INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
Lian, Richard Chiho

DESIGN OF A FAULT-TOLERANT DISTRIBUTED OPERATING SYSTEM
BASED ON NESTED ATOMIC ACTIONS

The Ohio State University

University Microfilms International 300 N. Zeib Road, Ann Arbor, MI 48106

Copyright 1984

by

Lian, Richard Chiho

All Rights Reserved
DESIGN OF A FAULT-TOLERANT DISTRIBUTED OPERATING SYSTEM
BASED ON NESTED ATOMIC ACTIONS

DISSERTATION

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

By
Richard Chiho Lian, B.S.E.E., M.S.

* * * * *

The Ohio State University
1984

Reading Committee:
Ming T. Liu, Chairman
Dennis Leinbaugh
Neelam Soundararajan

Approved By

Ming T. Liu
Adviser
Department of Computer
and Information Science
To My Parents
ACKNOWLEDGMENTS

During my four and half years of graduate education at The Ohio State University, the primary source of direction has been provided by my advisor, Professor Ming T. Liu. He has suggested the research direction to pursue, provided opportunities for me to obtain design and implementation experiences in local area computer networks, encouraged me to clarify my ideas during the course of this research, and contributed in a significant way to the development of this dissertation. His many hours of reading and re-reading drafts of this document have resulted in a much better document than I alone would have produced. His help is greatly appreciated.

I would like to thank Dr. Dennis Leinbaugh and Dr. Neelam Soundararajan for serving on my reading committee. They have made many suggestions that have improved the quality of the presentation and the technical content of the dissertation. Their understanding and support throughout this entire process is much appreciated.

This research work grew out of discussions in early 1981 among Dr. Ming T. Liu, Dr. Duen-Ping Tsay, and this author to design a network operating system for a local-area computer network. The model presented in this dissertation represents a significant departure from that in the initial discussions but still embodies much of the original concept. Their help is appreciated.

Thanks are also due to all the members in the Reliable Distributed Systems group: Dr. Kyungsook Lee, Lin Chiu, Ahmed Elmagarmid, Bhushan Gupta, I-en Liao, Pan-Foo Liu, Young Oh, Mark Shaw, Amit Sheth, Anoop Singhal, and Shu-Jen Wang. They provided many useful suggestions during the course of my research and have created a cooperative atmosphere in which to work.

I would also like to thank the Instruction and Research Computer Center for the financial support as a Graduate Research Associate throughout my four and half years at The Ohio State University. I am especially grateful to Delbert W. Waggoner, Associate Director of the IRCC, for his support, encouragement, and
friendship while he was Manager of the IRCC/CIS Computing Laboratory.

The work reported in this dissertation was supported in part by U.S. Army CECOM, CENCOMS, Fort Monmouth, New Jersey, under Contract No. DAAB07-83-K-K542. The views, opinions and/or findings contained in this dissertation are those of the author and should not be construed as an official Department of the Army position, policy or decision.

Finally, I would like to thank my loving wife, Jean, for her encouragement, patience, understanding, and assistance during all phases of this work.
VITA

December 10, 1948 . . Born - Ilan, Taiwan, R. O. C.

1970 . . . . . . B.S.E.E., National Chiao Tung University, Hsinchu, Taiwan R. O. C.

1972 . . . . . . M.S., National Chiao Tung University, Hsinchu, Taiwan R. O. C.

1972-1973 . . . . Officer
Automatic Data Processing Section,
Chinese Marine Corps R. O. C.

1973-1974 . . . . Senior System Programmer
Computer Center,
China Airlines Ltd., R. O. C.

Computer Center,
China Airlines Ltd., R. O. C.

1977-1979 . . . . Chief, Programming and Development Section,
Computer Center,
China Airlines Ltd., R. O. C.

1974-1976 . . . . Assistant Professor (part time)
Department of Electronic Engineering,
Lung Hwa Technical College, R. O. C.

1979-1984 . . . . Graduate Research Associate Instruction and Research Computer Center,
The Ohio State University Columbus, Ohio

1981 . . . . . . M.S., The Ohio State University Columbus, Ohio
PUBLICATIONS


FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Computer Architecture
Professor Ming T. Liu

Studies in Computer Operating Systems
Professor Dennis W. Leinbaugh

Studies in Programming Languages and Systems
Professor Jay Ramanathan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>VITA</td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiii</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Distributed Computing Systems</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Characteristics</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Objectives</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Importance of Reliability in Distributed Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Achieving High Reliability</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Fault Prevention</td>
<td>7</td>
</tr>
<tr>
<td>1.3.2 Fault Tolerance</td>
<td>9</td>
</tr>
<tr>
<td>1.3.3 Fault Prevention versus Fault Tolerance</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Motivation, Objectives, and Contributions of Research</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Organization of Dissertation</td>
<td>13</td>
</tr>
<tr>
<td>2. RELATED WORKS</td>
<td>15</td>
</tr>
<tr>
<td>2.1 Recovery Blocks</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Conversation</td>
<td>17</td>
</tr>
</tbody>
</table>
6.1.1 The Client/Server Model. .......................... 117
6.1.2 Communication Primitives. ....................... 118

6.2 Analysis of the Intercell Communication Model
   6.2.1 Reliability. .................................... 124
   6.2.2 Performance. .................................. 125

6.3 Distributed Commitment. .............................. 128
   6.3.1 Local Commitment. ............................. 129
   6.3.2 Global Commitment. ............................ 132

6.4 Analysis of Commit Protocol. .......................... 136
   6.4.1 Robustness. ................................... 136
   6.4.2 Performance. .................................. 137

6.5 Summary. ........................................... 138

7. SYSTEM ARCHITECTURE AND PROTOCOL STRUCTURE. .... 140
   7.1 Design Goal. ..................................... 140
   7.2 System Structure. ................................ 141
   7.3 Naming. .......................................... 145
   7.4 Communication. .................................. 148
   7.5 Protection. ...................................... 149
   7.6 Fault Tolerance. ................................ 150
      7.6.1 Detecting Node Failures and Network Partitions, and Reconfiguring the System
   7.6.2 Restarting from Node Crash. .................... 155
   7.6.3 Surviving Node Crashes or Network Partitions
7.6.4 Tolerating Residual Software Design Faults

