THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學
Pao Yue-kong Library
包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

The Hong Kong Polytechnic University

Department of Computing

Design of Fault Tolerant Mobile Agent Systems

By

YANG JIN

A Thesis Submitted in Partial Fulfillment of

the Requirements for the Degree of

Doctor of Philosophy

July, 2006

Pao Yue-kong Library
PolyU • Hong Kong

THS
LG
S1
·H577P
COMP
2006
Yang

Certificate of Originality

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signature)

YANG Jin (Name of Student)

Abstract

Combining the characteristics of distributed computing and mobile computing, mobile agent (MA) becomes a new computing model and has a great potential of being used in many areas. For MA to be widely deployed, one of the primary concerns is fault tolerance. So far, there is a lack of a systematic approach to developing fault tolerant MA systems. In this research, we target at developing a framework and its associated algorithms for providing fault tolerance in an MA system. The framework is based on a hierarchical system architecture consisting of six layers: local fault tolerance support, reliable MA migration, reliable message delivery, fault tolerant MA execution, MA group, and application-level fault tolerance. For these layers, algorithms based on the following mechanisms are developed: (1) Failure detection, (2) Checkpointing, (3) Primary backup, (4) MA transaction.

Failure detection caters for the local fault tolerance support layer. We identify the problems with the popular heartbeat failure detector (HBFD) and show that it is unfeasible for a large scale network environment. We then propose a new approach to implementing FD, called notification-based FD (NTFD). Instead of sending heartbeat messages periodically as HBFD does, NTFD sends failure notification messages only when the failure of a process is detected locally. Comparing with HBFD, NTFD achieves higher efficiency and scalability, guarantees 100% accuracy and provides a much lower probability of false detection. We also propose the design of a hybrid FD which combines the advantages of HBFD and NTFD.

Checkpointing and primary-backup mechanisms provide support at the reliable MA migration and fault tolerant MA execution layers. With respect to the checkpointing-based

approach, we first design three checkpoint placement algorithms for MA systems. We also design communication induced checkpointing (CIC) based algorithm for MA systems, which is well integrated with the independent checkpointing for reliable MA migration. For the primary-backup based approach, we propose efficient algorithms (RMAA and AMAA) for fault tolerant execution of MA by introducing parallel processing, which reduces the overhead and improves the execution speed dramatically.

MA transaction enforces the tasks in an MA application to be executed in a transactional way, maintaining the system consistency during the abort process of a failed MA, the re-execution of non-idempotent operations, and the execution of a group of MAs. Different from most existing works, which are theoretical studies on how to model and implement MA transactions, we propose a realistic solution which integrates MA transactions with the real execution environment of MAs. We adopt a two-level nested transaction model for MA transactions. Based on this model, system architecture and algorithms for transactional execution of single MA and multiple MAs using different commitment models are designed. We also propose two path-pushing style deadlock detection algorithms to detect the possible deadlock in MA transactions.

In summary, this thesis makes the following contributions: (1) A framework for providing fault tolerance in an MA system. (2) New approaches (NTFD and Hybrid FD) to implementing FD. (3) Checkpoint placement algorithms and CIC based algorithms for MA systems. (4) Efficient backup-based algorithms (RMAA and AMAA) for fault tolerant MA executions. (5) Models and mechanisms for MA transactions, with the support for deadlock prevention and deadlock detection.

Keywords: Mobile agent; Fault tolerance; Failure detector; Checkpointing; Primary-backup; Transaction.

Publications

1. Jin Yang, Jiannong Cao, Weigang Wu, Chengzhong Xu, “Efficient Algorithms for Fault Tolerant Mobile Agent Execution” To appear in *International Journal of High Performance Computing and Networking (IJHPCN).*

2. Jin Yang, Jiannong Cao, Weigang Wu, Chengzhong Xu, “Models and Mechanisms for Mobile Agent Transactions”, *International Journal of Wireless and Mobile Computing (IJWMC),* “Accepted for publication”.

3. Jin Yang, Jiannong Cao, Weigang Wu, Chengzhong Xu, “Parallel Algorithms for Fault-Tolerant Mobile Agent Execution”, Proceedings of the *6th International Conference on Algorithms and Architectures (ICA3PP’05),* 246-256, 2-5 October, 2005, in Melbourne, Australia.

4. Jin Yang, Jiannong Cao, Weigang Wu, Chengzhong Xu, “A Framework for Transactional Mobile Agent Execution”, Proceedings of the *4th International Conference on Grid and Cooperative Computing (GCC’05)* 1002~1008 November 30-December 3, 2005, Beijing, China.

5. Jin Yang, Jiannong Cao, Weigang Wu, Corentin Travers “The Notification based Approach for Implementation Failure Detector”, Proceedings of the *First International Conference on Scalable Information System (InfoScale 2006),* May, 29 2006, Hong Kong.

6. Jin Yang, Jiannong Cao, Weigang Wu, "Checkpoint Placement Algorithms for Mobile Agent System", Proceedings of the *5th International Conference on Grid and Cooperative Computing (GCC'06)* October 21- October 23, 2006, Changsha, China.

7. Jin Yang, Jiannong Cao, Weigang Wu, "CIC: An Integrating Solution for Checkpointing in MA Systems", To appear in Proceedings of the *2nd International Conference on Semantics, Knowledge and Grid (SKG2006).* November 1-3, 2006, Guilin, China.

8. Jiannong Cao, Liang Zhang, Jin Yang, Das, S.K.; "Reliable Mobile Agent Communication Protocol", Proceedings of the *24th International Conference on Distributed Computing Systems (ICDCS'04)* 468 – 475, March 23-26, 2004, Tokyo, Japan.

9. Jiannong Cao, Jin Yang, Weigang Wu, Chengzhong Xu, "Exception Handling in Distributed Workflow Systems Using Mobile Agents" Proceedings of the *2005 IEEE International Conference on e-Business Engineering (ICEBE 2005),* October 18-20, 2005, Beijing, China.

10. Weigang Wu, Jiannong Cao, Jin Yang, "A Scalable Mutual Exclusion Algorithm for Mobile Ad Hoc Networks", Proceedings of the *Fourteenth International Conference on Computer Communications and Networks (ICCCN'05),* October 17-19, 2005, San Diego, California USA.

11. Weigang Wu, Jiannong Cao, Jin Yang, Michel Raynal, "A Hierarchical Consensus Protocol for Mobile Ad Hoc Networks", Proceedings of the *14th Euromicro Conference on Parallel, Distributed and Network based Processing (PDP 2006),* 15~17th February 2006, Montbéliard, France.

12. Weigang Wu, Jiannong Cao, Jin Yang, “A Fault Tolerant Mutual Exclusion Algorithm for Mobile Ad Hoc Networks”, Elsevier, Pervasive and Mobile Computing. (Revised version submitted)

13. Weigang Wu, Jiannong Cao, Jin Yang, Michel Raynal, “Design of Efficient Consensus Protocols for Mobile Ad Hoc Networks”, IEEE, Transaction on Computers. (Under major revision)

Acknowledgements

The completion of this thesis would not have been possible without the guidance and help of my supervisor Prof. Jiannong Cao. Therefore, I would like to express my deepest thanks and gratitude to my supervisor Prof. Jiannong Cao firstly. I would like to thank him for his kind supervision and continuous support during my Ph.D. study at the Hong Kong Polytechnic University. He is a very kind and optimistic person with positive attitudes even facing difficult situations and challenges. Each time when I feel depressed during my study, he will come and encourage me to be more confident in dealing with the problems. He is also an academician with acuminous insight and strictness attitude in doing research. I have learned a lot from him, both in terms of the scientific knowledge as well as the proper way to conduct research work. The experience as his student in these three years is of great benefit to me for the rest of my life.

Another professor I would like to thank is Prof. Chengzhong Xu. Prof. Xu has given many valuable advices for this research through email. When he visited HongKong, we discussed this research project several times and he helped me to find out some important aspects for further improvement. His guidance is of great help for my study and this research.

I would like to thank many of my colleagues: Weigang Wu, Jinling Wang, Yifeng Chen, Yu Huang, Xiaopeng Fan, Hui Cheng, Kirk Huang, Yi Xie, Xiapu Luo and Jie Pan. I appreciate very much for their feedbacks, discussions, assistances, advices, and supports. They also share the pleasure of life with me in the Hong Kong Polytechnic University.

Last, but not the least, I wish to express my deepest appreciation to my parents for their support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

The objectives of this research are to investigate the issues of fault tolerance support in mobile agent (MA) systems and to develop a systematic framework for designing a fault tolerant MA system. This chapter provides an introduction to some basic concepts related to MA systems and explains the motivations for this research firstly. Then we describe our proposed framework and algorithms designed to achieve fault tolerance in MA systems. Finally we summarizes the contribution of this research work and outlines the organization of this thesis.

1.1 Basic Concepts of MA Systems

A Mobile Agent (MA) is a program that can migrate from host to host in a network of heterogeneous computer systems to execute tasks specified by the agent's owner. Characteristics of MAs include mobility, autonomy, asynchrony, encapsulation of protocols, adaptability, and support for mobile computing (disconnected operations). As indicated in [CHD95], while none of the individual advantages of MAs is overwhelmingly strong, their aggregate advantages are overwhelmingly strong. Combining the characteristics of distributed computing and mobile computing, MA provides an alternative and complementary way to solve problems in a networked environment that fits more naturally with the real world than traditional approaches. The use of MAs offers three particular benefits. First, because an MA can package a conversation and dispatches itself to a destination host, thus taking the advantages of being in the same site as the peer, interacting with the peer locally, and autonomously making decisions, an MA-enabled

algorithm can reduce remote communications and make use of the most up-to-date system information for decision-making. Second, MA can encapsulate protocols and can be dynamically dispatched, allowing us to design a proactive and scalable program adaptive to the current system configuration and state. Third, MA can support disconnected operations by carrying out tasks for a mobile user temporarily disconnected from the network. After being dispatched, the MAs can operate independently, asynchronously and autonomously.

MA can be employed in many different applications, including information searching and filtering, network management, mobile communication, e-commerce, negotiation, and parallel processing [CHD95, HAG98, HIL97, SIL97, DAS99, XU00a]. It also has been found especially suitable for structuring network and distributed services that require intensive remote real-time interactions [FUN99, XU99, CAO01a]. Furthermore, strategies are proposed for using a collection of cooperating MAs as a means for maintaining inter-site relationships to solve various problems including dynamic, peer-to-peer networking and distributed coordination [MIN99, SCH97, CAO01b].

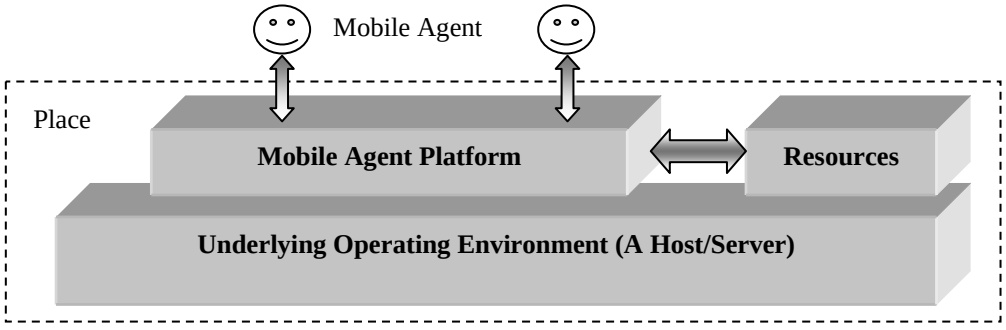


Figure 1.1 Place, MA platform and Host/Server

An MA cannot carry out its execution without the help of the systems it running on. An *MA Platform* (MAP) manages the lifecycle of MAs, supporting agent creation, interpretation, execution, migration, and termination, as well as location tracking,

communication and management. It also provides a gateway through which agents can access resources available at a host. *MA ID* is a unique value and it is required for identification, management operations, and locating. MAP, host and related resources form a *place* for MAs, which is a context in which an MA executes. Figure 1.1 illustrates the relationships between the MA, place, MAP and host.

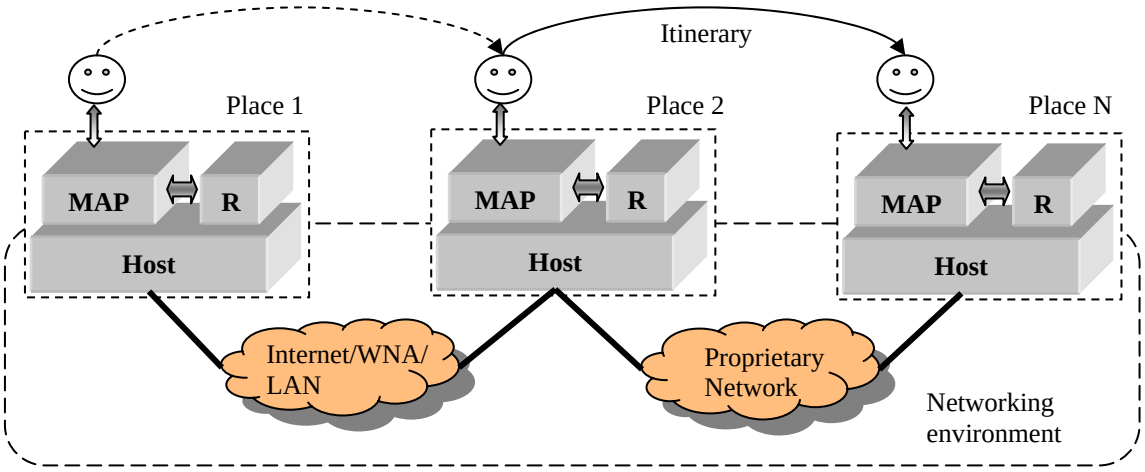


Figure 1.2 Mobile Agent System

An MA system (Figure 1.2) can contain one or more places and a place can host one or more MAs. Since each MA platform will be associated with a host or server and form a place for MAs, we hereafter make no distinction between place, MA platform, server and host. There are two ways to determine order in which an MA visits places: *static* and *dynamic*. The static visiting order is predefined in an *itinerary* which is stored in an MA. The dynamic visiting order has no predefined itinerary. The next place to be visited is determined by an MA according to the execution context on the current place. Figure 1.2 also illustrates the running environment for an MA. A group of hosts are connected by communication networks, which can be Internet/WAN, LAN or proprietary networks. The communication networks can provides various degrees of QoS.

Many platforms, languages and toolkits exist to develop MA code [GRR01, WAN99]. Until recently, MA systems were developed primarily on script languages like Tcl [GRR95], Telescript [WHI94], and Tacoma [JOH95]. The latest proliferation of MA technology is mostly a result of the popularity of Java. The Java virtual machine and its dynamic class loading model, coupled with several other Java features, most importantly serialization, remote method invocation, and reflection, facilitate the construction of first class MA systems. Many systems have been developed, most notably including Aglet [LAN98], Ajanta [TRI99], Concordia [WAN97], D'Agent [GRR98], Mole [STR98], Odyssey [WHI94], and Voyage [OBJ]. MA systems that provide Web connectivity have also been developed, allowing Web servers to host MAs [FUN99]. In our research work, we test and evaluate our proposed algorithms and software prototype using Naplet [NAP], a flexible, Java-compliant MA system developed by us.

1.2 Fault tolerance in MA Systems

It has been agreed that MAs are essential for structuring and coordinating distributed applications running in the error prone environment of open and distributed networking systems [CHD95, LAN99, CAO01a]. Many potential applications such as e-commerce, systems management, and active messaging [STR00] require MA systems to be able to tolerate some failures in order to provide a better quality of service. This makes fault tolerance a key issue in designing an MA system, with the goal being to guarantee that the execution of MA can be recovered and continue once a failure occurs.

The particular characteristics of MAs as mentioned in Section 1.1, however, pose new challenges for fault tolerant computing. The challenges lie on two aspects. Firstly, traditional fault tolerance approaches in conventional distributed systems are neither directly applicable nor sufficient for MA scenarios because their models are for stationary

processes and focus only on maintaining their states at local sites. However, MAs travel over a network along predefined or dynamically determined itinerary, bring their states and data with them, and are asynchronous and autonomous. Therefore, the following features which distinguish MAs from processes in conventional distributed systems should be considered in the design of the fault tolerant algorithms for MA systems. (1) Mobility: An MA may travel a large network and visit a large number of hosts, resulting in a very long execution period; message delivery for MA is also significantly different. (2) Autonomy: An MA may decide and change its activities dynamically, all by itself. (3) Encapsulation of protocols: An MA may carry protocol related information during its migration, which may bring convenience for the algorithm design. Secondly, existing works on fault tolerance in MA computing treat various fault tolerant issues in isolation and there is a lack of systematic approach to designing fault tolerant MA systems. This research is motivated by these observations and seeks to develop a systematic framework for providing fault tolerance support to MAs. New and integrated methods are needed to provide fault tolerance support at various levels of an MA system.

Before we develop the system architecture for a fault tolerant framework, we must analyze the MA systems and clarify what kind of failure we should tolerate. As shown in Figure 1.2, an MA system consists of MAs, places and networks. All of them can suffer failures. In this research, we focus on the fault tolerance for the execution of MAs. As to the fault tolerance of places and networks, plenty of solutions have already been proposed, such as the backup servers and redundant links or routes. Therefore, we assume the failed host or communication link will recover later, or can be substituted by their backups, so as to provide a workable infrastructure for the execution of MAs. However, the failures of places and networks will affect MAs. For example, if a host failed, all the contents (including the running MAs) in its RAM will be lost. As a result, eliminating these

affections is our concern. To have a better understanding of the fault tolerance for the execution of MAs, we firstly investigate the computing model of MAs.

The computing model for an MA is illustrated in Figure 1.3. Failures that can happen in the operations of an MA include: (1) Failures in message passing: Messages exchanged between MAs may be delayed, duplicated or lost; (2) Failures during the migration of an MA: an MA can be lost or damaged while migrating between hosts due to host failure or various network failures; (3) Failures during the execution on a host: an MA can fail during its execution on a host for a number of reasons, for example the crash of the MA itself or the failure of a host.

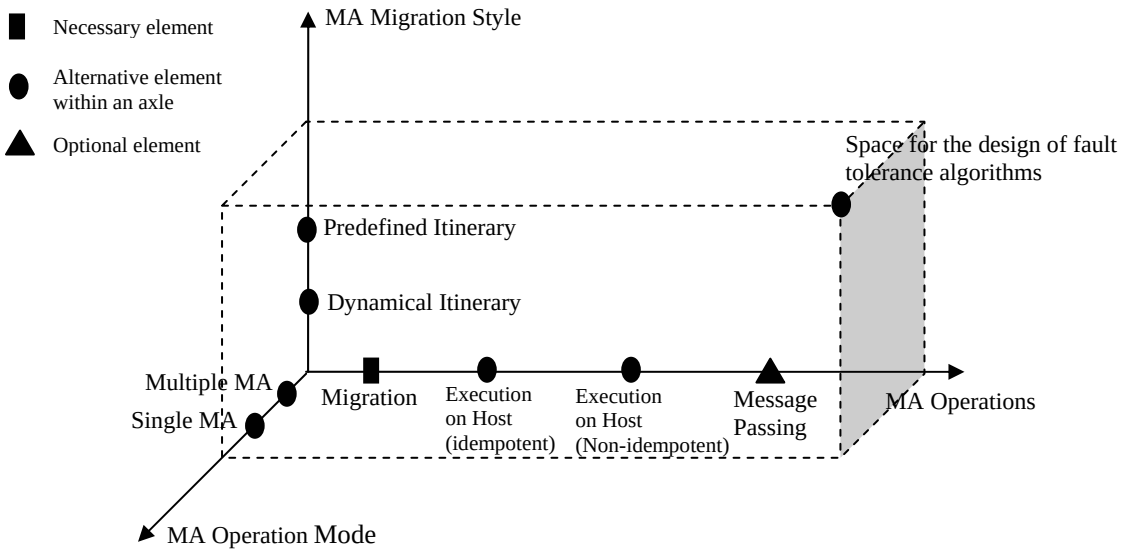


Figure 1.3 MA Computing Model

In distributed systems, to get a better grasp on how serious a failure actually is, failures are classified into a hierarchy of increasing difficulty level, which includes (1) *crash failure* (fail-stop): a processor fails by stopping to respond starting from a single point in time; (2) *send-omission failure*: a processor fails either by stopping forever, or by forgetting to send some of its messages; (3) *general-omission failure*: a processor fails either by stopping forever, or by dropping messages to be sent or to be received; (4) *timing failure*: messages

may arrive too early or too late or are lost, or processor crashes; and (5) *arbitrary failure* (Byzantine failure): processor can behave in whatever way it likes. In this research, we consider the failure during the migration of an MA to be the *general-omission failure* and the failure during the execution of an MA to be the *crash failure*. Depending on the different system models, the failure of message passing can be either *send-omission failure*, *general-omission failure* or *timing failure*.

In the following section, we describe in greater detail the requirements on fault tolerance support that an MA system must satisfy, describe a hierarchical architecture (framework) for designing system or fault tolerance support, and describe the components of the proposed system.

1.3 A Framework for Designing Fault Tolerant MA Systems

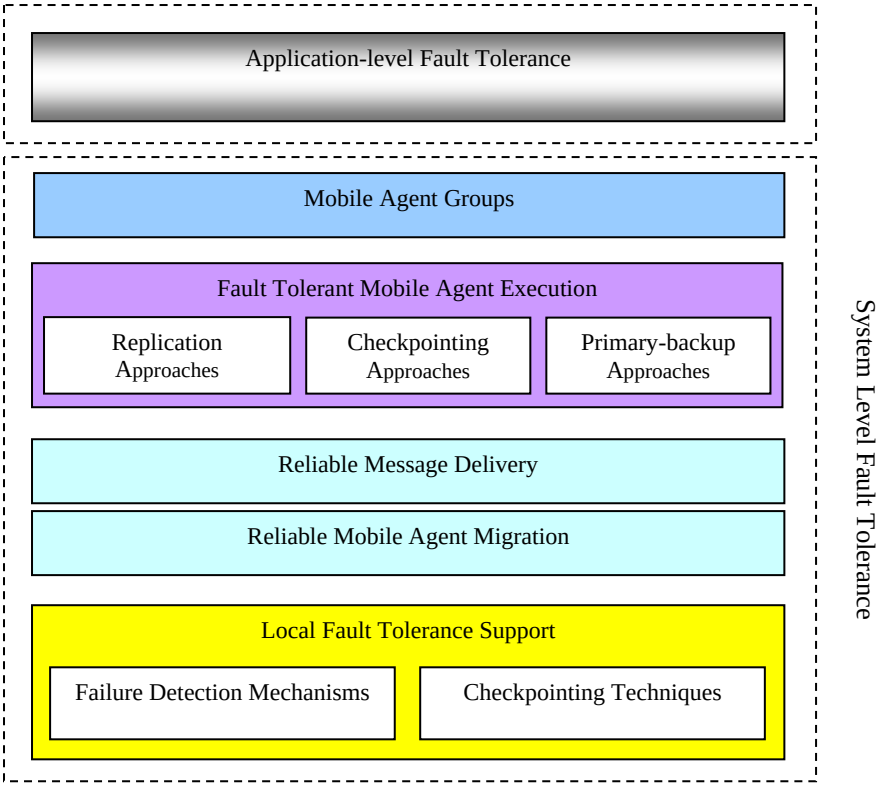


Figure 1.4 Hierarchical Architecture

The framework for providing fault tolerance in an MA system that is described in this section is based on a hierarchical system architecture consisting of six layers (Figure 1.4). It is a generic architecture model wherein each separate layer is devoted to a different function. In this respect it is novel. Figure 1.4 illustrates our proposal. The functions of a layer are built on top of the lower layers.

1.3.1 Local Fault Tolerance Support

The bottom layer of the proposed framework provides local support for fault tolerance and support for the algorithms in the upper layers. This bottom layer undertakes two main tasks. First, it uses checkpointing to guarantee the persistence of states associated with MAs. Checkpointing stores the state and data of an MA into stable storage and recovers the MA if it fails. For example, in the event of an emergency shutdown, the system may implicitly checkpoint all locally existing agents [PEI97]. Similarly, to avoid the loss of migrating agents that might occur as a result of a network link failure or host crash, the agent is checkpointed to a stable storage before being dispatching itself to a new site and this checkpoint is retained until the agent arrives at the new site. The second task of the bottom layer is to provide a failure detection mechanism on the local MA platform. Normally, fault tolerance mechanisms are triggered when a failure is detected but currently available systems provide little support for MA failure detection. In this thesis, we present a new failure detection mechanism which is especially feasible and efficient for MA systems.

1.3.2 Reliable MA Migration

Above local fault tolerant support is the reliable MA migration layer. This layer is responsible for ensuring the safe migration of an MA from the current host to a new host. This layer provides a transport service that guarantees reliable migration. The functionality

of this layer depends on the services provided by the bottom layer and the communication networks. We have mentioned that checkpointing can provide support for reliable migration.

1.3.3 Reliable Message Delivery

This layer keeps track of the movement of MAs and ensures that a message destined for an agent will be safely delivered to it. Most of the available MA systems support remote agent communications but few take fault tolerance into account. In this area there are two primary issues: tracking the location of MAs and delivering messages to them. Most existing MA message communication facilities rely on one of two approaches: a central home server approach or a forwarding pointer based approach. Neither can guarantee reliable message delivery without synchronization between the message-relay objects and the MAs. Based on our previous work on a flexible and adaptive mailbox-based communication scheme [FEN01], we have developed protocols for fault tolerant message delivery that are also efficient [CAO04b, CAO05]. The mailbox-based scheme allows decoupling between the mailbox and its owner agent, providing abstractions by separating different concerns in tolerating agent migration faults and message passing faults.

1.3.4 Fault Tolerant MA Execution

The main purpose of this layer is to guarantee the livingness of the MA during its execution. To do this we can adopt three types of fault tolerance technologies: replication, checkpointing, and primary-backup. Replication based approaches have been well and widely studied, which require both the MA and the MA platform to be replicated and execute several MAs simultaneously for the same task. The biggest advantage of replication based approaches is that the execution of an MA based application will not be

blocked even if some replicated MAs fail. However, maintaining the consistency of the replicated MAs is costly.

In this thesis, we focus on the checkpointing and primary-backup based approaches. Checkpointing based approaches rely on the checkpoints made by the bottom layer. Once a failure is detected, the saved checkpoint will be incarnated as a new MA so as to continue the execution. Primary-backup based approaches must maintain one or several backups for the working MA. The working MA and the backup will each detect if the other fails and create a new MA to replace the failed one. Both checkpointing and primary-backup rely on the failure detection mechanism provided by the bottom layer. A common problem for checkpointing and primary-backup based approaches is how to maintain the property of exactly once execution during the recovery. As we know, an operation can be classified as either an idempotent or non-idempotent operation. Idempotent operations such as reading an entry from a database can be carried out many times without changing the system's status. No inconsistency is introduced into the system. Non-idempotent operations, however, such as the operation of withdrawing money from a bank, will change the status of the system. If a process fails during the execution of a non-idempotent operation, it is not possible to simply redo it because to do so would produce an inconsistent result. Therefore, to prevent such inconsistencies is one of our main tasks. In this thesis, we preserve the consistency during recovery by MA transaction.

1.3.5 MA Group

The MA group layer supports cooperation between MAs by providing means for reliable group communications [MAC01] and transactional execution. One widely used mechanism for supporting reliable and consistent execution of collections of cooperating processes in distributed systems is the process group. The MA group similarly provides fault tolerant

message delivery and guarantees consistency for a collection of cooperating MAs. The MA group concept, however, must also deal with the issue of the mobility of a group's member agents. Although the concept of mobility group has been proposed, no realistic algorithms and protocols have yet been developed. Based on our work on reliable multicast protocols, MA multicast communications and a mailbox-based message delivery scheme, we have developed reliable multicast protocol for MAs [CAO04b, CAO05]. In this thesis, we continue to seek algorithms and protocols for transactional execution for an MA group.

1.3.6 Application-level Fault Tolerance

This layer is the highest layer in the hierarchy. This layer builds its own application dependent fault tolerance schemes using the mechanisms provided at the underlying system layers. The fault tolerance mechanisms in this layer depend on the specific applications so they cannot be generalized to all the applications. Since this research work aims to seek system level fault tolerance approaches, we will not design algorithms for a specific application. However, during our research work we did notice that some of our proposed algorithms were particularly efficient and suitable for a class of applications. For example, some algorithms are especially efficient for applications whose operations are all idempotent. So as not to ignore the background and context of different types of applications and to provide some guidance for the design of the algorithms, we thus classify the applications as either operation-idempotent applications or as operation-non-idempotent applications.

1.4 Contributions of the thesis

In this section, we describe our contributions in this thesis, which consists of several groups of algorithms designed for the fault tolerance of MA systems. Before the design of the algorithms, we have identified the following four issues as essential for our framework:

- *Efficient primary-backup based algorithms for fault tolerant MA execution*
- *Integrated checkpointing-based algorithms for fault tolerant MA execution*
- *Efficient and scalable failure detection mechanism*
- *MA Transaction*

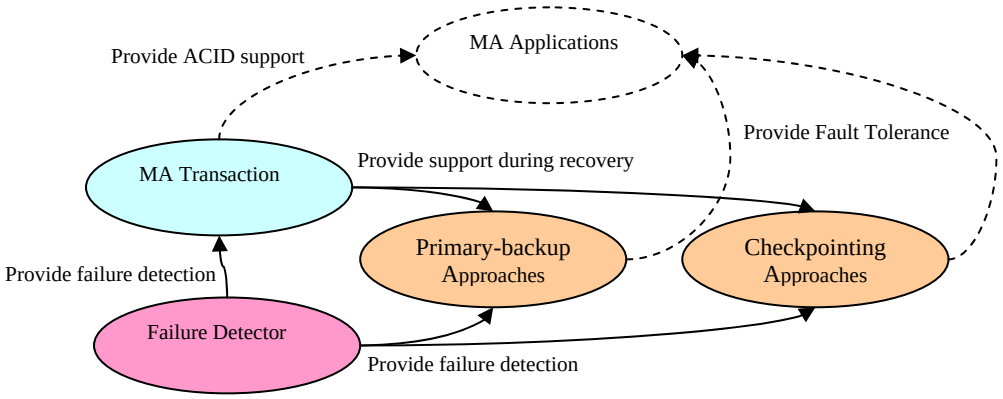


Figure 1.5 Relations among the Issues

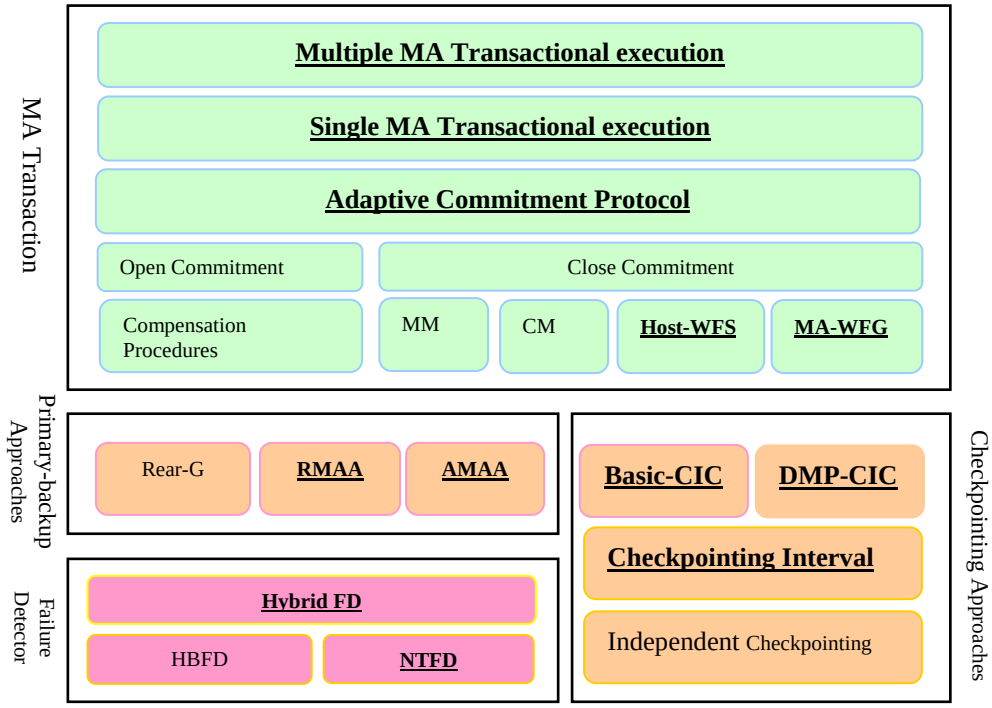


Figure 1.6 Our Proposed Algorithms for the Framework

The colored ovals in Figure 1.5 show how these issues are related. Checkpointing and primary-backup techniques are two widely adopted fault tolerant schemes for the fault tolerance of MAs. Failure detector is a building block providing the failure detection service for both checkpointing and primary-backup based algorithms. MA transaction guarantees the system consistency for the two schemes during their recovery and the ACID [GRJ93] properties for MA based applications. Corresponding algorithms have been proposed for each of these issues. They are arranged into four blocks in Figure 1.6, and each block corresponds exactly to one issue. We describe these issues and our designed algorithms in the following subsections.

1.4.1 Primary-backup based Algorithms for Fault Tolerant MA Execution

Primary-backup is a popular scheme for the MA fault tolerance [JOH95, TAO00, KOM02, PEA03, PLE03]. In our research work, we focus on how to make it more efficient. The basic idea of the primary-backup based approaches is to maintain some backups of the working MA. The working MA and the backup will each detect the failure of the other through FDs. Once a failure is detected, a new MA is created by the living working MA or backup to replace the failed one. Existing primary-backup schemes have shortcomings. The first is the overheads caused by the backups. The backups do nothing except monitoring and keeping synchronization with the working MA, which increases the system's overhead and dramatically slows down the execution speed. The second problem comes from the failure detection mechanism (heartbeat based FD: HBFD) adopted in existing works. HBFD requires that the peers keep on exchanging the heartbeat messages. This characteristic not only incurs message costs and causes false detection, but also needs modification to cater for the MA environment, because HBFD cannot work properly due to no message can be delivered during the period of migration for an MA (we call this period

the dumb period). To solve these problems, parallel processing is introduced in our newly propose primary-backup based algorithms (RMAA and AMAA). This reduces the system's overheads and improves the system's performance. We also design the handover procedures to solve the dumb period problem. Our newly proposed notification based FD (NTFD) is integrated with RMAA and AMAA to reduce the message cost and the false detection caused by HBFD.

1.4.2 Checkpointing-based Algorithms for Fault Tolerant MA Execution

Checkpointing is an operation in which the real-time execution state of a process is saved to stable storage [DAV69]. When an MA fails, it need not restart from the beginning but from the latest checkpoint. It is naturally to adopt checkpointing in MA systems: serializing an MA for the migration to the next host effectively constructs a checkpoint. Different checkpointing libraries have been developed for various platforms [PLA95, SIV98]. Considering our MA platform is a Java based system, we employ the serialization mechanism provided by Java to construct the checkpointing primitives.

Checkpointing in MA system has some similar features as the process migration via checkpointing, but they are different in several aspects. Basically, process migration aims to balance the workload among a group of hosts (usually in a LAN), while an MA can travel a very large area and retrieval/update the information on the visited hosts. The migration of a process happens during or before the execution of the process and is triggered by a central or distributed algorithm (i.e., load balance algorithm), while an MA fully controls its migration along an itinerary and its migration starts after it finishes its execution on a host. Therefore, it is necessary to redesign the checkpointing based algorithms in MA systems.

For checkpointing-based approaches, independent checkpointing is a popular solution to keep the persistent state of an MA. However, checkpointing involves high costly I/O operations, so how to achieve a good trade-off between the checkpointing cost and system performance is a critical issue. Excessive checkpointing would result in the performance degradation during normal operation. On the contrary, deficient checkpointing would incur expensive recovery cost upon a failure. There has been much research on how to determine the optimal checkpointing interval [YOU74, MAN75, GEL78, GEL79, TAN84, SHI87, NIC90, PAG01], which is referred to as the checkpoint placement problem if the optimal checkpointing interval cannot be achieved. Equidistant and equicost are the two well-known checkpointing strategies. The equidistant strategy considers a deterministic productive time between two neighboring checkpoints, while the equicost strategy allows a checkpoint to be made when the expected re-execution cost is equal to the checkpointing cost. With the occurrence of failures following a Poisson process, these strategies become identical and will result in the optimal checkpointing interval [TAN84] in conventional systems. Currently, for MA systems, there is no work done on how to determine a proper checkpointing interval for an MA and no study on how the above two strategies can be applied. In this research, we firstly analyze the behaviours of an MA system. Based on the analysis, we find that it can be modelled as a homogeneous discrete-parameter Markov chain, which is different from the models used in conventional systems. Therefore, the analytic methods and corresponding results for conventional systems cannot be adopted directly for an MA system. Based on our proposed model, we study the equidistant and equicost checkpointing strategies and propose three checkpoint placement algorithms for MA systems. Through simulations we evaluate the performance of our proposed algorithms and the result shows that the equicost strategy based algorithm is most suitable for an MA system.

If an MA group is concerned, independent checkpointing may suffer the domino effects. We introduce communication induced checkpointing (CIC) in MA systems. CIC has been proven to be a flexible, efficient and scalable scheme. We propose a deferred message processing CIC algorithm (DM-CIC) to support the recovery of a group of MAs. DM-CIC algorithm not only establishes the consistent recovery lines efficiently but also integrates well with the independent checkpointing for reliable MA migration.

1.4.3 Efficient and Scalable Failure Detection Mechanisms

Failure detection mechanism is basic to fault tolerance. Failure detector (FD) is an important component of the bottom layer of our framework. On the one hand, the fault tolerance mechanisms are triggered when a failure is detected; on the other hand, if an FD reports a false error, it will produce the duplicate computation. Therefore, an accurate and quick failure detection service is highly desirable. However, the running environment (e.g. Internet or LAN) of an MA is characterized by asynchrony (no bound on the process execution speed or message delay) and this makes it impossible to detect precisely whether a remote MA has failed or has just been very slow. Consequently, we can provide the failure detection service at only a certain level of quality. Recent years have seen many studies on the QoS and corresponding implementations of FDs [BER03, CHE02, BER02, GUP01, REN98, FAL05]. The QoS metrics include accuracy, quickness, efficiency and scalability. The accuracy of an FD refers to how well an FD provides accurate failure information. The quickness reflects the speed of an FD for failure detection. High efficiency requires the FDs to have low overheads. Finally, scalability is desirable for the large deployment of FDs in distributed systems.

Heartbeat-based failure detectors (HBFD) are widely used in existing works. The reasons for the popularity of HBFD are that its model and implementation are simple and it

preserves the strong completeness property. But HBFD can operate well only under certain conditions: (1) clocks are synchronized between the monitored process and the FD or the clock shift is bounded; (2) the heartbeat message should be delivered to the FD within a fixed period of time. For the first requirement, even if it is not unrealistic to deploy a clock synchronizing network, the cost could be very high. The second requirement actually implies that a synchronized system is required, which can not be achieved in an asynchronous distributed system. The problems with current HBFD implementations motivate us to design an alternative approach to implementing FDs. The model used in existing works considered a process and its local running environment as a single entity. In our research, we consider a more general model where the processes (MA) and the hosts on which the processes are running are separated, and an FD is installed on each host to monitor the processes on that host. This model covers the existing single entity model as a special case but allows us to propose different methods for implementing FDs. All the results of QoS study on FD in the literature are also applicable in the general model.

The alternative method we proposed uses a notification based approach (NTFD). NTFD works by letting the FD installed on the host send notification messages to the interested entities once a failure of the local process (MA) is detected. It has the following features: (1) Efficiency: message cost is low. (2) Scalability: message cost (network load) increases linearly with the number of process failures, no matter how many processes and FDs are involved. (3) Accuracy: once a failure notification message is sent out, the monitored process is really failed. NTFD will not make false detection. (4) Simplicity and easy to implement: No synchronized clocks are needed and NTFD can be implemented by using on-the-shelf hardware or software components. However, one problem with NTFD is that it cannot maintain the completeness property if the notification message cannot reach the target processes, whereas HBFD can preserve the completeness property well. Considering the advantages and disadvantages of the NTFD and HBFD approaches, we propose a

hybrid approach in which notifications are incorporated in heartbeat mechanism to improve the overall performance.

1.4.4 Models and Mechanisms for MA Transactions

MA transaction allows the transactional execution for the tasks carried out by a single MA and multiple MAs so as to guarantee the *Atomicity, Consistency, Isolation,* and *Durability* (ACID) properties [GRJ93] for the tasks. *Atomicity* enforces that all operations of a transaction must be done or none of them. *Consistency* guarantees that a system is transformed from a consistent state to another one. *Isolation* ensures that each transaction execution be isolated even though transactions are executed concurrently. *Durability* is the condition stating that once a transaction commits, its effects on the system are durable. For an MA system, *consistency* is necessary to maintain a global consistent state for the rollback procedure for a failed MA; *atomicity* can help one or a group of MAs either finish all of the assigned tasks or none of them; *isolation* supports multiple MAs to execute in the same MA system in a serialized way, and *durability* makes the final committed computing results available. Therefore, MA transaction can preserve the system consistency during the abort process of a failed MA and provide support for the re-execution of non-idempotent operations, and the execution of a group of MAs.

Existing work on MA transactions pays more attention on the theoretical discussions of MA transactions, such as the ACID properties, open/close transaction models [SIL97, SHE01, PLE03]. Few works consider the issues such as the system architecture (model) and performance evaluation when to implement MA transaction in a real MA execution environment. For an MA application which needs MA transaction support, normally the system below the MA platform already provides the local transaction support. Such transaction support mechanisms have been studied extensively for decades and have been

widely deployed, especially in centralized or distributed database management systems (DBMS). Hereafter we refer to the systems that already provide support for transactions as the base transaction systems. Most base transaction systems support Client/Server computing model. MA as a new computing model converts the repetitive remote communications between clients and servers into local operations by encoding the tasks of the client side into an MA and launching the MA to the server side, so as to improve the efficiency of base transaction systems dramatically. From above analysis we can draw a conclusion that an MA transaction should be deployed in the context of base transaction systems so as to integrate with these de-facto technologies. Accordingly, our MA transaction model is based on the support of base transaction systems, where the MA system acts as an upper layer and has no substaintial differences from other applications running on the base transaction systems.

