. A partial solution for application-level fault-tolerance mechanisms is proposed, though limitations exist.

7.2 Future work

Due to time limitation, not all features are implemented. The source code of the WAVE system is so huge that we do not understand every detail, and we should still keep exploring inside the “WAVE”.

The checkpointing algorithm should be further refined. It may be possible to improve the way that the Monitor differentiates between local and non-local communications. It will be also quite interesting if we can remove the constraints of reliable communication channel. Another possible improvement on the checkpointing algorithm is to make it work on dynamically grouped processes. Since the processes in a domain do not stay there indefinitely, it is difficult to judge whether the communications among processes are local or not. More checks should be made to adjust the counter accordingly.

Certainly, the security problems are also interesting. It remains to be seen whether it is possible for us to combine security problems with fault-tolerance issues, and make a general solution for building a reliable and secure system. Furthermore,
we may be able to turn security problems into a part of the universal fault-tolerance domain.

From Wavetella, we learned that it is not always efficient to do everything with Mobile Agent Systems, but we can extend the mobile agent's skills list. Possibly, more kinds of mobile agents can be introduced, they can do certain kind of work well, and they communicate efficiently through a special channel. Furthermore, they could have different priorities, which can make mobile agent systems like miniature human societies. Control agents maintain order; and worker agents carry data back and forth efficiently, just like a data packet; presentation agents are in charge of GUI. If all these were achieved, a 100% pure mobile agent system version of Wavetella could come into being.

There are always so many exciting mysteries to be explored in the mobile agents' world. We enjoy this work thoroughly.
Reference:


[GLV01] S. Gonzales-Valenzuela, V.C.M. Leung, and S.T. Vuong “Multipoint-
to-Point Routing With QoS Guarantees Using Mobile Agents”, MATA2001 –
The 3rd International Workshop on Mobile Agents for Telecommunication
Applications, Montreal, August 2001.


URL: http://java.sun.com/j2se/1.3/docs/api/index.html

Aglets”, IEEE Internet Computing, July 1997.


[NH94] Raymond T. Ng and Jiawei Han. “Efficient and Effective Clustering
Methods for Spatial Data Mining”, Proc. 20th VLDB, 1994, pp. 144-155


[PMG98] David Poole, Alan Mackworth, Randy Goebel, Computational
Intelligence, Oxford University Press, 1998

Williams (Eds.): Encyclopedia for Computer Science and Technology,

Exactly-Once Property of Mobile Agents”, Proc. of the 17th IEEE
Symposium on Reliable Distributed Systems, SRDS’98, pp. 100 – 108,
October 1998

70


[SSL] OpenSSL, URL: http://www.openssl.org/


Appendix A

A Brief Introduction of WAVE Language

The WAVE language syntax and semantics are summarized in figure A.1, the syntax and primitives in the current WAVE language definition is described. Words in small letters denote syntactic categories. Brackets represent zero or more repetitions with a corresponding delimiter; square brackets embrace an optional construct; and a vertical bar separates alternatives; “Id” means “letter or digit”

\[
\begin{align*}
\text{wave} & \to \{\text{move,}\}\}
\text{move} & \to \text{unit}\{\text{act unit}\}|\text{rule}(\text{wave})
\text{rule} & \to \text{SQ}|\text{OS}|\text{OP}|\text{AS}|\text{AP}|\text{RP}|\text{WT}|\text{CR}
\text{unit} & \to \{\text{dot,}\}|\text{N}\{\text{ld}\}|\text{F}\{\text{ld}\}|\text{C}|\text{A}|\text{P}|\text{S}|\text{L}|\text{T}
\text{act} & \to \#|\sim|/-|<|\leq|+|-|*|/|\%|\&|:\::|=|?|!
\text{dot} & \to [±]\{\text{digit}\}|\text{string}|@|[±]|$
\end{align*}
\]

a) WAVE Syntax
b) Wave Semantics

Figure A.1: Core WAVE

A WAVE program (wave) is a set of "moves", which represent sequential (period delimiter) and parallel or unordered (comma delimiter) composition of space-time actions over KN. Moves could be elementary actions over two local information units, or waves again in parenthesis, which provides recursive semantics. In such
a sequence, the first move is called “head”, and the remaining is the “tail” of the wave. The waves are always processed head by head, without the traditional program counter.

“unit” is the data component part in a wave, which can be data values or variables. Data values are represented as linear vectors of character strings, or elements being separated syntactically by semicolons. Variables include Nodal variables, Frontal variables and Environmental variables, each start with a big letter defining the variable type.

“act” works on two data “units”, neither can be empty. The result is usually kept in the first unit. The most popular act in WAVE is hop operation (“#”), which helps wave program navigate around nodes.

Wave in parenthesis may be optionally preceded by a control rule. The control rule may set the constraints to the replication of the wave by processing its head, or set logical conditions on the branches that the wave is split into. Thos control rules are:

- SQ (SeQuence), AS (And Sequential), OS (Or Sequential): The branches of the wave’s head are developed sequentially. The wave program obtained from one branch has to terminate before the next branch is activated. The “AS” and “OS” rules add additional conditions on the echoes returned from the terminated waves of the branches.

- WT (WaIting), AP (And Parallel), OP (Or Parallel): The branches of the wave’s head are developed in parallel as without a control rule. The “or” and “and” rule has the same functionalities as that in sequence rules. “WT” waits for the completion of all branches, and can be used for synchronization.

- RP (RePetition): This control is replication itself, and can be used to implement loops.
- CR (CRerate): the rule creates new nodes and links of the KN.
Appendix B

Screenshots from Experiments

Test 1

1. Screenshots before crash

a) Screenshot on Koff

b) Screenshot on Albani
c) Screenshot on Carlsberg

Figure B.1: Screenshots before Crash

2. Screenshots after recovery
Figure B.2: Screenshots after recovery (After Carlsberg went down)

The screenshot of Test 2 is more or less the same as Test 1.
Appendix C

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Authentication Center</td>
</tr>
<tr>
<td>CA</td>
<td>Certification Authority</td>
</tr>
<tr>
<td>CRL</td>
<td>Certification Revocation List</td>
</tr>
<tr>
<td>GND</td>
<td>General Network Distance</td>
</tr>
<tr>
<td>HCMP</td>
<td>Heartbeat Co-Monitoring Protocol</td>
</tr>
<tr>
<td>HMP</td>
<td>Heartbeat Monitoring Protocol</td>
</tr>
<tr>
<td>KN</td>
<td>Knowledge Network</td>
</tr>
<tr>
<td>MLDMA</td>
<td>Multi-Layered Dual-Monitor Architecture</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>WAVE</td>
<td>The mobile agent paradigm.</td>
</tr>
<tr>
<td>wa</td>
<td>Wave agent</td>
</tr>
<tr>
<td>WH</td>
<td>Wave Host</td>
</tr>
<tr>
<td>WI</td>
<td>Wave Interpreter</td>
</tr>
<tr>
<td>WSS</td>
<td>Wave Secure System</td>
</tr>
</tbody>
</table>