7.6.5 Achieving Damage Confinement

7.7 Protocol Structure

7.7.1 Service Layer

7.7.2 Service Support Layer

7.7.3 IPC Layer

7.7.4 Request Scenario

7.8 Summary

8. PERFORMANCE ISSUES

8.1 System Overhead

8.2 System Performance

8.3 Performance Enhancement

8.3.1 Computation Structuring

8.3.2 Algorithm Enhancement

8.3.3 Architectural Support

8.4 Summary

9. SUMMARY AND FUTURE WORK

9.1 The Cell Model and Its Applications

9.2 The Underlying Mechanisms of the Cell Model

9.3 Experiment

9.3.1 System Functions

9.3.2 Implementation

9.3.3 Results

9.4 Areas for Future Work
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Reliability of a Distributed System</td>
<td>5</td>
</tr>
<tr>
<td>2.</td>
<td>System Production</td>
<td>8</td>
</tr>
<tr>
<td>3.</td>
<td>Decomposition of a Computation X</td>
<td>23</td>
</tr>
<tr>
<td>4.</td>
<td>The Cell Model for a Fault-tolerant DOS</td>
<td>25</td>
</tr>
<tr>
<td>5.</td>
<td>Comparisons Between Cells and Moss' Model</td>
<td>27</td>
</tr>
<tr>
<td>6.</td>
<td>The Target Environment</td>
<td>31</td>
</tr>
<tr>
<td>7.</td>
<td>Modeling the Target Environment</td>
<td>34</td>
</tr>
<tr>
<td>8.</td>
<td>Network Partitions</td>
<td>37</td>
</tr>
<tr>
<td>9.</td>
<td>Modeling a Fault-tolerant DOS</td>
<td>40</td>
</tr>
<tr>
<td>11.</td>
<td>Cooperating Cells</td>
<td>49</td>
</tr>
<tr>
<td>12.</td>
<td>The Agent Tree for Cooperating Cells</td>
<td>50</td>
</tr>
<tr>
<td>13.</td>
<td>An Example for File Transfer</td>
<td>54</td>
</tr>
<tr>
<td>14.</td>
<td>A Synchronization Problem</td>
<td>57</td>
</tr>
<tr>
<td>15.</td>
<td>The Client/Server Model</td>
<td>67</td>
</tr>
<tr>
<td>16.</td>
<td>The Transaction Model</td>
<td>74</td>
</tr>
<tr>
<td>17.</td>
<td>The Life Span of a Request in a Cell</td>
<td>76</td>
</tr>
<tr>
<td>18.</td>
<td>An Example for Concurrency Control</td>
<td>85</td>
</tr>
<tr>
<td>Section Number</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>19</td>
<td>Commit IDs and Commit Conditions</td>
<td>86</td>
</tr>
<tr>
<td>20</td>
<td>An Example of Multiple Test Messages</td>
<td>90</td>
</tr>
<tr>
<td>21</td>
<td>An Example of Interprocess Cooperation</td>
<td>97</td>
</tr>
<tr>
<td>22</td>
<td>The State Diagram of a Cell</td>
<td>106</td>
</tr>
<tr>
<td>23</td>
<td>The State Diagram of a Request</td>
<td>107</td>
</tr>
<tr>
<td>24</td>
<td>The Types of Requests</td>
<td>111</td>
</tr>
<tr>
<td>25</td>
<td>The Control Flow of Asynchronous Communication</td>
<td>119</td>
</tr>
<tr>
<td>26</td>
<td>The Control Flow of Synchronous Communication</td>
<td>120</td>
</tr>
<tr>
<td>27</td>
<td>The Control Flow of Quasi-asynchronous Communication</td>
<td>121</td>
</tr>
<tr>
<td>28</td>
<td>An Example for Calculating Response Time</td>
<td>127</td>
</tr>
<tr>
<td>29</td>
<td>An Example for Distributed Commitment</td>
<td>131</td>
</tr>
<tr>
<td>30</td>
<td>Possible Commit Actions for the DPCP</td>
<td>133</td>
</tr>
<tr>
<td>31</td>
<td>Possible commit sequences for a Tree of Nested Atomic Actions</td>
<td>135</td>
</tr>
<tr>
<td>32</td>
<td>A Physical Structure of the Cell-based DOS</td>
<td>142</td>
</tr>
<tr>
<td>33</td>
<td>A Logical Structure of the Cell-based DOS</td>
<td>143</td>
</tr>
<tr>
<td>34</td>
<td>Naming of Request Identifiers</td>
<td>146</td>
</tr>
<tr>
<td>35</td>
<td>Service Redundancy</td>
<td>157</td>
</tr>
<tr>
<td>36</td>
<td>Application of Recovery Block</td>
<td>158</td>
</tr>
<tr>
<td>37</td>
<td>Application of Recovery Block</td>
<td>159</td>
</tr>
</tbody>
</table>
1.1 Distributed Computing Systems

In the past decade, the price-performance revolution in microelectronics as well as the development of interconnection and communication technology has removed the need for concentrating computing power and thus has provided a good incentive for distributed processing. This new flexibility in system design enables the functions of a complex system to be divided physically as well as logically. The trend toward the development of distributed systems is becoming more and more obvious in many applications due to the requirements of sophisticated applications, the convenience of decentralized activities, and the advantages of economical considerations.

1.1.1 Characteristics

The technical and commercial literature contains many definitions of distributed systems. One of the most widely cited definitions is given by Enslow [ENSL78]. He defined five essential characterizations of a distributed system:
- A multiplicity of dynamically assignable physical and logical resources.
- A physical distribution of such resources interacting through a communication network.
- A high-level operating system that unifies and integrates the control of the distributed resources.
- System transparency, permitting services to be requested by name only.
- Cooperative autonomy, characterizing the operation and interaction of resources.