In an MA transaction, the MA may travel in a large network and visit different base transaction systems that have different interfaces and provide different services, which implies that the execution time of an MA can be very long. On another hand, two phase locking (2PL) is a popular concurrency control protocol and can guarantee the strict isolation between different transactions. Long execution time plus 2PL can easily produce deadlock. To date there has been no study evaluating the probability of the occurrence of deadlock, and the deadlock prevention or detection in MA transactions. With these problems in mind, we propose a system model and architecture for MA transactions which integrates the base transaction systems. We also design an adaptive commitment model with algorithms for the transactional execution of both single and multiple MAs. Through simulations we evaluate the probabilities of deadlock occurrence and the impact of deadlock on the performance of MA transactions. We then propose algorithms for deadlock prevention and algorithms for deadlock detection (Host-WFS, MA-WFG).

1.4.5 Summary

In summary, we have developed the following framework, algorithms and software packages for the fault tolerance of MA systems.

1. A framework for providing fault tolerance in an MA system, which is based on hierarchical system architecture consisting of six layers.

2. A new approach to implement FD: notification based FD (NTFD). Compared with HBFD, NTFD exhibits a much higher probability of achieving a better balance between completeness and accuracy properties. We also propose the design of a hybrid FD which combines the advantages of HBFD and NTFD.

3. New efficient primary-backup based algorithms (RMAA and AMAA) for the execution of MAs have been proposed. Failure detection service which is specially designed for MAs has been integrated into these algorithms.

4. Three checkpoint placement algorithms for the independent checkpointing of single MA. Basic-CIC algorithm (Basic-CIC) and deferred message processing CIC (DM-CIC) algorithm are proposed.

5. New model and mechanisms for MA transactions. Algorithms for transactional execution of single MA and multiple MAs using different commitment models have been developed. Deadlock prevention and deadlock detection have been studied and an efficient deadlock detection algorithm has been presented. The simulation results show

our newly designed deadlock detection algorithm is particularly efficient for MA transactional execution.

1.5 Structure of the Thesis

This chapter introduces the background, motivation and contributions of this research, explains the layers of the hierarchical architecture in our proposed framework, and the algorithms designed for this framework. The remaining chapters of the thesis are organized as follows.

Chapter 2 briefly presents the literature review of the relevant topics and provides some necessary preliminary knowledge of failure detection mechanism, fault tolerance techniques (replication, checkpointing, and primary-backup) and transaction processing.

Chapter 3 presents the FD techniques. We first prove HBFD is unfeasible for large scale network environments and then propose a new approach (NTFD) to implementing the FD based on a general model for FD. We also analyze the trade-off of achieving QoS of FD with the results showing that NTFD has a much higher probability of achieving a better balance between completeness and accuracy, yet provides a much lower probability of false queries with lower system cost. Based on this analysis, we propose the design of a hybrid FD which combines the advantages of HBFD and NTFD. Since the FDs we discussed in this chapter are general FDs and not only cater for MA systems, we introduce it before other techniques. In Chapter 4, we will introduce the integration of failure detection service with MA systems.

Chapter 4 presents our proposed efficient primary-backup based algorithms for fault tolerant MA execution (RMAA and AMAA). We also integrate the failure detection

services with our algorithms. A handover procedure is designed for FDs to deal with the mobility of MAs.

Chapter 5 presents two set of checkpointing-based algorithms. One set of algorithms aim to determine the checkpoint placement for the independent checkpointing of a single MA. The other set of algorithms provide consistent checkpointing mechanism for a group of MAs.

Chapter 6 provides the models and mechanisms for MA transactions. We analyze the realistic execution environment of MAs and propose algorithms for transactional execution of single and multiple MAs. Through simulations, we study the probability of the occurrence of deadlocks and the impact of deadlocks on the MA systems. We propose deadlock prevention algorithms but mainly focus on deadlock detection algorithms. Related simulations are performed to evaluate the performance our proposed algorithms.

Chapter 7 gives the conclusions and a discussion of future works.

Chapter 2
Literature Review

In this chapter, we provide a literature review for the previous works on fault tolerance of MA systems, beginning each section by outlining some basic concepts. The organization of this chapter is as follows. Firstly, a brief overview is given in Section 2.1. Section 2.2 describes checkpointing based approaches. Section 2.3 describes replication based approaches. Section 2.4 describes primary-backup based approaches. Section 2.5 introduces failure detection as a building block for the fault tolerance of MA systems. Section 2.6 presents a review for MA transactions. Finally, Section 2.7 summarizes this chapter.

2.1 Overview

Checkpointing, replication, primary-backup, failure detection mechanism and transaction are the primary techniques for the design of fault tolerance in conventional distributed systems. These techniques can also be applied to the fault tolerance of MA systems, but the special characteristics of MAs (mobility, autonomy, asynchrony, encapsulation of protocols, adaptability, etc.) must be considered. Many existing works have proposed fault tolerance algorithms for MA systems based on these techniques. Two survey papers [PLE04, QUW05] have summarized existing works from different viewpoints. In [PLE04], authors focus on the replication and MA transaction techniques. In [QUW05], authors classify the existing works into two categories according to the two widely used techniques in MA systems: checkpointing and replication.

In this chapter, we classify previous work on the design of fault tolerance in MA systems according to whether they have dealt with the three widely used fault tolerance techniques, checkpointing, replication and primary-backup, or whether they deal with the two infrastructure functionalities that support fault tolerance techniques, failure detection, and MA transaction. Among them, failure detection for MA systems has gained little concentration. In most previous work, failure detection is usually simply a matter of sending *"I am alive"* or *Ping* messages, which incurs a high message cost and is in any case unreliable. To propose a new approach to implementing failure detection, we make a literature review for the failure detection techniques in traditional system. In the following sections, we will review the previous works which had adopted each of these techniques.

2.2 Checkpointing based Approaches

A checkpoint is the copy of an MA's code and state stored on stable storage. Normally, an MA system periodically saves checkpoints for the MAs. When the failure of an MA is detected by the system, it will recover the MA by rolling back to its last checkpoint. Make a checkpoint is easy in MA system: serializing an MA for the migration to the next host effectively constructs a checkpoint. Since nearly all existing MA platforms are Java based, accordingly nearly all the existing works adopt Java serialization techniques to make checkpoints. Existing works on checkpointing schemes focus on three issues: (1) what kind of checkpointing techniques they use, (2) what kinds of failures they tolerate and (3) how to deal with consistency during the recovery. This thesis also considers a less-studied issue, that of how to determine checkpoint placement in MA systems. Since to our knowledge there has been to date no work dealing with this, in the following sections we shall provide some basic background to the topic, reviewing how this problem is dealt with in conventional systems. We begin by describing some commonly used checkpointing techniques.

2.2.1 Concepts and Definitions

There has been considerable research in the area of checkpointing, for example, as it applies to scientific calculation and distributed systems. Checkpointing can be classified as application transparent or application non-transparent, depending on whether the making of a checkpoint requires the intervention of the application or of the programmer. In application transparent checkpointing, the system automatically takes checkpoints and automatically recovers from failures. Obviously, this approach has the advantage of relieving programmers of some complex programming tasks. In this research, we discuss only application transparent checkpointing. Note too, that unless otherwise stated, the term "checkpointing" will hereafter denote only application transparent checkpointing. In the following, we describe three commonly used transparent checkpointing schemes: Independent, Coordinated, and Communication-induced.

1) Checkpointing Schemes

[ELN02] classified checkpointing techniques as being either independent (or uncoordinated), coordinated, or communication-induced. Independent checkpointing is the simplest of these schemes. It allows processes to take checkpoints periodically without any coordination with other processes. Independent checkpointing, however, suffers from the domino effect [RAN75]. That is, if it is used in a group of processes that communicate by messages. The messages can induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes in a system, these dependencies may force some of the processes that did not fail to rollback, creating what is commonly called rollback propagation. Rollback propagation may extend back to the initial state of the computation, causing the loss of all the work performed before a failure.

In order to avoid the domino effect, [CHY85] proposed coordinated checkpointing in which processes send checkpoint coordination messages in order to synchronize their checkpointing activities. In this way, a globally consistent set of checkpoints is always maintained in the system. Coordinated checkpointing involves high message overheads. This makes it unsuitable for mobile computing systems with networks containing low bandwidth wireless channels. A further disadvantage of coordinated checkpointing is that process execution may have to be suspended during coordination, resulting in performance degradation.

Another checkpointing scheme to overcome the domino effect is communication-induced checkpointing (CIC), which has been proposed in [RUS80]. In CIC, processes make two kinds of checkpoints, basic and forced. Basic checkpoint is a kind of independent checkpoint, so there is no coordination for this type of checkpoint. Forced checkpoint is made to maintain the consistency recovery line. Instead of exchanging coordination messages to coordinate the checkpoints like coordinated checkpointing, CIC piggybacks protocol specific information in application messages. Processes use the piggybacked information to decide whether a forced checkpoint should be taken or not.

Compare with the independent and coordinated checkpointing, CIC has several advantages. Firstly, communication-induced checkpointing is intuitively believed to be scalable since it does not require the processes to participate in a global checkpoint. However, in [ALV99], through several experiments for a class of compute intensive distributed applications made by author, they got conclusions that CIC does not scale well as the number of processes increases. The occurrence of forced checkpoints at random points within the execution due to communication messages makes it very difficult to predict the required amount of stable storage for a particular application run. Also, this unpredictability affects the policy for placing basic checkpoints and makes communication-induced protocols cumbersome to

use in practice. Furthermore, the study shows that the benefit of autonomy in allowing processes to take basic checkpoints at their convenience does not seem to hold. In all experiments, a process takes at least twice as many forced checkpoints as basic, autonomous ones. Secondly CIC scheme allows processes themselves make decision to take checkpoints. Therefore the process can choose the right time to make the checkpoints. Obviously, this characteristic provides the autonomy and flexibility. For example, system or processes can choose the time to save checkpoint when the images of the processes are small [LIC90, PLA95]. This also makes it possible to integrate with other algorithms in a system which adopted other checkpointing schemes. The hierarchical architecture of our framework also makes it possible to save overheads by integrating the algorithms in different layers so as to reduce the number of forced checkpoints. We will discuss a case of integration in Chapter 5.

2) Definitions

Here we provide some necessary definitions and theories for checkpointing, which have been widely accepted by existing works [RAN75, RUS80, CHY85, LIC90, ALV99]. We will focus more on CIC because CIC has several advantages.

Definition 2.1 *Recovery line:* Upon a failure, the system is restored to the most recent consistent set of checkpoints, which form a recovery line.

Definition 2.2 *Basic checkpoints:* A process may take a basic checkpoint any time during the execution. The basic checkpoints of different processes are not coordinated to form a global consistent checkpoint.

Definition 2.3 *Forced checkpoints:* To guard against the domino effect, a CIC protocol piggybacks protocol specific information to application messages that processes exchange. Each process examines the information and occasionally is forced to take a checkpoint according to the protocol.

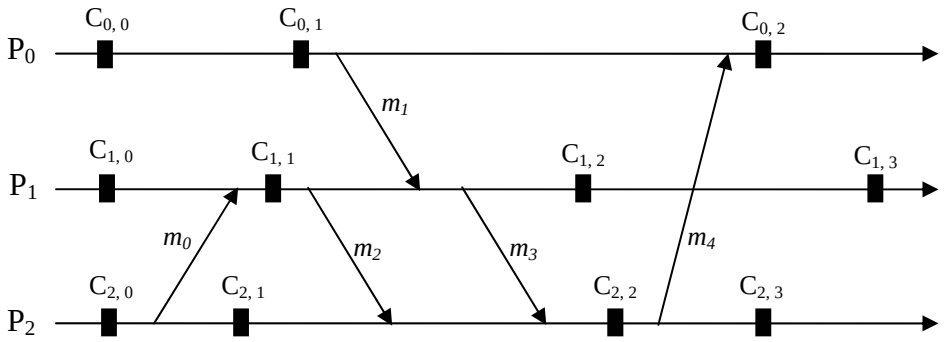


Figure 2.1 Checkpoints in a distributed computing environment

Definition 2.4 *Useless checkpoints:* A useless checkpoint of a process is one that will never be part of a global consistent state. In Figure 2.1, checkpoint $C_{2,2}$ is an example of a useless checkpoint. Useless checkpoints are not desirable because they do not contribute to the recovery of the system from failures, but they consume resources and cause performance overhead.

Definition 2.5 *Checkpoint interval:* A checkpoint interval is the sequence of events between two consecutive checkpoints in the execution of a process.

Definition 2.6 *Z-paths:* A Z-path (zigzag path) is a special sequence of messages that connects two checkpoints. Let → denote Lamport's happen-before relation [LAM78]. Given two basic checkpoints $C_{i,m}$ and $C_{j,n}$, a Z-path exists between $C_{i,m}$ and $C_{j,n}$ if and only if one of the following two conditions holds:

1. $m < n$ and $i = j$; or
2. There exists a sequence of messages $[m_0, m_1, \ldots, m_z]$, $z > 0$, such that:
   a) $C_{i,m} \rightarrow send_i(m_0)$;
   b) $\forall\, l < z$, either $deliver_k(m_l)$ and $send_k(m_{l+1})$ are in the same checkpoint interval, or $deliver_k(m_l) \rightarrow send_k(m_{l+1})$; and
   c) $deliver_j(m_z) \rightarrow Cj,n$ where $send_i$ and $deliver_i$ are communication events executed by process p_i. In Figure 2.1, $[m_1, m_2]$ and $[m_1, m_3]$ are examples of Z-paths.

Definition 2.7 *Z-cycles:* A Z-cycle is a Z-path that begins and ends with the same checkpoint. In Figure 2.1, the Z-path $[m_4, m_1, m_3]$ is a Z-cycle that involves checkpoint $C_{2,2}$.

Z-cycle is important for CIC. In [NET95], authors proved that a checkpoint is useless if and only if it is part of a Z-cycle. Therefore, if we can guarantee that there is no Z-cycle in a distributed system, then there is no useless checkpoint, which means the domino effect can be avoided. According to the methods to preventing the Z-cycle, CIC protocols can be classified model-based and index-based protocols. Model-based CIC protocols aim to prevent the formation of specific checkpoint and communication patterns that may lead to the creation of a Z-cycle. Therefore a model is needed to detect the possibility that such patterns could be forming in the system. The MRS model proposed in [RUS80] requires all message-receiving events precede all message-sending events within every checkpoint interval. Another model in [BAR81] forces the process to take a checkpoint before every message sending event. In the contrast, index-based CIC protocols eliminate the useless checkpoint by making the forced checkpoint according to the piggybacked information in application messages. In [BRI84], a typical index-based CIC algorithm has been proposed. We will discuss CIC algorithm in details in Chapter 5.

In the following sections, we first review checkpointing schemes on MA systems according to the type of checkpointing scheme that they have adopted and review the types of failures that these schemes tolerate. We then discuss previous work on checkpoint placement.

2.2.2 Independent Checkpointing

Independent checkpointing is the simplest type of checkpointing scheme and has been used to guarantee the persistence of state associated with MAs. Independent checkpointing has two main goals: First, it seeks to guarantee reliable migration. It does this by checkpointing the agent to stable storage before dispatching an MA to a new host. This copy is kept until the agent arrives at the new host. Second, it seeks to tolerate agent crash on a host. It does this by checkpointing a replica MA into stable storage upon its arrival at each host or during its execution on the host. These two approaches have been adopted in many works [STR98b, JOH99, SIL00, MOH00, SIV00, WU01, PAR02, CHN03] as well as in some MA systems. Concordia [WAN97] utilizes proxy objects and a persistent object store to insulate applications from system or network failures. However, the task of checkpointing and recovery of MAs is left to the programmer. Ara [PEI97] and Aglets [LAN98] offer a similar means for an MA system to create a checkpoint, which is stored on some persistent media (e.g. disk). Independent checkpointing is easy to implement. However, the domino effect greatly constrains its usage in environments where the MA has interactions with others, i.e., the multiple MAs cooperation applications.

In order to remove the domino effect, message logging has been used in tandem with independent checkpointing. [PAL00] used receiver based logging to log messages so as to ensure they could be regenerated during the re-execution phase. The advantage claimed for receiver based logging against sender based logging is faster recovery. Another claimed advantage is that recovery and pruning of the message log can be done autonomously,

without interaction with other user agents. But how to implement the recovery was not described. [LYU03] adopted the similar idea. The focus of these two works is on the replication scheme, with the role of checkpointing being that of an assistant scheme. Another independent checkpointing strategy that is assisted by a message logging scheme is found in [OSM04], although the authors call this checkpointing scheme a communication pair independent checkpointing strategy.

2.2.3 Coordinated Checkpointing

Another way to restrict the influence of failures to the other agents in the same communication group is to use coordinated checkpointing or communication-induced checkpointing. In coordinated checkpointing, a coordinator initiates all the group members to start the checkpointing process. This coordinates message passing so as to produce a consistent system snapshot. In [DAL98], a checkpoint manager (CM) monitors all the agents inside a cluster of machines. The CM, which is assumed to be very reliable, is responsible for keeping track of the agents and for restarting the agents when there is a node failure. Although the authors did not describe in detail how to make coordinated checkpoints, they did provide architecture for centralized checkpointing coordination. The main problem with this approach is the fact that the CM is a single point of failure, and message cost for a centralized way could also be very high.

[GEN00] presents a detailed coordinated checkpointing procedure in which one particular checkpoint server acts as a checkpoint coordinator. Each process maintains one permanent checkpoint, belonging to the most recent consistent checkpoint. During each run of the protocol, each process takes a tentative checkpoint, which replaces the permanent one only if the protocol terminates successfully. The coordinator process starts a new consistent checkpoint by taking a tentative checkpoint, and broadcasting an *Initiate* message. Upon

receiving an *Initiate* message, a process takes a tentative checkpoint. It sends a *cpTaken* message to the coordinator. When the coordinator (root) process receives a *cpTaken* message from all processes (its children), the coordinator (root) broadcasts a *Commit* message (to its children). When a process receives a *Commit* message, it makes its tentative checkpoint permanent and discards its previous permanent checkpoint.

2.2.4 Checkpoint Placement

Determining the optimal checkpointing interval (the optimal checkpoint placement scheme) has been studied for a long time. Most works focus on the uniprocessor systems [YOU74, MAN75, GEL78, GEL79, TAN84, SHI87, NIC90, PAG01]. They use execution time as the basic metric to evaluate the optimal checkpointing interval, and adopt the equidistant or equicost checkpointing strategies. A common assumption is that the normal execution time of the target program without checkpointing is known in advance.

In [YOU74], author proposed a first-order approximation to the optimum checkpoint interval. The author assumed a system in which a failure is detected as soon as it occurs, the checkpointing interval is fixed, the checkpointing time is constant, and no failures occur during error recovery. In addition to these assumptions, the author adopted the equidistant strategy and assumed that the occurrence of failures are essentially random (a Poisson process), with the failure rate λ. Then the mean time T_f between failures is $T_f = 1/\lambda$, and the density function $P(x)$ for the time interval of length x between failures is given by $P(x) = \lambda e^{-\lambda x}$. This failure assumption has been used by most of the papers [GEL78, GEL79, WON96, ZIV97, CXY03].

In [TAN84], the authors relaxed the above assumptions in three ways: by considering general failure distributions, by allowing checkpointing intervals to depend on the

reprocessing time and the failure distribution, and by allowing failures to occur during checkpointing and error recovery. They first discussed the equidistant checkpointing strategy and found that the system availability resulting from using the strategy depended only on the mean of the failure distribution. Then, the equicost strategy was introduced which is a failure-dependent and reprocessing-independent checkpointing strategy. For *Weibull* failure distributions, the authors showed that the equicost strategy achieved higher system availability than the equidistant strategy.

Instead of using the execution time as a metric, in [ZIV97], the authors presented an online algorithm for the placement of checkpoints. This algorithm keeps track of the size of the state of a program and a checkpoint is made when it is small.

Solutions have also been developed for parallel and distributed systems [WON96, PLA98] and mobile computing systems [CXY03]. In [WON96, PLA98], the optimal checkpointing interval in synchronous checkpointing for multiple processes is considered based on a mean failure time. In [CXY03], the authors derived an approximation to the optimal message number interval between checkpoints. In mobile computing environments, as part of the total application execution time, messages passing time is affected by link bandwidth, making it difficult to predict the execution time of a program. Therefore, to determine the checkpoint placement the authors utilized the received computational message number. It is assumed that the inter-failure time is exponentially distributed.

A common characteristic of all of these works is that, in the system model, the execution of the programs to be checkpointed is continuous and has a long execution period, and a uniform failure rate during the entire execution of the program is known in advance. In our study of MA systems, however, the execution of an MA is discrete in time because the MA executes for a while at a host, and then stops execution to migrate to another host. Such an

execution model looks similar to a continuously executing process with high fail-stop rate, but the two models are essentially different. The failure of a continuously executing process is totally unpredictable, while the migration of an MA is fully controlled by the MA. This results in a quite different system model which required new solutions which will be described in detail in Chapter 5.

2.3 Replication based Approaches

Although checkpointing ensures that the MA will not lost, its requirements of stable storage together with the high cost of I/O operation, and especially the possibility of blocking make it unacceptable for some applications. This has led to research into alternative approaches to achieving fault tolerance. Instead of storing the snapshot of an MA into stable storage like checkpointing, replication approaches replicate the agent and the place (hosts or servers providing the same services). There have been a number of works based on this approach [ROT98, SIL98, PLE00, PLE01, PLE03]. In [PLE01], the authors classify these works as either spatial-replication-based (SRB) or temporal-replication-based (TRB) approaches. SRB approaches send the replicas of the agents to a set of places at the next stage. TRB approaches attempt an agent execution in one place. If the execution fails, the agent is sent to another place. In [PLE03], the same authors offer another classification: the commit-after-stage and commit-at-dest approaches. Commit-after-stage approaches make the modifications at the agent and the place permanent and visible to other agents during or immediately after every execution stage. In contrast, commit-at-dest approaches generally commit the modifications only at the end of the entire agent execution.

The distinctive advantage of replication based approaches is that it guarantees the unblocking of the execution of an MA. The idea is intuitive: servers are replicated, so there

is more than one MA platform. MA is also replicated and each replica is sent to a replicated server and executes the task on it. Therefore, if one server crashes, other servers and MAs can still finish the execution. An obvious problem of this, however, is how to guarantee exactly-once execution. Existing works solve this problem using transaction and consensus.

Transaction processing is adopted in [ROT98] and [SIL98]. In [ROT98], exactly-one execution is guaranteed by applying a voting protocol and distributed transaction. A transactional message queue mechanism ensures that the following three operations executing on a place constitute a transaction: a place (1) retrieves an MA from its input queue; (2) executes the agent; (3) sends the agent to the places at the next stage. The transaction can only be committed (made permanent) by the place which has received a majority of votes. However, according to [PLE00], this solution can produce blocking if a failure occurs during the leader election. For example: the leader fails immediately after its election but before committing the transaction. As the leader can no longer resign by itself and thus no other leader can be elected to commit its transaction, the execution of MA is blocked. This could be a problem, but the transaction processing monitor (TPM) is able to monitor the failure of the leader and this would trigger a new round of elections.

In [SIL98], another leader election algorithm and 3PC were proposed as a way of improving the algorithm in [ROT98]. However, as this particular combination of leader election and transaction model may lead to a violation of the exactly-once property, it relies on a so-called distributed context database to prevent more than one concurrent leader, thereby enforcing the exactly-once property. In this approach, the commit decision is made in collaboration with the distributed context database, a leader election protocol, and the 3PC. The use of the combination of transaction and leader election to model

fault-tolerant MA execution imposes a rather high maintenance cost on the distributed context database and introduces a higher latency.

Instead of using transaction, consensus is employed in [PLE00, PLE01, PLE03]. In these works, fault tolerant and exactly-once MA execution are modeled as a sequence of agreement problems (denoted by $AgrPb_n$, $n \geqslant 0$). At every stage S_i the following operations are performed: (1) one (or potentially multiple) place executes the agent; (2) all replica agents running on the set of places M_i of stage S_i reach an agreement on the computation result; (3) finally the agreed agent is broadcasted to all the places at the next stage. Operations (1) and (2) are carried out by Deferred Initial Value Consensus (DIV) algorithm. DIV consensus algorithm assumes that a majority of participants does not fail, which is the first building block of the FATOMAS system. An example in [PLE03] can help us understand these two steps. Suppose the execution of an MA spans four stages (from S_0 to S_3). Note that at stage S_2, place p_0 fails, which causes p_1 to take over the execution. Solving an agreement problem leads all places in M_2 to agree on p_1 as the place that has executed the agent. This would be of particular importance if p_0 had been erroneously suspected by the other places in M_2. Finally, (3) is an instance of the reliable broadcast problem of reliably forwarding the agent to the next stage. A simple protocol is that every place in M_i broadcasts the result to every place in M_{i+1}. However, this incurs significant overhead, so only a majority of the places in M_i broadcast to all places in M_{i+1}. As DIV consensus assumes that a majority of places in M_i do not fail, it is ensured that at least one place actually sends the agent. Traditional reliable broadcast protocols assume a *1-to-m* communication scheme where *one* process broadcasts a message to *m* destination processes. In this case an *r-to-m* communication schema is needed: *r* senders have the same message to reliably broadcast to *m* destinations.

From above discussion we can see that although replication based approaches can provide the non-blocking execution environment for MAs, the cost is very high: replicated servers are needed; transaction or consensus algorithm is necessary.

2.4 Primary-backup based Approaches

While replicated servers are sometimes not available, there are no limits on MA backups. Backup MA is used in primary-backup based approaches to safeguard the failure of the working MA. The backup MA follows the working MA along the itinerary. Once a failure of the working MA is detected, a new working MA is created by the backup MA to replace the failed one. Note the backup in the primary-backup approach is not like the replica in the replication approach, where the replica simultaneously carries out the same task on the replicated server as the working MA. Normally backup MA will not perform the same work as the working MA, and its task is just to safeguard the failure of the working MA.

Most of the primary-backup based algorithms in the literature are based on the same rear-guard model. A working MA is followed by one or several backups, called the rearguard agents [JOH95]. If the working MA fails, the rearguard agent will continue the job for the failed MA. However, no implementation is described in [JOH95]. Later works made improvement on and reported implementations of this model. In [JOH99], the authors implemented their idea of the rearguard agent on their TACOMA MA platform. In addition to the rearguard agent, checkpointing provided by TACOMA is also adopted in [JOH99] to guarantee the persistent state of an MA. Reliable failure detection and reliable broadcast are needed to detect the failures and deliver messages between the backup MAs and the working MA.

It is not necessary for the name of the backup MA to be that of the rearguard agent. Another work [LYU03] which stems from [JOH99] refers to the rearguard agent as the witness agent. An improvement is that [LYU03] gives up the reliable broadcast and uses peer-to-peer communication in order to reduce costs. In [TAO00], the authors presented a "sliding window" mechanism. Before each migration of an MA, a specific number of backups of the MA are created lest it collapses or disappears. It can be said that agent backups simply play the role of rearguard agents. The size of the window is adjustable and determines the number of backups used. [KOM02] uses a "surrogate" agent, which is just another name for the rearguard agent. An MA will leave a surrogate on each host it visits. Once a surrogate learns that the working MA has failed, it will recreate an agent to continue the job. A mobile shadow scheme is proposed in [PEA03] which employs a pair of backup MAs, the master and the shadow. [PLE03] proposes a pipeline model in which a backup agent follows a working agent and runs on a witness place. Again, the shadow and the witness agent are acting as rearguard agents.

The rearguard agent is limited in that it only guards the failure status of the working MA, and maintains consistency with the working MA in order to continue the work should the working MA fail. It is possible to improve the system performance by letting the backup MA undertake tasks that can be done concurrently with the working agent. In [CAO03], the authors improve the system execution speed by using two reverse MAs to execute in parallel in reverse itinerary. In [QIH03], two MAs executing in reverse itinerary to speed up execution and improve fault tolerance. The first of these works however focuses on achieving load balance and the second on improving sensor network's performance. Neither addresses the issue of fault tolerant execution of MAs.

In summary, although the rear-guard algorithm provides fault tolerance for an MA system, it is not efficient. Also, conventional heartbeat-style failure detectors are costly and also

introduce the problem of false detections. All of these problems will adversely affect the system performance. In addition to these problems, another crucial consideration is that many data retrieval applications such as network management demand fast data collection. Data submitted late usually is not useful, and can even be harmful to the system, so it is an important requirement of fault tolerance algorithms that they be efficient.

2.5 Failure Detection Techniques

As we mentioned in Section 1.4.3, failure detection, or FD, is a building block for the fault tolerant execution of MAs. As such, it appears in nearly all the algorithms or schemes discussed above. An FD with good QoS is highly desirable. In this section, we review the existing works on the QoS of FDs, beginning with some preliminary background.

2.5.1 Properties and QoS of FDs

1) Properties of FD

Chandra and Toueg proposed the concept of unreliable failure detector, which is characterized by the completeness and accuracy properties [CHA96]. The accuracy property restricts the mistakes a failure detector can make, while completeness represents the capacity of suspecting an actually crashed process. To be more specifically, there are two completeness properties:

- ***Strong Completeness***: Eventually every crashed process is permanently suspected by every correct process;
- ***Weak Completeness***: Eventually every crashed process is permanently suspected by some correct process;

And four accuracy properties:

- ***Strong Accuracy***: No process is suspected before it crashes;
- ***Weak Accuracy***: Some correct process is never suspected;
- ***Eventual Strong Accuracy***: There is a time after which correct processes are not suspected by any correct process;
- ***Eventual Weak Accuracy***: There is a time after which some correct process is never suspected by any correct process;

Table 2.1 shows these two sets of properties, which produce eight combinations. They classify FDs into different classes.

Table 2.1 TFD Classes in [CHA96]

Completeness	***Accuracy***			
	Strong	***Weak***	***Eventual Strong***	***Eventual Weak***
Strong	Perfect P	Strong S	Eventual Perfect ◊P	Eventual Strong ◊S
Weak	Q	Weak W	◊Q	Eventual ◊W

2) QoS Specifications for HBFD

Paper [CHE02] is a representative work for the QoS specifications and analysis of heartbeat-style FD (HBFD). A basic HBFD algorithm in [CHE02] is described as follows. At regular time intervals (η time units), process p sends heartbeat messages $m_1, m_2, m_3, \ldots$ to another process q. Let σ_i be the sending time of message m_i. q has a sequence of receiving time points $\tau_1 < \tau_2 < \tau_3 < \ldots \tau_i$ obtained by shifting the sending time σ_i forward by δ time units ($\tau_i = \sigma_i + \delta$), where δ is a fixed parameter of the algorithm. Time points τ_i, together with the arrival times of the heartbeat messages, are used to determine the output of the failure detector at q. Consider the time period $[\tau_i, \tau_{i+1})$. At time τ_i, the failure detector

at q checks whether q has received some message m_j with $j \geq i$. If so, the failure detector trusts p in the period $[\tau_i, \tau_{i+1})$. Otherwise, it starts suspecting p at time τ_i. Hereafter we call this algorithm as the basic-HBFD algorithm.

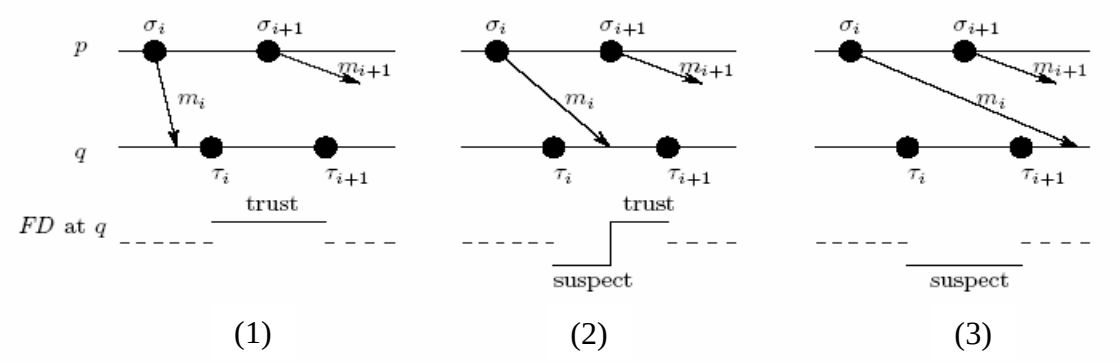


Figure 2.2 Three scenarios of the failure detector output [CHE02]

Figure 2.2 shows the three scenarios of the outputs of the basic-HBFD. Among them, (2) shows the *T-transition*, where the output of FD changes from suspect to trust. In contrast, an *S-transition* occurs when the output changes from trust to suspect. Before configure the parameters for basic-HBFD, authors of [CHE02] proposed QoS metrics for FD. We consider five metrics. The first one measures the speed of a failure detector. It is defined with respect to the runs in which p crashes. The rest are used to specify the accuracy of an FD.

Detection Time (T_D): Informally, T_D is the time that elapses from p's crash to the time when q starts suspecting p permanently. More precisely, T_D is a random variable representing the time that elapses from the time that p crashes to the time when the final *S-transition* occurs and there are no transitions afterwards. If there is no such final *S-transition*, then $T_D = \infty$; if such an *S-transition* occurs before p crashes, then $T_D = 0$.

Mistake Recurrence Time (T_{MR}): This metric measures the time between two consecutive mistakes. More precisely, T_{MR} is a random variable representing the time that elapses from an *S-transition* to the next one. If no new *S-transition* occurs, then $T_{MR} = \infty$.

Mistake duration (T_M): This metric measures the time it takes the failure detector to correct a mistake. More precisely, T_M is a random variable representing the time that elapses from an *S-transition* to the next *T-transition*. If no *S-transition* occurs, then $T_M = 0$; if no *T-transition* occurs after an *S-transition*, then $T_M = \infty$.

Average Mistake Rate (λ_M): It measures the rate at which a failure detector makes mistakes, i.e., it is the average number of *S-transitions* per time unit. This metric is important to long-lived applications such as group membership and cluster management, where each mistake (each S-transition) results in a costly interrupt.

Query Accuracy Probability (P_A): This is the probability that the failure detector's output is correct at a random time. This metric is important to applications that interact with the failure detector by querying it at random times.

3) The Equations between QoS Metrics and Network Behaviour

The networking model of HBFD [CHE02] assumes that processes *p* and *q* are connected by a link that does not duplicate messages, but may delay or drop messages. Note that the link here represents an end-to-end connection and does not necessarily correspond to a physical link. The message loss and message delay behavior of any message sent through the link is probabilistic, and is characterized by the following two parameters: (1) message loss probability P_L, which is the probability that a message is dropped by the link; (2) message delay time *D*, which is a random variable with range $(0, \infty)$ representing the delay

from the time a message is sent to the time it is received, under the condition that the message is not dropped by the link. The expected value *E(D)* and the variance *V(D)* of D is finite. Processes *p* and *q* have access to their own local clocks. For simplicity, in [CHE02], no clock drift is assumed.

For basic-HBFD, we assume that the heartbeat messages satisfy the following message independence properties: (1) the message loss and message delay behavior of any message sent by *p* is independent of whether or when *p* crashes later; and (2) there exists a known constant Δ such that the message loss and message delay behaviors of any two messages sent at least Δ time units apart are independent. We assume that the inter-sending time η is chosen such that $\eta \geq \Delta$, so that all heartbeat messages have independent delay and loss behaviors. Let $\tau_0 = 0$, and τ_i, $i \geq 1$, the following definitions and propositions are used in the equations proposed by [CHE02], and they are all with respect to failure-free runs.

Definition 1

(1) For any $i \geq 1$, let k be the smallest integer such that for all $j \geq i + k$, m_j is sent at or after time τ_i.

(2) For any $i \geq 1$, let $p_j(x)$ be the probability that q does not receive message m_{i+j} by time $\tau_i + x$, for every $j \geq 0$ and every $x \geq 0$; let $p_0 = p_0(0)$.

(3) For any $i \geq 2$, let q_0 be the probability that q receives message m_{i-1} before time τ_i.

(4) For any $i \geq 1$, let $u(x)$ be the probability that q suspects p at time $\tau_i + x$, for every $x \quad [0, \eta)$.

(5) For any $i \geq 2$, let p_S be the probability that an *S-transition* occurs at time τ_i.

Proposition 1

(1) $k = \delta/\eta$.

(2) For all $j \geq 0$ and for all $x \geq 0$, $p_j(x) = p_L + (1 - p_L)Pr(D > \delta + x - j\eta)$.

(3) $q_0 = (1 - p_L)Pr(D < \delta + \eta)$.

(4) For all $x \in [0, \eta)$, $u(x) = p_0(x)\, p_1(x) \ldots p_k(x)$.

(5) $p_S = q_0 \cdot u(0)$.

Equations

The FD implemented by basic-HBFD has the following properties:

(1) $T_D \leq \delta + \eta$.

If $p_0 > 0$ and $q_0 > 0$ (the non-degenerated case), then we have:

(2) $E(T_{MR}) = 1/\lambda_M = \eta / p_S$

(3) $E(T_{MR}) = (1- P_A)* E(T_{MR}) = (1/P_A)* \int_0^{\eta} u(x)dx$

(4) $P_A = 1-(1/\eta) \int_0^{\eta} u(x)dx$

If $p_0 = 0$ or $q_0 = 0$ (the degenerated case), then we have: in failure-free runs, (1) if $p_0 = 0$, then with probability one q trusts p forever after time τ_1; (2) if $q_0 = 0$, then with probability one q suspects p forever.

From these equations, we can derive many useful properties of the QoS of the FD: (1) When δ increases, the detection time increases linearly; (2) When η decreases, the network bandwidth used by the failure detector increases linearly. Therefore, with a small (linear) increase in the detection time or in the network cost, we can get a large (exponential) increase in the accuracy of the new failure detector.

2.5.2 Implementation of FDs

In [CHE02], the considerations of the FD implementation include: (1) How fast the failure detector detects actual failures and (2) How well it avoids false detections. HBFDs with different QoS properties (Detection time, Mistake recurrence time, Mistake duration) have

been implemented to satisfy these considerations. But the authors did not consider the scalability and message cost of the FDs.

A severe problem with HBFD is its poor scalability due to the high message cost. On the other hand, we can not reduce the message exchange frequency if a quick failure detection capability is required. In order to reduce the message cost and improve the scalability, people adopt Gossip protocol in HBFDs [REN98, WAS03]. In a group of FDs, each FD maintains a list of its known FDs. Every T_{gossip} seconds, each FD increments its own heartbeat counter and selects another FD randomly to send its list to. For a monitored process, each time only one heartbeat message is sent out no matter how many FDs need the process' heartbeat message. So the heartbeat flows are reduced dramatically and a better scalability is achieved. However, the heartbeat messages from a source process may be forwarded by many intermediate FDs before it reaches the destination FD. This may generate a large variation of the end-to-end message delays. It therefore can not guarantee the accuracy and also can have a very long detection time. Another constraint is that membership information needs to be maintained between the FDs.

In [GUP01], the failure detection service is organized in a hierarchical structure to support large-scale applications. It consists of two levels: local and global, mapped upon the network topology. The system is composed of local groups mapped upon a LAN, bound together by a global group. Each group is a detection space, which means that every group member watches on all the other members of its group. Every local group elects exactly one leader which will participate in the global group. The leader will summarize the heartbeat messages from its group and exchange the summarized messages among the leaders so as to reduce the message overhead. However, it adds the cost to construct the groups and elect the leader.

2.6 MA Transactions

Transaction is a basic technique for providing fault tolerant MA execution. It has the particular advantage of being able to guarantee the properties of *Atomicity, Consistency, Isolation,* and *Durability* (ACID), which not only provide the rollback mechanism for non-idempotent operations, but also directly support the upper layer MA applications which require the ACID properties. Existing works on MA transaction have focused either on using transaction to achieve fault tolerance or on implementing transaction so as to provide ACID services for MA based applications (i.e., e-commerce and network management). The works related to these two aspects will be reviewed in the following two subsections. Since few works to date have discussed the problem of deadlock in MA transaction, at the end of this section, we provide a literature review of deadlock detection in distributed system.

2.6.1 Transactions as an MA Fault Tolerance Mechanism

Transaction processing is adopted in [ROT98] and [SIL98] to help ensure that exactly-one execution is guaranteed. We have discussed these two works in the earlier discussion of the replication-based approach. These two works use transaction to guarantee the atomicity and consistency of the system. In [VOG97, VOG97b], distributed transaction processing is implemented to guarantee the migration of MAs and preserve the exactly-once execution property. The transaction semantic for the agent migration avoids the duplication of an agent as well as its loss. In case of error situations or host crashes, the recovery and rollback mechanisms provided by the transaction allow a reliable and consistent resumption of the agent transport. However, these works did not mention how to implement MA transaction.

In [PLE03], the authors discuss two schemes for undoing agent actions, the pessimistic and the optimistic execution. Pessimistic execution executes each stage of the agent's tentatively, making modifications permanent only when it is guaranteed that the agent neither crashes nor is erroneously suspected. In this situation modifications can be undone simply by flushing them out. Pessimistic execution will lock the accessed data items until the MA has finished executing all of its tasks along the itinerary. Other agents can access a locked data item only once the agent holding the lock has committed (i.e., made permanent) its modifications. Optimistic execution immediately makes its modifications to the place permanent and visible to other agents and does not lock data for long periods. However, this can make the task of undoing modifications more complex because other agents may have read the modified data. One response to this problem is that of the compensating transaction [GRJ81, GAR87]. Compensating transactions undo transactions semantically and allow modifications to the data to be committed immediately. If they need to be aborted later, only the compensating transaction need be run. Generally, pessimistic execution is based on the use of a close sub-transaction model and optimistic execution is based on the use of an open sub-transaction model.

2.6.2 Implementation of MA Transactions

Work to date on the implementations of MA transactions still stay on the stage of theoretical discussions. Some works focus on how to maintain the ACID properties, the support of open or close transaction model, etc. Some works have discussed only MA transaction within existing distributed transaction standards and implementations.

In [STR00], a mechanism for the application-initiated partial rollback of the agent execution is presented for providing the exactly-once execution of MAs. The rollback mechanism uses compensation operations to roll back the effects of the agent execution on

the resources and uses a mixture of physical logging and compensation operations to roll back the state of the agent. The introduction of different types of compensation operations allows performance improvements during the agent rollback. In [SIL97], the authors analyzed the requirements of MA transaction and proposed an MA transactional execution model based on open nested transaction. A single agent migrates from site to site to execute assigned tasks. Each local sub-transaction may commit or abort. If a local sub-transaction commits, the MA will migrate to the next site and start a new sub-transaction. If any sub-transaction aborts, the whole MA transaction will abort. Since the previous sub-transactions have already committed, compensation is needed. The authors later introduced the fault tolerance support for their MA transaction model, so as to increase the rate of commitment in [SIL00]. A similar MA transaction model is adopted in [PLE02]. Authors also introduced fault tolerance mechanisms based on MA platform redundancy. The idea is that, instead of sending the MA from one place (platform) to another, the MA is replicated and sent to a set of places at the same time. This allows the set of replicated MAs to proceed with the transaction notwithstanding the failures of some MA systems. An agreement protocol is proposed to prevent possible multiple executions of the agent code. The agreement protocol requires all MAs to reach an agreement that only one MA can commit its execution results. In the approach proposed in [SHE01], authors introduced the parallel transactions which run over different itineraries but with the same destination. The computing results will be combined at the destination. For realistic applications, however, it is hard to identify and execute parallel transactions. The implementation of this approach depends on the available resources and the server settings.