The distributed system being considered in this study is a collection of autonomous and heterogeneous computers networked together. All these computer systems (component subsystems) are capable of solving their own problems, and may use resources of or be cooperating with other computer systems (component subsystems) in the network to perform services. The resources might be specialized peripheral devices, processors with increased capabilities, large amounts of primary memory, particular software packages not available at the local site, databases maintained in different computer systems, and so on.

1.1.2 Objectives

The objectives and the potential benefits of distributed systems include:

- High system performance

Short response time and high throughput can be achieved through the partitioning of global system functions into
tasks that each of several component subsystems can handle individually.

- high reliability and availability

If several component subsystems cooperate in a decentralized manner to perform a task without relying on a single component subsystem, it becomes possible to take full advantage of redundancy in hardware, software, and data to improve the overall reliability and availability.

- resource sharing

Physical and logical resources are optimally (and/or dynamically) allocated in different component subsystems to achieve optimal resource sharing.

- load sharing

The workload in component subsystems can be automatically and dynamically adjusted based on a system-wide policy and loading situation.

- high flexibility

Component subsystems may be added or removed from the system whenever needed without disrupting the system operation.

Proper design of a distributed system provides a very good opportunity to achieve and realize the aforementioned system objectives and advantages.

1.2 Importance of Reliability in Distributed Systems

High reliability and availability are potential advantages of a distributed system over a centralized system. However, if a distributed system is not properly designed, its reliability, availability, and even performance will be worse than those of its
centralized counterpart.

The following illustrates this concept. Assuming that an application is implemented by a centralized system, if the failure rate of this system is $f$, then the reliability of this application is $1-f$. If we implement the application in a distributed environment which consists of $n$ component subsystems and if the failure rate of each component subsystem remains the same as $f$, the reliability of the application can be quite different as shown in the following:

Best case: $\prod_{i=1}^{n} (1-f)$

Worst case: $\prod_{i=1}^{n} (1-f)$

Figure 1 shows the possible ranges of the reliability in the distributed system. The best case occurs when the system is properly designed and thus can keep an application running as long as one of the component subsystems is up. The reliability is much better than that of its centralized counterpart. The worst case occurs when the system is improperly designed so that a failure in a component subsystem causes the failure of the application. The reliability is much worse than that of its centralized counterpart.

Moreover, distributed systems also introduce a new dimension of problems such as communication failures or the inconsistent system state due to node failures or network partitions. An unreliable communication network may cause internode messages to be lost,
corrupted, out of order, arbitrarily delayed, and so on. The failures or isolations of some nodes in the system may cause the system state to be inconsistent, thereby greatly complicating the reliability problem.

1.3 Achieving High Reliability

Basically, there are two approaches to achieving reliability in computing systems: fault prevention and fault tolerance [ANDE81]. Fault prevention, also called fault intolerance, prevents system failures by ensuring that all possible causes of unreliability have been removed from the system before reliance is placed on its operation. There are two aspects of fault prevention: fault avoidance and fault removal.

Fault avoidance is concerned with design methodologies and techniques which aim to avoid the introduction of faults during the design and construction of a system. Fault removal is concerned with checking and verifying the implementation of a system and removing any faults which are therein exposed.

Fault tolerance is based on the assumption that some faults still exist or some failures may occur in the system. Incorporation of fault tolerance techniques guarantees acceptable system behavior should faults or failures occur. The techniques on fault prevention and fault tolerance are described in Subsections 1.3.1 and 1.3.2, respectively.
1.3.1 Fault Prevention

As mentioned above, fault prevention consists of two aspects: fault avoidance and fault removal.

Fault avoidance techniques are those design principles and practices whose objectives are to prevent errors from ever existing in the system. Most of the techniques focus on the individual phases in the system production as shown in Figure 2 and are concerned with the prevention of errors at each phase. These software/hardware engineering methodologies and techniques fall into the following categories:

- managing and minimizing complexity
- achieving greater precision during translation in each phase
- improving the communication of information
- detecting and removing translation errors

These methods are concerned with the detection of errors at each translation step rather than deferring the discovery of errors until program instructions or hardware modules are produced and tested.

Fault removal techniques for software mainly include the following:

- testing
  
  the process of executing a program or a part of a program with the intention or goal of finding errors.
Figure 2: System Production
- proof
  an attempt to find errors in a program without regard to
  the program's environment.

- verification
  an attempt to find errors by executing a program in a test
  or simulated environment.

- validation
  an attempt to find errors by executing a program in a
given real environment.

Fault removal techniques for hardware/firmware mainly include
emulation and field testing.

Most fault removal techniques involve detecting failures as
soon as possible after they arise. Immediate detection has two
benefits: it can minimize the effect of the error as well as the
later difficulty a human will have in take information about an
error and locating and correcting it.

1.3.2 Fault Tolerance

According to Avizienis' definition, fault tolerance is the
architectural attribute of a digital system that keeps the logic
machine doing its specified tasks when its host, or physical system,
suffers various kinds of failures of its components [AVIZ78]. Fault
tolerance techniques usually can be identified as: (1) fault
detection; (2) fault diagnosis and treatment; (3) computation
recovery; and (4) fault containment.
Fault detection is a technique to determine if a fault exists by using hardware and/or software mechanisms. Fault diagnosis and treatment is to locate, identify, and remove a fault. System reconfiguration is a typical method used in fault treatment. Computation recovery tries to repair damage caused by an existing fault and restores the system to a consistent state. Fault containment intends to prevent fault-damaged information from propagating through a system after a fault occurs but before it is detected.

Most fault tolerance techniques depend upon the effective deployment and utilization of redundancy. The introduction of redundancy into a system may affect its cost and performance. The decision to employ fault tolerance in a computer system usually involves trading off the cost of failure against the cost of implementing fault detection and recovery. The cost of failure includes both the expense of unscheduled maintenance and the losses associated with incorrect or unperformed computations.

1.3.3 Fault Prevention versus Fault Tolerance

Fault prevention could be the best strategy for software reliability because the best way to achieve reliability is to avoid creating errors in the first place. However, experience has shown that complete elimination of all possible faults in a non-trivial system is not feasible or at least not economical. Design
inadequacies and software faults might still exist in a complex computing system until the complete correctness of a system can be guaranteed. Furthermore, in a reliable distributed system, fault prevention has been found to be inadequate to deal with the aging and deterioration of hardware components. For such reasons, fault tolerance is often required if high reliability is expected. Consequently, fault tolerance and fault prevention are better regarded as complementary rather than competitive approaches to system reliability [RAND79].

In general, due to the complexity of systems and the nature of failures in distributed systems, their system-level fault tolerance technique is far more important than that in centralized systems. The fault tolerance feature, therefore, is becoming an essential attribute of a reliable distributed system.

1.4 Motivation, Objectives, and Contributions of Research

As indicated by the previous discussion, the incorporation of fault tolerance techniques into a distributed computing system is required in order to support resilient computing. Since a distributed operating system serves as a backbone of distributed computations, the design of a fault-tolerant distributed operating system (FTDOS) is an essential step toward achieving highly reliable distributed computing.

According to the conception in this study, a fault-tolerant
distributed operating system must provide recoverable atomic services to the users. The computations performed in the service must be successfully done or not done at all in spite of various failures such as node crashes, medium corruption, communication failures, and failures due to residual software design faults. Any failure in the middle of a service will not cause the total loss of the service. The service can be restarted, if it is possible, with the least amount of computation loss.

The purpose of the research as presented in this thesis has been to design a fault-tolerant distributed operating system which satisfies the aforementioned properties. The cell model, which combines three basic concepts, modular decomposition, nested atomic actions, and exception handling, is proposed as a conceptual framework to design a fault-tolerant distributed operating system.

According to the cell model, a fault-tolerant distributed operating system consists of a collection of fault-tolerant components called cells. Each cell provides functions to deal with normal activities (providing services) and exceptions (surviving failures). A user's request will be performed by a single cell or a group of cooperating cells which may reside over different nodes. The services performed in a cell are atomic actions. Theoretically, there is no restriction on the number of cells in a node.

Based on the aforementioned concept, the following issues need to be addressed:
- how to construct the cell model, and what mechanisms are required for supporting the model
- how to construct a fault-tolerant distributed operating system based on the cell model

The results of our research include a novel cell model and the mechanisms for supporting the model. These mechanisms include concurrency control algorithms, commit protocols, recovery algorithms, and communication primitives. Various design issues of a fault-tolerant distributed operating system such as naming, synchronization, communication, protection, fault tolerance, and performance are addressed. The protocol structure based on the model is also presented.

1.5 Organization of Dissertation

In this dissertation, related works such as various approaches to fault tolerance are described in Chapter 2. The target environment, where a number of nodes are connected together through a communication network, and its underlying assumptions are addressed in Chapter 3. Chapter 4 presents the cell model and a discussion of various design issues of the model. The mechanisms for supporting the model, including concurrency control, recovery, communication, and commitment, are introduced in Chapters 5 and 6. The system architecture and its proposed protocol structure are addressed in Chapter 7. The performance issues of this cell-based system and the approaches to improving system performance are
discussed in Chapter 8. Finally, a summary and future work are given in Chapter 9.
Chapter 2

RELATED WORK

In this chapter, various techniques for fault tolerance based on system structuring are discussed. These techniques include recovery blocks, conversations, N-version programming, transactions, atomic actions, and nested atomic actions.

In the past decade, researchers at the University of Newcastle-Upon-Tyne, England, have investigated extensively the recovery block based on the backward error recovery scheme to tolerate residual software design faults for centralized systems [ANDE76, HORN74, RAND75]. The concept of recovery blocks is described in Section 2.1. An extension of the recovery block to a multiprocess environment called conversation is discussed in Section 2.2 [KIM82, RAND75, RUSS79]. Another approach based on majority voting called N-version programming is described in Section 2.3.

Next, the concept of atomic actions, a generalization of the concept of transaction introduced by database designers, is described in Section 2.4. The atomic action has been widely accepted as a conceptual framework for error detection and recovery in a multiprocess environment. Finally, the refinement of the
concept of atomic actions, called nested atomic actions, to provide a better degree of recoverability and concurrency is discussed in Section 2.5. Different approaches to implementing the concept of nested atomic actions are also addressed in this section.

2.1 Recovery Blocks

The recovery block scheme for providing software fault tolerance in sequential programs was first introduced by Horning, et al., in 1974 [HORN74]. A recovery block consists of a conventional block which is provided with a means of error detection (an acceptance test) and zero or more stand-by spares (the additional alternatives). A possible syntax for recovery blocks is as follows [RAND75]:

\[
\text{<recovery block>} ::= \text{ensure <acceptance test> by} \\
\hspace{1cm} \text{<primary alternate>} \\
\hspace{1cm} \text{<other alternatives>} \\
\hspace{1cm} \text{else error} \\
\text{<primary alternate>} ::= \text{<alternate>} \\
\text{<other alternate>} ::= \text{<empty> l <other alternate>} \\
\hspace{1cm} \text{else by <alternate>} \\
\text{<alternate>} ::= \text{<statement list>} \\
\text{<acceptance test>} ::= \text{<logical expression>} 
\]

The main problem of the recovery block scheme used in a
multiprocess environment is that if the recovery points of these processes are not properly coordinated, then an intolerable long sequence of rollback propagations, called the domino effect [RAND79], can occur.

2.2 Conversation

The conversation, proposed by Randell, is a variation of the recovery block scheme for interacting processes [RAND75]. A conversation is a two-dimensional recovery block which provides a means for coordinating the recovery block structures in a multiprocess environment. It can be viewed as a recovery block which spans two or more processes and provides a means for coordinating the recovery block structures of the various processes. Randell imposed two restrictions:

- All the processes (within a conversation) must satisfy their respective acceptance tests and none may proceed until all have done so, and

- While conversations may be nested, they must not intersect.