Several works have been reported on supporting MA transaction within existing distributed transaction standards. [VOG98] presents an extension of the OMG-OTS model with multiple MAs. This model uses the transaction model proposed in [SIL97] to guarantee ACID. OTS services are used to help the MA monitor the resources and observe the

resources in a preparation phase so as to minimize the probability of aborts and rollback/compensation, which is the major problem in open nested transactions. Although they chiefly aim to provide fault tolerance for MAs, some works also provide the implementations. In [ROT98], the implementation is based on conventional transaction technology used in X/Open DTP and CORBA OTS. DPT is also adopted in [VOG97]. It is true that MA transactions share some characteristics with distributed transactions. But normally, the platforms that support the distributed transaction model (DPT/XA; OTS/COBRA; JTS/JTA) are only deployed within one organization, such as within a company. In an MA transactional execution, the MA may migrate across several organizations/companies, and the hosts visited by the MA can host a centralized database system or a distributed database system that supports a distributed transaction model. MA transaction cannot only rely on these standard models. Therefore, a more general system model is needed. Such a model is proposed in this research and will be described in Chapter 6.

2.6.3 Deadlock Detection

Deadlock may occur during the transactional execution of MAs. A deadlock is a situation in which subsets of MAs are waiting for some other MAs to release resources. No progress is possible, unless this situation is broken. Four necessary conditions must hold for the occurrence of a deadlock: Mutual exclusion, Non-preemption, Hold-and-wait and Circular wait. Deadlock handling can be divided into three types: Deadlock prevention, Deadlock avoidance and Deadlock detection and resolution. We mainly focus on the deadlock detection in our research. Distributed deadlock detection algorithms can be classified into four categories: Path-Pushing (WFG-based) [OBE82, ASH02], Edge-Chasing (Probe-based) [CHY82, MIT84], Diffusing Computation [CHY83] and Global State

Detection [BRA83]. The former two types of algorithms are widely adopted in database and distributed systems.

Path-pushing algorithms maintain an explicit Wait-for-Graph (WFG). A WFG consist of a set of processes $\{P_1, P_2,\ldots, P_n\}$ as the node set. An edge (P_i, P_j) exists in the graph if and only if P_i is waiting for a resource held by P_j. Each site periodically builds a local WFG by collecting its local wait dependencies, then searches for a cycle in the WFG and tries to resolve these cycles. After that every site sends its local WFG to its neighboring sites. Each site updates its local WFG by inserting wait dependency received, and detects cycles in the updated WFG. The updated WFG is passed along to neighboring sites again. This procedure will repeat until some sites finally detect the deadlock or announce the absence of deadlock. The most famous algorithm in this category is Obermarck's algorithm [OBE82] implemented in System R*. In a newly proposed algorithm targeting at handling deadlocks in MA systems [ASH02], a "Detection Agent" is dispatched to all resources held by its target "Consumer Agent" for collecting deadlock information. The gathered information is finally returned to the "Shadow Agent" which is responsible for monitoring that Consumer Agent and detecting deadlock cycles.

Instead of explicitly building the WFG, Edge-chasing algorithms send a special probing message to detect deadlocks. A process (initiator) sends probes to processes holding the locks it is waiting for. A process receiving a probe message forwards it to all the processes it is waiting for. The probe message contains information to identify the initiator. If the initiator receives a probe sent by itself, it can announces a deadlock because the probe must have traveled a cycle. This idea was originally proposed in [CHY82] with the correctness proof presented in [KSH91]. Similarly in the algorithm of [MIT84], a probe consists of a single number that uniquely identifies the initiator. The probe travels along the edges in the opposite direction of global WFG. When it returns to its initiator, a deadlock is detected.

Making use of the mobility and data encapsulation of MA, in [CAO04], authors proposed a novel algorithm which adopts the ideas of both Path-pushing type and Edge-chasing. It is based on WFG and MAs are working like the probes. A framework, called MAEDD (MA Enabled Deadlock Detection), for distributed deadlock detection using MAs is proposed. In MAEDD, MAs are dispatched to collect and analyze deadlock information distributed over the network sites, detect deadlock cycles in the system, and then resolve the deadlocks.

2.7 Summary

In this chapter, we first classified fault tolerance techniques adopted in MA systems and then used this classification to review related works. For the convenience of readers, we also presented some fundamental concepts related to these techniques.

In summary, we compared the three well-known fault tolerance approaches adopted in MA systems, checkpointing, replication and primary-backup. Checkpointing is a popular scheme for the fault tolerance of MA systems. It is simple to make checkpoints in Java based MA systems, and also reduced communication costs compared with replication based schemes. However, there are difficulties associated with checkpointing. First, it depends on stable storage. As we know, the I/O operation (i.e. hard disk I/O) is costly. Some devices even have no stable storage available, such as some switches, or mobile devices like PDAs or smart phones, so checkpointing is not applicable. Second, in the event of an agent server crash the checkpoint is unavailable for an unknown time period until the agent server recovers, which results in blocking of the execution of the MA. Third, while excessive checkpointing degrades performance, insufficient checkpointing risks expensive recovery overheads. It is important to make a good judgment as to the frequency of checkpointing. Finally, if the operations that MAs have finished are idempotent, the

system can recover from the latest recovery line. However, if the operations are non-idempotent, the system cannot simply recover, but must also remove the effects associated with the failed agent.

Replication-based approaches have the advantage of being non-blocking but they are not absolutely non-blocking. For example, we can imagine a situation in which all the links from S_i to S_j are broken or all the places that hold MA a_i have crashed: in such a case, the execution of the MA will be blocked. But such catastrophic failures are rare and replication-based approaches can cover for most failures and guarantee non-blocking with a very high probability. Once again, however, there is a tradeoff: the maintenance of replicated servers and the implementation of complicated algorithms such as consensus, leader election and reliable broadcast to support the exactly-once property. This all makes replication-based approaches complicated and this is why all the replication-based approaches in the literature have been proposed only for single MA scenarios.

In [KIM02, PAR02], authors evaluate the performance of independent checkpointing approach and replication approach. Their conclusion is that checkpointing approach exhibits a very stable performance. Blocking time is the only factor affecting checkpointing approach. In contrast, many factors can greatly affect replication approach: failure rate, network delay, agent working time on a place, and the number of replicas. By selecting proper values for these factors, the system can achieve a desirable performance. Otherwise the replication approach can be a burden to the system. Compared with checkpointing approach, replication approach incurs much higher overhead during normal execution.

Primary-backup based approaches make a tradeoff between replication-based approaches and checkpointing-based approaches. Neither server replication, nor stable storage is

required. However, the primary-backup based approach does require an accurate failure detection mechanism. Otherwise, false suspicions would lead to duplicate executions, which is unacceptable in non-idempotent operations. There are two ways to solve this problem. One is to seek better failure detectors. The other way is to undo the non-idempotent operations performed by the failed agent, which can be done through the transactional execution of MAs.

Chapter 3
Notification based Approach to Implementing Failure Detectors

In this chapter, we introduce our proposed notification based approach to implementing failure detectors and present the performance comparison between NTFD and HBFD. This chapter is organized as follows. Section 3.1 presents the problems of HBFD and our motivations. In Section 3.2, we propose a general system model for the design of failure detectors. Based on the general system model, we propose NTFD and its enhancement in Section 3.3. Section 3.4 presents the analysis and comparison of the performance between NTFD and HBFD. A hybrid FD is proposed in Section 3.5. We summarize this chapter in Section 3.6. Please be noted that our proposed FD is a general FD and it is applicable to all types of processes in distributed systems. MA is a type of process when it is executing on a host. Therefore, MA can also be monitored by the FDs proposed in this chapter.

3.1 Motivations

HBFD works by periodically exchanging heartbeat messages between the FDs. According to [CHE02], we have to maintain a certain frequency of the message exchanging if we want to provide desired failure detection latency. However, high frequency of message exchanging will increase the message overhead. Therefore, the message cost and the QoS requirements of HBFD are conflict. Besides this problem, other problems with HBFD will be discussed in the following subsections.

3.1.1 Synchronization

Let us analyze the commonly used HBFD algorithm -- basic-HBFD (Chapter 2 Section 2.5.1). At process q, $\tau_i = \sigma_i+\delta$, that is to say τ_i is set according to σ_i, the sending time of m_i at p. This can only be achieved if synchronized clocks exist between p and q. To circumvent this requirement, a modified algorithm is proposed in [CHE02] which lets q obtain τ_i by estimating the expected arrival time of the heartbeat message. As we know, in an asynchronous system like Internet, we can not always make an accurate estimation of the expected arriving time of a message. Therefore, HBFD can make false detections. Although we can ignore this problem if the clock drift is small within a short period, we can not ignore it if HBFD needs to run for a long period as required in many systems. The solution is to make the clocks synchronized or use some algorithms to bypass it, but it either costs too much or loses the accuracy of failure detection.

3.1.2 The Impact of Heartbeat Messages

Based on the assumption that the network behaviour follows some probabilistic distribution, authors of paper [CHE02] construct FDs that can guarantee the accuracy property in probabilistic meaning. However, the problems are:

(1) We don’t know exactly the probabilities of message loss and message delay, so the only way is to estimate them.

(2) The probabilities of message loss or delay may change due to the change of network traffic workload. This problem is more obvious in a mobile computing environment. For example, an HBFD monitors a process on a mobile unit. Due to the random mobility of a

mobile unit, the behaviours of the messages are hard to estimate. A solution is to keep reevaluating the probabilities of message behaviours, but it is costly.

(3) A severe problem is that HBFD itself can change the behaviour of messages in a network. To explain this problem, we study a typical execution environment of FD shown in Figure 3.1. Several LANs are connected by routers. In LAN_1 there are several HBFDs monitoring the processes in LAN_2, LAN_3 and LAN_4. Processes been monitored send heartbeat messages to HBFDs and all the messages will be forwarded by the customer edge routers and the central router. From the view of message processing, a router can be viewed as a queue. We assume that the queue is an M/M/1 queue.

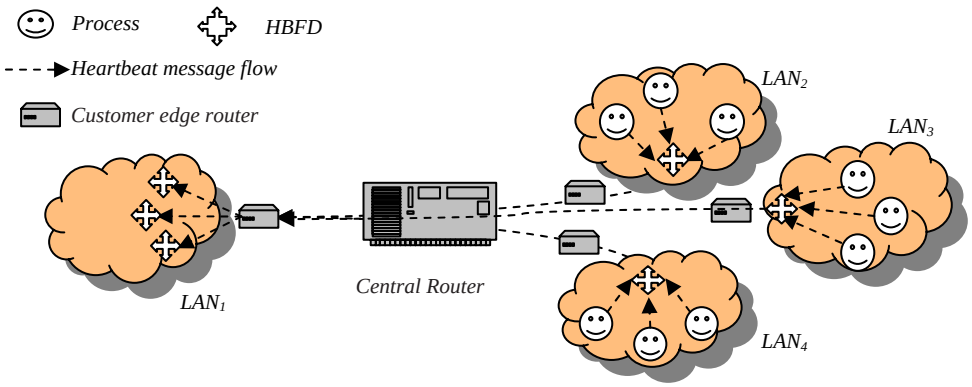


Figure 3.1 Running Environment of HBFD

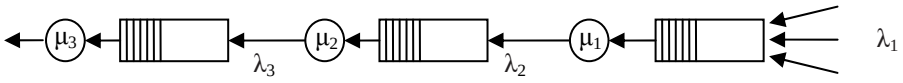


Figure 3.2 Queue Model

The message queue model for the running environment of HBFDs is illustrated in Figure 3.2. A heartbeat message is forwarded by two customer edge routers and one central router. So there are three queues in Figure 3.2. We use queuing theory to analyze the influence to

the network behaviour caused by HBFDs. For each M/M/1 queue in Figure 3.2, let λ denote the message arriving rate and μ denote the message processing rate. The expected message processing time E[T] (from the moment a message enters the first queue to the moment it leaves the last queue) is given by equation (1):

$$E[T] = \sum_{i=1}^{n} 1/(\mu_i - \lambda_i) \qquad (1)$$

Equation (1) shows that if λ increases, the message processing time will increase, and consequently the message delay will increase. Let D_{init} be the initial message delay for an HBFD. Based on D_{init}, the running configuration of the HBFDs can be calculated according to [CHE02] and then exchange of heartbeat messages starts. The exchange of heartbeat messages will increase the message delay and D_{init} will be no longer applicable. It is true that one HBFD will have only trivial impact on the message delay of the system, and we can also make compensation for it in advance. But if too many HBFDs are operating in the system, too many heartbeat messages will be generated in the system and the influence to the message delay can no longer be ignored. Although we can add a safety margin [BER02] to D_{init}, the added message delay can still possibly exceed the safety margin. Another solution is to reevaluate the message delay from time to time, but it is too costly. HBFD's accuracy property depends on the stability of the network behaviour, but the fatal problem is that HBFD itself (especially the HBFDs deployed in large scale) can change the behaviour of the network.

3.1.3 Our Motivations

The problems with current HBFD implementations motivate us to design an alternative approach to implementing FD. It has the following desired features:

- *Efficiency*: Message cost is low.
- *Scalability*: Message cost (network load) increases linearly with the number of process failures, no matter how many processes and FDs are involved.
- *Accuracy*: The output will not oscillate between the suspicious or trust status of a monitored process.
- *Simplicity and easy to implement*: No synchronized clocks are needed and the FD can be implemented by using on-the-shelf hardware or software components.

To achieve such an FD, we firstly propose a new system model for FDs which is particularly suitable for MA systems.

3.2 System Model

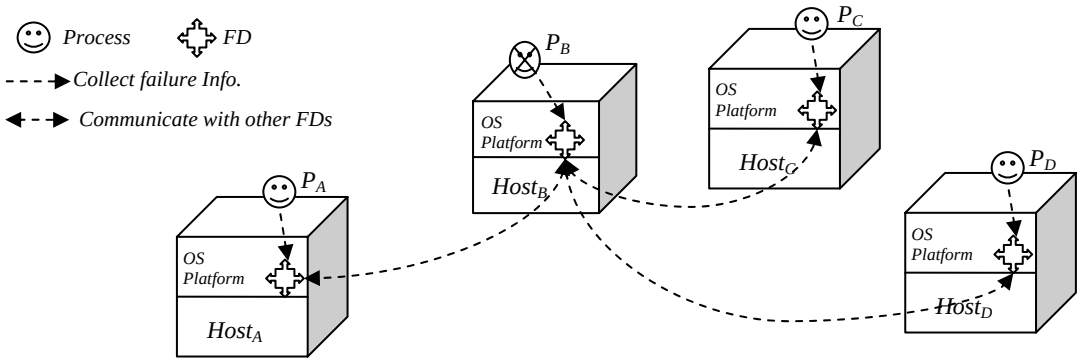


Figure 3.3 General FD Model

Nearly all the existing works are based on the conventional FD model that considers a process and its local running environment as a single entity. In practice, however, failure detection is usually provided by the system. FDs are independent module/threads and separated from the user processes. For example, the Linux kernel 2.4.x provides watchdog drivers that include software watchdogs and drivers for hardware watchdog. In MA systems, MA is running in an environment provided by MA platform. Catering for these

scenarios, as shown in Figure 3.3, we propose a general model where the failure detection service is separated from the monitored processes and provided by independent FD modules. FD collects the failure information of the processes being monitored on the same host and communicates with FDs on other hosts.

The running environment of FD is a distributed system consisting of a finite set of N hosts (N>1). On each host, there are some running processes. Processes on the same host can communicate with each other (intra-host communication) by using shared memory or messages. On each host, there is a clock referenced by all the local processes. We assume that the intra-host communication is synchronized and reliable so that FD will not make false failure detection for a process within a host. The hosts are connected by unreliable links. Processes on different hosts communicate with each other (inter-host communication) through message passing. Messages can be delayed or lost. There is no global synchronization mechanism and there is no bound on the clock drift among all the clocks. So the system is a pure asynchronous system. However, we assume that message delay and message loss follow some probabilistic distributions. The hosts or processes can fail only by crash (prematurely halting).

This general model covers the existing single entity model as a special case when the monitored process and the FD module are combined. However, the model also allows us to propose different approaches to implement FDs. As described later, the failure of the underlying host and the FD module can be handled by using replication or backup techniques.

3.3 Notification-based FD (NTFD)

The problems with HBFD motivate us to propose an alternative approach, called notification based FD (NTFD), to implementing FDs based on our newly proposed system model. Since the problems are caused by the unstable network behaviour and the periodically exchanged messages, the NTFD approach avoids them by locally detecting the failures of the processes being monitored on the same host. Once a failure is detected, NTFD sends notification messages to the interested receivers.

3.3.1 The Design of NTFD

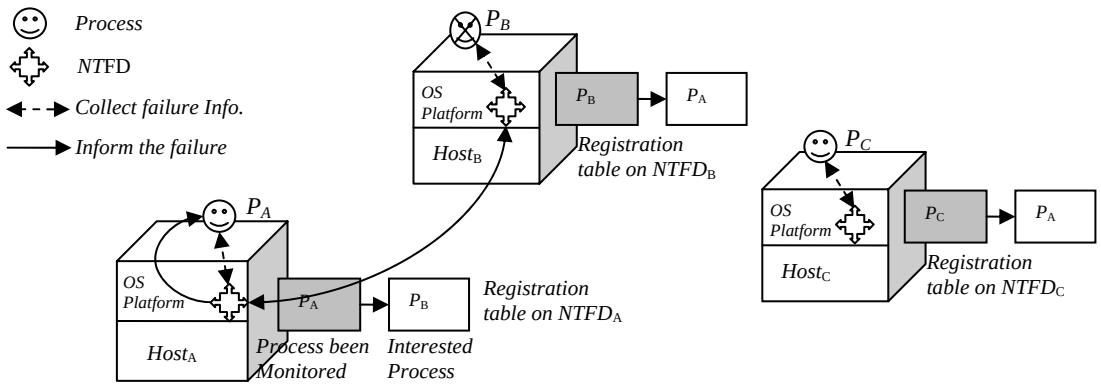


Figure 3.4 NTFD

Figure 3.4 shows the system structure of NTFD. Each NTFD maintains a registration table (Figure 3.5) for all the processes that it is monitoring. Each process can make a registration in the registration table to get the failure status of other processes. For example, in Figure 3.4, P_A makes a registration on $NTFD_B$ to get the failure status of P_B. $NTFD_B$ detects the failures of P_B through local failure detection mechanisms. Once $NTFD_B$ detects that P_B failed, it sends notifications to P_A and all processes registered for the failure status of P_B.

Each process been monitored occupies an entry of the registration table. In Figure 3.4, P_B occupies one entry in the registration table of $NTFD_B$. Other processes' IDs will be

attached to this entry if they make a registration for the failure status of P_B. So P_A's ID is attached to the entry for P_B.

The registration procedure has three steps: (1) P_A asks for the failure detection service from local $NTFD_A$. Within the service request description, P_A provides the address of the target process P_B and other optional information like the number of notification messages that should be sent to P_A, etc; (2) $NTFD_A$ accepts the request of P_A and composes a registration message and then sends it to $NTFD_B$; (3) Once received the registration message, $NTFD_B$ will allocate a new entry in its registration table for P_B if it is the first registration message for P_B. Otherwise, $NTFD_B$ just appends the ID of P_A to the entry for P_B.

The deregistration procedure also has three steps: (1) P_A asks for the deregistration service (stop monitoring the failure of P_B) from $NTFD_A$; (2) $NTFD_A$ sends a deregistration message to $NTFD_B$; (3) When the deregistration message arrived, $NTFD_B$ searches the entry for P_B and removes the ID of P_A from that entry.

3.3.2 The Implementation of NTFD

Since NTFD and its monitored processes reside on the same host, the failure can be detected at machine instruction level, code module level, process level or system level. In this thesis, as an example, we use the watchdog technique to detect local failures at process level (Figure 3.5). Watchdog is a variation of heartbeat-based FD but it uses the shared memory to increase or decrease the heartbeat (HB) counter, so there is no message passing between the monitored processes and the watchdog process.

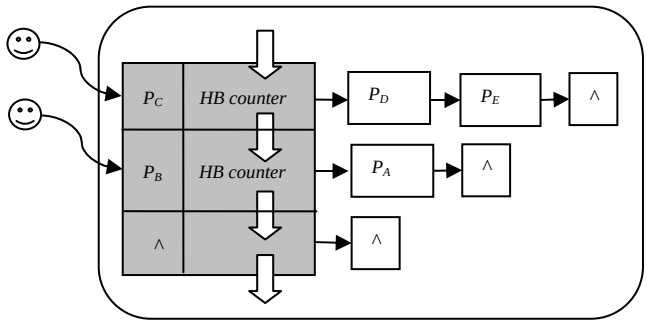


Figure 3.5 The Watchdog and Registration Table of NTFD

NTFD regularly decreases the HB counters in its registration table (The first column of the registration table is expanded to hold the HB counter). The monitored processes are designed to reset their HB counters at particular points in their execution cycle. If NTFD discovers that a HB counter has reached a certain threshold during its decrementing operation, it signals an error and then sends the notification message to the processes according to the registration table. In our model, all the processes within a host are synchronized. According to the discussion on the QoS of HBFD in [CHE02], there is no false detection of HBFD under a synchronized system. Therefore, NTFD will not make a false detection within a host.

A NTFD can be a software module and run on system level (kernel process) or user level (user process), and it can also be a hardware unit installed in the host. Watchdog is easy to build with the off-the-shelf software or hardware. For example, the Linux kernel 2.4.x includes watchdog drivers for software watchdog and drivers for several types of hardware watchdog boards. The algorithm in pseudo-code format for executing NTFD is illustrated in Algorithm 3.1.

Algorithm 3.1 NTFD

```
//A NTFD gets a Request Msg for the failure detection service
if (msg = getMsg() = = "failureStatusRequest" )
     { if ( watchdog = askWatchdog() = = NULL)
               watchdog = OS.createWatchdog();
          watchdog.Add(msg.ProcessID);
       }
//A Process renews its HB counter
if (watchdog = askWatchdog() != NULL)
        watchdog.renewHBcounter(Proc. ProcessID);
else
   PrintError();
//The Watchdog Process
while (proc = regTable.Process.hasNext() )
               {proc.HBcounter --;
                  if (proc.HBcounter < 0)
                       AssertFailure(proc.ID);
             }
// Assert Failure for a failed Process. Its ID is failure.ID;
registedProc = regTable.getRegistedProc(failure.ID)
while (procID = registedProc.hasNext() )
                 { sendNotificationTo (ProcID);
                 }

// Struct of registedProc
struct registedProc
{int procID;
  struct registedProc * next;
}

//Struct of Registration table
struct regItem
{ int PorcID;
     int HBcounter;
     struct registedProc * Pointer;
     struct regItem * next;
}
```

3.3.3 Advantages and Disadvantages of NTFD

NTFD solves the problems with HBFD described in Section 3.1. First, HBFD and the monitored processes reside on different hosts and reference different clocks, causing the synchronization problem. However, NTFD and the monitored processes reside on the same host and reference the same clock, so no synchronization is needed. Second, when a failure is detected by NTFD, NTFD will send the failure notification messages to the registered processes on the remote hosts. Therefore, the message cost only increases linearly with the

number of failed processes. Comparing with the periodical message exchange in HBFD, NTFD is much more efficient and has much less influence on the network traffic. Unlike HBFD approach, message delay will not cause false detection in NTFD, but only affect the failure detection time. These features not only solve the problems with HBFD, but also make NTFD scalable.

However, one problem with NTFD is that the remote process can not be informed of the failure, if the notification message is lost or the NTFD (or host) fails. This results in the loss of the completeness property. For the loss of notification message, we can reduce the message loss probability by using the send-and-retry method to send multiple notification messages. The multiple notification messages can be transmitted through different source routed routes in order to make the messages independent from each other. Hereafter, we assume that the notification messages are independent. For the failure of NTFD, we can build a self-restart FD to solve this problem [HUA95]. Finally, in order to solve the problem of host failure, we propose enhanced NTFD at the next section.

3.3.4 NTFD-E: An Enhancement to NTFD

NTFD-E is an enhanced version of NTFD for handling host failures. Once the host failed a separate backup NTFD is needed to send out the notification messages. The separate backup NTFD runs on a hardware board with a CPU and a networking interface. The hardware board is installed close to the host and monitors the host through proprietary connection, e.g. COM port. Now with the hardware price continuously dropping, such a hardware board only costs us tens of dollars and one such hardware board can also be shared by several local hosts. If the host is configured with a backup system/host, the backup NTFD can also run on the backup system/host.

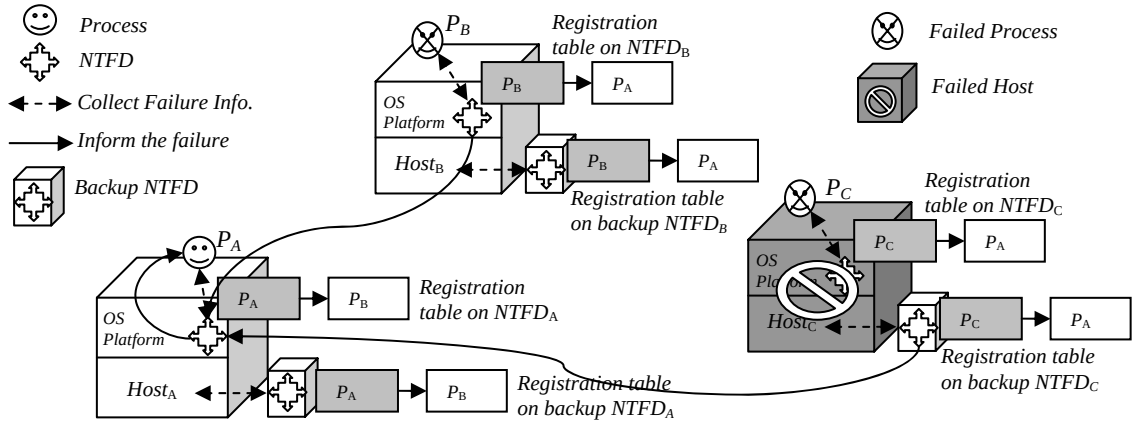


Figure 3.6 Enhanced NTFD

Figure 3.6 shows the scenario of NTFD-E. Each host is configured with a separate hardware module which provides the running environment for a backup NTFD. Except the backup NTFD, all other configurations are the same as the NTFD system. Backup NTFD has a copy of the registration table. Its task is to detect the host failure (i.e., using the traditional heartbeat algorithm). Once a failure is detected, the backup NTFD will send out the notification messages to all the registered processes in its registration table. NTFD will collect the working status of backup NTFD. If backup NTFD failed, NTFD will try to restart it or report this failure to the administrator.

It is possible that the host and the backup NTFD fail at the same time so that no notification message could be sent out. But the probability is very low. Another problem with NTFD-E is the same as NTFD: although we can reduce the probability of loss of the completeness to a very low level by sending multiple notification messages for one failure, there still exists the probability that all the failure notification messages are lost and therefore the completeness property cannot be preserved. As we know, it is an unsolvable problem in an asynchronous distributed system. The conclusion is that NTFD and NTFD-E can only preserve completeness property in probabilistic meaning, while HBFD can

maintain the accuracy property in probabilistic meaning. It is necessary to make a comparison between these two approaches.

3.4 Performance Comparisons between HBFD and NTFD

To evaluate the performance of the two types of FD implementations, we adopt the same set of parameters used in [CHE02] for HBFD. The parameters are listed in Table 3.1.

Table 3.1 Parameter Table

Parameter	***Description***
δ	*A fixed interval for HBFD to get the fresh point* $\tau_{i.}$ $\tau_i = \sigma_i + \delta$ *(refer Section 3.1).*
η	*The heartbeat messages inter-sending time; fixed to be 1 (second)*
k	$k = \lceil \delta/\eta \rceil$
P_L	*Message loss probability. Message non-loss probability is* P_{NL} *(=* $1-P_L$*)*
D or T_D	*Message delay time;* $P_D(D \leq x) = 1 - e^{-x/E(D)}$ $(x \geq 0)$
$E(D)$	*Average message delay time; set to be 0.02 (second); a small value compared with* η
f_H	*Host failure probability; 0.001 if not specified.* f_H *can impact NTFD's completeness*

We compare NTFD/-E (hereafter we use NTFD/-E to denote both NTFD and NTFD-E) and HBFD on how well they preserve the two properties of FD, the QoS metrics like quickness and efficiency, and our newly proposed metric: the probability of false report (PFR). For NTFD/-E, as mentioned in Section 3.3, we assume that NTFD/-E will not make false detection within a host. The failure detection time of HBFD is: $\delta \leq T_D \leq \delta + \eta$ [CHE02]. Within a host we can set η and δ to be μs level. Comparing with the value of η and δ (several *ms* or *s*) in a network environment, we ignore NTFD/-E's failure detection time within a host.

3.4.1 Completeness/Accuracy Properties

Any crashed process will stop sending heartbeat messages and then HBFD can claim a failure. Therefore, HBFD can preserve the strong completeness property. But HBFD can not maintain even the weak accuracy property because HBFD can make the false detection to every process due to message loss or delay. On the contrary, NTFD/-E can maintain the strong accuracy because if a failure notification message for a process is received, the process must have failed. However, NTFD/-E cannot preserve even the weak completeness due to host failure (for NTFD) or the loss of all the notification messages.

For most applications, a preferred FD should preserve both the accuracy and completeness properties with a high probability. In order to make a comparison between HBFD and NTFD/-E, we assume that there are *2*N* processes and *N* FDs in a system. Among the *2*N* processes, we assume *N* processes work well and *N* processes failed, in order to check how well HBFDs maintain the accuracy property for the *N* living processes (as shown in Figure 3.8) and to evaluate how well NTFD/-Es guarantee the completeness property for the failed *N* processes (as shown in Figure 3.7).

For NTFD/-E, if a process failed, we assume that *K* notification messages are sent out to each registered process. For NTFD, we assume that the host failure probability f_H is 0.001. For HBFD, we check how well it maintains the accuracy property during the period of *K* heartbeat messages. We assume that an HBFD will claim a process failure if it cannot receive the heartbeat message within *t* time unit.

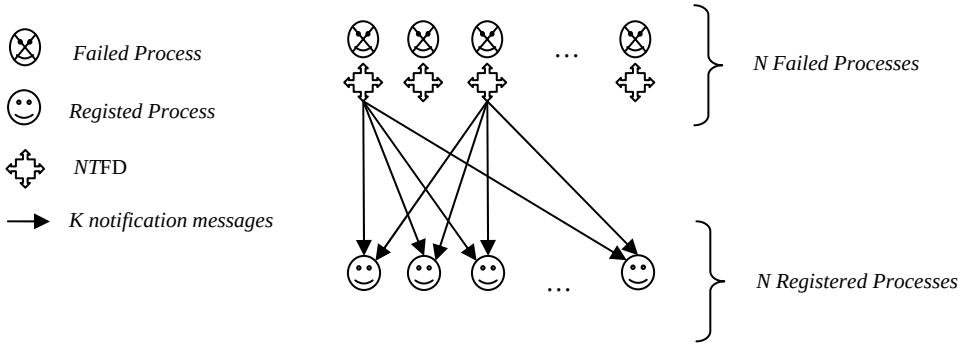


Figure 3.7 The Scenario of NTFDs when *N* Processes Failed

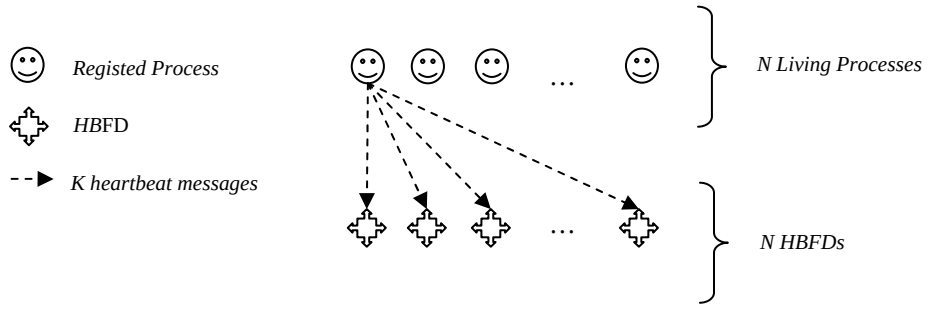


Figure 3.8 The Scenario of HBFDs for Monitoring *N* Living Processes

Table 3.2 Expressions to Calculate Completeness (C) and Accuracy (A) Properties

		NTFD-E (send K msgs)	**NTFD (send K msgs)**	**HBFD**
C	**Strong**	$\{[1-(P_L)^K]^N\}^N$	$\{(1-f_H)[1-(P_L)^K]^N\}^N$	100%
	Weak	$\{[1-(P_L)^K]^N\}^n \quad (0<n\leqslant N)$	$\{(1-f_H)[1-(P_L)^K]^N\}^n \quad (0<n\leqslant N)$	100%
A	**Strong**	100%	100%	$[(1-P_L)P_r(D<t)]^{NKN}$
	Weak	100%	100%	$1-\{1-[(1-P_L)P_r(D<t)]^{NK}\}^N$

For NTFD-E to maintain strong completeness, it requires that at least one notification message from each failed process reach all NTFD-Es. For weak completeness, all NTFD-Es should receive at least one notification messages from some failed process.

Comparing with NTFD-E, NTFD must consider the influence caused by the host failure. For our assumed system, the probabilities of maintaining two properties of FD for NTFD/-E and HBFD are listed in Table 3.2.

For the weak accuracy property of N HBFDs, $[(1\text{-}P_L)\Pr(D< t)]^{NK}$ is the probability that one living process is not suspected by all the N HBFDs and $\{1 - [(1\text{-}P_L)\Pr(D< t)]^{NK}\}$ is the probability that one living process is suspected by one or more HBFDs of the N HBFDs, while $\{1 - [(1\text{-}P_L)\Pr(D< t)]^{NK}\}^N$ is the probability that all the living processes are suspected by one or more HBFDs of the N HBFDs. Therefore, the probability that some correct process is never suspected is: $1- \{1 - [(1\text{-}P_L)\Pr(D< t)]^{NK}\}^N$ (refer Table 3.2).

In order to show the difference clearly, we plot the probability expressions of Table 3.2 in Figures 3.9/3.10/3.11/3.12 with the following settings: N=10 (there are 20 processes and 10 FDs in a system: 10 processes work well while 10 processes failed); K=10; n=1.

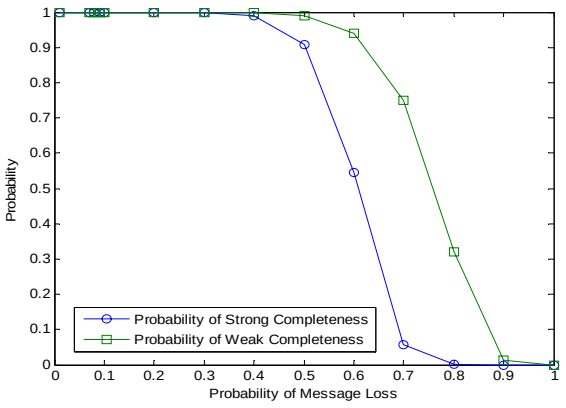


Figure 3.9 The Probability of Maintaining NTFD-E's Completeness with f_H=0.001

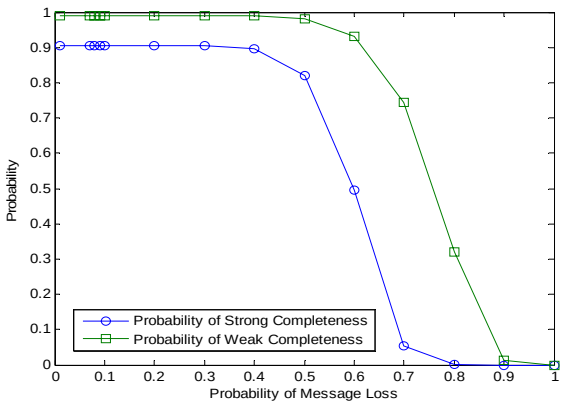


Figure 3.10 The Probability of Maintaining NTFD's Completeness with f_H=0.001

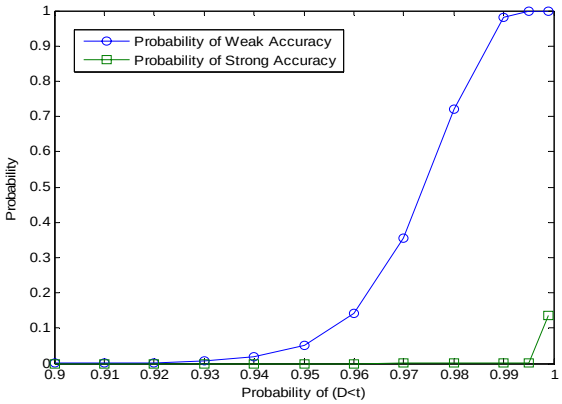


Figure 3.11 The Probability of Maintaining HBFD's Accuracy

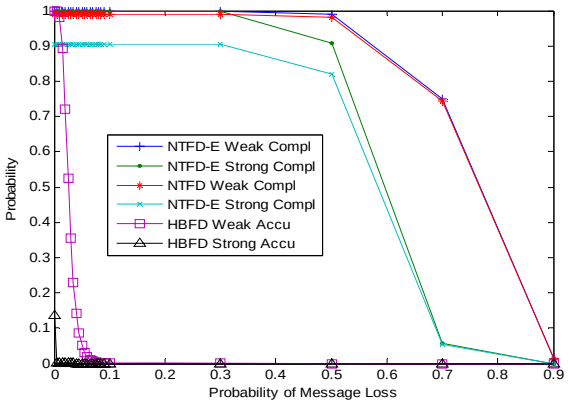


Figure 3.12 Comparison of the Probability of Maintaining FD's Properties

Figures 3.9 and 3.10 show the capabilities of NTFD-E and NTFD to maintain the completeness according to the expressions in Table 3.2. The capabilities are only affected by the probability of message loss: P_L. We assume that P_L ranges from 0.01 to 1. From Figure 3.9, we can see that NTFD-E has strong capability to maintain strong/weak completeness. Figure 3.10 shows that the capability of NTFD to maintain the weak completeness is similar to NTFD-E, but it is weak in maintaining strong completeness comparing with NTFD-E due to the possible host failure.

Figure 3.11 shows the capability of HBFD to maintain the accuracy property with a spectrum of message delay probabilities ($Pr(D<t)$ ranges from 0.9 to 0.999) and a static message loss probability (P_L = 0.001). From the results we can see that HBFD can maintain weak accuracy well when the probability of $(D<t)$ is greater than 0.99. But it is nearly impossible to maintain the strong accuracy no matter how much the probability of $(D<t)$ is. In Figure 3.12, we plot the probability for HBFD to maintain the accuracy property with different message loss probabilities (from 0.001 to 0.9) and a fixed message delay probability ($Pr(D<t)$ = 0.001). Figure 3.12 shows that HBFD can maintain the weak accuracy well only when P_L <0.01, and it is difficult to maintain the strong accuracy. For comparison, we also plot the capability of NTFD/-E to maintain completeness in Figure

3.12. We can see that NTFD/-E have a much higher probability to maintain completeness than the capability of HBFD to maintain accuracy.

3.4.2 Efficiency, Quickness and Probability of False Report (PFR)

The capability to maintain the completeness and accuracy properties is not enough to show all the advantages or disadvantages of NTFD/-E and HBFD. For example, the system overhead caused by different FDs is obviously different: HBFD keeps on sending heartbeat messages, while NTFD/-E only sends failure notification messages upon the failure of the monitored processes. Due to the different failure notifying mechanisms, failure detection time is also different. In this section, we compare the performance of FDs in efficiency (system overhead), quickness (failure detection time) and our newly defined metric: probability of false report (*PFR*).

1) Efficiency and Quickness

For an HBFD serving for a process which needs other *n* processes' failure status, the number of message exchanged within its execution duration is:

$$N_{msg} = \sum_{i=1}^{n} (T_i / \eta) \qquad (2)$$

T_i: The execution duration of Process i
n: The number of monitored processes
η: heartbeat messages inter-sending time (Table 3.1).

Equation (2) shows that the message cost of HBFD is linear with the execution duration of each monitored process.

Under the same scenario, the message cost of a NTFD or NTFD-E can be calculated by equation (3), which shows the message cost of NTFD/-E is linear with the number of failed

processes, which is much less than HBFD. Therefore, NTFD/-E are much more efficient on the message cost than HBFD.

$$N_{msg} = k * F \qquad \begin{array}{l} k\text{: number of notification messages for a failed process} \\ F\text{: The number of failed process. } (0 \leqslant F < n) \end{array} \qquad (3)$$

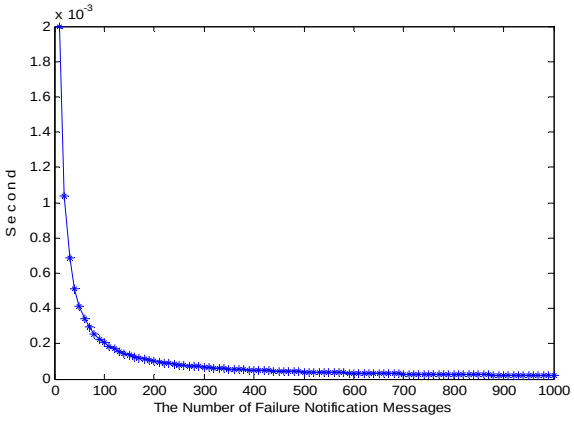


Figure 3.13 Failure Detection Time of NTFD

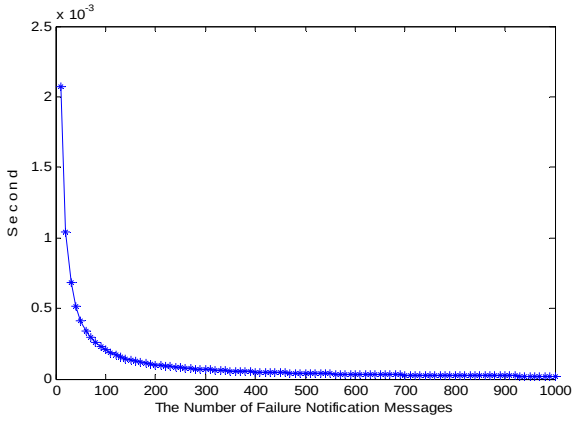


Figure 3.14 Failure Detection Time of NTFD-E

For NTFD, suppose k failure notification messages are sent out for a failed process to a registered process, the number of expected arrived messages is $N_k = k*(1-P_L)*(1-f_H)$. For n arrived failure notification messages, the probability distribution function for the first arrived message is $F_{min}(x) = 1- e^{-nx/E(D)}$. Therefore, for k notification messages, the expected arriving time of the first arrived message is $E_k(D)=E(D)/[k*(1-P_L)*(1-f_H)]$, which is the failure detection time of NTFD (T_{D_NTFD}). If we remove the impact of host failure, we get the failure detection time for NTFD-E: $T_{D\_NTFD\text{-}E} =E(D)/[k*(1-P_L)]$. Figure 3.13 and Figure 3.14 plot the failure detection time of NTFD and NTFD-E with following settings: k ranges from 10 to 1000, $E(D)$ = 0.02s, f_H = 0.001, P_L =0.001.