A further investigation on conversation was reported in [RUSS79]. Russell defined conversations as synchronous and asynchronous depending on whether there is a need for synchronization at the exit of a conversation or not. He proposed two possible language constructs for specifying conversations: name-link recovery block, and multiprocess recovery block. Three
additional mechanisms for specifying the conversation schemes were proposed by Kim [KIM82]: conversation monitor, abstract data type conversation, and concurrent recovery block.

One major limitation of the conversation scheme is that the number of processes which will interact at a given time is quite often not known in advance in many applications [SHRI78]. Both recovery blocks and conversations provide static computation structures. They are usually inadequate in distributed computing environments.

2.3 N-version Programming

N-version programming is a fault tolerance scheme based upon the N-Modular Redundancy (NMR) structure: N versions of a program which have been independently designed to satisfy a common specification are executed and their results are checked based on a majority of the rest of the system [AVIZ77, CHEN78].

This technique can only be used in an environment which supports multiprocessing. In stand-alone or loosely coupling systems, application of this technique might not be very effective due to the requirement of checking or voting. Another potential problem in the N-version programming scheme is that the maximum allowable range used in a check of inexact voting may be difficult to determine and may change between executions of a version.
2.4 Atomic Actions

As described above, the recovery block scheme can only be used in a uniprocess environment. Although the conversation and N-version programming schemes can be applied to a multiprocess environment, the computation structure provided is static. A new fault tolerance scheme based on system structuring is necessary to achieve flexibility and convenience.

An abstraction called atomic action, which provides a conceptual framework for error detection and recovery in a multiprocess environment, was first formalized by Lomet [LOME77], and later expanded by Randell [RAND79] and Reed [REED79]. An atomic action is a generalization of the concept of transaction introduced by database designers. It is a computation that cannot be decomposed from the point of view of computations outside of the atomic action although it might be composed of primitive computational steps executed at different times and in different places.

An atomic action can be characterized by the following properties:

- failure atomicity

  The computations defined in an atomic action are either completely done or not done at all in spite of various failures.

- permanence
Once an atomic action commits, its result will never fail.

The atomic action has been widely accepted as a conceptual framework for error detection and recovery, especially in distributed systems. Gray [GRAY79] proposed a model to implement atomic transactions in a distributed database environment based on the following: (1) consistency locks; (2) do-undo-redo logic and the write ahead protocol; and (3) the two-phase commit protocol. An alternative to implementing an atomic action was proposed by Reed [REED78, REED79] based on a temporal environment and commit records. Lampson and Sturgis [LAMP79, LAMP81a] proposed a hierarchy of more reliable abstract machines as an approach to implementing atomic transactions. The Argus project at MIT is adding transaction constructs into the CLU language for reliability [LISK82]. Researchers at Carnegie-Mellon University also have conducted a study of designing a transaction kernel: an abstract machine that supports transactions [SPEC83]. The Clouds Project at Georgia Tech uses the concepts of actions and objects to develop a global operating system [MCKE83].

2.5 Nested Atomic Actions

Incorporating the concept of atomic actions into a distributed environment, we found the following problems:

- lack of recoverability

A distributed system consists of more than one component
subsystem. Each subsystem may have its independent failure rates and occurrences. A computation may cover more than one subsystem. Based on the property of atomic actions, if failures occur in the middle of the computation, the whole computation will be totally lost. This is unacceptable in some applications.

- lack of concurrency

A computation in a distributed environment may run on more than one computer. Due to the property of sequential processing in an atomic action, the computation cannot take advantage of this extra concurrency provided by the distributed environment.

To resolve these problems, the concept of nested atomic actions, an extension of the atomic action toward finer-grain recoverability and greater concurrency, has been proposed and it is gaining more and more attention [LIAN83a, MOSS81, REED78, SVOB83]. The nested atomic action decomposes a computation into a tree-like structure such that each tree leaf, each subtree, and the whole tree preserve the failure atomicity. The whole tree still preserves the property of permanence.

Shown in Figure 3 is a computation X based on the concept of nested atomic actions. In the computation, the tree leaves X1, X2, and X4, the subtree X3, and the whole tree X should have the property of failure atomicity. The whole tree will still have the property of permanence. Each nested atomic action in the tree is allowed to run concurrently, so a better degree of concurrency can be achieved. If there is a failure in the middle of any nested atomic action, the computation can be restarted from the beginning of that particular nested action, not restarted from the beginning.
of the whole computation; thus a higher degree of recoverability can be obtained.

Additional overhead is required to coordinate nested atomic actions and to record recovery information in the implementation of nested atomic actions. It involves the heavy use of stable storage to record computation states and extra messages required to coordinate commitments in the nested atomic action. Although the overhead is very expensive, other alternatives might be equally or even more expensive if a highly reliable system is needed.

The concept of the nested atomic action was first mentioned by Reed [REED78] in terms of possibilities and dependent possibilities. Moss [MOSS81, MOSS82] proposed a model to implement nested atomic actions based on the transaction model. Each node consists of a transaction manager which will be responsible for transaction management. The model is intended to be used in a general transaction-oriented system. Moss used the two-phase commit protocol and transaction images to guarantee the "all or nothing" property and used a two-phase locking protocol to resolve the concurrency control problem. The periodic query is used to resolve the orphan problem. However, the problem of handling duplicate requests was not addressed.

By extending our previous task model [LIU82, TSAY81, TSAY83], we propose an alternative to implementing nested atomic actions based on the following considerations [LIAN83a]:
Figure 3: Decomposition of a Computation X
generalizing the concept of transaction in a resource-sharing distributed operating system environment,

combining the concepts of virtual machines [LAMP79, LAMP81a] and structured computation [GRAY79, REED79] in the implementation of atomic actions for a general object-based environment,

allowing a dynamic restructuring of nested atomic actions to cope with node failures and network partitions,

minimizing the overhead involved in implementing nested atomic actions and improving the degree of concurrency, and

facilitating architectural support.