For HBFD, the expected failure detection time is: $T_D= \eta/2+\delta$ [CHE02]. The expression shows that the failure detection time of HBFD is bounded and increases linearly with δ and η. In Internet environment, normally, η is set to be 1 second and δ can be the expected

delay of the heartbeat messages [CHE02], which makes T_D of HBFD much longer than T_D of NTFD/-E. Please be noted that we cannot set η too small; otherwise the false detection of HBFD will be very high.

2) The PFR (Probability of False Report)

From the users' point of view, the most important thing is whether the detection results of an FD are correct or not. Therefore, we define a new metric: Probability of False Report (PFR). PFR means that when a user queries an FD for the status of a process, the result reported by the FD is wrong.

PFR can be calculated by dividing the total period which an FD gives wrong detection results by the entire execution time (T) of the FD. We assume that the MTBF (Mean Time Between Failures) of a host is 30 days (= 259200 seconds) and the MTBF of a process is 1 day (= 86400 seconds). Suppose the reliabilities of the hosts and the processes follow the exponential distribution: $F_x(T) = 1\text{-}e^{-T/m}$. m is MTBF; x can be h (host) or p (process). Within the system's execution time T, the probability of a host failure occurring at or before t is: $F_h(t) = 1 - e^{-t/m} = 1 - e^{-t/259200}$; the probability of a process failure occurring at or before t is: $F_p(t) = 1 - e^{-t/m} = 1 - e^{-t/86400}$. We assume that $F_p(t)$ includes the possible process crash caused by host failures. Hereafter, we calculate the PFR for each type of FDs and plot their average PFR within the different execution durations which range from T = 50 to T = 1600 (seconds).

2.1) The PFR of NTFD/-E

Message delay and messages loss cause the false detection of NTFD-E. We already considered these two factors in $T_{D\_NTFD\text{-}E}$. Therefore, the PFR of NTFD-E can be calculated by: $PFR_{NTFD\text{-}E} = F_p(T) * T_{D\_NTFD\text{-}E} / T$.

In addition to the message delay and messages loss, host failure affects the PFR of NTFD severely: if a host failed, the query result will be false permanently. Therefore, two conditions may result in the PFR of NTFD. One condition is that the process failed before the host fails and the notification messages have been sent out. The other condition is that the process does not fail while the host fails and results in permanent false report. PFR of NTFD can be got by summing up the results of these two conditions. For simplicity, we calculate an approximate PFR of NTFD. The approximate PFR is greater than the real PFR of NTFD, and it is enough to illustrate the performance of NTFD. At time t of a NTFD's execution duration, the corresponding PFR can be calculated by the following expression:

$$F_p(T)*T_{D\_NTFD}/T + (T-t)/T * \lim_{\Delta t \to 0}(F_h(t+\Delta t) - F_h(t)) .$$

The first part $F_p(T) * T_{D\_NTFD}/T$ computes the part of PFR due to the process failure. Please be note that since here the system already holds that the host does not fail, T_{D_NTFD} equals to $T_{D\_NTFD\text{-}E}$ in this part. $(T-t)/T$ approximates the part that the process works well but the host fails at t and results in permanent false report. Since t ranges from 0 to T, we should sum them up. Therefore, we get NTFD's PFR in equation (4).

$$PFR_{NTFD} = F_p(T) * T_{D\_NTFD}/T + \int_0^T [(T-t)/T] * F'_h(t)dt \qquad (4)$$

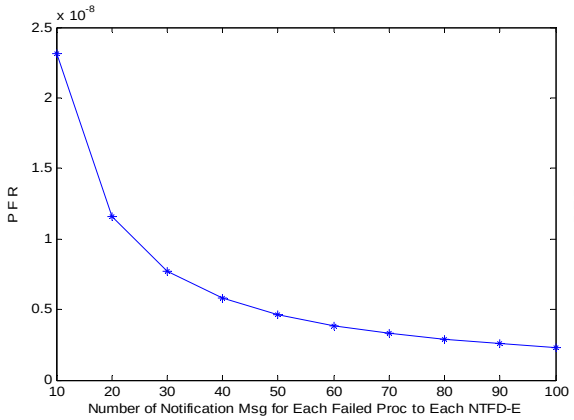


Figure 3.15 PFR of NTFD-E

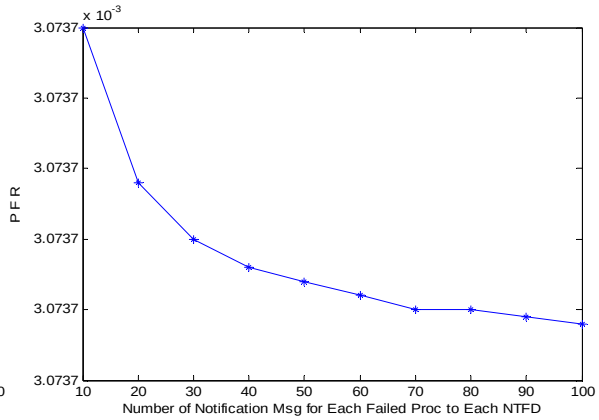


Figure 3.16 PFR of NTFD

In order to make a clear comparison, we draw the PFR of NTFD/-E in Figures 3.15 and 3.16. Compare the two figures we can see that the PFR of NTFD-E is much lower than that of NTFD, because the host crash can cause permanent false report for NTFD.

2.2) The PFR of HBFD

Firstly, we describe a special phenomenon of HBFD which does not exist in NTFD/-E. The phenomenon is that HBFD will make false detection even though the monitored process works well. The PFR under the process failure-free condition (PFR_{no_f}) equals to $(1\text{-}P_A)$. P_A is the query accuracy probability of HBFD proposed in [CHE02]. PFR_{no_f} can be calculated by equation (5) and it is plotted in Figure 3.17. Figure 3.17 shows that PFR_{no_f} always decreases with δ increasing.

$$PFR_{no\_f} = (1\text{-}P_A) = (1/\eta) * \int_0^{\eta} \mu(x)\,dx$$
$$= (1/\eta) * \int_0^{\eta} \prod_{j=0}^{k} (P_L + (1-P_L)P_r(D > \delta + x - j\eta))dx \qquad (5)$$

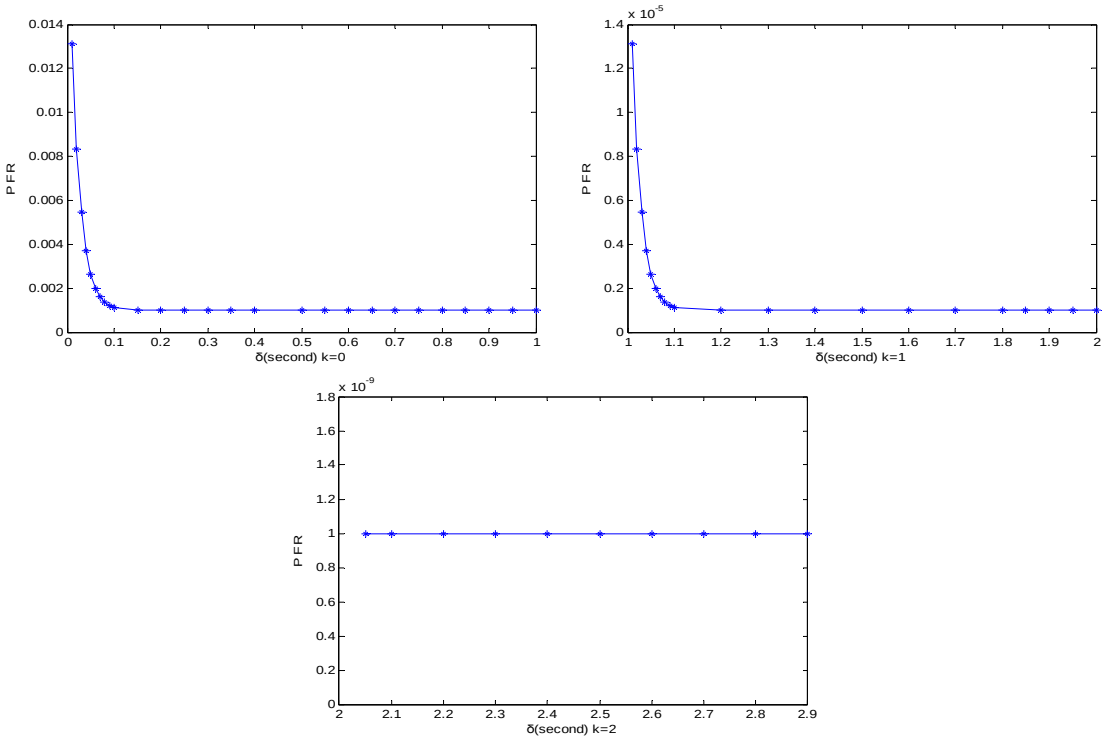


Figure 3.17 PFR of HBFD with Different δ under Failure-free Condition

However, process or host can crash and the failure can only be detected after the period of failure detection time: T_D. During T_D, HBFD cannot assert the failure so as to cause the false report. Combining PFR_{no_f}, the PFR of HBFD can be calculated by equation (6).

$$PFR = \int_0^T (1/T) * [((t\ DIV\ \eta) + \delta) + PFR_{no\_f} * t] * \lim_{\Delta t \to 0} (F_p(t) - F_p(t - \Delta t)) / \Delta t dt$$
$$= \int_0^T (1/T) * [((t\ DIV\ \eta) + \delta) + PFR_{no\_f} * t] * F'_p(t) dt \qquad (6)$$

In order to visualize the comparison, we plot the PFR of HBFD calculated by equation (6). To simplify the calculation of equation (6), here and hereafter, we adopt the average value of ($t\ DIV\ \eta$), which is ranging within (0, η), to be $\eta/2$. Figure 3.18 illustrates that the PFR of HBFD decreases with δ increasing within $0<\delta<1$. When δ increases to 1 (which makes $k=1$), the PFR decreases sharply because at this point HBFD makes false detection only when two consequence heartbeat messages are lost or delayed. But from the point $\delta=1$, the

PFR begins to increase. Figure 3.18 cannot reflect this trend due to the too large scale. We show this tendency in Figure 3.19. The reason is that from $\delta=1$, PFR_{no_f} is not the main factor affecting the overall PFR. Failure detection time ($T_D \leqslant \delta +$ ⁄) becomes the main factor affecting the PFR of HBFD. With δ increasing, the longer T_D makes the PFR of HBFD increase.

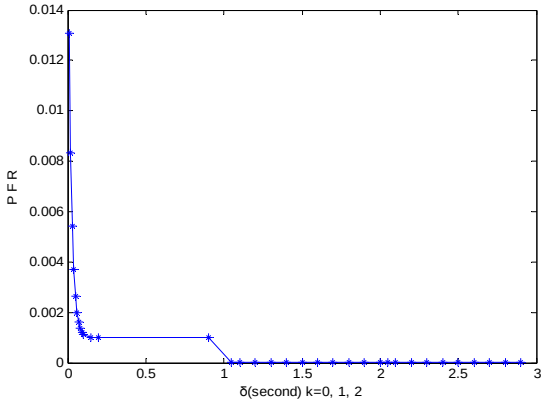


Figure 3.18 PFR of HBFD with $0<\delta<3$

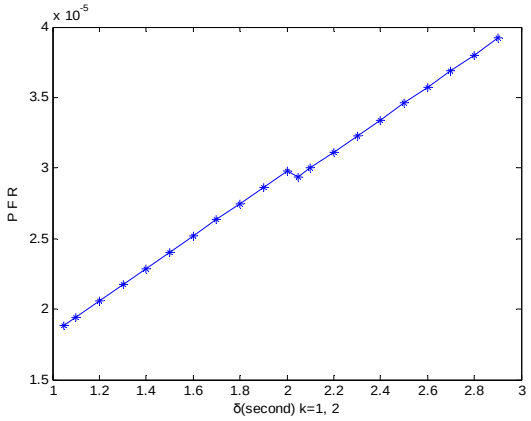


Figure 3.19 PFR of HBFD with $1<\delta<3$

3.5 A Hybrid Approach to Designing Failure Detector

From Figure 3.17 we can see that with δ increasing, the PFR of HBFD under failure-free condition always decreases due to the less impact of message loss and delay. That is a good property and especially valuable in the Internet environment where the probabilities of message delay and loss are difficult to predict.

However, if the failures of processes or hosts are concerned, the PFR of HBFD will not always decrease and it depends on the parameter δ. The PFR of HBFD decreases only when δ is small ($0<\delta<1$). After δ increases to a specific value (in our case, after δ reaches 1), the PFR of HBFD stops decreasing and begins to increase, because the increasing failure detection time becomes the main factor to affect PFR (Figure 3.19).

Above analysis tells us that to improve HBFD's PFR, we must reduce the failure detection time of HBFD when δ is increasing. However, it is impossible for HBFD, because the failure detection time of HBFD has a linear relation with δ ($T_D \leqslant \delta + \eta$). Section 3.4 tells us that the failure detection time of NTFD is determined by the fastest failure notification message, which can be much shorter than the failure detection time of HBFD. This characteristic helps us build hybrid FD to reduce the PFR of HBFD.

In hybrid FD, the failure notification mechanism of NTFD is incorporated into HBFD. All the operations of hybrid FD are the same as HBFD except the failure notification mechanism. There are two criteria for hybrid FD to assert a failure. One is the same as HBFD. Another criterion is that when a failure notification message arrives, hybrid FD will assert a failure. In hybrid FD, we increase δ to reduce the impact of message loss and delay, while failure notification mechanism helps us achieve quick failure detection. The PFR of hybrid FD can be calculated by equation (7).

$$PFR = \int_0^T [T_D * P_r(T_D < \delta + \eta/2) + \lambda_M * t]/T * \lim_{\Delta t \to 0} (F_p(t) - F_p(t - \Delta t)) / \Delta t dt$$
$$+ \int_0^T [((t \ DIV \ \eta) + \delta) * P_r(T_D > \delta + \eta/2)]/T * \lim_{\Delta t \to 0} (F_p(t) - F_p(t - \Delta t)) / \Delta t dt \qquad (7)$$

Figure 3.20 shows that the PFR of hybrid FD with different δ. Compare Figure 3.20 with Figures 3.18 and 3.19 we can see that hybrid FD not only reduces its PFR greatly, but also keeps stable (The PFR will not increase with δ increasing). These improvements are achieved by the short failure detection time provided by the failure notification mechanism of NTFD.

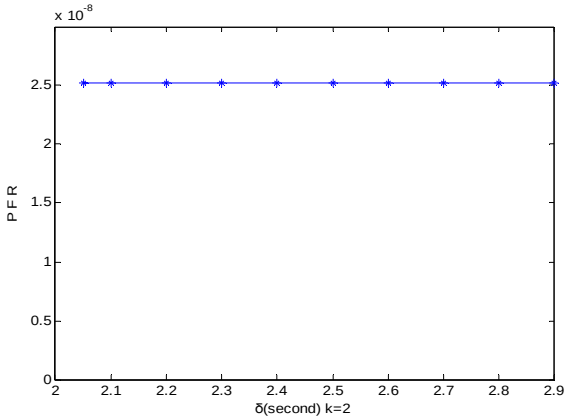


Figure 3.20 PFR of Hybrid FD

3.6 Summary

In this chapter, we proposed a new approach to implementing FDs, called NTFD, based on our newly proposed a general system model. Performance analysis shows that NTFD has advantages on all the QoS aspects except the completeness property. The biggest advantage of NTFD is that it can preserve the strong accuracy property, which is valuable for many applications. Based on the analysis of trade-offs in achieving QoS of FDs, we proposed a hybrid FD which combines the advantages of both the NTFD and HBFD approaches.

Chapter 4
Efficient Primary-Backup based Algorithms

In this chapter, we introduce our proposed efficient primary-backup based algorithms for the fault tolerant execution of MAs. This chapter is arranged as follows. Section 4.1 presents the system model. Section 4.2 describes our proposed efficient primary-backup based algorithms. In Section 4.3, two types of failure detectors are integrated with our proposed primary-backup based algorithms. Section 4.4 presents the performance analysis for our proposed algorithms. The analysis results are validated by simulations. Section 4.5 summarizes this chapter. In Appendix, we give a sample implementation for ours proposed algorithms in a Java based MA system.

4.1 System model

In this chapter, we consider the system model of an MA system that consists of one or several independent MAs and a group of hosts. The independent MAs do not communicate with each other, and each MA executes and migrates along a predefined or self-initiated itinerary. With a self-initiated itinerary, the MA only knows the first host it will visit and the following hosts will be determined by the execution results on previous hosts, whereas the agent knows all the hosts it will visit with a predefined itinerary. The hosts are connected by the communication links and the links are not reliable. An MA launched by a user is called the working MA. A working MA can have one or several backups. The backups of a working MA will monitor the failure status of the working MA through FDs. MAs can communicate with the FDs by message passing. We assume that the messages

from FDs can be delayed or duplicated, but they will be delivered to the destination eventually.

For simplicity, we assume all the operations executed by the MAs are idempotent, so the exactly once execution property is not necessary to be considered in this chapter. For non-idempotent operations, transaction support is needed to maintain the system consistency during recovery. We will introduce MA transaction in Chapter 6.

4.2 Efficient Primary-backup Based MA Fault Tolerance Algorithms

The main idea to improve the efficiency of primary-backup based algorithms is to introduce parallel processing between the working MA and its backups. According to whether the MA's itinerary is predefined or not, we propose two primary-backup based algorithms, namely Reverse MAs Algorithm (RMAA) and Alternate MAs Algorithm (AMAA).

4.2.1 RMAA

RMAA is well suited for MA applications with a predefined itinerary and no requirement on the host visiting sequence. One typical example is information retrieval. In RMAA, the original predefined itinerary is the forward itinerary, while the reverse itinerary is an itinerary that reverses the sequence of hosts in the forward itinerary. There are two MAs in RMAA. One is called the Forward MA (FMA) which will visit hosts according to the forward itinerary, and another is called the Reverse MA (RMA) which will visit hosts according to the reverse itinerary. In this section, unless otherwise stated, we only discuss one pair of MAs: FMA and RMA.

Figure 4.1 illustrates the RMAA scheme. The pair of MAs (FMA and RMA) is dispatched by the MA platform at the user side simultaneously. They execute concurrently along their own itineraries until they reach the two neighboring hosts (e.g., Host 2 and Host 3 in Figure 4.1), which indicates that all the hosts on the itinerary have been visited. The two MAs will then return to the MA platform on the user side.

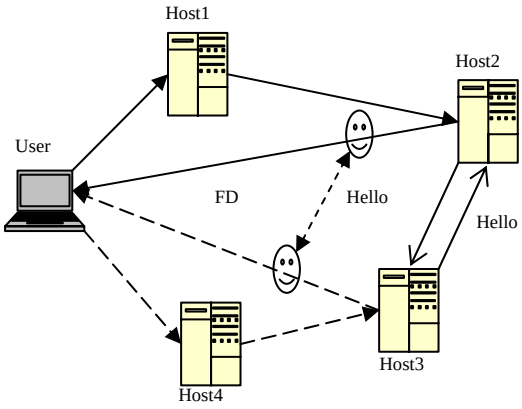


Figure 4.1 The Execution of RMAA

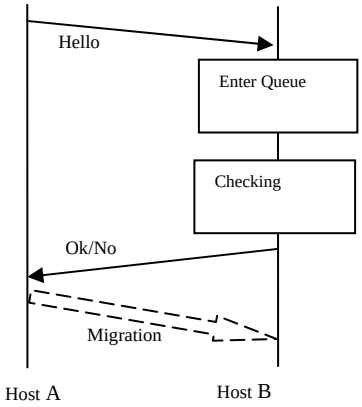


Figure 4.2 Landing Procedure

In order to prevent the failure of both MAs due to the failure of a host, the two MAs are not allowed to land on the same host. For this purpose, a landing procedure is needed (Figure 4.2). The two MAs send the coordination message “Hello” before migrate to the next host. The “Hello” message is put into a queue on the MA platform of the next host, which is to ensure that the host only accepts one MA with the earlier “Hello” in the pair of MAs. An MA can migrate to the host only if it has received an “Ok” message as response from the host. If both MAs send the “Hello” message to the host simultaneously, the host will receive both of them. But in the queue, one will precede another. For the sender of the later “Hello” message, the MA platform will reply it with a “No” message. When the MA receives a “No” message, it knows that another MA is already on the neighboring host. So it will go back to the user host. The MA which got the “OK” message will return user host too after it finishes its execution.

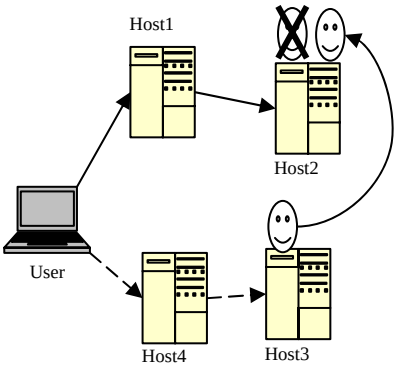


Figure 4.3 Failure Handling of RMAA

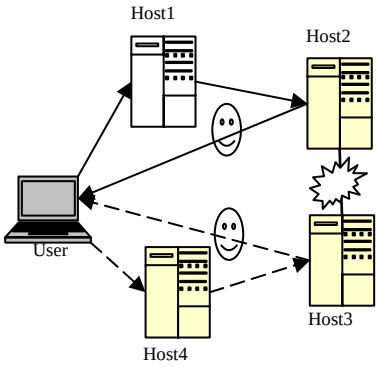


Figure 4.4 Itinerary Partition

Same with the rear-guard algorithm, we assume that the FMA and RMA will not fail at the same time. During the execution of the pair of MAs in RMAA, one MA may fail during its execution or migration. The failure detector will detect the failure and inform another MA, and the living MA will generate a new MA to replace the failed MA (Figure 4.3). For this purpose, FMA and RMA should keep each other's computing results (this is the same with rear-guard algorithm). A distinguishing advantage of the RMAA algorithm is that it can handle the itinerary partition due to a link failure. In Figure 4.4, the itinerary is partitioned into two separated sections. It is obviously that the pair of MAs can finish their tasks if they will not fail.

RMAA can be implemented at the system level in a way transparent to the application programmer. What the programmer needs to do is just to provide the MA's task and itinerary to RMAA. RMAA will create FMA and RMA to finish the users' task. The algorithm in pseudo-code format for executing RMAA is illustrated in Algorithm 4.1.

The RMAA scheme can be easily extended to accommodate n ($n \geq 1$) pairs of MAs to speed up the execution in large-scale networks. The original itinerary can be separated into n sections, and on each section RMAA is executed.

Algorithm 4.1 *RMAA*

```
//RMAA is a class which implements all the functions of RMAA algorithm. User just needs to create
//a RMAA object and provide the Task and Itinerary to the RMAA object.
1. RMAA rmaa = new RMAA (Itinerary, Task); //RMAA creates 2 members: a FMA and a RMA;
2. rmaa.Launch(); //FMA and RMA are launched;
//FMA and RMA execute the same code in parallel. We only describe FMA's execution.
3. if (rmaa.FMA.tryMigration() = = OK) //will not encounter RMA
       {rmaa.FMA.migration(); //migrate to next host
        result = rmaa.FMA.Task.start();
        rmaa.FMA.synchronize(result); //synchronize the computing result for failure handling.
        goto 3; //Finish the execution on current host, then try to go to next host.
   }else   //will encounter RMA if migrate to the next host. So FMA returns home.
      rmaa.FMA.returnHome();

// Pseudocode for the MA failure handling. Suppose ma gets a message from failure detector.
if (msg = ma.getmessage() = = MA_Failure) //get asynchronous message from failure detector
     ma1 = ma.clone();   //this ma will clone a new ma according to the failed ma's infomation.
     ma1.migration(msg.host, failureMA_id); //the cloned ma migrates to the host.

//After the cloned MA lands on the host, it will check the reported ma is really failed or not.
if (ma1.check(failureMA_id) = = ReallyFailed) //if the reported ma really failed, its job will be
     ma1.resumeFailedma();                    //continued.
```

4.2.2 AMAA

A predefined itinerary is necessary for RMAA. But one of the fundamental features for MA is its autonomy, which allows an MA to dynamically determine the next host to visit without a predefined itinerary. RMAA is not applicable under such a context while the rear-guard algorithm can still work. But the rear-guard algorithm is not efficient. Therefore, an efficient algorithm which can determine it itinerary dynamically is desirable.

For an MA application without predefined itinerary, an agent needs to compute the next stop before every migration. Accordingly, we divide an MA's operations into two sections (Figure 4.5): *CalNextStopOps* contains all the necessary operations that have to be done in order to get the next stop; *RestOps* includes the rest operations. The border between these

two sections can be different in different applications. Some applications can determine the next stop in the first few steps; some get it later.

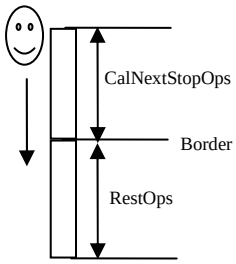


Figure 4.5 An MA's Operations

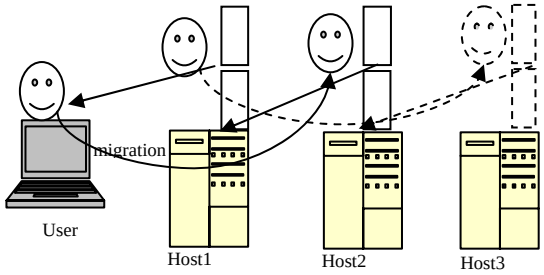


Figure 4.6 The Execution of AMAA

AMAA involves two MAs. One MA which is on the head is called the Leading MA (LMA); the other MA which is behind the LMA is called the Slave MA (SMA). The two MAs should arrange their operations into the two sections as shown in Figure 4.5. Figure 4.6 shows the execution process of AMAA. The MA platform on the user's host launches two MAs. One lands on the first stop and becomes LMA. The other waiting on the user side becomes SMA. When the LMA has worked out the next stop, it sends a message to the SMA. SMA then migrates to the next stop and becomes the new LMA to start its execution. The former LMA becomes SMA now. When the new LMA determines the next stop, it sends a message to SMA. Now the SMA may or may not finish the *RestOps*. When SMA finishes the *RestOps*, it will migrate to the next stop. The process will continue until the task is finished. As we can see, LMA and SMA execute at alternative hosts in the network.

Same with the rear-guard algorithm and RMAA, the failure detector will inform the failures of MAs, and the living MA will generate a new MA to replace the failed MA. However, different from RMAA, AMAA can not handle the itinerary partition. AMAA can also be implemented at the system level. Users need not provide the itinerary, but the

task is required to separate into two sections as we described. Algorithm 4.2 illustrates the pseudo-code for AMAA

AMAA can be easily extended to involve n ($n \geq 2$) MAs. Among the n MAs, One acts as the LMA. The rest *n-1* MAs form a sequence of SMAs. When the LMA determines the next stop, it informs the last SMA. The last SMA migrates to the next stop and becomes the LMA. Previous LMA becomes the first SMA of the sequence of SMAs.

Algorithm 4.2 *AMAA*

```
//AMAA is a class which implements all the functions of AMAA algorithm. User just needs to create
//a AMAA object and provides the Task to the RMAA object.
1. AMAA ma[ ] = new AMAA (Task); //AMAA creates 2 members: an LMA and a SMA;
2. NextHost = FirstHost; ma[0].end = false; ma[1].end = false; //Initiation;
3. ma[0].goto 4; ma[1].goto 9; //ma[0] is current LMA and ma[1] is current SMA.
//ma[0] and ma[1] share the same code from 4 to 9. In the following, "ma" can be ma[0] or ma[1].
4. ma.migration(NextHost);
5. NextHost = ma.Task. CalNextStopOps();
6. if (NextHost ≠ NULL)
       ma.informSMA(NextHost); //After the current SMA get this message, it will migrate
   else                        // to next host and becomes the new LMA. This ma becomes the new RMA.
       {ma.informSMA(NULL); //No next host, so inform SMA to return home.
        end = true; //This mark will make LMA return home
       }
7. result = ma.Task. RestOps(); //Finish the rest operations.
8. ma.synchronize(result); //synchronize the computing result
9. if (end = = ture) //No next host.
       ma.returnHome();
   else if (ma.getNextHost() ≠ NULL) //SMA get the next host which is sent by LMA
           {goto 3;                          //SMA will migrate to the next host.
           } else    //No next host
              ma.returnHome(); //it is time to go home.

// Pseudocode for the MA failure handling in AMAA is the same with RMAA.
```

4.3 Failure Detection Mechanisms in MA Systems

The function of failure detection is a fundamental requirement for primary-backup based fault tolerance algorithms and HBFDs are widely implemented in realistic systems. But as

mentioned in previous chapters, problems of HBFDs include the false detection, high message cost, and the dumb period for MA applications. In the following subsections, we propose the handover procedure for the problem of dumb period, and for the false detection and message overheads of HBFD, we introduce our newly proposed NTFD for MA systems.

4.3.1 HBFD in MA Systems

Figure 4.7 shows the HBFDs in an MA system. Each HBFD keeps collecting the status of the monitored MA in a predefined frequency on the same host. For each piece of collected status information, HBFD will construct a heartbeat message and send it to the interested HBFDs recorded in the registration table.

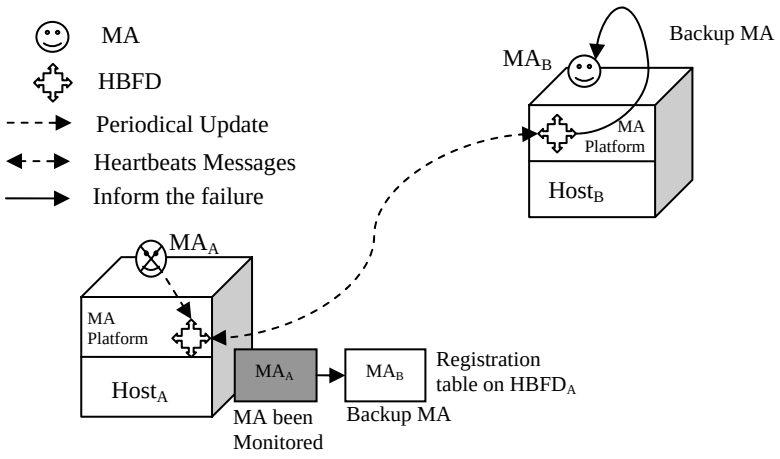


Figure 4.7 HBFD in MA System

For example, in Figure 4.7, MA_A makes a registration in the registration tables of $HBFD_A$. A complete registration table is shown in Figure 4.9. The registration procedure has three steps: (1) MA_A asks for the failure detection service from the local HBFD: $HBFD_A$. Within the service request description, MA_A nominates its backup MA's ID and address (MA_B, $Host_B$). (2) When $HBFD_A$ received the registration request, it will allocate a new entry in

its registration table for MA_A. The service requestor (MA_A) will be put in the first column of this row. The backup MA's ID and address will be put in the following columns of the entry. Then $HBFD_A$ sends the request to $HBFD_B$ on $Host_B$. (3) When $HBFD_B$ receives the request, it starts to receive the heartbeat messages sent from $HBFD_A$ to detect the failure of MA_A.

Before an MA migrates to a new host, it will make a deregistration on the current host. The deregistration procedure also has three steps: (1) MA_A sends a deregistration request (stop monitoring the failure of MA_A) to $HBFD_A$. (2) $HBFD_A$ will traverse the registration table to find the entry for MA_A ID and delete it. Then $HBFD_A$ sends the request to $HBFD_B$. (3) When $HBFD_B$ receives the request, it stops receiving the heartbeat messages sent from $HBFD_A$.

In Figure 4.7, MA_A periodically updates its status on $HBFD_A$ and $HBFD_A$ sends these collected updates in the heartbeat messages to $HBFD_B$. If MA_A failed, the updating operation will stop. So there is no heartbeat message to $HBFD_B$ and $HBFD_B$ will assert a failure. The frequency of the updates and the heartbeat message can impact the QoS of the HBFD. For the implementation of an HBFD with certain degrees of QoS, please refer to [CHE02].

For the problem of dumb period, a handover procedure is needed. A simple solution is simply provided by the registration and deregistration procedures: before an MA starts migration, it makes a deregistration, the remote HBFD stops the failure detection operations. The problem for this scheme is that, if the MA is lost during migration, the HBFD can not detect it. An enhanced scheme is based on reliable MA migration. When an MA starts a migration, it sends its clone to the next host and waits until the clone landing on the next host. During the migration process, the waiting MA can keep sending heartbeat

messages to the HBFD. After the clone lands on the new host, it informs the waiting MA which will then hand over the task of heartbeat message exchanging to the clone. Through this scheme, the dumb period problem can be solved and the HBFD can keep working on the monitoring task.

False detection is an inherent problem of HBFDs. What we can do is to add a checking procedure. When a new MA is generated to replace the failed MA, the new MA should check the status of the failed MA. If the new MA finds a false detection, it will kill itself to avoid the duplicated execution. For the message overheads caused by HBFDs, we can not reduce it only through decreasing the frequency of heartbeat messages exchanging, if we want to maintain quick failure detection. According to [CHE02], decreasing the frequency of heartbeat message exchanging will make the time of failure detection longer. These two problems can be solved by our newly proposed NTFD in Chapter 3.

4.3.2 NTFD in MA Systems

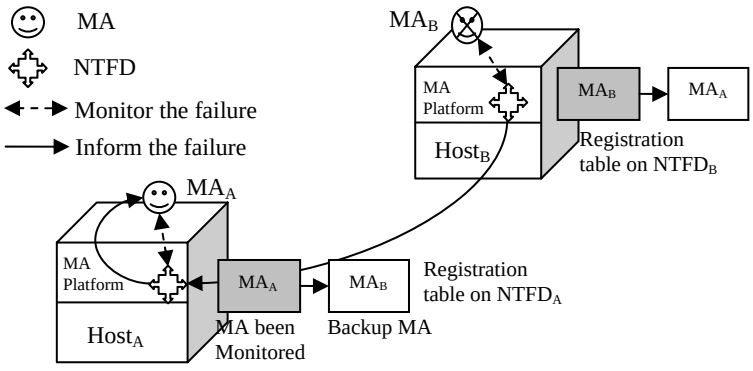


Figure 4.8 NTFD in MA System

NTFD is an alternative approach to implementing failure detector with 100% accuracy and very low message cost. For NTFD, the underlying failure detection mechanism on an MA platform will detect the failure status of local MAs, and once a failure is detected, a failure

notification will be sent to the backup. Figure 4.8 shows the NTFD in an MA system. In NTFD approach, all the MAs running on an MA platform will be monitored by a NTFD running on the host.

Each NTFD maintains a registration table for the MAs that it is monitoring. Each MA will make a registration to record the address of its backup MA. For example, in Figure 4.8, MA_A and MA_B are each other's backup and they are monitored by $NTFD_A$ and $NTFD_B$ respectively. MA_A makes a registration in the registration table of $NTFD_A$ and nominates MA_B as its backup MA. MA_B does the same procedure on $NTFD_B$ except that its backup MA is MA_A. Before an MA migrates to a new host, it will deregister at the current host. The registration procedure and the deregistration procedure are similar. Figure 4.9 shows a complete registration table.

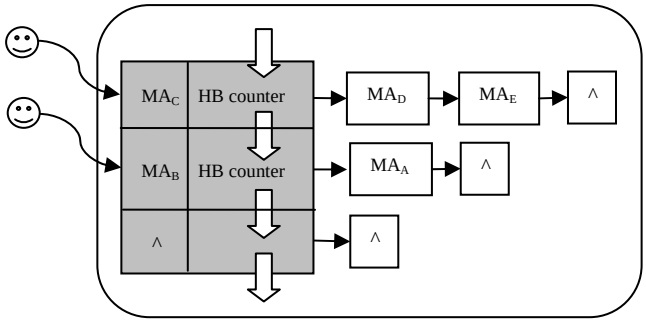


Figure 4.9 Registration Table for FDs in MA System

NTFD also has the dumb period problem, so it needs the same handover procedure as in HBFD. But comparing with HBFD, NTFD is more efficient and scalable, and can guarantee the strong accuracy property. However, a problem for NTFD is that it can not guarantee the completeness property as well as by HBFD, because the backup MA can not acknowledge the failure if the failure notification message is lost. However, sometimes the accuracy property is more important. For example, in RMAA or AMAA, if one MA failed and the backup MA does not know the failure, the backup MA may finish the task by itself

if it will not fail. But if HBFD makes a false detection, the backup MA may cause the duplicated operations, which is unacceptable in some applications like the e-commerce. In Chapter 3, we have concluded that the overall performance of NTFD is better than HBFD.

4.4 Performance Analyses and Evaluations

In this section, we first make an analytic analysis on the execution time for the different primary-backup based fault tolerant MA execution algorithms, and then describe the result of our simulation study.

4.4.1 Analysis on Execution Time and Message Cost

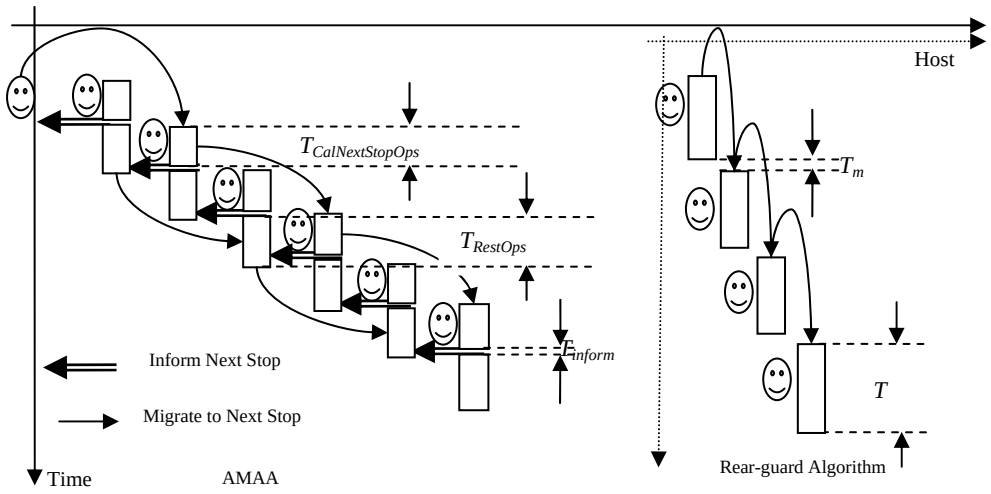


Figure 4.10 Execution Time Comparisons

In the following discussion, we assume that the execution time T on each host for an MA is the same. N is the number of hosts. The time for an MA to migrate from the current host to the next stop is T_m. T_{Task_exe} is the total execution time for each algorithm. For rear-guard algorithm, when the working MA starts a migration, it will inform the rear guard MA to follow it. We assume that the time needed for this operation is T_{inform}. In RMAA, $T_{landing}$ is

the time needed by the landing procedure. In AMAA, like the rear-guard algorithm, T_{inform} is the time needed by the operation of LMA informing SMA of the next stop, and according to Figure 4.5, we assume that the time taken by each *CalNextStopOps* is $T_{CalNextStopOps}$; the time taken by each *RestOps* is $T_{RestOps}$. It is obviously that $T = T_{CalNextStopOps}+T_{RestOps}$. For simplicity, we do not consider the cost of synchronization messages, because they are needed by all of our discussed algorithms.

For the rear-guard algorithm, the whole task is finished by the single working MA and no parallel processing is involved (Figure 4.10). So we have: $T_{Task\_exe} = N(T+T_m+ T_{inform})$. For RMAA, the FMA and RMA execute in parallel, so ideally the execution time is: $T_{Task\_exe} = N(T+T_m+T_{landing})/2$. AMAA allows partial parallelism in MA execution, and its execution time depends on how much job is done in parallel. From Figure 4.10, we can figure out how to compute the execution time for AMAA as shown in Formula (1) and (2).

$$T_{Task\_exe}=\begin{cases} N\,(T_m+ T_{CalNextStopOps} + T_{inform})+ T_{RestOps} & T_{RestOps} < T_m+ T_{CalNextStopOps} + T_{inform} \quad (1) \\ (N+1)(T_m+ T + T_{inform}) & T_{RestOps} \geq T_m+ T_{CalNextStopOps} + T_{inform} \quad (2) \end{cases}$$

The parameter $T_{RestOps}$ determines the degree of parallism that can be achieved. If we can increase the $T_{RestOps}$, the execution time of AMAA will be reduced. However, the reduction in the execution time is bounded that the total time will be no less than $(N+1)(T_m+T+T_{inform})/2$, if $T_{RestOps}$ is greater than $T_m+T_{CalNextStopOps}+T_{inform}$. We define this $T_{RestOps}$ as the AMAA critical value. For AMAA involving n MAs, it is easy to see that the task execution time will be $(N+1)(T_m+T+T_{inform})/n, (2\leq n \leq N,\ T_{RestOps} \geq T_m+T_{CalNextStopOps} +T_{inform})$.

Table 4.1 Execution Mode, Execution Time and Message Cost Comparisons

Algorithm	***Itinerary***	***Execution Mode***	***Theoretical Execution Time (N hosts)***		***Message cost (N hosts)***	
			2MAs	***nMAs (n>2)***	***2MAs***	***nMAs (n>2)***
RearGMA	Self-initiate	Non-parallel	$N(T+ T_m + T_{inform})$	$N(T+ T_m + T_{inform})$	*2N*	*2N*N*
RMMA	Predefined	Full-parallel	$N(T+ T_m+ T_{landing})/2$	$N(T+ T_m+ T_{landing})$/n	*2N*	*2N*
AMAA	Self-initiate	Partial-paral.	$(N+1)(T_m +T+ T_{inform})/2$	$(N+1)(T_m +T+ T_{inform})/n$	*N*	*2N/n*

Table 4.1 summarizes the execution modes, execution time and message cost for all the algorithms discussed in this chapter. Note that for AMAA in Table 4.1, we assume that $T_{RestOps}$ is set as the critical value. We can see that RMAA can provide the fastest execution speed (T_{inform} is almost the same with $T_{landing}$). But RMAA needs a predefined itinerary and also requires the system to allow a random hosts accessing sequence. These requirements make RMAA inflexible. AMAA has the same degree of flexibility as the rear-guard algorithm, but its execution time can only be shortened if the next stop can be calculated quickly (then $T_{RestOps}$ will becomes bigger). If AMAA can only determine the next stop at the last step of its operation ($T_{RestOps} = 0$), the execution time will be the same as the rear-guard algorithm: $T_{Task\_exe} = N(T_m + T_{CalNextStopOps}+T_{inform})+T_{RestOps} = N(T_m+T+T_{inform})$. But normally the $T_{RestOps}$ will not be zero, because an MA has to perform some routing operations on a host at last, such as deregistration, release resources, etc. So we can always gain the partial parallelism so as to shorten the execution time. The results in Table 4.1 are just the theoretical values. In practice, the real execution time will be longer due to various overheads. We will compare the realistic execution time in the simulation.