According to the model, a distributed operating system can be implemented based on a collection of fault-tolerant components called cells [LIAN83b]. Each cell can be viewed as a generalized server which provides predefined services for users. Each cell can handle normal activities (providing services) and exceptions (surviving failures). The cell model takes advantage of some existing and novel mechanisms to construct a fault-tolerant distributed operating system. The relationships of those mechanisms and concepts, which constitute the central part of the fault-tolerant distributed operating system, are shown in Figure 4.

An adaptive concurrency control algorithm based on synchronization constraints specified in each cell is used to resolve the concurrency control problem. According to the proposed algorithm, the semantic knowledge about typed requests and transactions is used not only to increase the degree of concurrency but also to allow for interprocess cooperation within a transaction
Figure 4: The Cell Model for a Fault-tolerant DOS
are made possible. The proposed commit protocol allows dynamic system restructuring, improves system performance in spite of failures, and provides additional flexibility and survivability. The fault tolerance property provided by the cell model not only allows a cell to survive node crashes and network partitions, but also has the provision for tolerating some residual design faults. In addition, the orphan problem is avoided in the environment by using the proposed commit protocol and the problem of duplicate requests can be easily resolved by using the proposed active request log.

The cell model relieves users of the burden of coordinating recovery actions among different modules. It also facilitates the structuring of existing atomic actions to form a new function and provides the utilities of modularity, extensibility, and reusability. A comparison between the cell model and Moss' model is shown in Figure 5.

2.6 Summary

In this chapter, various fault tolerance schemes based on system structuring such as recovery blocks, conversations, N-version programming were discussed. The static computation structures provided by these schemes are inadequate in many applications, hence the concept of atomic actions, which supports a dynamic computation structure, was proposed. The atomic action has been widely accepted
<table>
<thead>
<tr>
<th>Item</th>
<th>Moss</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Model</td>
<td>Transaction</td>
<td>Server (Task)</td>
</tr>
<tr>
<td>Synchronization (concurrency control)</td>
<td>Pessimistic approach</td>
<td>Adaptive approach</td>
</tr>
<tr>
<td>Recovery</td>
<td>Do-undo-redo logic +</td>
<td>Do-undo-redo logic +</td>
</tr>
<tr>
<td></td>
<td>Checkpointing</td>
<td>Checkpointing</td>
</tr>
<tr>
<td></td>
<td>+ Transaction Images</td>
<td>+ Active request logs</td>
</tr>
<tr>
<td>Communication</td>
<td>Synchronous</td>
<td>Quasi-asynchronous</td>
</tr>
<tr>
<td>- Duplicate request</td>
<td></td>
<td>Active request log</td>
</tr>
<tr>
<td>- Orphan processes</td>
<td>Inquiry</td>
<td>Do not exist</td>
</tr>
<tr>
<td>Commit</td>
<td>Two-phase commit</td>
<td>DPCP</td>
</tr>
<tr>
<td>Support arbitrary object structure</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Application</td>
<td>Transaction processing or databases</td>
<td>General purpose operating systems</td>
</tr>
</tbody>
</table>

Figure 5: Comparisons between Cells and Moss' Model
as a conceptual framework for error detection and recovery in a multiprocess environment.

In order to provide a better degree of concurrency and recoverability, the concept of nested atomic actions was proposed by several researchers. However, this further decomposition of an atomic action may introduce additional overhead. The cell model was proposed to implement the concept of nested atomic actions in a distributed operating system environment. This Model provides a conceptual framework to structure a fault-tolerant distributed operating system based on the concepts of modular decomposition, nested atomic actions, and exception handling. The model aims to survive anticipated failures such as node crash, medium corruption, and communication failures; to tolerate unanticipated failures such as residual design faults; to confine damages and errors; to facilitate reconfiguration; and to achieve reusability and extensibility.
Chapter 3

MODELING THE TARGET ENVIRONMENT

This chapter describes the characteristics and assumptions of the target environment. The target environment can be modeled in terms of processes, objects, and messages. The Task model, which has been proposed to model a distributed operating system, is briefly described. Also addressed are the failure model and how to model a fault-tolerant distributed operating system based on cells.

In Section 3.1, the target environment, consisting of nodes and a communication network, is described. Basic assumptions that concern nodes and communication are also addressed. The Task Model which was proposed to construct a distributed operating system is described in Section 3.2. The failure model is presented in Section 3.3, where the failures due to node, medium, communication, and residual software faults are described. Section 3.4 presents the basic concept of this research. The cell model, an extension of the previous Task model, is used to decompose a fault-tolerant distributed operating system into a group of cooperating fault-tolerant components, called cells. It aims to provide robust services for users. Finally, a summary of this chapter is given in
Section 3.5.

3.1 The Target Environment

Our target system is comprised of a number of nodes that are connected together via a communication network as shown in Figure 6. No specific topology is implied in the network. However, it is assumed that every pair of nodes in the system can communicate in both directions, though perhaps not directly. Nodes may go down (crash) and up (recover), and new nodes may be added to and old nodes may be removed from the network.

A node consists of one or more processors and some storage. Additionally, a variety of peripherals may be attached to any node. There are two types of storage in each node: stable and volatile storage. The stable storage, which survives node crashes and medium failures, can be characterized by the following properties:

- Update atomicity
  Every update in the storage should be completely done or not done at all.

- Failure resistance
  Once an update is completed, its result should survive various failures.

There are several ways to construct the stable storage; notable among them is the stable storage system proposed by Lamport and Sturgis [LAMP79]. Based on their concept, the disk storage not used as volatile storage for the processor state is converted into stable
Figure 6: The Target Environment
storage with the same operations as disk storage, but with the property that no errors can occur. The key idea is to use storage redundancy to achieve failure resistance and to use proper algorithms to guarantee update atomicity.