The message cost of the algorithms mainly includes the coordination message cost for the synchronization and for informing the next stop, which is illustrated in Table 4.1. Please note that the message cost in Table 4.1 does not consider the message cost caused by failure detector, because failure detection is a kind of service provided by the platform in

our system. We will consider the message cost caused by failure detector in simulation study.

4.4.2 Simulation Results

In order to compare the realistic execution time in a real environment, we performed simulations of the Rear-Guard algorithm (RearG), RMAA and AMAA on the Naplet MA platform [Nap]. For simplicity, we only implement two MAs in each algorithm. The simulations are carried out on a PC with Pentium 4 CPU (2.5GHz), 256MB RAM. The software environment is: Window XP, Java version 1.4, and Naplet MA platform. Five Naplet MA platforms are installed on the PC and we simulated the algorithms with the MA travelling 15, 25, 35, 45, 55, 65, 75, 85, 95, 105 nodes respectively using different failure detection services (HBFD and NTFD). The number of MA failures is set to be 1/20 of the total number of hosts that have been visited and the failures are uniformly distributed along its itinerary, which indicates that for every 20 hosts, there is one failure occurring. The exchange frequency of the heartbeat messages is 5 messages per second. For AMAA, we set the $T_{RestOps}$ to be its critical value, which means that an MA will send out the next stop message in the middle of its execution.

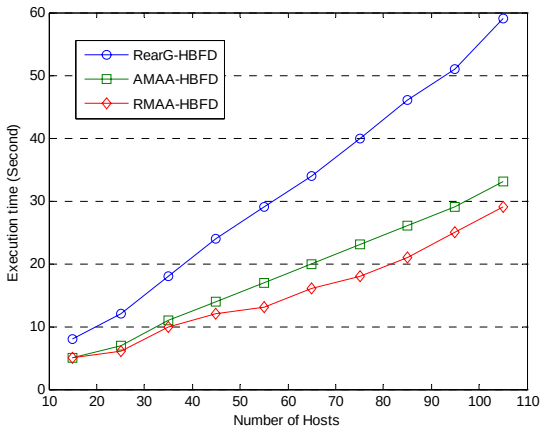


Figure 4.11 Execution Time with HBFD

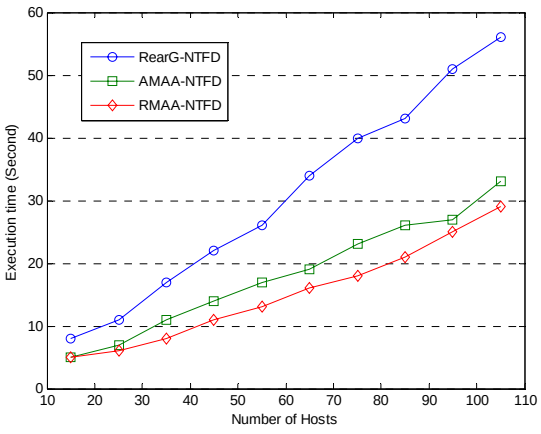


Figure 4.12 Execution Time with NTFD

From the simulation results in Figure 4.11 and 4.12, we can see that the execution time of RMAA and AMAA takes about half of the rear-guard algorithm's execution time. But AMAA is a little slower than RMAA due to the partial parallelism and the synchronization cost such as informing of the next stop. Comparing Figure 4.11 and 4.12 we can see that using different failure detectors will not make much difference in the algorithms' execution time. This is because the interface between a monitored MA and HBFD is almost the same as the interface between a monitored MA and NTFD in our implementation, which results in the similar execution time for an algorithm using different failure detector.

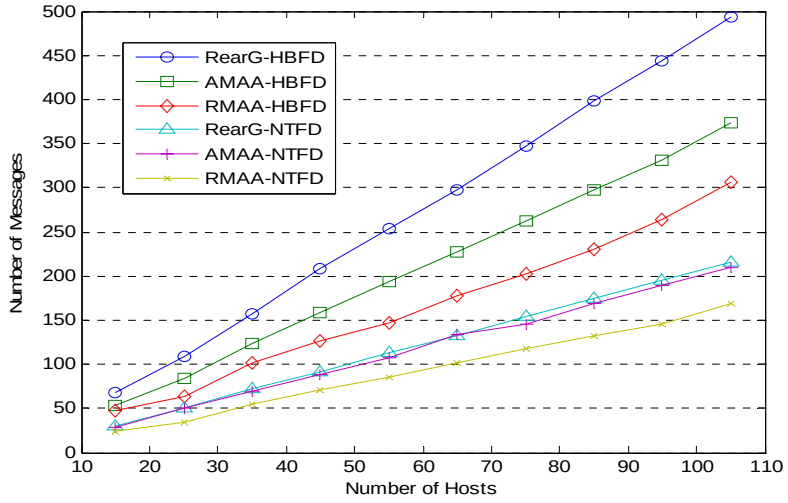


Figure 4.13 Message Cost Comparison

However, different types of failure detectors make big differences on the message cost. Figure 4.13 shows that the messages cost increases much more quickly with HBFD. That is because the heartbeat messages cost is in direct proportion to the execution time and takes up the most part of the exchanged messages during MA's execution. Longer execution time will cause more heartbeat messages. But the message cost caused by NTFD becomes very low, because NTFD's message cost has no relation with the execution time but is in proportion to the times of MA failure. Under NTFD, the message cost mostly depends on the fault tolerant algorithms' own message cost (Table 4.1). That is why in Figure 4.13, the

message costs of Rear guard MA and AMAA are nearly overlapped, because their own message costs are all *2N*.

4.5 Summary

In this chapter, we firstly proposed two efficient primary-backup based algorithms for fault tolerant MA execution. The algorithms allow for parallel processing and provide tolerance of MA failure. Then we discussed the failure detection techniques for primary-backup based MA fault tolerance algorithms. We introduce our proposed NTFD in MA systems. Analytic and simulation results show that the proposed algorithms can improve system's execution speed dramatically.

Appedix: Implementation in a Java Based MA System

```
/**************************************************
A Genaral MA(G-MA) class. It has the basic functions for an MA
**************************************************/
pulic abstract class GenaralMA
{
   public GenaralMA ( Itinerary iti, Tasks t) {…}
   public void MigrateToNext (Addr addr) {…}
   public void LandOn () {…}
   public void ProcessingMsg (Message msg) {…}
}
```

//Implementation of RMAA

```
//RMAA is independent of MA platform. RMAA is a class which implements all the functions of
//RMAA algorithm. User just needs to create a RMAA object and provides the Task and Itinerary to the
//RMAA object.

/*******************************************
  RMAA_MA class
*******************************************/
pulic class RMAA_MA extends GenaralMA
{
RMAA_MA peerMA;
State state ;
public RMAA_MA(Itinerary iti, Tasks t)
        {
           super(iti, t);
}
```

```
public void AssignPeerMA(RMAA_MA, peer)
{
        peerMA = peer ;
}

public void MigrateToNext(Boolean fixfailure, Addr addr)
{
 if (fixfailure == ture)
{
super.MigrateToNext(addr); //call G-MA's migration process
state = fixfailure;
}
 else
    {
     if (CkMeeting() != Yes)    //check the meeting event
        {result = sync(peerMA.ID); //synchronize the computing //results
         super.MigrateToNext();    //call G-MA's migration process
}
 else
        BackHome();    //if the MAs meet, both MAs return home.
}
}

public void LandOn ()
{//After the cloned ma lands on the host, it will check the reported ma is really failed or not.
        if (state == fixfailure AND Check(peerMA) = = ReallyFailed )
           {flush();    //if the reported ma really failed, its job will be continued
            resume();
           }
        else
        super.LandOn();
}

public void ProcessingMsg (Message msg)
{
 if (msg = = MA_Failure) //get asynchronous message from failure detector
                {ma = clone(result);    //Clone a new ma according to the failed ma's infomation.
                 ma. MigrateToNext (true, msg.addr); //the cloned ma migrates to the host.
                }
}
}

/*********************************************
 RMAA class
*********************************************/
pulic class RMAA
{
   RMAA_MA fma, rma;
   public RMAA(Itinerary iti, Tasks t)
   {
        iti = getForwardIti();
        fma = new RMAA_MA(iti, t);
        iti = getReverseIti();
        rma = new RMAA_MA(iti, t);
```

```
            fma.AssignPeerMA(rma) ;
            rma.AssignPeerMA(fma) ;
    }

public Launch()
     {
       fma. MigrateToNext ();
       rma. MigrateToNext ();
     }
}

/*********************************************
  Using RMAA class in user applications
*********************************************/
1. RMAA rmaa = new RMAA (Itinerary, Task); //RMAA creates two members: a FMA and a RMA;
2. rmaa.Launch(); //FMA and RMA are launched;

//Implementation of AMAA
//AMAA is independent of MA platform. AMAA is a class which implements all the functions of
//AMAA algorithm. User just needs to create a AMAA object and provides the Task and Itinerary to the
//AMAA object.

/*********************************************
  AMAA_MA class
*********************************************/
pulic class AMAA_MA extends GenaralMA
{
AMAA_MA peerMA;
State state ;
public AMAA_MA(iti, Tasks t)
        {
           super(iti, t);
}

public void AssignPeerMA(AMAA_MA, peer)
        {
           peerMA = peer ;
}


public void MigrateToNext(Boolean fixfailure, Addr addr)
{
 if (fixfailure == ture)
     {
       super.MigrateToNext(addr); //call G-MA's migration process
       state = fixfailure;
     }
 else
     {
       if (CkMeeting() != Yes)    //check the meeting event
          {result = sync(peerMA.ID); //synchronize the computing results
           super.MigrateToNext();    //call GenaralMA's migration process
          }
       else
          BackHome();    //if the MAs meet, both MAs return home.
```

```
        }
  }

  public void LandOn ()
  {//After the cloned ma lands on the host, it will check the reported ma is really failed or not.
          if (state == fixfailure AND Check(peerMA) = = ReallyFailed )
             {flush();    //if the reported ma really failed, its job will be continued
              resume();
             }
          else
          super.LandOn();
  }

  public void ProcessingMsg (Message msg)
  {
   if (msg = = MA_Failure) //get asynchronous message from failure detector
                  {ma = clone(result);    //Clone a new ma according to the failed ma's infomation.
                   ma. MigrateToNext (true, msg.addr); //the cloned ma migrates to the host.
                  }
             }

   public void NotifyNextStop()
         {
          msg = new Message(NextStop, peerMA);
          SendMsg(msg);
         }

   public void waitNotification(Message msg)
         {
          NextStopAddr = msg.addr;
         }
  }

  /*********************************************
  AMAA class
  *********************************************/
  pulic class AMAA
  {
     AMAA_MA lma, sma;

     public AMAA(Addr First_host_addr, Tasks t)
          {
            lma = new AMAA_MA(First_host_addr, t);
            sma = new AMAA_MA(null, t);
            lma.AssignPeerMA(sma) ;
            sma.AssignPeerMA(lma) ;
           }

     public Launch()
          {
           lma. MigrateToNext ();
           rma. waitNotification();
          }
  }
```

```
/*******************************************
  Using AMAA class in user applications
*******************************************/
1. AMAA amaa = new AMAA (NULL, Task); //AMAA creates two members: a FMA and a RMA;
2. amaa.Launch(); //LMA and SMA are launched;
```

Chapter 5
Checkpointing-based Algorithms

In this chapter, we aim to solve two problems with checkpointing-based algorithms for the fault tolerance of MA systems: (1) Determining the checkpoint placement for independent checkpointing and (2) Checkpointing-based algorithm for MA group. For the first problem, we introduce our proposed checkpoint placement algorithms, and for the second problem, we present our proposed communication induced checkpointing (CIC) based algorithms. For simplicity, in this chapter we assume all the operations executed by the MAs are idempotent, so the property of exactly once execution is not necessary to be considered. For non-idempotent operations, transaction support is needed to maintain the system consistency during recovery. We will introduce MA transaction in Chapter 6. This chapter is arranged as follows. Section 5.1 describes our proposed three checkpoint placement algorithms and related simulations. Section 5.2 describes our newly designed CIC based algorithms for MA group, which includes a basic CIC algorithm (Basic-CIC) and a deferred message processing algorithm (DM-CIC). Section 5.3 summarizes this chapter.

5.1 Checkpoint Placement Algorithms for MA System

Many checkpoint placement algorithms have been proposed for conventional computer systems. However, there is no work done on how to determine a proper checkpoint placement for an MA and no study on how the two widely used strategies (equidistant and equicost) can be applied. Algorithms for checkpoint placement in conventional computer systems cannot be ported directly to MA systems because the system model of the MA system is different from that of conventional systems. In an MA system, the MA carries

out its assigned tasks on the hosts along its itinerary. The tasks are separated by the migration operations. Within each migration operation, the MA terminates its execution on the previous host and prepares for the migration. The preparation includes releasing the allocated resources (stack, memory) and packing the code and data sections of the MA into an image. Then the image will be transmitted to the next host. When the next host receives the image, it will perform system defined checking (i.e., CRC checking) to guarantee that the image is not damaged during the transmission, and incarnate the image to a new MA if the image passes the checking. The new MA will continue the execution on this new host. Since the MA is executing on a new host, the execution environment is totally new and has no any relation with the previous host; the stack and memory for this agent is reestablished. Therefore, we can claim that the failure of the MA on the current host is independent with its failure on the previous host. This characteristic corresponds exactly to the Markov chain property. Consequently, the execution of an MA in an MA system can be modelled as a discrete-parameter Markov chain. In the following subsections, we first define this model, and then propose checkpoint placement algorithms for MA systems.

5.1.1 System Model

We consider an MA system model where a single MA executes and migrates along a predefined or self-initiated itinerary. With a predefined itinerary, the agent knows all the hosts that it will visit, while with a self-initiated itinerary, the agent only knows the first host it will visit and the following hosts are determined by the execution results on the previous host. The itinerary consists of N hosts, $Host_0$, $Host_1$,…, $Host_{N-1}$, and a home node, *Home*. *Home* launches the MA to $Host_0$ reliably, so we consider that the MA starts its execution on $Host_0$.

We assume that an MA can take an independent checkpoint right after it lands on a host and before starts its execution. It cannot take a checkpoint during its execution on a host. After the agent finishes its execution on a host, it will migrate to the next host according to the itinerary. This process will continue until all the hosts have been visited. During the migration, only the code and data (computing results) of an MA on previous host will be transmitted to the next host. On the new host or during the recovery process from a checkpoint, the MA is incarnated and its execution environment is reconstructed. Based on these observations, we assume that all the failures of an MA are independent from each other. Accordingly, the execution of an MA is modelled as a discrete-parameter Markov Chain.

For simplicity, we assume that the stationary transition probabilities of this Markov Chain are fixed. Therefore, it is a homogeneous discrete-parameter Markov Chain. Let *I* be the state space and *T* be the parameter space, both are finite and discrete.

$$I = \{0, 1, 2, \ldots N+1\}, N>1.$$

$$T = \{0, 1, 2, \ldots\}.$$

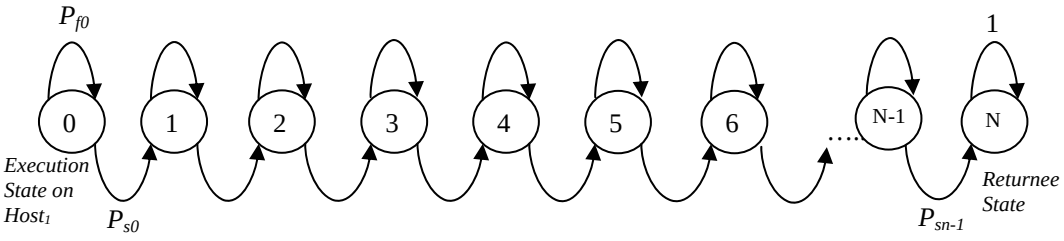


Figure 5.1 State Transition Graph

Figure 5.1 illustrates the state transition graph for an MA's execution. In state space *I*, state *i* ($0 \leqslant i \leqslant N-1$) denotes the *Execution State* of an MA on $Host_i$. Failure may happen during the execution and migration of the MA with probability P_{fi} on $Host_i$. P_{fi} is independent from each other. A failure is detected immediately and the agent recovers and starts the

re-execution from the latest checkpoint with probability 1. Such a failure-recovery process is called a *failure-recovery round*, which may happen X_i times on $Host_i$. X_i is a random variable and its distribution is listed in Table 5.1.

Table 5.1 Distribution Table of X_i

X_i	0	1	2	3	…	N	…
p	$1- P_{fi}$	P_{fi}	$(P_{fi})^2$	$(P_{fi})^3$	…	$(P_{fi})^n$	…

The probability of successfully finishing the execution on the current host and landing on a new host is P_{si} ($P_{si} = 1-P_{fi}$). P_{si} is independent from each other. The last state $N+1$ is the *Returnee State*, which means that the MA has returned home. In *Returnee State*, we assume that the MA is able to recover under the user's control with probability 1, no matter what failure occurs.

The purpose of a checkpoint placement algorithm is to determine the proper (or optimal) checkpointing interval so as to reduce (or minimize) the system cost, which consists of the cost of making checkpoints and the cost of recovery when failures happen. The recovery cost includes the cost of incarnating the new MAs from the checkpoints and the re-execution cost. Table 5.2 shows the notions for the various types of cost in this model.

Table 5.2 Various Types of Cost

Parameter	*Description*
C_{sys}	Overall system cost
C_i	Recovery cost for one failure-recovery round on $Host_i$
C_{cp}	the cost of making a checkpoint
C_I	the cost of incarnating an MA from a checkpoint
E_i	the cost of the execution on $Host_i$
C_{Xi}	the re-execution cost on $Host_i$

C_{cp} is the cost to store a checkpoint on disk while C_I is the cost to read a checkpoint from the disk. Although the direction of the data flow is different, the cost is similar: both of them can be evaluated by the cost of I/O operations and have no relation with specific applications. However, E_i is related with the specific application on $Host_i$ because different application has different execution cost. Therefore, we assume that C_{cp} and C_I are known in advance. E_i is provided by the MA platform of $Host_i$. P_{fi} is also maintained by the MA platform of $Host_i$. An MA does not know E_i and P_{fi} before it retrieves them from the MA platform of $Host_i$.

5.1.2 Methodology

The algorithms proposed in the next sections are based on the equidistant and equicost checkpointing strategies. The principle of these checkpointing strategies is to seek a better balance between the expected recovery cost and the checkpointing cost. An optimal checkpointing interval can be achieved in conventional systems if the failure rate is the same during the entire execution duration of a program. Similarly, we can also get the optimal checkpointing interval for an MA if all the failure rates P_{fi} are the same. Otherwise we cannot get the optimal checkpointing interval.

In our model, checkpointing cost C_{cp} is known in advance, so we just need to derive the expected recovery cost within a checkpointing interval to seek the balance between C_{cp} and recovery cost. In following, we firstly determine the optimal checkpointing interval for an MA under an ideal condition (all the P_{fi} are the same), and then using the result to derive heuristics for designing checkpoint placement algorithms in a realistic MA system, where P_{fi} are independent in nature.

5.1.3 Expected Recovery Cost within a Checkpointing Interval

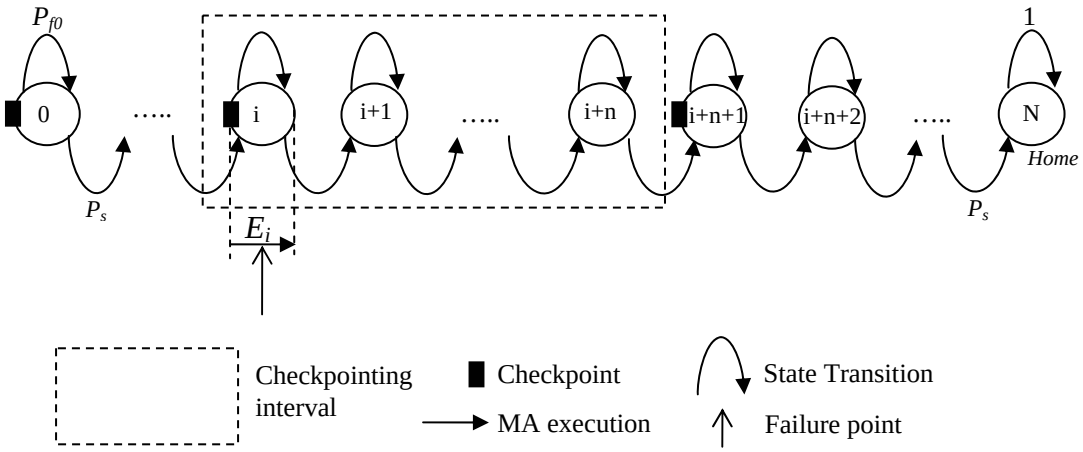


Figure 5.2 The Calculation of System Cost in a Checkpointing Interval

With the reference to Figure 5.2, suppose an agent takes a checkpoint on $Host_i$ before it initiates its execution, and the agent takes its next checkpoint on $Host_{i+n+1}$. The checkpoint interval is defined as the number of hosts between $Host_i$ and $Host_{i+n}$ (including $Host_i$ and $Host_{i+n}$). The probability of the MA failure on $Host_i$ is P_{fi}, and the number of failure-recovery rounds is X_i with its distribution listed in Table 5.1. The expected number of failure-recovery rounds (R_i) is given by Equation 1:

$$R_i = E(X_i) = \sum_{r=1}^{\infty} r * (P_{fi})^r = P_{fi}/(1-P_{fi})^2 \qquad (0 \leqslant i \leqslant N-1) \qquad (1)$$

The recovery cost C_i on $Host_i$ consists of the incarnation cost C_I and the re-execution cost C_{xi}, which refers to the cost of the execution starting from the checkpointing point to the failure point. Since the failure point is evenly distributed in the duration of an MA's execution on a host (as shown in Figure 5.2), the expected re-execution cost in one failure-recovery round is given by Equation 2.

$$E(C_{Xi}) = \int_0^{E_i} x(dx / E_i) = (1/E_i)\int_0^{E_i} x dx = E_i/2 \qquad (2)$$

Accordingly, the expected recovery cost on $Host_i$ is:

$$E(C_i) = R_i*(C_I + E_i/2) \qquad (3)$$

For a failure-recovery round on $Host_{i+1}$, since the recovery should start from $Host_i$, the expected recovery cost on $Host_{i+1}$ is: $E(C_{i+1}) = R_{i+1}*(C_I+E_i+E_{i+1}/2)$ and the expected recovery cost on $Host_{i+n}$ is: $E(C_{i+n}) = R_{i+n}*(C_I+E_i+\ldots+E_{i+n}/2)$.

Since the failures are independent, the total expected recovery cost for the checkpoint interval illustrated in Figure 5.2 is given by Equation 4 below ($C_{i,i+n}$ denotes the total expected recovery cost from $Host_i$ to $Host_{i+n}$):

$$E(C_{i,i+n}) = E(\sum_{k=0}^{n} C_{i+k}) = \sum_{k=0}^{n} E(C_{i+k}) \qquad (4)$$

Together with the cost for the checkpointing and execution cost within this interval, the overall system cost can be calculated by Equation 5:

$$C_{sys} = C_{cp}+E(C_{i,i+n})+(n+1)E_i = C_{cp}+\sum_{k=0}^{n} E(C_{i+k}) +(n+1)E_i \qquad (5)$$

5.1.4 Optimal Checkpointing Interval under an Ideal Condition

If the cost of checkpointing and the failure rate are fixed during the execution of a program, the optimal placement strategy would be to place the checkpoints in fixed equidistant intervals [TAN84, ZIV97]. In our model, C_{cp} is fixed, but P_{fi} are different from each other (the same to E_i). To derive the heuristic rules for checkpoint placement algorithms in a realistic MA system where P_{fi} are independent in nature, we first consider an ideal condition: all the P_{fi} and E_i are the same.

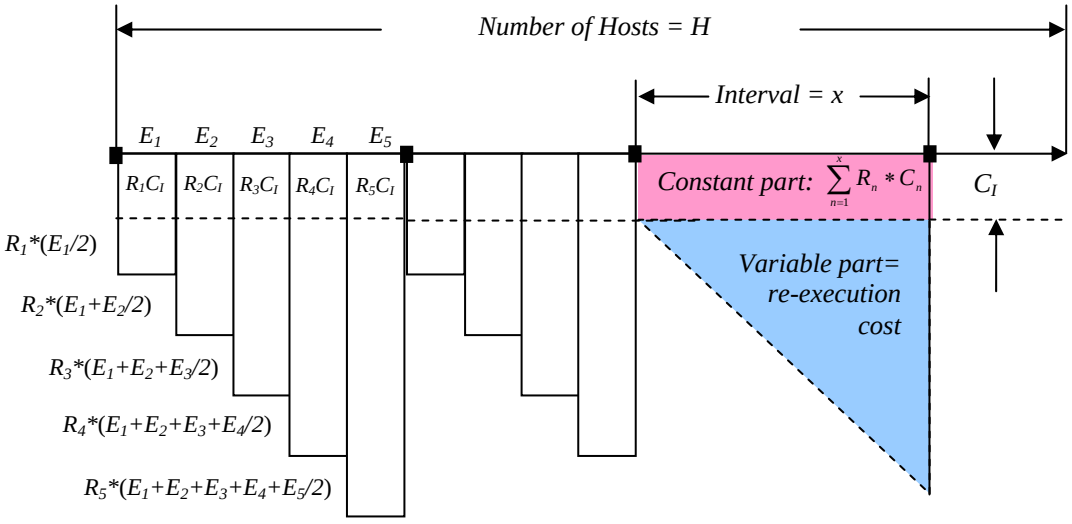


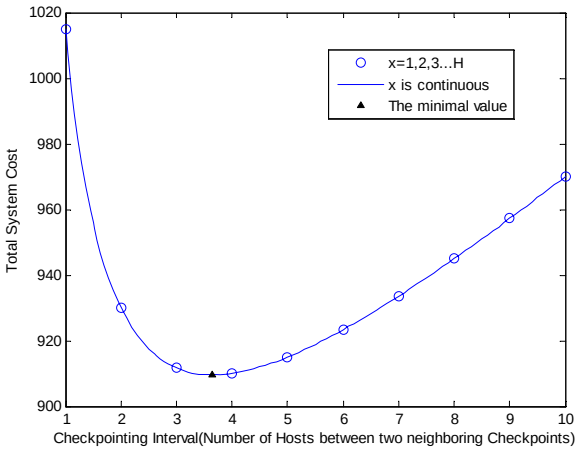
Figure 5.3 The Expect Cost in a Checkpointing Interval

Fixing P_{fi} makes the MA system to have a fixed failure rate. Having E_i with the same value on all the hosts makes the intervals equidistant if each interval contains the same number of hosts. Since we do not consider checkpointing in the middle of an MA's execution on a host, as shown in Figure 5.3, the granularity of the checkpointing interval is one host. To determine the optimal interval, we assume that an interval contains x hosts. Then the expected total system cost is calculated as follows:

$$C_{sys} = (H/x)[x*C_I + (RE+(x-1)RE)(x-1)/2+xER/2+C_{cp}]+H*E \qquad (x=1,2,3,\ldots N)$$
$$=(H/x)[x*C_I+x^2RE/2+C_{cp}]+H*E = H*C_I+xHRE/2+HC_{cp}/x+H*E \qquad (6)$$

To get x that produces minimal value for Equation (6), we consider that x is continuous as shown in Figure 5.4. Then we can get x by derivative.

$$C_{sys}' = REH/2 - C_{cp}H/x^2 \qquad (7)$$
$$C_{sys}'' = 2\,C_{cp}H/x^2 \qquad (8)$$



The curve is got with:
H=100;
C_I=2;
C_{cp}=2;
R=0.05;
E=6;

Figure 5.4 Different System Cost under Different Checkpointing Interval

Since $C_{sys}''>0$, C_{sys} has the minimal value. Let $C_{sys}'=0$, we can get the value of x to make C_{sys} minimal.

$$C_{sys}' = REH/2 - C_{cp}H/x^2 = 0$$

$$x = \sqrt{2C_{cp} / RE} \qquad (10)$$

With the optimal x derived from Equation 10, the cost of the expected total re-execution cost in an interval (Figure 5.3) can be calculated using Equation 11 below.

$$\sum_{i=1}^{x} C_{Xi} = (RE+(x-1)RE)(x-1)/2+xER/2 = C_{cp} \qquad (11)$$

Equation 10 implies that the optimal checkpointing interval is only related with C_{cp}, R and E, and has nothing to do with C_I. Equation 11 tells us that within an optimal checkpointing interval, the expected total re-execution cost (variable part in Figure 5.3) equals exactly to C_{cp}. The implications can be used as heuristic rules in designing checkpoint placement algorithm based on the equicost strategy. However, notice that Equation 11 is gotten under the assumption that x is continuous, but x is actually discrete. Therefore, the optimal value

should be the integer that is neighboring *x*. In Figure 5.4, the optimal interval should be *"3"* or *"4"*. We must determine the optimal interval by checking which value leads to a smaller result of Equation 6.

5.1.5 Checkpoint Placement Algorithms

In a real MA system, the failure rates cannot be the same on all the hosts and links, so we need to design algorithms working in general conditions. From the analysis in Section 5.1.4, we propose three checkpoint placement algorithms.

Algorithm 5.1: *Equidistant with failure rate estimation:* before an MA starts its travelling, it estimates a uniform failure rate for the MA system and decides the checkpointing interval by using Equation 10 in Section 5.1.4. The estimation can be made by using all the P_{fi} collected from the MA platforms on each host. The pseudo-code of these algorithms is shown in the following box.

Algorithm 5.1: ***Equidistant with failure rate estimation*** **(pseudo-code format)**

```
itinerary= MA.GetItinerary(); // Itinerary should be predefined {Host1; Host2; … HostH};
Pf[ ] = MA.GetFailureProbability(itinerary); //Collecting probabilities from each host;
AvrPf = MA.AveragePf(Pf[ ]);                 // Eastimating average failure rate;
R= AvrPf /Power(1- AvrPf);                   //Get R according Equation 1;

OptInterval_float = Sqrt(Ccp/R*E); //calculate the checkpointing interval according to Equation 10;
OptInterval_Ceiling=Ceiling(OptInterval_float); //Get two neighboring integer;
OptInterval_Floor=Floor(OptInterval_float);

OptC= OptInterval_Ceiling*H*R*E/2+H*Ccp/OptInterval_Ceiling //Calculate the system cost
OptF= OptInterval_Floor*H*R*E/2+H*Ccp/OptInterval_Floor      // according to Equation 6;

if (OptC< OptF)     //the integer that leads to smaller cost becomes the checkpointing interval;
   OptInterval= OptInterval_Ceiling;
else
   OptInterval= OptInterval_Floor;

MA.Context.CpInterval= OptInterval; //MA will carry the interval can do checkpointing
MA.Migration(itinerary);                  // every "OptInterval" hosts.
```

In Algorithm 5.1, to collect P_{fi} from all the hosts along the itinerary, a predefined itinerary is needed. Since we require only an estimated average failure rate, the interval obtained here is only an approximate optimal interval.

Algorithm 5.2: *Equidistant by enumeration:* before an MA start its travelling, it collects all the P_{fi} maintained by the MA platforms on the hosts, and enumerate the results of Equation 6 with the value of *x* from *1* to *H* to get the *x* that leads to the smallest result.

Algorithm 5.2: *Equidistant by enumeration* (pseudo-code format)

```
itinerary= MA.GetItinerary(); // Itinerary should be predefined { Host1; Host2; …HostH};
Pf[ ] = MA.GetFailureProbability(itinerary); //Collecting failure probabilities from each host;

for i=1 to itinerary.NumberOfHosts //Calculate R according Equation 1
    R[i]=Pf[i]/Power(1- Pf[i]);
    RC[i]=E* R[i];                  //RC[i] is the re-execution cost on a single host i;
end

for interval=1 to itinerary.NumberOfHosts //Enumerate the system cost with each interval;
    TC=Ccp; // Initiate the total cost: A checkpoint will be made at the first host of an interval;
    accumulateC=0; // Temporary variable to record the re-execution cost within an interval;
    currentC=0;   //Temp variable to count current cost with "index" number of host has passed;
    index=1; //Temporary variable to record how many host has passed;

    for host=1 to itinerary.NumberOfHosts
        for i=1 to index   //this loop count the re-execution cost if failure happen on this host;
            accumulateC=accumulateC+RC[host-i+1];
        end
        currentC = currentC + accumulateC;
        if (index==interval)   //Passed hosts equals to the interval, a checkpointing is made;
            TC=TC+Ccp+currentC; //and then count the system cost;
            accumulateC=0;   //Restore the temporary variable;
            currentC=0;      //Restore the temporary variable;
            index=1;         //Restore the temporary variable;
        else
            index=index+1; //Still not reach the interval boarder
        end
    end
    if (index<interval)      //reach the tail of the itinerary;
        TC =TC+currentC;
    end
    allresults[interval]=TC;
end
OptInterval = MiniIndex(allresults[]); //the minimal value's index is the checkpointing interval;

MA.Context.CpInterval= OptInterval; //MA will carry the interval can do checkpointing
MA.Migration(itinerary);                // every "OptInterval" hosts.
```

In Algorithm 5.2, a predefined itinerary also is necessary for the P_{fi} collection. All the possible intervals will be tried to get the best equidistant interval. Obviously, Algorithm 1 cannot give a better equidistant interval than Algorithm 2, but the time complexity of Algorithm 1 ($O(1)$) is much lower then that of Algorithm 2 ($O(H^2)$).

Algorithm 5.3: *Equicost:* MA calculates the variable part (C_{var}) of the re-execution cost before its execution on each host. A checkpoint is made when C_{var} is equal to C_{cp} or greater than C_{cp} (according to the heuristic rules described in Section 5.1.4).

Algorithm 5.3: *Equicost* (pseudo-code format)

```
//MA initiation at home node;
MA.context={Ccp, E, FirstHost=TRUE, TC=0, accumulateC=0, currentC=0, index=1}; //
MA.Launch(firsthost); //

//Landing Procedure. Checkpointing is made in this procedure;
if (MA.FirstHost== TRUE)
   TC=Ccp; //A checkpoint will be made at the first host in an interval;
   MA.FirstHost== FALSE;
end
Pf=MAP.getPf();       //Pf is maintained by the MA platform on a host;
R=Pf/Power(1- Pf);    //Calculate R according Equation 1
RC[index]=E*R;            //RC[i] is the re-execution cost on a single host i;

for i=1 to index          //this loop for the re-execution cost if failure happen on this host;
   accumulateC=accumulateC+RC[i];
end

currentC = currentC + accumulateC; //the re-execution cost for current past hosts;

if (currentC>=Ccp) // if the re-execution cost for past hosts equals or exceeds Ccp, a checkpoint is
made;
   TC=TC+Ccp+currentC; //record the system cost
   accumulateC=0; //Restore the temporary variable;
   currentC=0; //Restore the temporary variable;
   index=1; //Restore the temporary variable;
else
   index=index+1;    //Still not reach the interval boarder
   if (ReturnedHome=TRUE)    //reach the tail of the itinerary;
      TC=TC+currentC;
end
```

Compare with Algorithms 5.1 and 5.2, the big advantage of Algorithm 5.3 is that it does not require a pre-defined itinerary. The decision on checkpointing is made during the execution of an MA. As we know, a most prevailing characteristic of MA is the autonomy, which allows an MA to determine its itinerary dynamically. Algorithm 5.3 has no constraint for MA to maintain this characteristic.

5.1.6 Performance Evaluation using Simulations

We need to evaluate the performance of the three proposed algorithms to find out which algorithm is the best in the sense that achieves the lowest system cost. Since we do not assume any distribution for the occurrences of failures on the hosts, there is no suitable way to make an analytic analysis. Therefore, we have carried out simulations to simulate the three algorithms using the Markov model shown in Figure 5.2 so as to evaluate the performance of the algorithms.

In our simulations, we consider an MA system with 100 hosts. On each host, the failure probability P_{fi} is a random number ranging from 0 to 1. We adopt a uniform unit for the system cost, which can be the execution time or some other metrics. We assume $C_{cp} \geqslant 1$, $E \geqslant 1$ and $C_I = C_{cp}$. The execution cost E can be less than C_{cp} if MA's tasks on hosts are short and have no I/O operations; otherwise E is larger than C_{cp}. Since the cost of the constant part (Figure 5.3) is the same for all the three algorithms, we only compare their differences on the variable part C_{var}. The results shown in the following figures are obtained with C_{cp} set to 2 and the execution cost E ranging from 1 to 20. For each cost metric, the same simulation is performed 100 times to get an average value for the checkpointing interval and the variable part C_{var}.

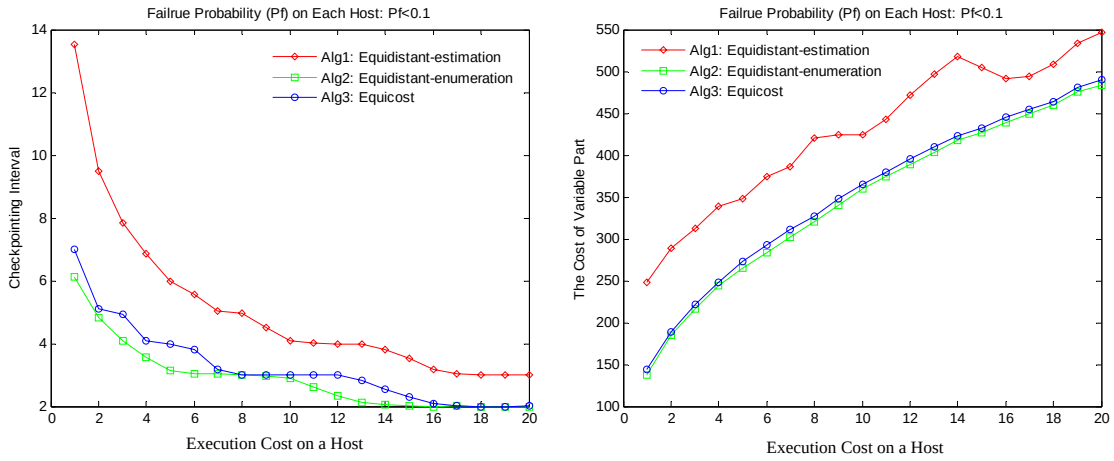


Figure 5.5 Checkpointing Interval and the Cost for Variable Part with $P_f<0.1$

Figures 5.5, 5.6 and 5.7 illustrate the average checkpointing interval and the corresponding variable part cost for the three algorithms under different failure probability ranges. A general tendency observed is that, with the failure probability decreasing, the checkpointing interval will become bigger. In terms of the system cost, Algorithm 5.2 (equidistant by enumeration) gives the best performance, while Algorithm 5.1 (equidistant with failure rate estimation) is always the worst. Algorithm 5.3 (equicost) leads to similar system cost with Algorithm 5.2, but it is much more flexible and suitable for an MA system as we discussed at the end of the last section.

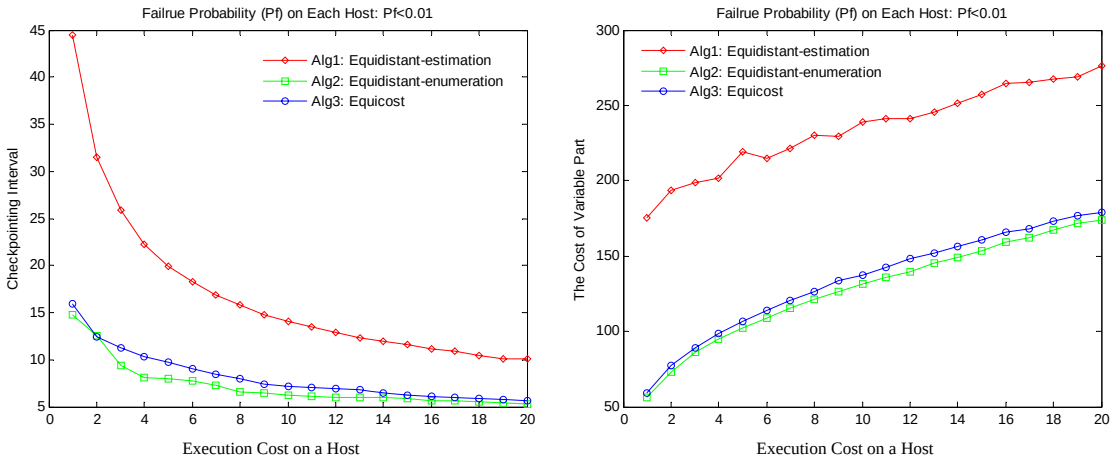


Figure 5.6 Checkpointing Interval and the Cost for Variable Part with $P_f<0.01$

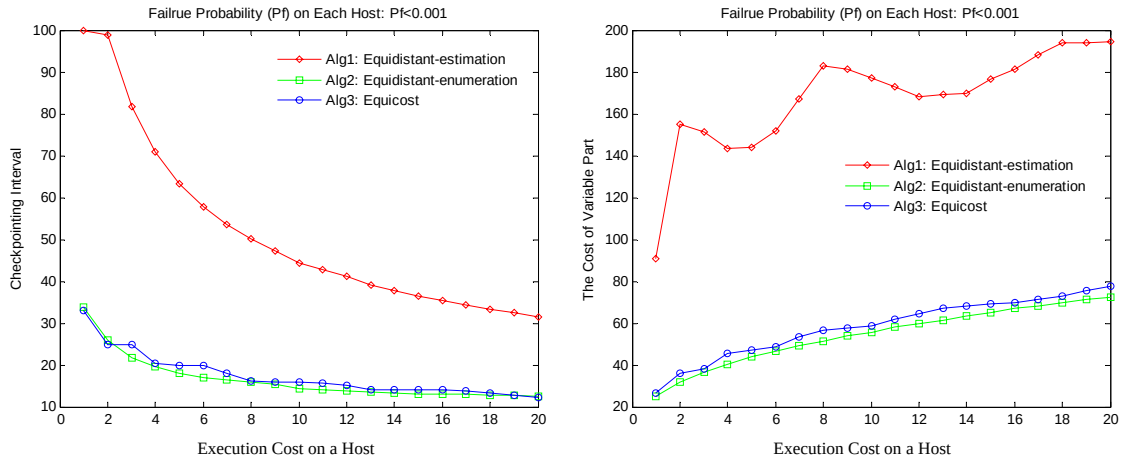


Figure 5.7 Checkpointing Interval and the Cost for Variable Part with P_f<*0.001*

5.2 CIC based Checkpointing Algorithms for MA Systems

If a group of MAs are concerned, independent checkpointing can not be applicable due to the domino effect. As we mentioned, CIC has been proven to be a flexible and efficient checkpointing scheme to prevent the domino effect, while allows each process to decide when to make checkpoint by itself. Up to now, we still can not find its application in MA systems. This motivates us to introduce CIC for the execution of MA group.