The concept of implementing the stable storage system has been used in reliable computing systems such as reservation and banking systems for a long time. One typical example is an international airline reservation system which the author was involved in its implementation [LIAN75]. The reservation database is stored in two identical sets of disk drives. Each update is performed one by one on each set. A data verification or cleanup procedure is performed periodically and at system restart time to make sure the data of both sets are correct and consistent. Whenever a system crash or bad writing occurs in the middle of the updating, it can always achieve the property of update atomicity and failure resistance. For instance, if a failure occurs in the middle of updating the first set, the update is undone after performing the cleanup procedure while restarting. If a failure occurs between updating the first and second set, the second set is updated after performing the cleanup procedure while restarting. If a failure occurs in the middle of updating the second set, the update at the second set is redone after performing the cleanup procedure while restarting. Therefore, the property of the stable storage system can be maintained.
Each node communicates with one another in the network via messages of variable sizes. Messages may be de-assembled into small packets and sent, then re-assembled into messages upon reception at their destinations.

3.2 Modeling The Target Environment

Nodes and the communication network constitute the central part of the environment. The communication can be modeled based on messages. The node can be modeled based on active entities (called processes) and passive entities (called objects).

A process, which is a schedulable unit for asynchronous computation, is an active entity which moves through the instructions of a procedure as the procedure is executed by a physical processor [BRIN73, DENN76]. Objects are abstract system resources, either logical or physical. In order to preserve its integrity, an object can only be accessed, be modified, or stand in relation to other objects in the way appropriate to that object [JONE78]. That is, to alter or even to determine the state of the object, an appropriate operator must be invoked. Processes in different nodes must communicate through messages. Each entity in a node may be implemented or realized on top of volatile or stable storage or both. Based on this abstraction, the target environment can be viewed as a set of processes and objects as shown in Figure 7.
Figure 7: Modeling the Target Environment
By using the Task Model [LIU82, TSA81, TSA83] to structure a distributed operating system, the entity universe is further divided into mutually exclusive sets. Each of these sets is called a task. A task consists of one or more processes and possibly some objects. Entities within a task form the address domain of that task. All the constituent entities, both processes and objects, must reside in the same physical node. A given task can assume the role of either a resource provider (called server) or a resource user (called client) or both at the same time. A process can refer directly to the objects in the address domain of its task, but can refer only indirectly to "non-private" objects (i.e., objects in other tasks) by sending messages to the appropriate task to which the objects belong.

3.3 The Failure Model

The failures in a system may include those at the component level (i.e., at the level of gates, flip-flops, registers, statements, instructions, and so on), at the module level (a bit slice, a microprogram memory, a program, and so on), at the subsystem level (where computers, communication, and peripheral elements are the building blocks), and finally, at the system level. Since many fault tolerance techniques such as coding, exception handling, and so on, have been well developed and built into existing systems especially at the component level, our research
mainly focuses on failures which exist in the system, subsystem, and module level. Such failures include:

- **Node failures**

  These occur when either a processor crashes or its main memory fails, thus temporarily ceasing normal operation. Node failures may cause the information in the volatile storage such as main memory to be lost.

- **Medium failures**

  These occur when a storage medium, such as disk storage, in the system fails or corrupts. Medium failures may cause the information stored in the medium to be lost or unrecoverable.

- **Communication failures**

  These occur when messages between nodes in the network are not transmitted properly due to broken links, hardware, or timing problems. The communication failures may cause a message being lost, duplicated, corrupted, out of order, or arbitrarily delayed. In addition, failures in communication links may result in a network partition where a whole system is partitioned into several mutually isolated subsystems as shown in Figure 8.

- **Failures due to residual software design faults**

  These occur when a system does not operate as it is intended. Most are due to design inadequacy of the system. These failures are usually difficult to anticipate, to detect, and to protect against.

According to Anderson and Lee [ANDER81], failures due to aging or physical properties of components that can be expected in advance, such as node, medium, and communication failures, are called anticipated failures; those which are difficult to predict in advance, such as residual software design faults, are called unanticipated failures.
Figure 8: Network Partitions
Protecting against all possible types of failures, in most cases, is impractical. So a first step in the system design is to specify the types of failures that will be handled properly. However, the failures that are expected to be rare or very hard to protect against cannot simply be ignored, and some mechanisms must be provided to prevent deterioration of the situation due to such failures.

With these in mind, we divide system failures into two categories: detectable and undetected. The anticipated failures such as node crashes, communication failures, and medium failures are generally detectable. Some residual software design faults also may be detectable if proper error detection mechanisms are provided. The undetected failures are those which cannot be detected by error detection schemes provided by the system; thus they cannot be recovered by the underlying recovery mechanisms. Mechanisms to achieve fault containment and to confine damages or errors due to undetected failures are required. Thus, the strategies for fault tolerance are based on the following concepts:

- to tolerate those detectable faults or failures
- to confine the propagation of the damages or errors due to undetected faults or failures
3.4 Modeling a Fault-Tolerant Distributed Operating System

Based on the concept of modular decomposition, a fault-tolerant distributed operating system is decomposed into a collection of fault-tolerant components which aim to serve requests as well as take care of various failures as shown in Figure 9. The component, called cell, can be viewed as a fault-tolerant server which incorporates fault tolerance features into the Task proposed by Tsay and Liu. Each cell has capabilities to provide normal services as well as to handle exceptions. Two types of exceptions are addressed: interface and failure. The interface exception will be raised when a request is submitted for certain services, but unfortunately rejected due to protection or syntactical errors. The failure exception will be raised when a request is aborted due to system failures such as node failures, medium corruption, communication failures, and so on.

The service provided in each cell has the atomic property. A user request may be performed by a single cell, or by a group of cooperating cells which implement the concept of nested atomic actions. Structure and various design issues of the cell model and how to construct a fault-tolerant distributed operating system are addressed in the following chapters.
Figure 9: Modeling a Fault-tolerant DOS
3.5 Summary

Our target environment is a collection of nodes, where each node consists of stable and volatile storages and communicates with others via a communication network. The stable storage is safe in the sense that it can survive node crashes or medium failures. Inter-node communication is through message passing.

Failures in the system are divided into undetected and detectable. Detectable failures can be further divided into unanticipated and anticipated. Our target system aims to survive anticipated failures such as node or communication failures and to have provisions to tolerate unanticipated failures such as those due to residual software design faults. Mechanisms to confine errors and damages due to undetected and detectable failures are also required.