5.2.1 System Model

We consider an MA system consists of a group of cooperating MAs which form an MA group. In this MA group, each group member (a single MA) has a global unique group ID and MA ID. Each group member executes and migrates along a predefined or self-initiated itinerary. Group members may crash during its execution and migration. However, we assume that the underlying execution environment of MAs will not crash, which can be guaranteed by many existing local fault tolerant technique such as primary-backup system. To guarantee MA's reliable migration or prevent the possible crash, we assume that each

MA can take an independent checkpoint before its migration. After the agent finishes its execution on a host, it will migrate to the next host according to the itinerary. This process will continue until all the hosts have been visited.

Group members communicate by message passing. Application messages can be delayed, replicated or lost. However, we assume that the messages to request the recovery of the group of MA will not lost, which will be guaranteed by the transport layer of an MA system. For simplicity, we assume that all the operations executed by the MAs are idempotent, so the exactly once execution property is not necessary to be considered in this paper. For non-idempotent operations, MA transaction support is needed to maintain the system consistency during recovery (please refer Chapter 6).

5.2.2 A Typical CIC Algorithm and Related Theorem

In [BRI84], a typical index-based CIC algorithm has been proposed. The algorithm assumes that each process p_i maintains a logical clock lc_i which functions as p_i's checkpoint timestamp. The timestamp is an integer variable with initial value 0 and is incremented as followings:

1. lc_i increases by 1 whenever p_i takes a basic checkpoint.
2. p_i piggybacks on every message m it sends a copy of the current value of lc_i. We denote the piggybacked value as $m.lc$.
3. Whenever p_i receives a message m, it compares lc_i with $m.lc$. If $m.lc > lc_i$, then p_i sets lc_i to the value of $m.lc$ and takes a forced checkpoint before processing the message.

The set of checkpoints having the same timestamps in different processes is guaranteed to be a consistent state.

In [NET95], authors proposed and proved the following theorem (for z-cycle, please refer Section 2.2.1, Chapter 2):

Theorem 5.1 (Consistency): *A set of checkpoints S, where each is from a different process, can belong to the same consistent global snapshot if and only if no checkpoint in S has a zigzag path to any other checkpoint (including itself) in S (falls in a z-cycle).*

Therefore, if we can guarantee that there is no Z-cycle in a distributed system, then we can always find a set of consistent checkpoints, which means the domino effect can be avoided.

We adopt this typical index-based CIC algorithm and integrate it with the existing algorithms in MA systems, such as the independent checkpointing which is to guarantee the reliable migration for MAs. In addition to the integration, we make an important improvement for our proposed CIC algorithm for MA system, which adopt the deferred message processing. This improvement produces a new algorithm: DM-CIC. DM-CIC can avoid the forced checkpoint so as to get a better trade-off on the system performance.

5.2.3 Basic-CIC Algorithm for MA Systems

Basic-CIC algorithm directly applies the traditional CIC algorithm to MA systems except that the MA platform will assist to make the checkpoints. Basic-CIC algorithm is described in Algorithm 5.4.

In Basic-CIC, each MA maintains a logical clock (*LC*) and it is initiated to be *0*. Basic checkpoint is made by MA according to a predefined frequency (defined in each MA). Before a basic checkpoint is made, *LC* is increased by 1 and the basic checkpoint is tagged with *LC*. The value of *LC* is piggybacked in every message sent out by MA. We denote the

piggybacked *LC* as *M_LC*. When an MA receives a message, if *M_LC* > *LC*, then sets *LC* to be the value of *M_LC* and takes a forced checkpoint tagged with *LC* before processing the message.

Failure detector helps to detect the failures. If an MA failed, it will be detected by the FD installed in the MA systems and the corresponding MAP will be informed. Then the MAP will trigger the recovery process which will inform all the MAs in this group to rollback to the set of checkpoints having the same timestamp (*LC*). The following Theorem 5.2 shows that Basic-CIC can build up a consistent global snapshot.

Algorithm 5.4: *Basic-CIC*

```
//Procedure 1: MA initiated and launched;
MA.LC=0;                          // logical clock is initiated to be 0
MA.Launch(firsthost); //

// Procedure 2:Execution on a host;
MA.Exe();                  //MA executes on current host
MA.LC= MA.LC+1;    // LC is increased by 1
MA.MakeCP(MA.LC);        //A basic checkpoint is made and tagged by LC.

// Procedure 3:Message processing: SendMsg();
Msg = CreateMsg (msgbody, MA.LC); //all messages should be tagged by LC
SendMsg(Msg);

// Procedure 4:Message processing: GetMsg();
Msg = GetMsg (); //Get a message from the system
if (Msg.msgbody == "RollBack")
      RollBackTo(Msg.LC);
else
   MA.ProcessingMsg(Msg.msgbody); //MA handle the message;
M_LC= Msg.LC;                                  //Get the piggybacked logical clock;
if (M_LC > MA.LC)                          //if piggybacked logical clock greater than local clock;
    { MA.LC=M_LC;                           //Update local clock;
    MA.MakeCP(M_LC);                 //a forced CP is made;
    }

// Procedure 5:A failure of MA is detected
Msg = CreateMsg ("RollBack", MA.LC); //rollback to the CP with timestamp LC
MAP.SendToAllMembers(Msg);    //
```

Theorem 5.2 (Consistency): *In Basic-CIC, a set of checkpoints S, where each is from a different MA is guaranteed to be a consistent state if they have the same timestamp.*

Proof: The proof is by contradiction. Assume that a set of checkpoints *S* created in Basic-CIC, each from a different MA, has the same timestamp *t* but is not a consistent state. Then in *S*, there must be at lest one checkpoint CP_z which is involved in a z-cycle. Then, there are two cases: the rest of the checkpoints *S'* (*S'*=*S*-CP_z != Φ) are either on the left side of the z-cycle or on the right side.

Case 1. If there is one checkpoint CP_l on the left side, then CP_l happens before CP_z: $CPl \rightarrow CP_z$. Since the timestamp is monotonically increasing, the timestamp of the checkpoints made before CP_z by the same MA should be smaller then the timestamp of CP_z. Under such a scenario, according to Procedure 4, a forced checkpoint CP_f with timestamp *t* should be made before CP_z. Therefore, the timestamp of CP_z should be greater than *t*, which contradicts with the assumption.

Case 2. If there is one checkpoint CP_r on the right side, then CP_z happens before CP_r: $CP_z \rightarrow CP_r$. Since the timestamp is monotonically increasing, the timestamp of the checkpoints made before CP_z by the same MA should be smaller then the timestamp of CP_r. Under such a scenario, according to Procedure 4, a forced checkpoint CP_f with timestamp *t* should be made before CP_r. Therefore, the timestamp of CP_r should be greater than *t*, which contradicts with our assumption.

Therefore, our assumption does not hold. The theorem holds.

According to the Basic-CIC algorithm, basic checkpoint is made by MA according to a predefined frequency. However, as we mentioned, an advantage of CIC is that it can prevent the domino effect while allowing processes considerable autonomy in deciding when to take basic checkpoints. An efficient implementation of CIC must therefore adopt a checkpointing policy that exploits this autonomy and translates it into a benefit. In general,

a good understanding of the application and of the execution environment is required. Specifically, for MA systems, we must exploit their characteristics. With these characteristics, we try to find how to benefit from the autonomy of CIC.

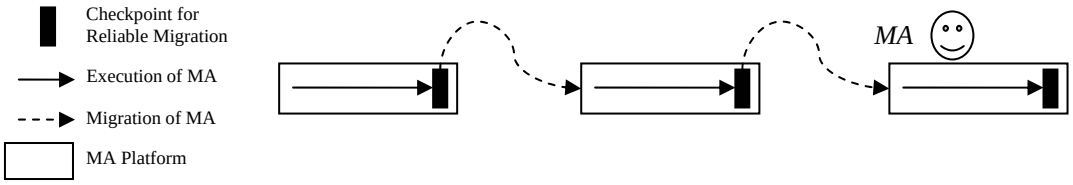


Figure 5.8 Reliable Migration of MA

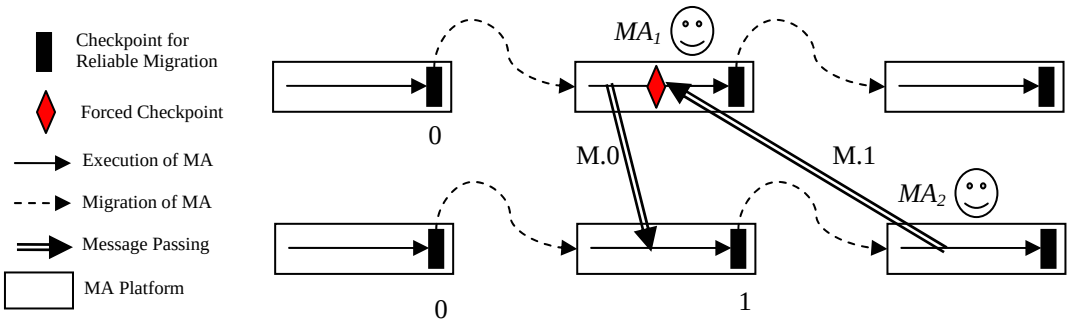


Figure 5.9 Basic-CIC Integrated with Reliable Migration

From the literature review (Section 2.2.2 of Chapter 2) we know that independent checkpointing is widely adopted in MA systems to provide reliable migration for MAs, where a checkpoint will be made before each migration operation. Figure 5.8 shows the scenario of reliable migration algorithm. Independent checkpointing scheme is adopted since the reliable migration is usually in terms of a single MA. If we can integrate the Basic-CIC algorithm and the reliable migration algorithm to let the independent checkpoint act as the basic checkpoint of Basic-CIC, we can save the operations to make basic checkpoints for Basic-CIC, so as to make the whole MA system more efficient. Figure 5.9 illustrates the integration of these two algorithms, where the checkpoints for reliable migration act as the basic checkpoints in Basic-CIC algorithm.

It is easy to implement such an integration. We only need to modify the Procedure 2 of Algorithm 5.4 and let each MA make basic checkpoint according to the frequency defined by the algorithm of reliable migration. Through this integration, an independent algorithm for reliable migration is not necessary. Basic-CIC will take the role to guarantee the reliable migration for each MA.

Even there is no independent checkpointing implemented for each single MA in an MA system, we can still benefit from CIC. Since CIC allows processes themselves to choose the right time to make the checkpoints, each member in an MA group can make the basic checkpoints at the right time: before the migration. At that moment, an MA will be serialized for the migration, which effectively constructs a checkpoint.

However, we still have one problem. Although we let the checkpoints for reliable migration act as the basic checkpoints in the Basic-CIC algorithm, we still need to make the possible force checkpoints during an MA's execution on a host. As shown in Figure 5.9, an MA may receive a message during its execution on a host, which will result in a forced checkpoint to be made and interrupt the MA's execution temporarily. Such interruption will affect the performance of the MA application severely if an MA will travel a large number of hosts. At next section, we try to improve the Basic-CIC algorithm by making further exploiting the characteristic of MA systems.

5.2.4 Deferred Message Processing based CIC (DM-CIC)

In an MA system, due to the mobility of MAs, messages are delivered in a purely asynchronous manner. A typical message delivery scheme is the forwarding point. Each MA leaves a point on each MAP it has visited to point to the next MAP that it has migrated to. Normally, a message destined to this MA will be sent to the home of this MA firstly,

and then follows the points to chase the MA. Based on the workload of the MA and the networks' QoS, the message can catch up the MA earlier or later. For the MA, if the message is not a strictly coordinated message (i.e., an MA does not have to receive it on a particular host), usually it is not important to receive the message earlier or later, on current host or on the next host. Considering this characteristic, we can avoid the forced checkpoint by deferring the processing of the message until a basic checkpoint is made.

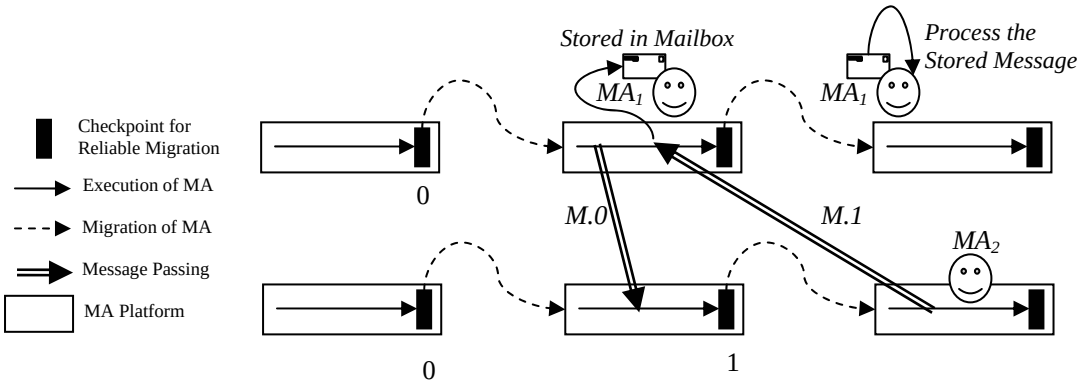


Figure 5.10 Deferred Message Processing based CIC (DM-CIC)

Figure 5.10 illustrates the scenario of the deferred message processing and Algorithm 5.5 shows the procedures of DM-CIC. In Figure 5.10, when MA_1 receives a message *M.1* and a forced checkpoint should be made according to CIC, instead of making a force checkpoint, MA_1 only accepts message *M.1* and does not process it. The received message which can result in a forced checkpoint is stored in the MA's associated mailbox (an MA's associated mailbox is a buffer in an MA). The received message will not be processed (that is to say: deferred) until MA_1 makes a basic checkpoints and lands on a new host. Through this way, we merge the basic checkpoint and the force checkpoint, but the price is that the messages which can result in a forced checkpoint can not be processed immediately when they reach the MA.

Algorithm 5.5: ***DM-CIC***

```
//Procedure 1: MA initiated and launched;
MA.LC=0;                          // logical clock is initiated to be 0
MA.Mbox=NILL;                     // Mailbox is initiated to be empty
MA.Launch(firsthost); //

// Procedure 2:Execution on a host;
if (MA.Mbox.PreviousHostMsg!=NILL)
   MA.ProcessingMsg(MA.Mbox.PreviousHostMsg);     //Process the stored previous host messages
MA.Exe();                                         //MA executes on current host
MA.LC= MAX(MA.LC+1,MAX( MA.Mbox.PreviousHostMsg));      // LC is increased by 1
MA.MakeCP(MA.LC);                 //A basic checkpoint is made and tagged by LC.

// Procedure 3:Message processing: SendMsg();
Msg = CreateMsg (msgbody, MA.LC);       //all messages should be tagged by LC
SendMsg(Msg);

// Procedure 4:Message processing: GetMsg();
Msg = GetMsg ();  //Get a message from the system
if (Msg.msgbody == "RollBack")
      RollBackTo(Msg.LC);
else
   M_LC= Msg.LC;                  //Get the piggybacked logical clock;
   if (M_LC > MA.LC)              //if piggybacked logical clock greater than local clock;
      MA.Mbox.Store(Msg);         //MA store the message in its mailbox;
   else
         MA.ProcessingMsg(Msg);

// Procedure 5:A failure of MA is detected
Msg = CreateMsg ("RollBack", MA.LC);      //rollback to the CP with timestamp LC
MAP.SendToAllMembers(Msg);
```

The main differences between Basic-CIC and DM-CIC lie at Procedures 2 and 4. In Procedure 4, when the piggybacked logical clock greater than the local clock, instead of making a forced checkpoint like Basic-CIC, the message will be stored in the mailbox. In Procedure 2, the stored message will be processed before the execution of an MA on a new host. The following theorem shows that DM-CIC can build up a consistent global snapshot.

Theorem 5.3 (Consistency): *In DM-CIC, a set of checkpoints S, where each is from a different MA is guaranteed to be a consistent state if they have the same timestamp.*

Proof: The proof is by contradiction. Assume that a set of checkpoints *S* created in DM-CIC, each from a different MA, has the same timestamp *t* but is not a consistent state. Then in *S*, there must be at lest one checkpoint CP_z which is involved in a z-cycle *C*. Then, there are two cases: the rest of the checkpoints *S'* (*S'*=*S*-CP_z != Φ) are either on the left side of the z-cycle or on the right side.

Case 1. If there is one checkpoint CP_l on the left side, then CP_l happens before CP_z: $CP_l \rightarrow CP_z$. According to Procedures 2 and 4 of DM-CIC, the received message (forming the z-cycle *C*) on the left side of CP_z will not be processed before CP_z, and the processing will be deferred after CP_z. So the z-cycle *C* is broken and does not exist.

Case 2. If there is one checkpoint CP_r on the right side, then CP_z happens before CP_r: $CP_z \rightarrow CP_r$. According to Procedures 2 and 4 of DM-CIC, the received message (forming the z-cycle *C*) on the left side of CP_r will not be processed before CP_r, and the processing will be deferred after CP_r. So the z-cycle *C* is broken and does not exist.

Therefore the z-cycle *C* does not exist and our assumption does not hold. The theorem holds.

The advantage of DM-CIC is very clear: forced checkpoint has been avoided, so as to saves the system's execution time. However, the advantage is achieved on the price of sacrificing the messages' real-time processing. In the next section, we evaluate the performance of DM-CIC and Basic-CIC thought simulation.

5.2.5 Performance Evaluation using Simulations

In our simulation, we consider the behaviour of an MA which belongs to a group of MAs (=20). The group of MAs communicates by message passing. The MA will migrate in a

network which consists of 100 hosts. Three metrics are adopted to evaluate the performance of Basic-CIC and DM-CIC: the time of the delayed message processing, the whole execution time of an MA and the time for the system recovery.

We decompose an MA's execution into three procedures: task execution (T_{exe}=50ms), checkpointing (T_{cp}=10ms), and migration (T_{mg}=10ms). An MA can process messages during T_{exe}, while the message processing will be blocked during T_{cp} and T_{mg}. Since only the messages causing the forced checkpoint can impact the performance of the CIC algorithms, we only consider such messages in our simulations. We assume these messages are sent and received randomly (following the exponential distribution) during the running of MAs' execution. The recovery time for a forced checkpoint (T_{rf}=10ms) and a basic checkpoint (T_{rb}=2ms) is different, because a basic checkpoint is under the prepared state for migration, while forced checkpoint is make during MA's execution, and all the system contexts have to be stored.

In order to get better average simulation results, we execute each condition 5000 times with the same settings. To make a clear comparison, we also implement an algorithm that does not make forced checkpoint: when there is a message which can cause forced checkpoint coming in, the algorithm just ignores it.

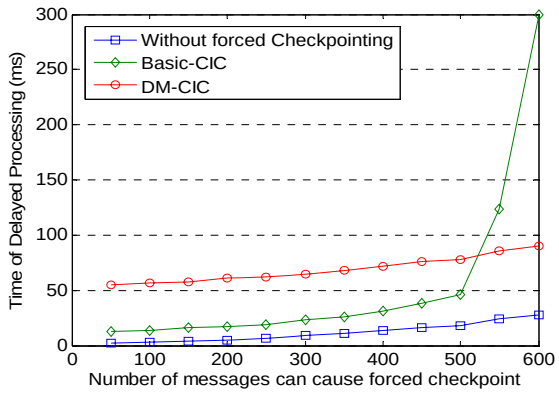


Figure 5.11 The Delayed Message Processing

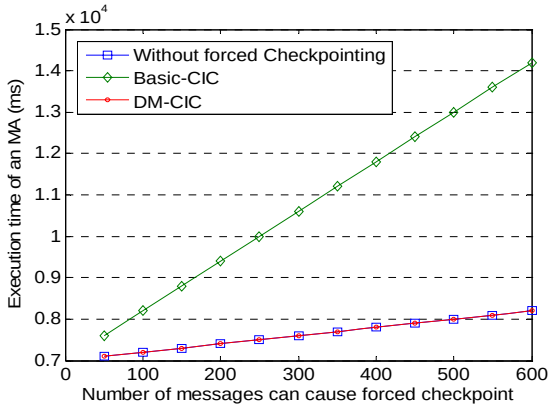


Figure 5.12 The Whole Execution Time of an MA

Figure 5.11 illustrates the time of the delayed processing for each message which can cause forced checkpoint. In DM-CIC, all the processing for the messages which can cause forced checkpoint will be delayed, but the delay is stable. Basic-CIC performs well when few messages arrive. But its performance degrades sharply when more messages come in, because the too many operations for forced checkpointing will block the message processing dramatically. The operations of forced checkpointing also prolong the MA's whole execution time in Basic-CIC, which is shown in Figure 5.12. However, due to avoid the forced checkpoint, the execution time of DM-CIC is exactly the same with the algorithm without forced checkpointing. Therefore, in Figure 5.12, they are overlapped.

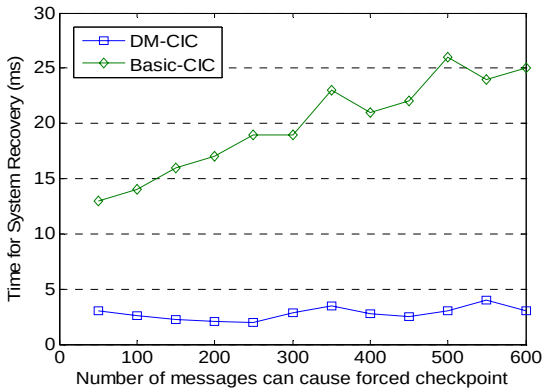


Figure 5.13 The System Recovery Time

Figure 5.13 illustrates the simulation results for the time of the system recovery. It is clear that Basic-CIC costs more time to recover the system to a consistent state than DM-CIC, because the time to resume a forced checkpoint is longer than a basic checkpoint. For Basic-CIC, another phenomenon is that the time to recover the system becomes longer with more forced checkpoints made in the system, because more forced checkpoints are involved in the recovery process. On the contrary, the recovery time for DM-CIC is always stable since only basic checkpoints are involved.

5.3 Summary

An MA system has a different system model from conventional computer systems in solving the checkpoint placement problem. In this chapter, we have proposed to model the problem for MA systems using the homogeneous discrete-parameter Markov chain. Based on this model and two widely adopted checkpointing strategies, we designed three checkpoint placement algorithms for MA systems. Through simulations we found out that the algorithm based on equicost checkpointing strategy achieved the best trade-off between checkpointing cost, system performance, and flexibility. We also introduced CIC for MA group, which is proved to be a flexible, efficient and scalable checkpointing scheme among the other schemes. DM-CIC has been designed to integrate with the independent checkpointing scheme for reliable MA migration.

Chapter 6
Mobile Agent Transaction

In this chapter, we describe the design and evaluation of our proposed architecture and algorithms for MA transactions. The chapter is organized as follows. Section 6.1 defines the system model for MA transaction and proposes the corresponding system architecture. Section 6.2 introduces algorithms for the transactional execution of both single MA and multiple MA applications. Section 6.3 addresses the issue of handling distributed deadlock in MA transactions. Section 6.4 provides a summary of this chapter.

6.1 System Model and Architecture

In this section, we firstly define a two-level nested transaction model for MA transaction based on the analysis of a realistic MA execution environment. Using the model, we develop the system architecture for the implementation of MA transactions.

6.1.1 System Model

An MA transaction can be modeled as a two-level nested transaction. The MA itself controls the top-level transaction which involves functionalities such as reliable migration, distributed deadlock detection, and commitment protocols. Nested within the top-level transaction are the lower-level transactions, called subtransactions which are supported by the base transaction systems. Each MA transaction is assigned a global unique identifier, called a *transaction ID*. Each MA also has a unique global *MA ID*.

The execution environment of an MA transaction involves a collection of servers/hosts identified as S_i ($i=1, \ldots, n$). Each server is deployed with a base transaction system, which supports functionalities such as locking (2PL, Timestamp Ordering Protocol, Optimistic Protocol), local deadlock detection, logging/shadow file, and recovery. Task descriptions are encoded in a single MA or a group of MAs. The MA migrates between the servers according to a predefined or self-initiated itinerary. Once an MA lands on a server $S_{i,}$ it starts its execution and initiates a subtransaction according to the predefined task description. On a server there is no difference between an MA initiated subtransaction and other transactions. When the subtransaction is ready to commit, the MA can choose to either commit it immediately or wait until all the subtransactions are ready to commit, in line with the relevant commitment model adopted by the top-level transaction.

We define the information related to the transaction in an MA as the *MA's transaction context*, which includes the itinerary of the MA, MA ID, transaction ID, and intermediate states/computing results. Since the failures of the server and the MA platform can be handled by the traditional backup-based fault tolerance techniques, we assume that the server and the MA platform will not fail. However, an MA may crash during its execution and migration. Also, messages may be delayed but we assume that a message will ultimately be delivered to its destination.

6.1.2 System Architecture

The architecture of MA transaction corresponds to the two-level nested transaction model in that it consists of two layers: a *base transaction layer* and an *MA transaction layer* (Figure 6.1), which respectively manage the low-level transactions and the top-level transactions. Base transaction systems constitute the base transaction layer, which provides the support for subtransactions executed on the local host. On he other hand, the MA

system governs the MA transaction layer. Its functionalities include initiating a subtransaction on each server according to the itinerary, managing the commitment process, concurrency control (resource allocation management and distributed deadlock detection), and adapting the transaction support primitives of base transaction systems, such as the "transaction_begin, transaction_end, commit, abort" etc., to a set of uniform primitives for the MA transaction layer. The following describes the base transaction layer, MA transaction layer, MA transaction context, and failure detector.

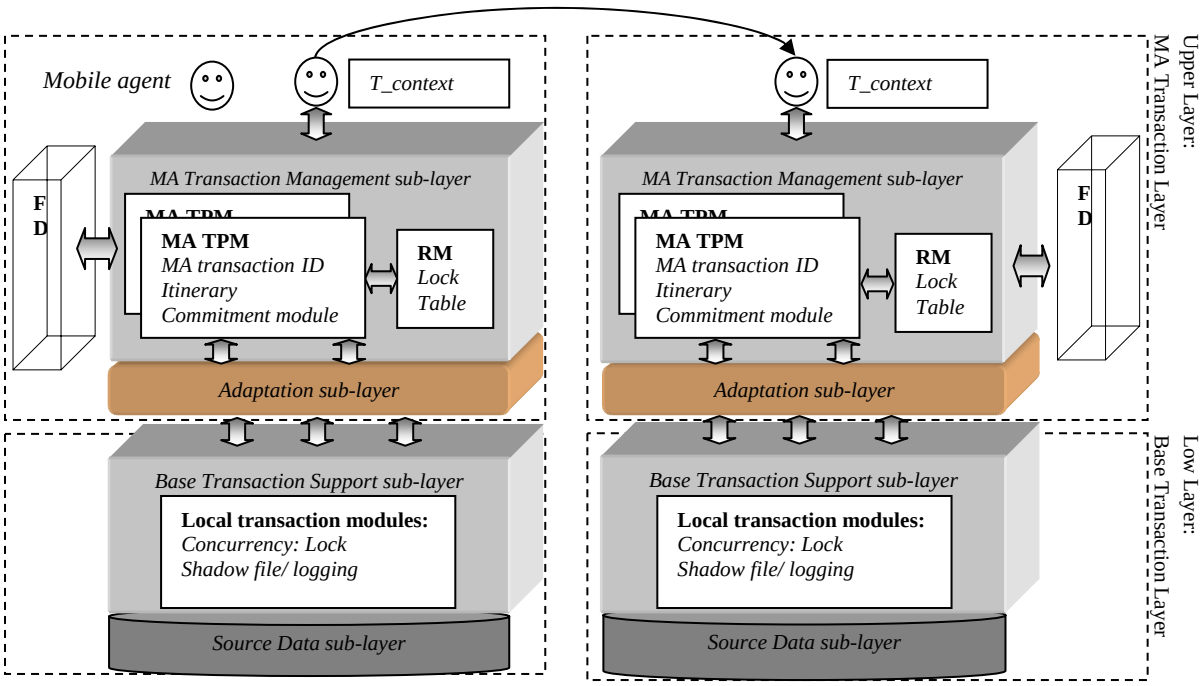


Figure 6.1 The Architecture of MA Transaction

1) Base Transaction Layer

As shown in Figure 6.1, we model the base transaction layer as two sub-layers. The lower layer is the *Source Data* sub-layer. Source data can be an ordinary file, a table, some simple database (i.e. ACCESS), or other structured data, such as MIB. This sub-layer provides access only to the source data and does not provide transaction support. The

upper sub-layer is the *Base Transaction Support* sub-layer, which provides APIs for manipulating source data transactionally. The transaction supports include local concurrency control mechanisms (2PL, Timestamp Ordering Protocol, Optimistic Protocol), logging/shadow file, and commit/abort subtransactions.

2) MA Transaction Layer

The MA transaction layer is implemented within the MA platform which is on top of the base transaction systems (Figure 6.1). From the viewpoint of the base transaction systems, the MA transaction layer has no difference from other applications running on top of the base transaction systems. The advantage of this architecture is that the base transaction systems do not need to be modified. MA transaction layer can be separated into two sub-layers. One is the *Adaptation* sub-layer which adapts the transaction support primitives of different base transaction systems to a uniform set of primitives for the upper sub-layer. The other is the *MA Transaction Management* sub-layer which, with the help of the set of uniform primitives, allows MA transactions to be initiated, committed or aborted. *MA Transaction Management* sub-layer contains two software modules: the *Transaction Processing Monitor* (*TPM*) and the *Resources Manager* (*RM*), which cooperate to control the top-level transaction.

The *TPM* manages the commitment process to preserve the *Atomic* and *Duration* properties for top-level transaction. Its management tasks include:

(1) Managing the execution context of MA transaction. This includes storing the MA's transaction context such as the Transaction ID and itinerary.

(2) Managing the states of uncommitted subtransactions. Should there be an incoming commitment command, the *TPM* will process the command and commit the corresponding subtransaction.

(3) Participating in the commitment process. In an MA transaction that adopts the close commitment model, each *TPM* will act as a participant when the top-level transaction is committed.

Table 6.1 Locking Table

Resource ID	***Holder's Transaction ID***	***Waiter's Transaction ID***	***Registered Event***	***Extendable area***
001	MA1	MA2	TO / Release	

The *RM* is responsible for the concurrency control that guarantees the *Consistency* and *Isolation* properties for MA transaction. One popular and representative concurrency control algorithm is *two phase locking* (*2PL*). *2PL* is used as the concurrency protocol in our system. However, in our proposed architecture, it is easy to replace *2PL* with some other algorithms since concurrency control is implemented in the *RM*, which is designed as a separate module. In order to detect possible distributed deadlocks produced by *2PL*, the *RM* maintains a locking table, shown in Table 6.1, which provides the necessary information for deadlock detection algorithms.

3) MA Transaction Context

A special characteristic of MA transactions is that an MA is able to carry information as it travels. As mentioned before, we refer to this information related to MA transactions as the MA transaction context. This context typically includes the following information

(1) *Transaction ID*: A unique transaction identifier that differentiates between MA transactions.

(2) *Itinerary*: This implies a sequence of servers involved in the MA transaction.

(3) *Commitment model*: The owner of the MA can specify the commitment model for an MA transaction and store it in the MA transaction context. The MA transaction context

will be stored in the *TPM* when an MA lands on a server to help a *TPM* complete its management tasks.

4) Failure Detector

Normally, MAs will migrate between several servers and initiate subtransactions on each server. If the close commitment model is adopted, the subtransaction on the first server it visits must wait until the MA finishes the subtransaction on the last server it visits. Before the MA transaction is committed, however, two kinds of failures may occur: the failure of the MA and the failure of the uncommitted subtransaction. A failure detector will detect such failures. In the case of an MA failure, the failure detector detects it and informs the *TPM*. The *TPM* can then carry out some predefined operations to handle this event. In the case of the failure of an uncommitted subtransaction (i.e., victim of deadlock resolving), the failure detector will directly inform the MA of this failure. The failure detector is a configurable, self-restarted software module executing on the same host as the MA Platform. If the host does not crash, the failure detector is always available. The failure detector can adopt the widely used heartbeat based failure detector [CHE02] or our own newly proposed notification based failure detector [YAN06].

6.2 The Execution of MA Transaction

In this section, we firstly introduce the two MA execution modes and the three MA commitment models. Then, we propose algorithms of MA's transactional execution under different execution mode and using different commitment models.

6.2.1 MA Execution Modes

The MA has two execution modes: Single MA (*SMA*) and Multiple MAs (*MMA*). In *SMA*, only one MA is launched to carry out a user assigned task. In contrast, *MMA* dispatches more than one MA. Accordingly, MA transaction can be classified as either single MA transactional execution (*SMA Transaction*) or multiple MA transactional execution (*MMA Transaction*). The advantage of *MMA* is that launching multiple MAs will speed up the execution of MA transactions, but at the same time *MMA* will consume more resources than *SMA* and require additional cooperation and coordination. For example, in the commitment process, *MMA* requires a coordinator while SMA, which dispatches just one MA, does not.

6.2.2 Commitment Models in MA Transactions

Different commitment models (open or close) classify nested transactions as either open nested transactions [WEI92, CHR94] or close nested transactions [MOS85]. In the open commitment model, every subtransaction (open subtransaction) makes its results visible to other transactions as soon as its computation has successfully terminated. This means that the commitment is independent of the outcome of its parent transaction. In contrast, in close commitment model, every subtransaction (close subtransaction) does not make its updates visible to other transactions until its parent transaction commits. For simplicity, hereafter, the open (close) commitment model is also referred to as open (close) commitment. In the following we briefly outline some of the limitations of the open and close commitment models and then propose a new commitment model – an adaptive commitment model. This new commitment model allows subtransactions taking place in base transaction systems to be committed using either the open or the close commitment model.

1) Open/Close Commitment Models and Their Limitations

In the context of MA transactions, both open and close commitment models involve a number of tradeoffs. One advantage of the open commitment model, for example, is that a subtransaction will be committed as soon as it successfully finishes. This means that the resources will not be locked by the subtransaction for too long, greatly alleviating the problem of long-lived transaction. However, if a committed subtransaction has to be aborted, compensation processing is necessary [GAR87, GAJ93], which is not always available. The requirement of compensation processing is a big limitation of the open commitment model.

In contrast, the close commitment model does not require compensation processing. It keeps subtransactions in the state of waiting-commit until the top-level transaction is ready to be committed. As a result, the subtransaction on the first visited server has to wait until the MA finishes the subtransaction on the last server. The locked resources on the first server cannot be made available to other transactions for a long time, which may increase the probability of distributed deadlock [GAJ93]. In addition, the close commitment model requires a coordinator during the commitment processing. Two phase commitment protocol (*2PC*) is a widely used commitment protocol for the close commitment model in distributed transaction. MA transaction provides some relief to *2PC*. An MA will migrate to the next stop only when the current subtransaction is ready to commit. Consequently, when the MA reaches the destination, all the subtransactions are ready to commit. Therefore, we only need “one phase commitment” in a single MA’s transactional execution: the MA just sends the commitment command to all the subtransactions to commit the top-level transaction.

The tradeoffs associated with the different commitment models make the choice of commitment model dependent on a number of factors. For example, open commitment may reduce the possibility of deadlock, but the precondition is that the subtransaction be

compensable. Scarcity of a resource is another factor that affects the choice of the commitment model. Take a ticket selling service as an example: if there are plenty of tickets, people are allowed to book tickets firstly and pay for the booked ticket later. This situation corresponds exactly to close commitment. However, if few tickets remain but there are still many buyers, the booking service usually will be cancelled and the tickets are only sold to buyers who pay for the ticket in cash. This situation can be modeled as open commitment.

2) An Adaptive Commitment Model

As we mentioned before, an MA may travel over a large network and visit many servers belonging to different organizations. It is possible that $Server_A$ supports open commitment and allows a subtransaction on it to remain uncommitted, while $Server_B$ requires the subtransaction on it to be committed as soon as the subtransaction finished, allowing only close commitment. This suggests that an adaptive commitment model is needed to perform open commitment for the subtransactions on $Server_A$ and close commitment on $Server_B$. To support adaptive commitment, a base transaction system should provide a parameter to indicate what kind of commitment model it supports. As an example, in the following of this chapter, we adopt the scarcity of a resource as the criteria to determine the commitment model on a server.

An adaptive commitment protocol requires an effective estimation of the scarcity of resource. To calculate it, we define the Resource Scarcity Valve Value (*RSVV*). *RSVV* is a configurable parameter for each kind of resource. If the quantity of a resource is greater than its *RSVV*, the close commitment model is required. Otherwise, the open commitment model can be selected. We let *0* denote close commitment and *1* denote open commitment. The required commitment model (*RCM*) for each type of resource is either *0* or *1*. *RCM* is

calculated by the *RM* and stored in the *TPM*. Suppose a subtransaction needs *N* types of resources on a server. The commitment model for this subtransaction (*SCM*) on this server would be: $SCM = RCM_1 \cap RCM_2 \cap \ldots \cap RCM_n$.

The owners of MAs can also specify their required commitment model (*UCM*) before the MAs are launched. Table 6.2 lists the final decisions made when *UCM* conflicts with *SCM*.

Table 6.2 Final Commitment Model

SCM \ UCM	***Open***	***Close***	***Adaptive***
Open	Open	Conflict -> (Abort)	Open
Close	Conflict -> (Abort)	Close	Close

The commitment model for each uncommitted subtransaction is stored in *TPM* on each server and is also carried by an MA's transaction context. When the MA decides to commit/abort the top-level transaction, it will perform corresponding commitment/abort operations according to the different commitment models stored in its transaction context.

6.2.3 Algorithms for MA Transaction's Execution

In this subsection, we propose the algorithms of MA's transactional execution under different execution mode and using different commitment models.

Table 6.3 The Modes of MA Transactions

	Open Commitment	***Close Commitment***	***Adaptive Commitment***
Single MA (SMA)	Open SMA Transaction	Close SMA Transaction	Adaptive SMA Transaction
Multiple MA (MMA)	Open MMA Transaction	Close MMA Transaction	Adaptive MMA Transaction

As shown in Table 6.3, the three commitment models (Open, Close, and Adaptive) and the two MA execution modes (SMA and MMA) produce six combinations, which will result

in six algorithms. These six combinations, however, can be derived from two basic combinations: SMA with open commitment (open SMA transaction) and SMA with close commitment (close SMA transaction). In the following, we firstly propose the algorithms for these two basic combinations, and then describe how to derive the algorithms for other four combinations from these two basic combinations.

1) Algorithms for SMA Transaction

Figure 6.2 shows the execution scenario for an *SMA* transaction. The following provides the algorithms for open SMA transactions (Algorithm 6.1) and close SMA transactions (Algorithm 6.2). The algorithm for adaptive SMA transactions is derived from Algorithms 6.1 and 6.2.

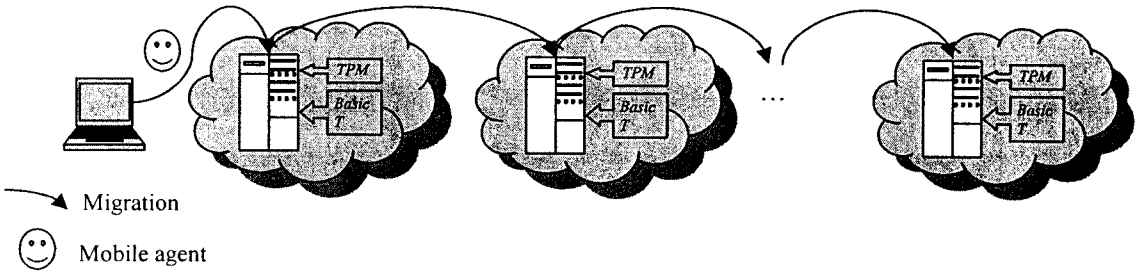


Figure 6.2 The Execution of an SMA Transaction

Algorithm 6.1 provides the open SMA transaction algorithm in pseudo-code format, which accommodates five procedures. In Procedure 1, user initiates and launches the MA to the first server. A unique transaction ID will be generated, itinerary and user required commitment model will be assigned (*UCM=Open*). Itinerary, transaction ID and user required commitment model compose the MA's transaction context and is carried by the MA. Itinerary can be predefined or self-initiated.

Algorithm 6.1 *Open SMA Transaction* (pseudo-code format)

```
// 1. Procedure for Initiating and Launching an MA
  MA.task = { Transaction_begin; Task1; Task2; … Taskn; Transaction_end }; //Tasks to be executed;
  MA.itinerary = { Server1; Server2; … Servern };   // Itinerary can be predefined, or self-initiated;
  MA.UCM = open;                                    // Itinerary, UCM and Transaction ID compose
  MA.T_id = Tid_generator();                        // the transaction context for this MA transaction;
  Si = MA.itinerary.getNext();      //Get the first server;
  Result = MA.migrateTo(Si);        //Migration to the first server;
  If (Result = fail)                //If failed, generate an alarm to alert the user
       Alarm("Mission cannot be started"); // that the transaction cannot be started;

// 2. MA's Entry Point on Each Server: Execute Subtransaction on server Si;
  TPM = Si.getTPM();                //MA gets the TPM of server Si
  TPM.insert(T_id, UCM, itinerary);   //TPM allocate an entry for this MA transaction
  Result = TPM.execute(task.Taski);   //TPM help to initiate the subtransaction
  If (Result == commitable )
     Goto 3          //Go to commit this subtransaction;
  Else
     Goto 5;         //Go to abort this subtransaction and then abort the top-level transaction;

// 3. Procedure for Committing Current Subtransaction
  If (Si = MA.itinerary.getNext() ==NULL)       //Get the next stop. If there is no next stop, it means that we
     Goto 4;                  //finished all the subtransactions. Therefore, we go to commit top-level transaction;
  Result = MA.migrateTo(Si);
  If (Result = fail)
     Goto 5; //If the next server is unreachable, go to abort this subtransaction and the top-level transaction;
  Else
     TPM.commit(T_id); //Only when the migration is successful, we commit this subtransaction;
    Goto 2; //Here is a logical Goto. Actual scenario is the MA finishes its execution on current server,
            //and starts its execution on the next server.

// 4. Procedure for Committing the Top-level Transaction
  Itinerary = TPM.getItinerary(T_id);
  While (S = Itinerary.getNext() != NULL)
        {CommitMsg = {"Commit", T_id}; //Compose a commitment command
          SendTo(S.TPM, CommitMsg);       //Send commitment commands to all the subtransactions.
        }

// 5. Procedure for Aborting Current Subtransaction and Top-level Transaction
  TPM.abort(T_id);                  //Abort current subtransaction;
  Itinerary = TPM.getItinerary(T_id);   //Get MA's itinerary from the context stored in TPM
  VisitedServers = Itinerary.getPrevious(Si); //Get the servers that MA have visited
  While (S = VisitedServers.getNext() != NULL)
        {AbortMsg = {"Abort", T_id};       //Compose a commitment command
          SendTo(S.TPM, AbortMsg);          //Send commitment commands to all the previous subtransactions.
        }

//Asynchronous message processing: corresponding to procedures 4 and 5
Msg =MAP.getMessge();               //MA platform is responsible to get a message;
If Msg.content = = "Commit"
   TPM.commit(msg.T_id);            //"Commit" command will flush the context stored in this TPM;
If Msg.content = = "Abort"
   TPM.abort(msg.T_id);             //"Abort" command not only flush the context stored in this TPM;
   TPM.compensate(msg.T_id);        //but also trigger the compensation proc for this committed subtransaction.
```

Algorithm 6.2 *Close SMA Transaction* (pseudo-code format)

```
// 1. Procedure for Initiating and Launching an MA
  MA.task = { Transaction_begin; Task1; Task2; … Taskn; Transaction_end }; // Tasks to be executed;
  MA.itinerary = { Server1; Server2; … Servern };    // Itinerary can be predefined, or self-initiated;
  MA.UCM = close;                                    // Itinerary, UCM and Transaction ID compose
  MA.T_id = Tid_generator();                         // the transaction context for this MA transaction;
  Si = MA.itinerary.getNext();       //Get the first server;
  Result = MA.migrateTo(Si);         //Migration to the first server;
  If (Result = fail)                 //If failed, generate an alarm to alert the user;
     Alarm("Mission cannot be started");    // that the transaction cannot be started;

// 2. MA's Entry Point on Each Server: Execute Subtransaction on server Si;
  TPM = Si.getTPM();                 //MA gets the TPM of server Si;
  RM = Si.getRM();                   //MA gets the RM of server Si, to record the locked resources;
  TPM.insert(T_id, UCM, itinerary);    //TPM allocate an entry for this MA transaction;
  Result = TPM.execute(task.Task,RMi);    //TPM help to initiate the subtransaction. Please be noted here RM
  If (Result == commitable )         //is also involve in the execution to record the locked resources;
     Goto 3          //Go to commit this subtransaction;
  Else
     Goto 5;         //Go to abort this subtransaction and then abort the top-level transaction;

// 3. Procedure for Migrating to the Next Stop
  If (Si = MA.itinerary.getNext() ==NULL)    //Get the next stop. If there is no next stop, it means that we
     Goto 4;                 //finished all the subtransactions. Therefore, we go to commit top-level transaction;
  Result = MA.migrateTo(Si);
  If (Result = fail)
     Goto 5; //If the next server is unreachable, go to abort this subtransaction and the top-level transaction;
  Else
     Goto 2; //Here is a logical Goto. Actual scenario is the MA finishes its execution on current server,
             // and starts its execution on the next server.

// 4. Procedure for Committing the Top-level Transaction
  Itinerary = TPM.getItinerary(T_id);
  While (S = Itinerary.getNext() != NULL)
        {CommitMsg = {"Commit", T_id}; //Compose a commitment command
          SendTo(S.TPM, CommitMsg);       //Send commitment commands to all the subtransactions.
        }

// 5. Procedure for Aborting Current Subtransaction and Top-level Transaction
  TPM.abort(T_id);                   //Abort current subtransaction;
  Itinerary = TPM.getItinerary(T_id);    //Get MA's itinerary from the context stored in TPM
  VisitedServers = Itinerary.getPrevious(Si); //Get the servers that MA have visited
  While (S = VisitedServers.getNext() != NULL)
        {AbortMsg = {"Abort", T_id};       //Compose a commitment command
          SendTo(S.TPM, AbortMsg);          //Send commitment commands to all the previous subtransactions.
        }

//Asynchronous message processing: corresponding to procedures 4 and 5
Msg =MAP.getMessge();              //MA platform is responsible to get a message;
If Msg.content = = "Commit"
   TPM.commit(msg.T_id);           //"Commit" command will flush the context stored in this TPM;
If Msg.content = = "Abort"
   TPM.abort(msg.T_id);            //"Abort" command flushes the context stored in this TPM;
```

Procedure 2 describes the operations performed when an MA lands on a server and begins to execute the corresponding subtransaction on this server. MA only needs to get the handle of *TPM* and let *TPM* manage the execution of this subtransaction. As a result, the complicated task of subtransaction management is shifted from the MA to the MA platform, which not only makes it easier to program the MA transactions, but also makes MA transactions more robust because *TPM* belongs to the system level module.

Procedure 3 presents the migration operation. A special design is that only after the MA lands on the next stop successfully, *TPM* on the current server will commit the subtransaction; Otherwise *TPM* can abort this subtransaction without executing compensation procedure for it. To let *TPM* manage the subtransaction, an MA should pass its transaction context to the *TPM*. When all the subtransactions are committed, the MA can commit the top-level transaction and let all the involved *TPMs* flush the transaction context of this committed MA transaction.

Procedures 4 and 5 provide the commitment and abort procedures respectively. A subtransaction may have to abort for a number of reasons such as deadlock, mismatch between *SCM* and *UCM*. Network failures may cause MAs to crash or prevent them from landing on the next stop. All these conditions can result in the aborting of the top-level transaction. Compensation operations are needed when a committed subtransaction is aborted. In fact, in Procedure five, it is not necessary to abort a top-level transaction if one or several its subtransactions have aborted. It depends on the commitment strategy adopted by the top-level transaction. For example, aborting some unimportant subtransactions will not impact on the final commitment of the top-level transaction. In this chapter, we adopt the strict commitment strategy which will not allow aborted subtransaction exist when commit the top-level transaction.

Algorithm 6.1 uses a predefined itinerary. This is similar to the implementation using a self-initiated itinerary except that adding each newly obtained next stop into the transaction context to prepare for the commitment of the top-level transaction when all the subtransactions are finished.

Algorithm 6.2 provides the close SMA transaction algorithm in pseudo-code format. The differences between close SMA transactions and open SMA transactions are in the resource management and the commitment procedure. Since we adopt *2PL* for close commitment model, it is possible to produce deadlock. Therefore, for resource management, top-level transaction must track the allocation of the resources to provide necessary information for deadlock detection algorithms. During the execution of the subtransaction, *RM* will take the responsibility for recording each locked resource in the locking table. In the commitment procedure, unlike open SMA transactions, no subtransactions will commit until the top-level transaction is committed.

In the adaptive SMA transaction algorithm, *UCM = Adaptive*, which, according to Table 6.2, means that there is no conflict with any *SCM*. The subtransactions are grouped as open commitment or close commitment. The commitment/abort procedure for open subtransactions is the same as that in Algorithm 6.1. The commitment/abort procedure for close subtransactions is the same as that in Algorithm 6.2.

2) Algorithms for MMA Transactions

Figure 6.3 shows a typical scenario for an MMA transaction in which two MAs (MA_1 and MA_2) are launched. During the execution, a child MA (MA_3) is spawned. All the MAs in an MMA transaction share the same transaction ID. The locking strategy, however, depends on different isolation policies. For example, in Figure 6.3, MA_1 and MA_2 belong to the

same MMA transaction. Whether MA_1 can access the resources locked by MA_2 or not depends on the isolation policy of this MMA transaction. If the MMA transaction requires isolation between the different members within it, MA_1 cannot access the resources locked by MA_2. However, if the MMA transaction requires isolation with other different transactions, MA_1 can access the resources locked by MA_2 since they belong to the same transaction. In this chapter, we apply the first policy, which is much stricter then the second one.

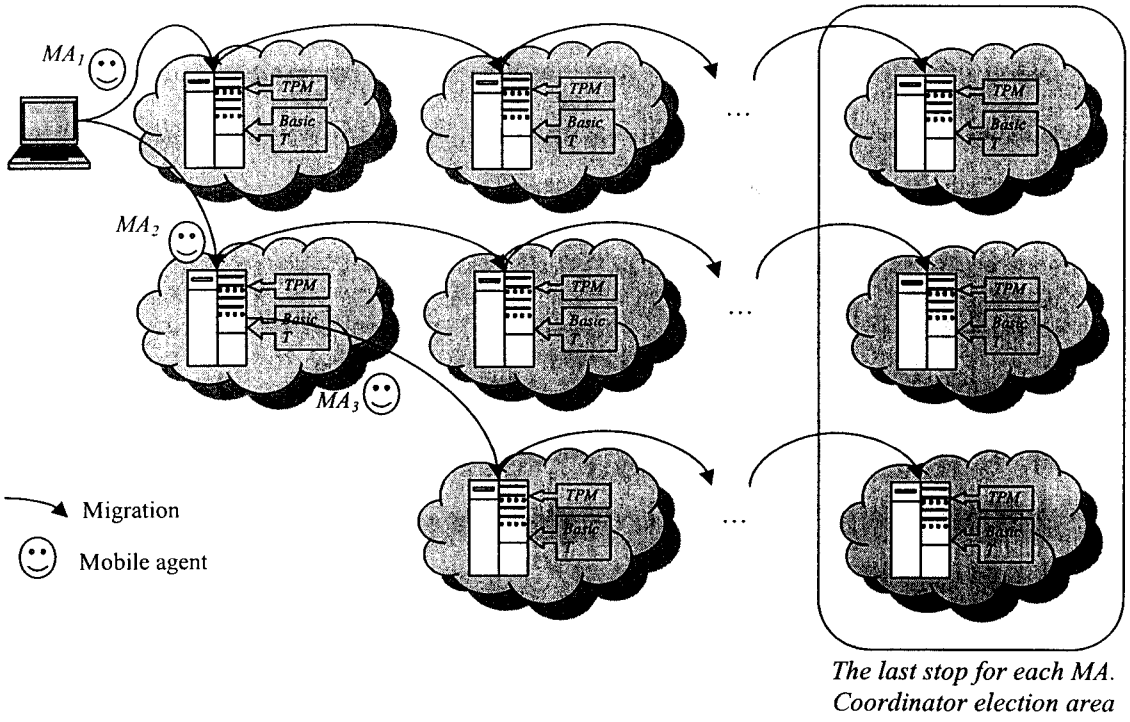


Figure 6.3 MMA transaction

Just as in an SMA transaction, there are also three MMA transaction commitment models: open, close and adaptive (Table 6.3). Each MA in an MMA transaction will execute an open, close or adaptive SMA transaction. Since more than one MA is involved, the commitment and abort procedures require additional operations.

The commitment procedure requires a coordinator to coordinate the final commitment of an MMA transaction. If an MA within the MMA transaction is ready to commit its top-level transaction, it must wait until all the other MAs are also ready to commit their top-level transactions. We can let one MA act as the coordinator. This coordinator can be the first MA that is ready or the MA with the highest MA ID, depending on the user's configuration. When an MA is ready to commit its top-level transaction, it reports its transaction context to the coordinator. The coordinator can get all the *TPMs* involved in this MA's SMA transaction from its transaction context. After the coordinator collects all the transaction contexts, the coordinator starts the commitment with all the participants (*TPMs*). According to the commitment model (close or open) recorded in each MA's transaction context, the commitment is the same as in Step 6 of both the open and close SMA transactions.

Each MA can abort its SMA transaction independently. It should also inform other MAs that they must stop their execution and abort their SMA transactions.

6.3 Deadlock Handling in MA Transactions

A deadlock is a situation in which subsets of processes are waiting for some other processes to release resources. No progress is possible until this situation is resolved. Deadlock handling can be divided into three types: *deadlock prevention*, *deadlock avoidance*, and *deadlock detection and resolution*. In this section, we first describe the scenarios of distributed deadlock and local deadlock in MA transactions. We then carry out simulations to study the probability of the occurrence of deadlocks in MA transactions and their affection on the MA system. We propose algorithms for deadlock prevention and deadlock detection, mainly focusing on the deadlock detection algorithms and their performance comparison.

6.3.1 Local Deadlock and Distributed Deadlock

Section 6.1.1 introduced the MA transaction execution environment. In such an environment, several MA transactions execute in parallel and each MA requests resources autonomously. If the requested resource is held by another MA, the requesting MA will be blocked until the requested resource is released. At any time, an MA can be only either active or blocked: active when it performs tasks normally or requests an idle resource; blocked if it requests a resource currently held by others. On being granted the requested resource, the MA will revert to the active state.

Different deadlock models have been proposed to describe a deadlock, including *one-resource* model, *and* model, *or* model, and *p-out-of-q* model. The most widely adopted model is the *one-resource* model, in which an MA handles only one outgoing request at a time, limiting it to being involved in just one deadlock cycle. Algorithms designed for deadlock detection under this model can be easily extended to other models. In this chapter, we consider only the one-resource model.

In MA transactions, a deadlock occurring within a subtransaction is called a *local deadlock*. A local deadlock will be detected and handled by the base transaction system, so it is not the concern of the top-level transaction which is managed by the MA transaction layer. On the other hand, a *distributed deadlock* is a deadlock that occurs between several subtransactions (as shown in Figure 6.4) and it cannot be detected by a base transaction system because the base transaction system has no knowledge of the global resource allocation situation. Distributed deadlocks will be detected and handled by the top-level transaction of an MA transaction. This is possible because the top-level transaction is fully

informed as to the status of resource allocation by recording all the resource allocation information in *RM* (see Section 6.2).

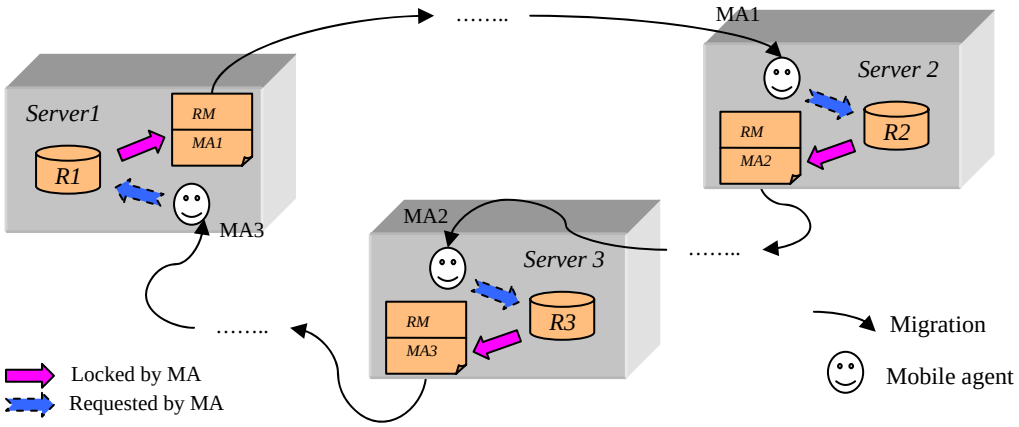


Figure 6.4 Distributed Deadlock in MA Transaction

As mentioned in Section 6.2.3, we choose strict isolation policy for MMA transaction, which implies the isolation is required between the different members within an MMA transaction. As a result, each MA in a MMA transaction can be viewed as executing an SMA transaction. Therefore, only SMA transactions need to be considered.

Distributed deadlock will not occur in open SMA transactions because a subtransaction is either committed or aborted immediately after it finishes local operations, releasing all the allocated resources. Close SMA transaction, however, keeps resources locked up until all the subtransactions have finished, which may produce the distributed deadlocks. Therefore, we only consider distributed deadlock occurring in close SMA transactions. For simplicity, we use MA transaction to stand for close SMA transaction in the following sections.

6.3.2 Evaluation of Deadlock Probability and Its Impact

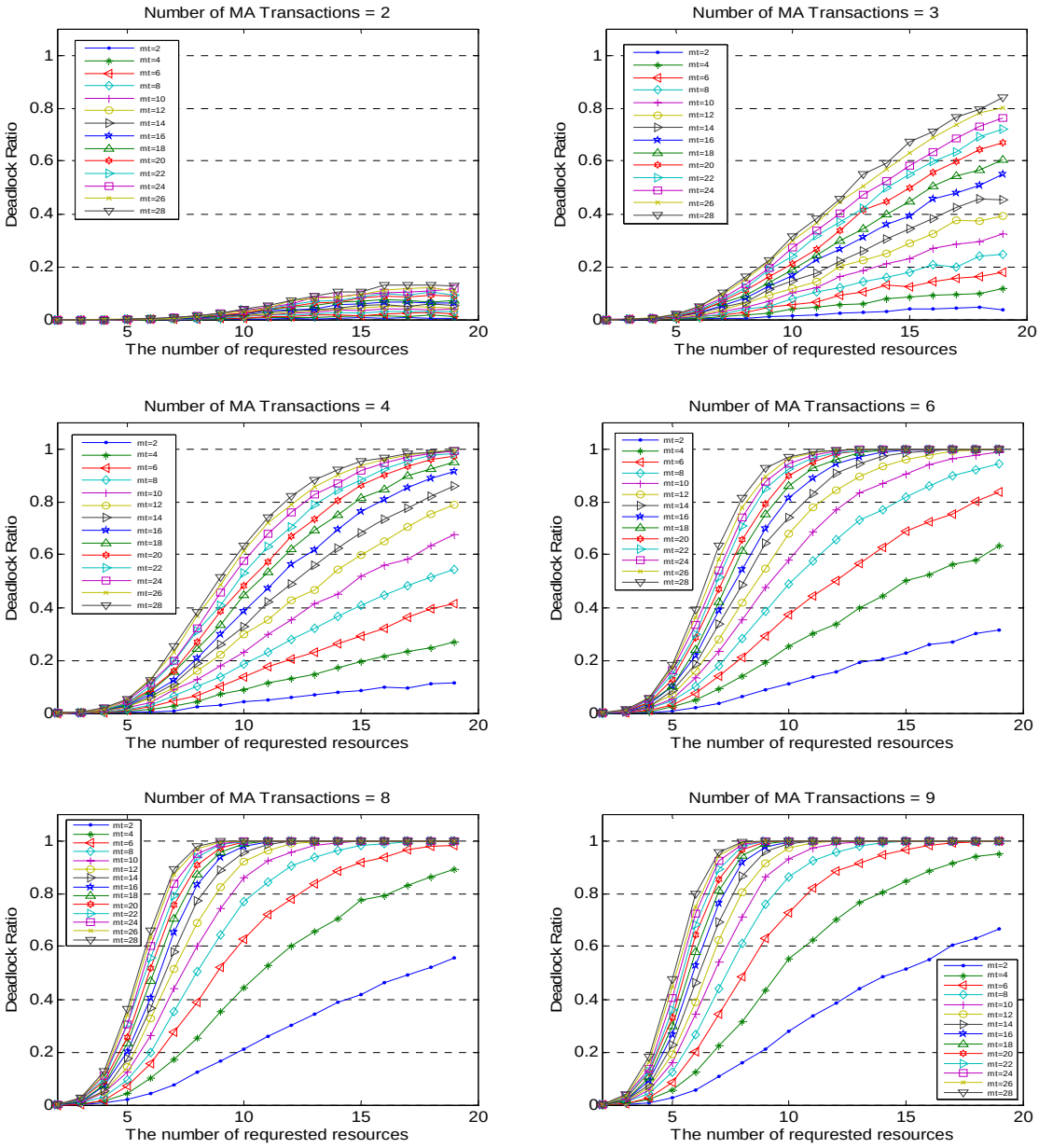


Figure 6.5 Probabilities of Deadlock in MA Transactions

In this section, we perform a simulation to evaluate the probability of deadlock occurrence in MA transactions. We simulate a system consisting of N servers ($2 \leqslant N \leqslant 25$). On each server there is only one resource. There are k ($2 \leqslant k \leqslant 10$) MAs simultaneously performing MA transactions on a subset of these N servers. Since we are considering the one-resource model, each MA applies only one resource on a server. If an MA gets the permission to use

a resource, it will lock the resource until it commits the MA transaction. Since a server only has one resource, the number of resources an MA locks is equal to the number of servers the MA has visited. To estimate the probability of deadlock occurrence, we use the *average deadlock ratio* as metric. In order to let the average deadlock ratio approximate the real probability, we execute each simulation 5000 times with the same settings, and then compute the deadlock ratio as (Number of deadlock occurrence)/5000.

We compare the probability of deadlock occurrence according to three parameters:
(1) Different degrees of concurrency. The number of MA transactions executing concurrently in an MA system ranges from *2* to *9*;
(2) Different migration time between two servers, which ranges from *2ms* to *28ms* (denoted as mt in Figure 6.5). This covers the migration time in networks with low latency such as LAN and those with high latency such as the Internet;
(3) Different number of resources being applied, ranging from *2* to *19*.

Figure 6.5 illustrates the simulation results. It can be observed that with a fixed number of concurrent MA transactions, the deadlock ratio increases with an increasing migration time, and more resources applied. If we increase the number of concurrent MA transactions, the deadlock ratio increases sharply.

Now let us consider the impact of deadlock so that, more specifically, we can answer the question, how many MA transactions will be blocked if a deadlock occurs without being resolved? There are two types of MA transaction that can be blocked. One type is the MA transactions involved in the deadlock ring and the other is the MA transactions that are applying the resource locked in the deadlock ring. To evaluate the impact of deadlock, we let the MA transactions run until all the MA transactions are either finished or blocked, and then we count the number of blocked transactions. By executing each simulation 5000

times, we obtained a spectrum of probabilities for the different numbers of MA transactions that had been blocked. Figure 6.6 shows these probabilities for the situations with a fixed mean migration time (MMT = 10 ms) and different numbers of MA transactions.

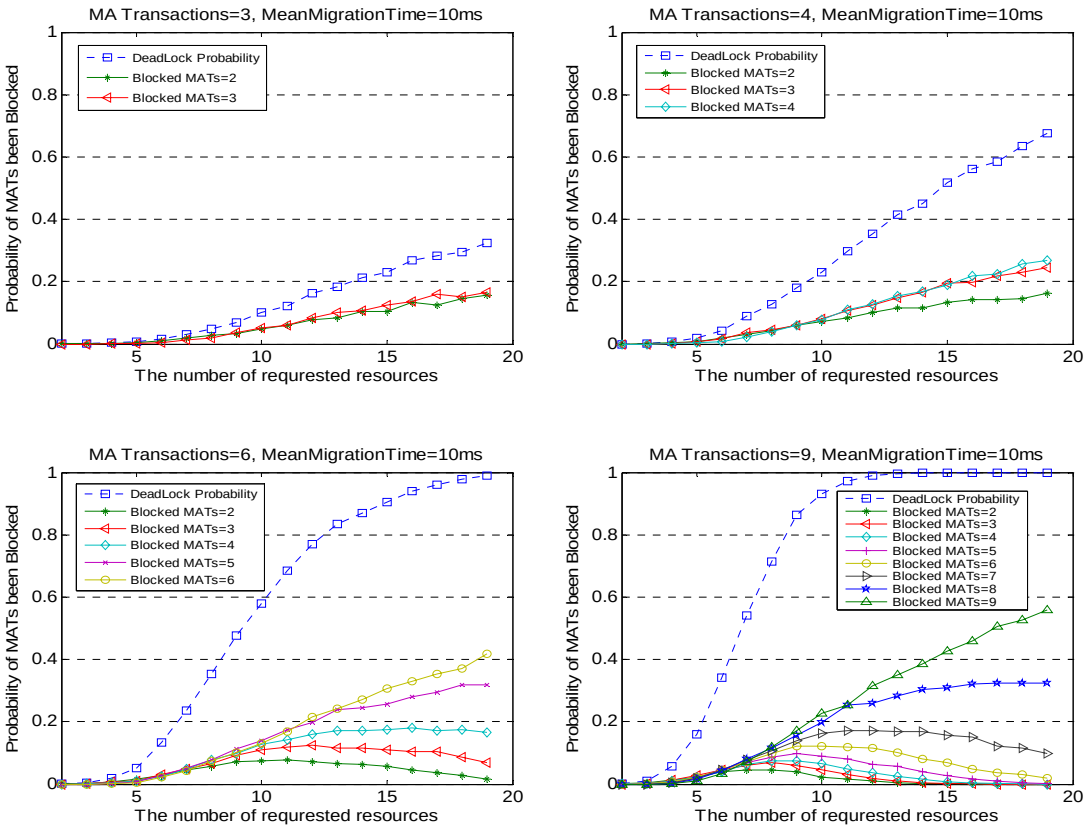


Figure 6.6 Probabilities for the Number of MA Transactions been Blocked

In Figure 6.6, the dotted line with square marks denotes the deadlock probability getting from Figure 6.5. We can see that with a higher deadlock probability, there are more MA transactions being blocked. One solution is to prevent the deadlock, for example by using the open subtransactions to reduce the distributed deadlock. Another solution is to detect and resolve the deadlock. We discuss these two solutions in the following sections.

6.3.3 Deadlock Prevention in MA Transactions

We consider two ways to deal with deadlock in MA transactions: either prevent it or detect and resolve it. The prevention of deadlock means breaking at least one of the four necessary conditions for deadlock: *mutual exclusion*, *non-preemption*, *hold-and-wait*, and *circular wait*. There are four basic schemes for doing this, each dependent on one of the four following specific requirements:

(1) An MA must acquire all the resources it needs before its execution;

(2) An MA must release all resources before acquiring any new resources;

(3) An MA commits suicide if it cannot complete;

(4) An MA must acquire resources in a pre-defined order.

We choose not to adopt the first scheme because it is difficult to satisfy since we need to know the required resources in advance and must hold up resources even when we do not use them. The second scheme is rejected because it can be in breach of the consistency and isolation properties of transaction. The third scheme is also rejected because it will induce a high abort ratio. The fourth scheme appears to be more suitable to MA transactions as it only requires all the MAs acquire the resources in a predefined sequence, which is not difficult for MA applications if their itineraries is adjustable.

Whereas the local deadlock on a server will be solved using a base transaction system, in our MA transaction model, a top-level transaction handles only distributed deadlocks. We can assign each server with a global unique ID, and all the MAs visit the servers in an ID increasing sequence. This prevents the occurrence of deadlock as there is no circular wait but it does so at the cost of the system's flexibility as an MA cannot visit servers according to its preferred sequence. We deal with this by setting up a coordinator. The coordinator

satisfies MA enquiries as to the current sequence for visiting hosts and ensures their itineraries have the same sequence. The VIP MA (MA has the highest priority) can decide the host visitation sequence and inform the coordinator. This scheme, however, is also not ideal in that it the coordinator provides a single point of failure. To this extent, it may be seen as demonstrating the limitations of deadlock prevention schemes, and justifying our focus on deadlock detection algorithms.

6.3.4 Deadlock Detection in MA Transactions

In this section, we propose deadlock detection algorithms. A deadlock detection algorithm must satisfy two properties:

(1) Safety: when it detects a deadlock, then there must be indeed a deadlock;

(2) Progress: when there is a deadlock, it must be detected eventually.

We first introduce a lemma and theorem which will be used to proof the safety property for all the deadlock detection algorithms introduced in the following sections.

Lemma 6.1: *If all transactions are 2PL, then a cycle in the WFG indicates a deadlock.*

The proof of Lemma 6.1 can be found in [WUU85]. A cycle can also be found in a group of MAs which perform MA transactions. If *2PL* is adopted, an MA will be blocked if the required resource is not available. A set of blocked MAs $\{MA_i, MA_{i+1},\ldots, MA_{i+n}\}$ can form a directed chain by the directed edges. If we follow the direction of the edges, we can start from an MA and return to the same MA, (i.e. MA_i->MA_{i+1}->… -> MA_{i+n}-> MA_i), then a loop (cycle) is formed. For example, in Figure 6.7, if MA_4 applies R_1, a loop is formed: MA_1->MA_2->MA_3->MA_4->MA_1.

Theorem 6.1: *If all MA transactions adopt 2PL, then a loop detected in the Waiting For Graph (WFG) by a deadlock detection algorithm indicates a deadlock.*

Proof: Since each blocked MA in the loop represents a transaction, and *2PL* is used in our discussed MA transactions, according to Lemma 6.1, when a deadlock detection algorithm detects the loop, there must be indeed a deadlock.

Two widely adopted deadlock detection mechanisms in distributed system are edge chasing and path pushing (introduced in Section 2.6.3 of Chapter 2). When an MA execute on a server, it has no difference with a process in distributed system. As a result, the edge chasing based deadlock detection algorithms for MA transactions are almost the same as the algorithms proposed for distributed systems. But when we implement them in MA systems, we must consider the specific data structure catering for the MA transaction. We need structures to store the WFG and the trace of an MA. The deadlock table in *RM* can store the WFG and the itinerary stored in *TPM* can provide the trace to be chased. Example deadlock tables are shown in Tables 6.4 and 6.5.

Table 6.4 MA Platform1's Locking Table

Resource ID	***Holder's Transaction ID***	***Waiter's Transaction ID***	***Registered Event***
MP1_R1	T_MA1	T_MA2	TO / Release

Table 6.5 MA Platform2's Locking Table

Resource ID	***Holder's Transaction ID***	***Waiter's Transaction ID***	***Registered Event***
MP2_R1	T_MA2	T_MA1	TO / Release

A significant drawback of edge chasing algorithms is that it is difficult to determine when to put them into operation. If we start the detection process once an MA waiting on a locked resource times out, it may be that there is no real deadlock. If there is a real

deadlock, however, we can detect it only after it has occurred and resources have been tied up. A second drawback is that edge chasing algorithms require a pre-defined itinerary or the trance of the MA must be maintained.

On the contrary, path pushing based deadlock detection algorithms do not have above drawbacks. In the following, we propose two path pushing based deadlock detection algorithms for MA transactions and then offer a comparison of their performances. The deadlock tables shown in Tables 6.4 and 6.5 provide the basic data structures for the following algorithms.

1) Path Pushing Based Deadlock Detection Algorithm 1: MA-WFG

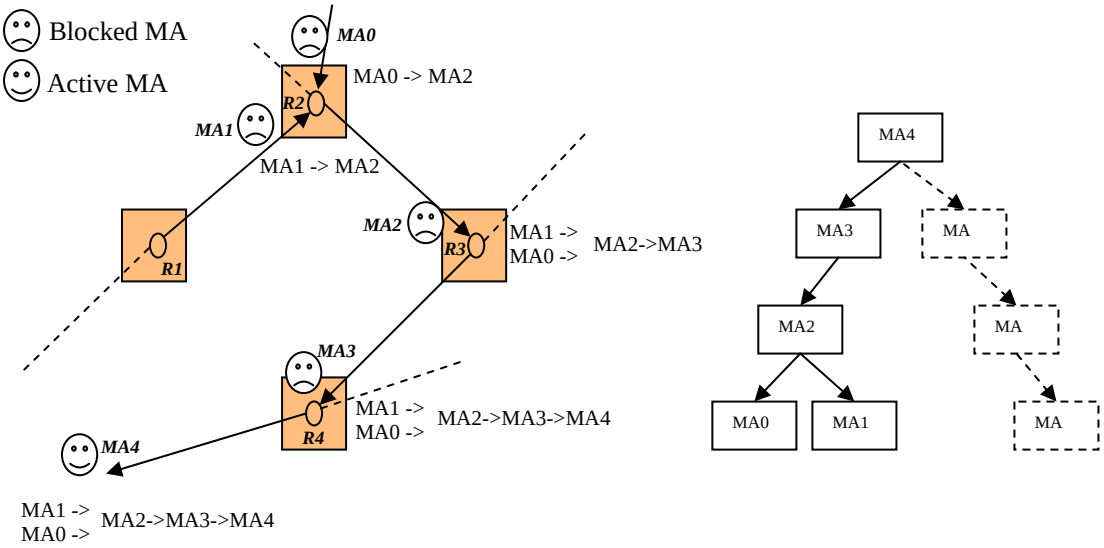


Figure 6.7 The Scenario of MA-WFG

The first algorithm, MA-WFG, lets each MA carry the WFG during its migration. A WFG in an MA transaction system consists of a set of MAs: $\{MA_1, MA_2,\ldots, MA_n\}$. An edge MA_i ->MA_j exists in WFG if and only if MA_i is waiting for a resource held by MA_j. The edge starts from the blocked MA and points to the resource's holder. For example, in Figure 6.7, MA_0 is blocked on R_2 which is locked by MA_2, so the WFG on MA_0 is MA_0->MA_2. In the

following Theorem 6.2, we show that the structure of each WFG is a tree in MA-WFG algorithm.

Theorem 6.2: *In our MA system model, prior to a deadlock being detected, there will be two types of MAs, active and blocked. The WFG carried by active MA or blocked MA is a directed tree.*

Proof: If the WFG is not a tree structure, there must be a cycle in the WFG. The cycle can either be like the loop, which, according to Theorem 6.1, will indicate a deadlock or the cycle will reverse the direction of one or several (but not all) of the edges. Then there is at least one MA waits two other MAs, which contradicts with our assumed one resource model for deadlock detection. So there is no cycle in the WFG. A directed graph without a cycle is a directed tree.

Algorithm 6.3 provides the MA-WFG algorithm in pseudo-code format and the following two theorems will illustrate that MA-WFG can guarantee both the safety property and the progress property of a deadlock detection algorithm.

Theorem 6.3 (For MA-WFG's Safety Property): *When MA-WFG detects a deadlock, there must be indeed a deadlock. That is to say, MA-WFG can guarantee the safety property of a deadlock detection algorithm.*

Proof: In the tree structure WFG of an MA, when a loop is identified, then according to line 6 of Algorithm 6.3 (MA-WFG), a deadlock will be asserted. Since *2PL* is adopted in the MA transactions running in MA-WFG, according to Theorem 6.1, the deadlock asserted by MA-WFG is indeed a deadlock.

Algorithm 6.3 *MA-WFG*

```
// 1. A MA (MAi) Applying a Resource (R);
1)  RM = Si.getRM();            //Get the RM from the platform to check resource's status;
2)  If (R.id ∈ RM.LockingTable)      //The resource has already been locked by another MA;
3)    {MA_locker = RM.getLoker(R.id);   //Get the locker;
4)     If (MAi.WFG.empty() == FALSE)   //If MAi's WFG is not empty;
5)       {MAi.WFG.addRoot(MA_locker); //Update MAi's WFG: let locker be the new root of MAi's WFG;
6)        If (MAi.WFG.loopChecking() == TRUE)   //Perform the loop checking;
7)          Alarm("Deadlock Detected"); //If a loop exists, a deadlock is detected.
8)        Else                                  //No loop, no deadlock.
9)          {WFGmsg = {"MA-WFG", MAi.WFG };      //Compose a WFG message
10)           SendTo(MA_locker, WFGmsg);}   // Send MAi's WFG to the MA that locked resource R;
          }
11)    Else                                   //If WFG is empty, compose a WFG for MAi;
12)      {MAi.WFG.add(MAi->MA_locker);   //Update MAi's WFG;
13)       WFGmsg = {"MA-WFG", MAi.WFG }; //Compose a WFG message
14)       SendTo(MA_locker, WFGmsg);     // Send MAi's WFG to the MA that locked resource R;;
         }
       }
15) Else     //The resource has not been locked by other MAs;
16)  {MAi.reserve(R);   //This MA can get this resource;
17)   RM.record(MAi, R.id); //RM records the allocation of this resource;
      }

//2. Asynchronous message processing. Suppose MAj receive this message.
18) Msg =MAP.getMessge();          //MA platform is responsible to get a message;
19) If (Msg.content = = "MA-WFG")
20)   { MAj.merge(Msg.WFG);          //Merge the incoming WFG with MAj's WFG;
21)     If (MAj.WFG.loopChecking() == TRUE)   //Perform the loop checking;
22)       Alarm("Deadlock Detected");   //If a loop exists, a deadlock is detected.
23)     Else                                  //No loop, no deadlock.
24)       While (MAw = MAj.getWaitedMA() != NULL)
25)         {WFGmsg = {"MA-WFG", MAj.WFG };      //Compose a WFG message
26)          SendTo(MAw, WFGmsg);     // Send MAj's WFG to all the MAs that are waiting MAj;
           }
      }
```

Theorem 6.4 (For MA-WFG's Progress Property): *When there is a deadlock in MA transactions, it must be detected eventually by MA-WFG. That is to say, MA-WFG can guarantee the progress property of a deadlock detection algorithm.*

Proof: If there is a deadlock in the MA transaction system, a loop must exist in the WFG. Suppose the loop consists of *n* MAs: MA_1->… -> MA_n -> MA_1 ($n \geqslant 2$). For this loop, suppose MA_n being blocked by MA_1 (MA_n -> MA_1) is the last step to form this loop. For the WFG (WFG_n) of MA_n, only two conditions exist before the last step. One condition is that WFG_n already contains MA_1. According to line 5~7 of Algorithm 6.3 (MA-WFG), a

deadlock will be detected. The second condition is that MA_1 is not contained in WFG_n. According to line 9~10, a new WFG: (WFG_n->MA_1) is formed and the new WFG will be sent to MA_1. According to line 24~26, (WFG_n->MA_1) will be forwarded along the loop, until it encounters some MA which belongs to (WFG_n->MA_1). According to line 20~22, a deadlock will be detected. If no deadlock is detected during the forwarding process, (WFG_n->MA_1) will be forwarded along the loop until it reaches MA_n. Since MA_n WFG_n, the deadlock can be detected according to line 5~7.

A problem for MA-WFG is that the MA must carry a tree during its migrations, which will increase the size of an MA and consume more bandwidth. Another problem is that it requires message passing between MAs, which is costly.

2) Path Pushing Based Deadlock Detection Algorithm 2: Host-WFS

Table 6.6 Locking Table for Host-WFS

Resource ID	***Waiter's Transaction ID (waiter column)***	***Holder's Transaction ID (holder column)***	***Registered Event***	***WFS (WFS column)***
MP2_R1	*T_MA1*	*T_MA2*	*TO / Release*	*MA3, MA4…*

In algorithm Host-WFS, we expand the locking table with one more column (as shown in Table 6.6) and the WFG is stored in the locking table. A WFG is been decomposed into three sections: a resource holder (stored in column 2 of Table 6.6), the set of MAs waiting for this resource (waiter set, stored in column 3 of Table 6.6), and the set of MAs that the resource holder is waiting for (wait-for set, stored in column 5 of Table 6.6). Since here the MAs that are waiting resources are represented as the type of set, we call the set as the wait-for set (WFS). To distinguish this algorithm from Algorithm 6.3, we call it Host-WFS.

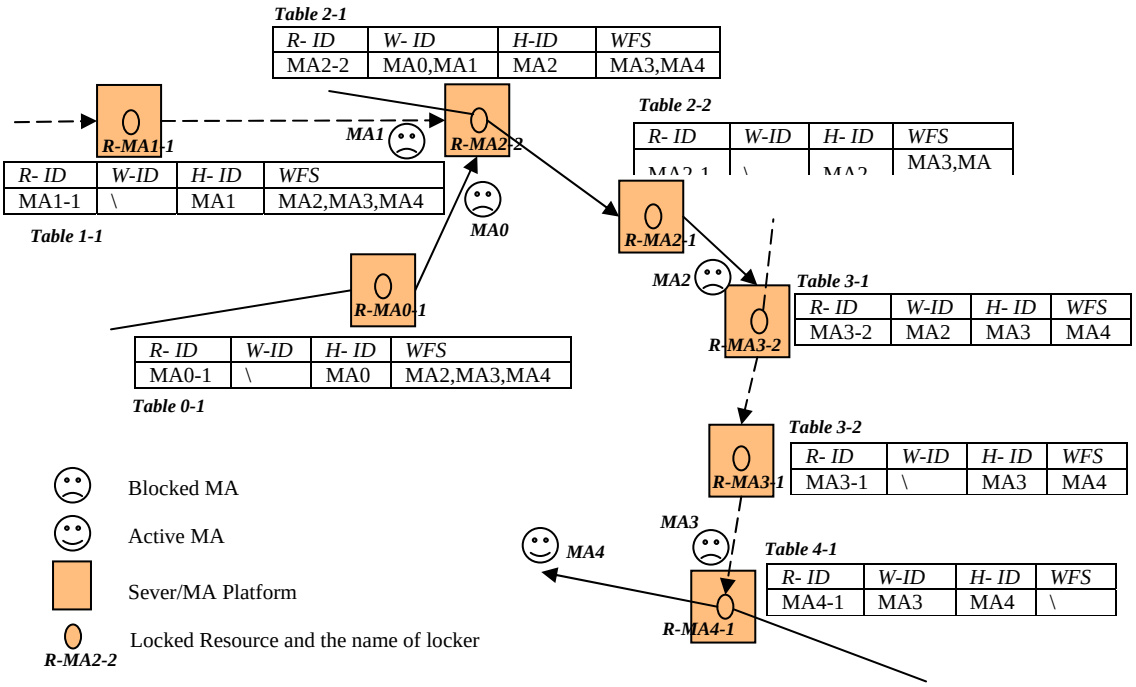


Figure 6.8 The Scenario of Host-WFS

Figure 6.8 illustrates the scenario applying Host-WFS. MA_4 holds resource (*R-MA4-1*) and its ID is recorded in the holder column of *Table 4-1*. When MA_3 is blocked on this resource, it becomes a waiter and its ID is recorded in the waiter column of *Table 4-1*. Then the holder's ID (MA_4's ID) will be forwarded to the WFS columns of all the locking table entries (in *Table 3-1, Table 3-2*) for the waiter (MA_3). If the waiter columns of these entries are not empty (i.e., the entry in *Table 3-1*), then the forwarding process will continue to build up the direct or indirect "waiting for" relations. In Figure 6.8, MA_4's ID is forwarded to all the deadlock tables because MA_3 is waiting for MA_4 directly, while others are waiting for MA_4 indirectly. If an MA is blocked and finds its ID is in the locking table's WFS column, a deadlock is detected. For example, in Figure 6.8, if MA_4 applies *R-MA4-1*, it will be blocked. Since its ID is in the WFS column of *Table 2-1*, a deadlock is detected. The algorithm of Host-WFS in pseudo-code format is illustrated in Algorithm 6.4.

Algorithm 6.4 *Host-WFS*

```
// 1. an MA (MAi) Applying a Resource (R);
1)  RM = Si.getRM();            //Get the RM from the platform to check resource's status;
2)  If (R.id ∈ RM.LockingTable)      //The resource has already been locked by another MA;
3)      {WFS = RM. LockingTable(R.id).getWFS();   //Get the locker;
4)       If (MAi.id ∈ WFS)   //If MAi's ID is in the WFG;
5)            Alarm("Deadlock Detected"); //A loop exists and a deadlock is detected.
6)       Else                                  //If WFG is empty, compose a WFG for MAi;
7)            {RM.LockingTable.addWaitingMA(R.id, MAi.id);   //add MAi's ID in WaitingMA_ID column;
8)             WFSmsg = {"Host-WFS", MAi.id, WFS }; //Compose a WFS message
9)             Itinerary = MAi.getItinerary();   //Get MA's itinerary;
10)            VisitedServers = Itinerary.getPrevious(Si); //Get the servers that MA have visited
11)            While (S = VisitedServers.getNext() != NULL)
12)                 SendTo(S, WFSmsg);     // Send WFS message to the MAs that MAi has visited;
               }
         }
13)  Else     //The resource has not been locked by other MAs;
14)     {MAi.reserve(R);   //This MA can get this resource;
15)      RM.record(MAi, R.id); //RM records the allocation of this resource;
        }

//2. Asynchronous message processing. Suppose MAj receive this message.
16)  Msg =MAP.getMessge();            //MA platform is responsible to get a message;
17)  If (Msg.content = = "Host-WFS")
18)     {   WaitingMA_id = Msg.getWaitingMA();
19)         If (LockingTableEntry = RM. LockingTable(WaitingMA_id) != NULL)
20)             {WFS = LockingTableEntry.getWFS();
21)              WaitingMASet = LockingTableEntry.getWaitingMASet ();
22)              WFS.merge(Msg.getWFS());          //Merge the incoming WFG with MAj's WFG;
23)              If (WaitingMASet ∩ WFS != Φ)   //Perform the loop checking;
24)                   Alarm("Deadlock Detected");   //If a loop exists, a deadlock is detected.
25)              Else                                //No loop, no deadlock.
26)                   If (WaitingMASet != Φ)
27)                        While (MAw = WaitingMASet.getNextWaitedMA() != NULL)
28)                             {WFSmsg = {"Host-WFS", MAw.id, WFS }; //Compose a WFS message
29)                              Itinerary = MAw.getItinerary();   //Get MA's itinerary;
30)                              VisitedServers = Itinerary.getPrevious(Si); //MA visited servers
31)                              While (S = VisitedServers.getNext() != NULL)
32)                                   SendTo(S, WFSmsg); // Send WFS msgs to MAs MAi has visited;
                                 }
              }
         }
```

Theorem 6.5 (For Host-WFS's Safety Property): *When Host-WFS detects a deadlock, there must be indeed a deadlock. That is to say, Host-WFS can guarantee the safety property of a deadlock detection algorithm.*

Proof: In algorithm Host-WFS, when a loop is identified (*WaitingMASet* ∩ *WFS* != Φ or *MAi.id* ∈ *WFS*), then according to line 4 and line 23 (Algorithm 6.4 Host-WFS), a

deadlock will be asserted. Since *2PL* is adopted in the MA transactions running in Host-WFS, according to Theorem 6.1, the deadlock asserted by Host-WFS is indeed a deadlock.

Theorem 6.6 (For Host-WFS's Progress Property): *When there is a deadlock in MA transactions, it must be detected eventually by Host-WFS. That is to say, Host-WFS can guarantee the progress property of a deadlock detection algorithm.*

Proof: If there is a deadlock in the MA transaction system, a loop must exist in the WFG. Suppose the loop consists of *n* MAs: MA_1-> MA_2…MA_{n-1}-> MA_n -> MA_1 ($n \geqslant 2$), and the last step to form this loop is that MA_n is blocked by resource R_1 which is locked by MA_1. Before the last step, the locking table entry for R_1 ($Entry_{R1}$) only has two possible conditions. One condition is that the MA_n's ID is in the WFS column of $Entry_{R1}$. According to line 4~5 of Algorithm 6.4, a deadlock is detected. The other condition is that the MA_n's ID is still not in the WFS column of $Entry_{R1}$. Because the chain "MA_1-> MA_2…MA_{n-1}-> MA_n" exists before the last step, according to line 7~12 and line 26~32 of Algorithm 6.4, the MA_n's ID will ultimately be forwarded to the WFS column of $Entry_{R1}$. According to line 23~24 of Algorithm 6.4, the deadlock is detected.

The advantage of Host-WFS is that it relieves the MA of the burden of carrying a WFG. On the contrary, the WFG is arranged in a form of "wait for set" and is distributed and stored on hosts. Therefore, the task of path pushing is finished by the hosts and the MA need not participate in the operation of path pushing. Intuitively, this feather provides the probability for the MA transactions and the path pushing executing in parallel, so as to make Host-WFS more efficient. We will report the result of performance comparison between MA-WFG and Host-WFS in next subsection.

3) Performance Comparison

The main metric used for evaluating and comparing the performance of path pushing style deadlock detection algorithms is message cost. Since WFG_messages can be piggybacked in MA-WFG and Host-WFS, different execution scenarios will produce different message overheads. Suppose the loop for a deadlock involves k blocked MAs, and between two blocked MAs, there are n_i ($0<i\leq k$, $n_i>0$) hosts. For a fair comparison, we adopt the forwarding pointer scheme for delivering the WFG_message in MA-WFG. The message cost ($C_{message\_cost}$) for both algorithms satisfies the following inequality

$$\sum_{i=1}^{k} n_i \leq C_{message\_cost} \leq MAX\ (\sum_{j=1}^{k} j * n_i)\ .$$

We can see that the upper bound and lower bound of the message cost are the same for both algorithms. It is obviously that an algorithm standing at the upper bound or lower bound will produce different message cost, but the problem is that we could not know which algorithm would stand on which side by theoretical analysis. Therefore, we carried out simulations to evaluate the performance of the two algorithms. The setup of the simulations is the same as the simulations reported in Section 6.3.2. In order to compare the performance of other types of algorithms as well, we also implemented two edge chasing style deadlock detection algorithms, namely *MME* [MIT84] and *CME* [CHY82].

Figures 6.9 and 6.10 illustrate the simulation results. From Figure 6.9, we can see that our proposed two path pushing algorithms (MA-WFG and Host-WFS) perform better than the two edge chasing algorithms (MME and CME) in terms of message cost when more resources (*>10*) are requested. For the two path pushing algorithms, Host-WFS performs better than MA-WFG when more resources (*>10*) are requested.

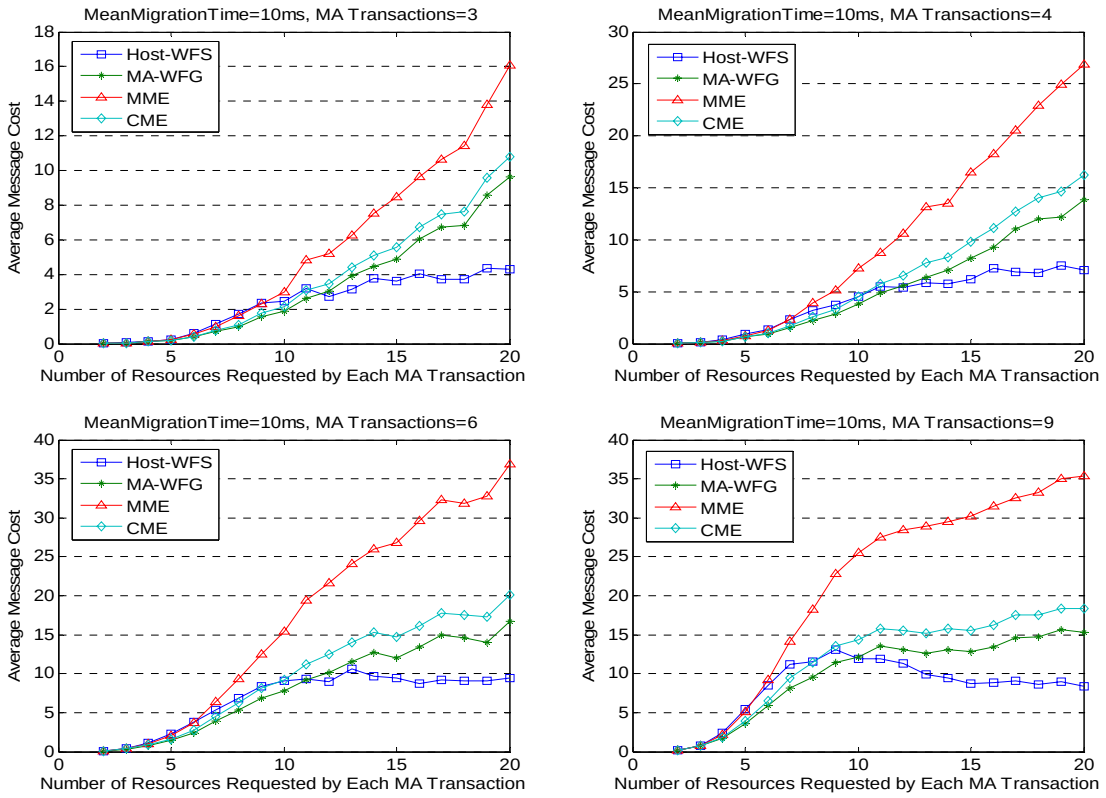


Figure 6.9 Message Cost for Four Different Deadlock Detection Algorithms

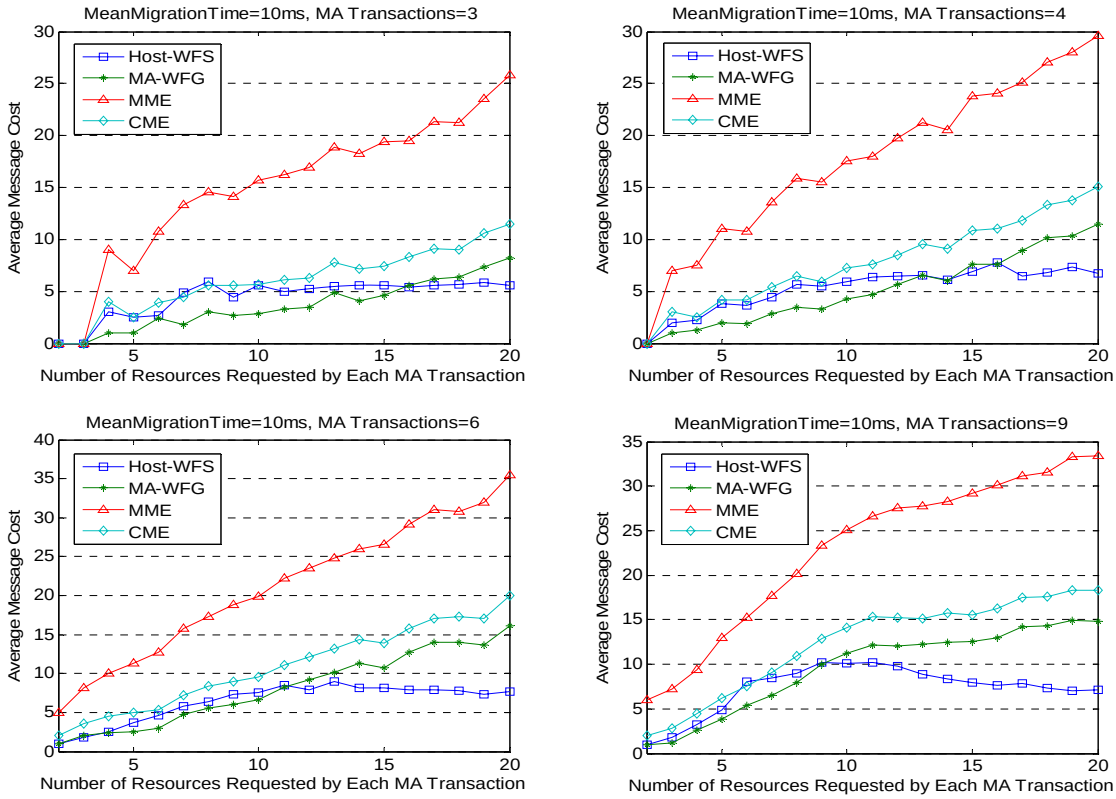


Figure 6.10 Message Cost for Detecting a Deadlock

Figure 6.10 shows the message cost for the detection of the first deadlock in an MA system. Different from Figure 6.9, path pushing style algorithms almost always achieve better performance than edge chasing style algorithms. For the two path pushing algorithms, generally Host-WFS performs better than MA-WFG when more resources are requested.

Table 6.7 The Threshold in Figure 6.10

Number of MA Transactions	***The Threshold Value on X-axes***
3	Requested Resources = 16
6	Requested Resources = 11
9	Requested Resources = 9

Table 6.7 shows the threshold values of the number of resource requests shown on the x-axes in Figure 6.10, at which the message cost of Host-WFS becomes less than that of MA-WFG. From Table 6.7, we can see that with more MA transactions involved, the threshold values become smaller. That is to say, Host-WFS has more chance to perform better than MA-WFG when more MA transactions executing concurrently. Considering that dozens or hundreds of MA transactions may execute concurrently in an MA system and each MA transaction may request dozens of resources, it is obviously that Host-WFS can achieve better performance than MA-WFG.

Another advantage of Host-WFS is that its message cost keeps stable (both in Figure 6.9 and Figure 6.10), while the message costs of other algorithms keep increasing. With the above analysis, we can draw a conclusion that Host-WFS is better suited for MA transactions.

6.4 Summary

Existing works on MA transaction support do not consider how to deploy MA transactions on top of the de-facto transaction support technologies. In this research, we have proposed a system model, system architecture, and related algorithms for MA transactions which integrate MA transactions with the widely deployed existing transaction support systems. The system model is a two-level nested transaction model. Accordingly, the system architecture is separated into two layers. The lower layer is the execution environment of the subtransactions, which is implemented by the existing transaction processing systems. The higher layer implements the top-level MA transaction involving functionalities such as commitment/abort control, resource management, and distributed deadlock detection. In this way, we integrate the MA computing model with the underlying de-facto transaction technologies, making use of the existing transaction processing system available on the hosts. Taking into account the characteristics of mobile agents, we proposed an adaptive commitment model for MA transactions and studied the issue of how to handle deadlocks for MA transactions. We further proposed two new path-pushing style deadlock detection algorithms. Simulation results show that the message cost of path-pushing style algorithms is lower than that of edge-chasing style algorithms, and in particular, the Host-WFS algorithm is particularly efficient for MA transactions.

Chapter 7
Conclusions and Future Works

In this chapter, we briefly summarize our work and outline directions for future research.

7.1 Conclusions

This research proposes a framework for fault tolerant MA systems. The fault tolerance techniques and the algorithms developed took into account the particular characteristics of MA systems. They allow the construction of an effective and efficient fault tolerant MA system with the support at different layers of system architecture. The system makes use of a failure detector and supports MA transaction. The failure detector provides failure notification and triggers recovery processing. MA transaction guarantees the system's consistency during recovery processing. Rollback is implemented using checkpointing and primary-backup.

We proposed a general model for implementing failure detectors (FDs) that is particularly suitable for MA systems. Our proposed failure detector, NTFD, uses failure notification in preference to the use of heartbeat message passing. As a result, NTFD is both highly scalable and economical in terms of network bandwidth. The biggest advantage of NTFD is that it is detects failures accurately -- a highly valued property in many applications. In e-commerce, for example, accurate failure detection prevents duplicated operations. NTFD has been shown to perform well in every area of QoS except completeness. Therefore, NTFD can provide at-most-once execution, which provides a solution to the problem of duplicate execution. In contrast, HBFD has the advantage of completeness. Following an

analysis of the trade-offs for achieving QoS of FDs, we proposed a hybrid FD which combines the advantages of both the NTFD and HBFD approaches.

Our proposed MA transaction architecture can be directly implemented in a realistic system because our design is based on an analysis of a realistic MA execution environment. The two layered system architecture can seamlessly integrate the MA system with the existing transaction platform without the need to modify the underlying transaction platform, making possible the large scale deployment of an MA system.

Our primary-backup based algorithms were designed with efficiency in mind. Using parallel processing results in shorter execution time. This improves the system performance, while at the same time, increases the chances for an MA to bypass host failures. Efficiency is also considered in the checkpointing-based algorithms for MA systems. Based on two checkpointing strategies, we proposed three checkpoint placement algorithms for MA systems and identified the best of the three algorithms through simulations. For MA groups, we adopted CIC for MA systems to benefit the flexible, efficient and scalable properties of CIC scheme. We also proposed DM-CIC, which combines the flexibility of CIC and the reliable migration operation of MA to create a more integrated and efficient checkpointing solution for MA systems.

7.2 Future Works

While the proposed system is complete in functionality, opportunities for further development nonetheless remain. We have currently identified the following directions for future work: guaranteeing accuracy and completeness properties of an FD; message logging; improvement of the primary-backup based algorithms; determining the optimal

checkpointing intervals; consolidating all the research results and incorporate them into a practical MA system.

Currently, neither HBFD, NTFD, nor hybrid FD can simultaneously demonstrate the capability to guarantee both accuracy and completeness properties. [FET03] proposed a solution to this dilemma, which is based on the lease protocol and sacrifices processes that have been falsely detected. In our future work, we will incorporate this method into our hybrid FD.

We will also study message logging for rollback recovery as one way to improve the QoS of fault tolerance. By logging and replaying nondeterministic events in their original order, a process can deterministically recreate its pre-failure state even if this state has not been checkpointed. In general, log-based rollback recovery enables a system to recover beyond the most recent set of consistent checkpoints. It is therefore particularly attractive for applications that frequently interact with the outside world, which consists of all input and output devices that cannot roll back [ELN02].

There is also room for improvement in the backup-based algorithms described in Chapter 4. One goal is to integrate them with MA transaction to support non-idempotent operations. We will also implement them in MA groups to deal with message exchange between group members.

As for the issue of determining checkpointing intervals, in this research we have considered only independent checkpointing of single MAs. However, coordinated or CIC checkpointing is needed in an MA group. In our future work, we will investigate ways to determine the checkpointing intervals for more complicated checkpointing schemes for MA groups.

Finally, up to current stage, our research results are mostly theoretical and evaluated through simulation. Although some algorithms have been implemented on a practical MA platform Naplet [NAP], they are not integrated. In the future, we plan to integrate all the algorithms in an MA platform and provide a convenience interface to the programmers (via a set of APIs) or users (via a GUI) to help them build fault tolerant MA applications.

References

[ALV99] Alvisi, L., Elnozahy, E. N., Rao, S., Husain, S. A., and Mel, A. D. “An analysis of communication-induced checkpointing”, In Digest of Papers, FTCS-29, The Twenty Nineth Annual International Symposium on Fault-Tolerant Computing (Madison,Wisconsin), 242–249. 1999.

[ASH02] B. Ashfield, D. Deugo, F. Oppacher and T. White, “Distributed Deadlock Detection in Mobile Agent Systems”, In Proc. 15th Inte’l Conf. on Industrial and Eng. Application of AI and Expert Sys., Caims, Australia, Jun. 17-20, 2002, pp. 146-156.

[BAR81] Bartlett, J. F. “A Non Stop Kernel”, In Proceedings of the Eighth ACM Symposium on Operating Systems Principles, 22–29. 1981.

[BER02] M. Bertier, O. Marin, and P. Sens., “Implementation and performance evaluation of an adaptable failure detector”, In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2002), pages 354–363, Washington, DC, June 23–26, 2002.

[BER03] Bertier, M.; Marin, O.; Sens, P., “Performance analysis of a hierarchical failure detector”, Proceedings, 2003 International Conference on Dependable Systems and Networks, 22-25 June 2003 Page(s):635 – 644

[BRA83] G. Bracha, S. Toueg, “A Distributed Algorithm for Generalized Deadlock Detection”, Tech. Rep. TR 83-558, Cornell University, Ithaca, N.Y., 1983.

[BRI84] D. Briatico, A. Ciuoletti, and L. Simoncini, “A Distributed Domino-Effect Free Recovery Algorithm”, In Proceedings of the IEEE International Symposium on Reliability Distributed Software and Database, pages 207~215, December 1984.

[CAO01a] J. Cao, G.H. Chan, W. Jia, and T. Dillon, "Checkpointing and Rollback of Wide-Area Distributed Applications Using Mobile Agents", Proc. IPDPS2001 - IEEE 2001 International Parallel and Distributed Processing, April 2001, San Francisco, USA.

[CAO01b] J. Cao, Alvin T. S. Chan, Jie Wu, "Achieving Replication Consistency using Cooperating Mobile Agents", Proc. IPPC 2001 Workshop on Wireless Networks and Mobile Computing, Valencia, Spain. Sep 2001.

[CAO03] Jiannong Cao, Yudong Sun, Xianbin Wang, Sajal K. Das. (2003) “Scalable load balancing on distributed web servers using mobile agents”, Journal of Parallel and Distributed Computing. 63(2003) 996-1005. May.

[CAO04] Cao Jiannong, Zhou Jingyang, Zhu Weiwei, Chen Daoxu and Lu Jian, “A Mobile Agent Enabled Approach for Distributed Deadlock Detection”, in Proceedings of the 3rd International Conference on Grid and Cooperative Computing (GCC'04), LNCS 3251,Wuhan, China, Oct. 21-24, 2004, pp. 535-542

[CAO04b] Jiannong Cao, Liang Zhang, Jin Yang, Das, S.K.; “Reliable mobile agent communication protocol”, Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS’04) 468 – 475, March 23-26, 2004 Tokyo, Japan

[CAO05] Jiannong Cao, Wei Xu, Alvin T.S. Chan, Jing Li, "A Reliable Multicast Protocol for Mobile Agent Communications", *Proc. 7th International Symposium on Autonomous Decentralized Systems (ISADS 2005)* (IEEE Computer Society Press), April 4-8, 2005, Chengdu, China.

[CHA96] Tushar Deepak Chandra and Sam Toueg, “Unreliable failure detectors for reliable distributed systems”. Journal of the ACM, 43(2):225–267, March 1996. A preliminary version appeared in Proceedings of the 10th ACM Symposium on Principles of Distributed Computing, August, 1991, 325–340.

[CHD95] D. Chess, C. harrison, and A. Kershenbaum, “Mobile Agents: Are They a Good Idea?”, IBM Research Report, RC 19887 (#88465) 3/16/95.

[CHE02] Wei Chen; Sam Toueg; Aguilera, M.K., “On the quality of service of failure detectors”, IEEE Transactions on Computers, Volume 51, Issue 5, May 2002 Page(s):561 - 580

[CHN03] X. Chen and M.R. Lyu. “Performance and Effectiveness Analysis of Checkpointing in Mobile Environments”, Proc. of the 22nd Int’l Symp. On Reliable Distributed Systems (SRDS’03), pp. 131-140, 2003.

[CHR94] Chrysanthis, P. and Ramamritham, K., “Synthesis of extended transaction models using ACTA”, ACM Trans. Datab. Syst. (TODS) 1994 19, 3, 450–491.

[CHY82] K. M. Chandy and J. Misra, “A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems”, In Proc.1st ACM Annual Symp. on Principles of Distributed Computing, Ottawa, Canada, Aug. 18-20 1982, pp. 157-164.

[CHY83] K. M. Chandy, J. Misra, L. M. Haas, “Distributed Deadlock Detection”, ACM Trans.on Computer Systems, Vol. 1(2), May. 1983, pp. 144-156.

[CHY85] Chandy, M., Lamport, L. “Distributed snapshots: Determining global states of distributed systems”, ACM Trans. Comput. Syst. 31, 1, 63–75. 1985.

[DAL98] M. Dalmeijer, E. Rietjens, D. Hammer, A. Aerts, M. Schoede, "A reliable mobile agents architecture," in Proc. 1st Int. Symp. Object-Oriented Real-Time Computing, Kyoto, Japan, Apr. 1998.

[DAS99] P. Dasgupta, N. narasimhan, L.E. Moser, and P.M. Melliar-Smith, "MAgNET: Mobile Agents for Networked Electronic Trading", IEEE Trans. On Knowledge and Data Engineering, Vol. 11, No. 4, July/Aug. 1999. pp. 509-525.

[DAV69] Jasper, David P., "A discussion of checkpoint/restart" Software Age (Oct. 1969), 9~14

[ELN02] E. N. Elnozahy, D. B. Johnson, Y. M. Wang. "A Survey of Rollback-Recovery Protocols in Message Passing Systems", ACM Computer Surveys, Volume 34, Number 3, September 2002 pp. 375-408

[FAL05] Falai, L.; Bondavalli, A, "Experimental evaluation of the QoS of failure detectors on wide area network." Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on 28 June-1 July 2005 Page(s):624 – 633

[FEN01] X. Feng, J. Cao, Jian Lu, Henry Chan, "An Efficient Mailbox-based Algorithm for Message Delivery in Mobile Agent Systems", to appear in Proc. 5th IEEE International Conference on Mobile Agents, Dec. 2-4, 2001, Atlanta, Georgia, USA.

[FET03] Fetzer, C.; "Perfect failure detection in timed asynchronous systems", IEEE Transactions on Computers Volume 52, Issue 2, Feb. 2003 Page(s):99 – 112

[FUN99] S. Funfrocken, "Integrating Java-based Mobile Agents into Web Servers under Secrity Concerns", Proc. 31st Hawaii Int'l Conf. On System Sciences (HICSS31), Kona, Hawaii, Jan. 1998, pp. 34-43.

[GAR87] H. Garcia-Molina and K. Salem, "Sagas," Proc. ACM SIGMOD Int'l Conf. Management of Data and Symp. Principles of Database Systems, pp. 249-259, 1987.

[GEN00] E. Gendelman, L.F. Bic, and M.B. Dillencourt. "An Application-Transparent, Platform-Independent Approach to Rollback-recovery for Mobile Agent Systems", Proc. 20th Int'l Conf. on Distributed Computing Syst., 2000.

[GEL78] E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems under Intermittent Failures," CO". ACM 21(6) pp. 493-499 (June 1978).

[GEL79] E. Gelenbe, "On the Optimum Checkpoint Interval," Journal of the ACM 26(2) pp. 59-270 (Apr. 1979).

[GRJ81] J. Gray, “The Transaction Concept: Virtues and Limitations,” Proc. Int’l Conf. Very Large Databases, pp. 144-154, 1981.

[GRJ93] J. GRAY AND REUTER, A. 1993. “Transaction Processing: Concepts and Techniques”. Morgan Kaufmann, San Mateo, CA.

[GRR95] R.S. Gray, "Agent Tcl: A Transportable Agent System", Proc. CIKM Workshop on Intelligent Information Agents, Baltimore, MN, USA, Dec 1995.

[GRR98] R. Gray, D. Kotz, G. Vigna, and D. Rus, “D’Agents: Security in a multi-language, mobile agent system”, in Mobile Agents and Security, Lecture Notes in Computer Science, Vol. 1419 (ed. G. Vigna), Springer-Verlag, 1998. pp. 154-187.

[GRR01] R.S. Gray, et al., “Mobile Agents: Motivations and State-of-the-Art Systems”, in Handbook of Agent Technology, (ed. J. Bradshaw), AAAI/MIT Press, 2001. http://agent.cs.dartmouth.edu/

[GUP01] Indranil Gupta, Tushar D. Chandra, German S. Goldszmidt, “On scalable and efficient distributed failure detectors” August 2001 Proceedings of the twentieth annual ACM symposium on Principles of distributed computing

[HAG98] L. Hagen, M. Breugst, and T. Magedanz, “Impact of Mobile Agent Technology on Mobile Communications System Evolution”, IEEE Personal Communications, Aug. 1998. pp.56-69.

[HIL97] S. G. Hild and J. H. Bischof, “Agent-based Multicast Routing”, IBM Research Report, RZ 2975 (#93021) 11/17/97.

[HUA95] Yennun Huang; Chandar Kintala; “Software fault tolerance in the application layer” Software Fault tolerance, John Wiley & Sons Ltd. 1995

[JOH95] Johansen, D.; van Renesse, R.; Schneider, F.B., “Operating system support for mobile agents”, Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on , 4-5 May 1995Pages:42 – 45

[JOH99] D. Johansen, K. Marzullo, F.B. Schneider, K. Jacobsen, and D. Zagorodnov, “NAP: Practical Fault-Tolerance for Itinerant Computations,” Proc. 19th Int’l Conf. Distributed Computing Systems (ICDCS ’99), June 1999.

[KIM02] Hyunjoo Kim; Yeom, H.Y.; Taesoon Park; Hyoungwoo Park; “The cost of checkpointing, logging and recovery for the mobile agent systems”, Proceedings. 2002 Pacific Rim International Symposium on Dependable Computing, 16-18 Dec. 2002

[KOM02] Komiya, T.; Ohsida, H.; Takizawa, M.; “Mobile agent model for distributed systems”, Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on , 2-5 July 2002 Pages:131 – 136

[KSH91] A. D. Kshemkalyani and M. Singhal, “Invariant-based Verification of a Distributed Deadlock Detection Algorithm”, IEEE Trans. on Software Engineering, Vol. 17(8) , Aug. 1991, pp. 789 – 799.

[LAN98] D.B. Lange and M. Oshima, “Programming And Deploying Java Mobile Agents With Aglets”, Addison Wesley, 1998.

[LAN99] D.B. Lange and M. Oshima, “Seven Good Reasons for Mobile Agents”, Communication of the ACM, Vol. 42, No. 3, March 1999. pp. 88-89.

[LAM78] L. Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”, Communications of the ACM, 21(7):558~565, July 1978.

[LIC90] Li, C. C. and Fuchs,W.K. “CATCH: Compiler assisted techniques for checkpointing”, In Digest of Papers, FTCS-20, The Twentieth Annual International Symposium on Fault-Tolerant Computing, 74–81. 1990.

[LYU03] M.R. Lyu and T.Y. Wong, "A Progressive Fault Tolerant Mechanism in Mobile Agent Systems," in Proceedings 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI2003), Orlando , Florida, July 27-30 2003, Volume IX, pp. 299-306.

[MAC01] R. J. A. Macedo and F.M. Assis Silva, “Integrating Mobility into Groups”, Proc. European Research Seminars on Advances in Distributed Systems, May 2001, Italy.

[MAN75] K. Mani Chandy, James C. Browne, Charles W. Dissly, and Werner R. Uhrig, “Analytic Models for Rollback and Recovery Strategies in Data Base Systems,” IEEE ransactions on Software Engineering SE-l(1) pp. 100-1 10 (March 1975).

[MIN99] N. Minar, K.H. Kramer and P. maes, “Cooperative Mobile Agents for Dynamic network Routing”, in Software Agents for Future Communication Systems, Springer-Verlag, 1999. ISBN 3-540-65578-6.

[MIT84] D. P. Mitchell, M. J. Merritt, “A Distributed Algorithm for Deadlock Detection and Resolution”, In Proc. ACM Symposium on Principles of Distributed Computing, New York, USA, 1984, pp. 282-284.

[MOH00] A. Mohindra, A. Purakayastha, and P. Thati, “Exploiting nondeterminism for reliability of mobile agent systems,” in Proc. Int. Conf. Dependable Systems Networks, Los Alamitos, CA, 2000, pp. 144–153.

[MOS85] Moss, J. 1985. “Nested Transactions: An Approach to Reliable Distributed Computing”, MIT Press, Cambridge, MA.

[NAP] “Naplet: A flexible and reliable mobile agent system for network-centric pervasive computing”, http://www.ece.eng.wayne.edu/~czxu/software/naplet.html

[NET95] Netzer, R. H. and Xu, J. “Necessary and sufficient conditions for consistent global snapshots”, IEEE Trans. Parallel and Distributed Syst. 6, 2, 165–169. 1995

[NIC90] Victor F. Nicola and Johannes M. Van Spanje, “Comparative Analysis of Different Models of Checkpointing and Recovery,” IEEE Trans. Software Engineering 16(8) pp. 807-821 (Aug. 1990).

[OBE82] Ron Obermarck, "Distributed deadlock detection algorithm",ACM Transactions on Database Systems (TODS) archive Volume 7 , Issue 2 (June 1982) Pages: 187 - 208

[OBJ] Objectspace, “Objectspace Voyager Core Technology”, http://www.objectspace.com

[OLI05] Oliner, A.J.; Sahoo, R.K.; Moreira, J.E.; Gupta, M.; “Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems”, Proceedings. 19th IEEE International Symposium on Parallel and Distributed Processing, 04-08 April 2005 Page(s):299b - 299b

[OSM04] Osman, T.; Wagealla, W.; Bargiela, A.; "An approach to rollback recovery of collaborating mobile agents", IEEE Transactions on Systems, Man and Cybernetics, Part C, Volume 34, Issue 1, Feb. 2004 Page(s):48 - 57

[PAG01] Gyung-Leen Park; Hee Yong Youn; Hyun-Seung Choo; “Optimal checkpoint interval analysis using stochastic Petri net”, Proceedings. 2001 Pacific Rim International Symposium on Dependable Computing. 17-19 Dec. 2001 Page(s):57 - 60

[PAL00] Holger Pals, Stefan Petri, and Claus Grewe, “FANTOMAS: Fault Tolerance for Mobile Agents in Clusters”, Parallel and Distributed Processing - Proceedings of 15 IPDPS 2000 Workshops, Cancun, Mexiko, May 2000, Cancun, Mexiko.

[PAR02] Taesoon Park; Ilsoo Byun; Hyunjoo Kim; Yeom, H.Y.; “The performance of checkpointing and replication schemes for fault tolerant mobile agent systems” 2002. Proceedings. 21st IEEE Symposium on Reliable Distributed Systems, 13-16 Oct. 2002

[PEA03] Pears, S.; Jie Xu; Boldyreff, C., "Mobile agent fault tolerance for information retrieval applications: an exception handling approach Autonomous Decentralized Systems", 2003. ISADS 2003. The Sixth International Symposium on , 9-11 April 2003 Pages:115 – 122

[PEI97] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents", Proc. 1st Int'l Workshop on Mobile Agents (Lecture Notes in Computer Science Vol. 1219), 1997, Berlin, Germany.

[PLA95] Plank, Janmes S.; Micah Beck; Gerry Kingsley; "Libckpt: Transparent Checkpointing under Unix", USENIX Winter 1995 Technical Conference New Orleans, Louisiana 16-20, Jan 1995

[PLA98] J. S. Plank and W. R. Elwasif. "Experimental assessment of workstation failures and their impact on checkpointing systems", In 28th International Symposium on Fault-Tolerant Computing, pages 48–57, Munich, June 1998.

[PLE00] S. Pleisch and A. Schiper, "Modeling Fault-Tolerant Mobile Agent Execution as a Sequence of Agreement Problems," Proc. 19th IEEE Symp. Reliable Distributed Systems (SRDS '00), pp. 11-20, Oct. 2000.

[PLE01] S. Pleisch and A. Schiper, "FATOMAS: A Fault-Tolerant Mobile Agent System Based on the Agent-Dependent Approach," Proc. Int'l Conf. Dependable Systems and Networks (DSN '01), pp. 215-224, July 2001.

[PLE02] Pleisch, S.; Schiper, A.; "Non-blocking transactional mobile agent execution", Proceedings of 22nd International Conference on Distributed Computing Systems, 2-5 July 2002 Pages:443 – 444

[PLE03] Pleisch, S.; Schiper, A., "Fault-tolerant mobile agent execution", IEEE Transactions on Computers, Volume: 52 , Issue: 2 , Feb. 2003 Pages:209 – 222

[PLE04] S. Pleisch and A. Schiper. "Approaches to Fault-Tolerant and Transactional Mobile Agent Execution–An Algorithmic View", ACM Computing Surveys, Vol. 36, No. 3, pp. 219-262, 2004.

[PHA98] V. A. Pham and A. Karmouch, "Mobile Software Agents: An Overview", IEEE Communications Magazine, July 1998. pp. 26-37.

[QIH03] Hairong Qi; Yingyue Xu; Xiaoling Wang. (2003) "Mobile-agent-based collaborative signal and information processing in sensor networks", Proceedings of the IEEE Volume 91, Issue 8, Page(s):1172 – 118 Aug.

[QUW05] Wenyu Qu, Hong Shen, Xavier Defago, "A Survey of Mobile Agent-Based Fault-Tolerant Technology," pdcat, pp. 446-450, Sixth International Conference on Parallel and Distributed Computing Applications and Technologies (PDCAT'05), 2005.

[RAN75] Randell, B. “System structure for software fault tolerance”, IEEE Trans. Softw. Engin. 1, 2, 220–232. 1975.

[REN98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. “A gossip-style failure detection service.” In Proceedings of Middleware’98, September 1998.

[ROT98] K. Rothermel, M. Straßer: “A fault-tolerant protocol for providing the exactly-once property of mobile agents”, Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems,20-23 Oct. 1998 Page(s):100 - 108

[RUS80] Russell, D. L. “State restoration in systems of communicating processes”, IEEE Trans. Softw. Engin. 6, 2, 183–194. 1980.

[SCH97] R. Schoonderwoerd, O. Holland and J. Bruten, “Ant-like Agents for Load Balancing in Telecom. Networks”, Proc. 1st Int’l Conf. On Autonomous Agents, CA USA, 1997. pp.209-216.

[SHE01] Sher, R., Aridor, Y., and Etzion, O. “Mobile transactional agents”, In Proc. of 21st IEEE Int. Conference on Distributed Computing Systems (ICDCS’01) 2001 (Phoenix, AZ). 73–80.

[SHI87] Kang G. Shin, Tein-Hsiang Lin, and Yann-Hang Lee, “Optimal Checkpointing of Real-Time Tasks,” IEEE Transactions on Computers C-36(11) pp. 1328-1341 (NOV. 1987).

[SIL97] F. M. de Assis Silva and Sven Krause, “A Distributed Transaction Model based on Mobile Agents”, Mobile Agents, (Eds.) K. Rothermel and R. Popescu, Springer, Proc. 1st International Workshop on Mobile Agents, Berlin, April 1997, pp. 198-209.

[SIL00] F.M. Silver and R. Popesch-Zeletin. “Mobile Agent-Based Transactions in Open Environments”, IEICE Transactions on Communications, E83-B(5), pp. 973-987, 2000.

[SIV98] L. Silva and J. Silva. “System-Level Versus User-Defined Checkpointing”, Proc. of the Symp. on Fault-Tolerant Computing, pp. 68-74, 1998.

[SIV00] L. Silva, V. Batista, and J. Silva, “Fault-tolerant execution of mobile agents,” in Proc. Int. Conf. Dependable Systems Networks, New York, June 2000, pp. 135–143.

[STR98] Markus Strasser, “Reliability concepts for Mobile Agents”, International Journal of Cooperative Information Systems (IJCIS), Volume 7, Number 4, 1998, pp. 355-382

[STR98b] M. Strasser, K. Rothermel, and C. Mailhofer. “Providing Reliable Agents for Electronic Commerce”, Trends in Distributed Systems for Electronic Commerce (TREC’98), LNCS 1402, pp. 241-253, 1998.

[STR00] M. Straasser and K. Rothermel, “System Mechanisms for Partial Rollback of Mobile Agent Execution”, Proc. 20th IEEE Int’l Conf on Distributed Computing Systems, 2000. pp.20-28.

[TAN84] Asser N. Tantawi and Manfred Ruschitzka, “Performance Analysis of Checkpointing Strategies”, ACM Transactions on Computer System 2(2) pp. 123-144 (May 1984).

[TAO00] Tao Shu; Cao Yang; Yin Jianhua; Xu Ning; “A mobile agent based approach for network management”. Communication Technology Proceedings, 2000. WCC - ICCT 2000. International Conference on , Volume: 1 , 21-25 Aug. 2000 Pages:547 - 554 vol.1

[TRI99] A. Tripathi, N. Karnik, et al, “Ajanta: A Mobile Agent Programming System”, Technical Report TR98-016, Dept. of Computer Science, University of Minnesota, April, 1999.

[TRK02] Trivedi, K.S., “Probability and Statistics with Reliablity, Queuing and Computer Science Applications (Second Edition)”, Prentice-Hall, Englewood Cliffs, N.J., 2002

[VOG97] H. Vogler, T. Kunkelmann, M.-L. Moschgath, “An Approach for Mobile Agent Security and Fault Tolerance using Distributed Transactions”, 1997 Int'l Conf. on Parallel and Distributed Systems (ICPADS'97), Seoul, Korea, 1997

[VOG97b] Vogler, H.; Kunkelmann, T.; Moschgath, M.-L.; “Distributed transaction processing as a reliability concept for mobile agents”, Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems, 29-31 Oct. 1997

[VOG98] Vogler, H.; Buchmann, A.; “Using multiple mobile agents for distributed transactions”, Proceedings. 3rd IFCIS International Conference on Cooperative Information Systems, 20-22 Aug. 1998 Page(s):114 – 121

[WAN97] D. Wang, et al, “Concordia: An infrastructure for collaborating mobile agents”, Proc. 1st Int’l Workshop on Mobile Agents, Lecture Notes in Computer Science, Vol. 1219, 1997. pp.86-97.

[WAN99] D. Wang, N. Paciorek, and D. Moore, “Java-based Mobile Agents”, Communications of the ACM, 42(3), March 1999, pp.92-102.

[WAS03] Szu-Chi Wang; Sy-Yen Kuo, “Communication Strategies for Heartbeat-Style Failure Detectors in Wireless Ad Hoc Networks”, . In Proceedings of International Conference on Dependable Systems and Networks, 2003. 22-25 June 2003

[WEI92] Weikum, G. and Schek, H.J. “Concepts and applications of multilevel transactions and open nested transactions”, In Database Transaction Models for Advanced Applications, 1992. A. Elmagarmid, Ed. Morgan-Kaufmann, San Mateo, CA, 515–553.

[WHI94] J. E. White, “Mobile Agents Make a Network an Open Plaftorm for Third-Party Developers”, IEEE Computer, 27(11), Nov. 1994, pp.89-90.

[WON96] K. F. Wong and M. Franklin, “Checkpointing in distributed systems”, Journal of Parallel & Distributed Systems, 35(1):67–75, May 1996.

[WU01] Chenggang Wu; Shaohui Liu; Bo Wang; Zhongzhi Shi; Hua Gu; “Configurable mobile agent and its fault-tolerance mechanism” Proceedings of International Conference on Computer Networks and Mobile Computing, 16-19 Oct. 2001 Pages: 380-389

[XU99] C. Xu and D. Tao, “Building Distributed Applications with Aglet”, http://www.cs.duke.edu/chong/aglet

[XU00a] C.Z. Xu and B. Wims, “A Mobile Agent Based Push Methodology for Global Parallel Computing”, Concurrency: Practice and Experience. Vol. 14(8), July 2000, pp. 705-726.

[YOU74] John W. Young, “A First Order Approximation to the Optimum Checkpoint Interval,” Communications of the ACM 17(9) pp. 530-531 (Sept. 1974).

[ZIV97] A. Ziv and J. Bruck. “An on-line algorithm for checkpoint placement”, IEEE Transactions on Computers, 46(9):976–985, September 1997.