The Task model has been proposed to model a distributed operating system called MIKE [LIU82, TSAY81, TSAY83]. A task is a group of processes and objects which perform services for users or realize a data type in the system. The cell model, an extension of the Task model, adds fault tolerance features to the Task in a distributed operating system environment. The cell model decomposes a fault-tolerant distributed operating system into a group of cooperating fault-tolerant components, called cells, which reliably communicate with one another and cooperatively perform services for users. A cell can be viewed as a fault-tolerant server or a fault-
tolerant task. The structure and various design issues of cells are addressed in the next chapter.
Chapter 4

CELLS

This chapter describes a fundamental fault-tolerant component, called Cell, in a fault-tolerant distributed operating system. A cell is a group of processes and objects which collectively implements a data type or provides certain services in the system. The computations defined in a cell are recoverable atomic actions. The structure and type of cells and various design issues of the cell model are addressed in this chapter.

In Section 4.1, the structure of a cell is presented. A cell consists of a unique process or process group called manager, its child processes called agents, and some protected objects. All constituents of a cell must reside in the same physical node. The relationships among a group of cooperating cells, and the group of corresponding agents in each cell, which forms a tree-like structure called the agent tree, are described in Section 4.2. This group of cooperating cells, which cooperatively performs services for users, implements the concept of nested atomic actions. Section 4.3 introduces two types of cells: type cell and service cell. The type cell implements a data type in the system. The service cell
provides predefined services for users. An example to explain how a group of cells cooperatively performs file transfer service is given in Section 4.4. Various design issues of the cell model including concurrency control, recovery, communication, commitment, and protection are addressed in Sections 4.6, 4.7, 4.8, 4.9, and 4.10, respectively. Finally, a summary of this chapter is given in Section 4.11.

4.1 Structure

A cell is a group of processes and objects which collectively implements a data type or provides certain services. The structure of a cell includes a unique cyclic (non-stop) process or process group called manager, zero or more descendant processes, which perform services on behalf of users, called agents, and protected objects. All the constituent entities in a cell, both processes and objects, form the address domain of that cell and must reside in the same physical node.

The functions of the manager in a cell include the following:

- Protection of local software and objects:

  Only authorized users are permitted to request services and to access objects in a cell.

- Scheduling of agent dispatching:

  Valid requests will be served by appropriate agents in a cell. They will be synchronized and scheduled by the cell manager.
- Manipulation of recovery information:

The recovery information will be manipulated by the cell manager to restore the correct object and cell state when failures occur.

- Management of inter-cell communication:

Inter-cell communication is trapped by the manager to monitor the progress of intercell interactions.

- Coordination of committed actions:

The cell manager is responsible for maintaining the atomic property while a group of cells cooperatively perform a service.

The manager will run, once the cell is created, until the cell is terminated at the request of its owner (either system or users).

An agent, a child process of the manager, will be created once a cell manager receives a valid request from a user. The "valid" request is used to refer to a request which is submitted by an authorized user and when the parameters of the request are acceptable to the cell. An agent will perform a service based on the input parameters of the request and on behalf of the user. It can be viewed as a worker process which provides services for users under the supervision of its manager. An agent is terminated by the manager once the service it provides is successfully completed or unfortunately aborted. Therefore, a cell can be viewed as a dynamic two-level process tree, with the manager as the root node as shown in Figure 10.

Direct communication among agents within a cell is prohibited, however, the communication through the manager is allowed. An agent
Figure 10: Process Structure in a Cell
can communicate with other cells through the manager based on intercell communication primitives.

There are three possibilities to implement a cell in a computing system. The considerations usually depend upon the characteristics of the environment. They include:

- **simplified case:**

  A cell is simply implemented by a single process or process group, the manager. There is no internal concurrency within a cell, and all incoming requests will be performed by the manager on a first come first served basis.

- **normal case:**

  A cell is implemented by a group of processes. Among them are, a unique process or process group called manager, and a number of child processes to perform services on behalf of users. There is internal concurrency within a cell. However, the maximum number of active agents in a cell is limited.

- **ideal case:**

  A unique process or process group called manager, an arbitrary number of child processes and some protected objects form a cell. The number of agents depends upon the number of incoming requests. There is no limitation on the number of active agents in a cell.

### 4.2 Cooperating Cells

As we mentioned in Section 4.1, a cell consists of the manager, agents, and protected objects. The computation which an agent performs is an atomic action. An agent can issue a sub-request to another cell for a service which will result in creating another
atomic action (agent) in that cell. This is a typical example of an inter-cell interaction. To extend the interactions to several levels, a computation which is invoked by an external request can be logically viewed as a tree-like structure consisting of multi-level nested atomic actions (agents). Figure 11 is an example of a group of cooperating cells performing a service for a user. Agent A1 asks cells B and C, on its behalf, to create child processes (agents) B1 and C1, respectively, to perform some functions. Similarly, agent B1 asks cells D and E to create child processes (agents) D1 and E1, respectively, to perform parts of its function. In the same example, cell A is also called a top-level cell from the viewpoint of the external request, and cells B, C, D and E are called nested cells. These cells form a reliable virtual machine which provides a service for the user. Figure 12 shows a tree, called the agent tree, which is a set of agents cooperatively performing a computation for an external request as shown in Figure 11. Each node in the tree is a corresponding agent in a cell. Each dotted circle in Figure 12 represents an atomic action which forms a recovery line. This group of cooperating cells implements the concept of nested atomic actions.

There are two types of requests in the model: top-level requests and nested requests. Top-level requests are those which are received by the top-level cells. Nested requests are those which are subsequently created by top-level requests or other nested
Figure 11: Cooperating Cells
Figure 12: The Agent Tree for Cooperating Cells
requests. The top-level request is also called a transaction. The nested requests submitted directly or indirectly by the same top-level request have the same transaction ID but different request IDs. Based on this definition, a transaction may contain more than one-level and more than one nested request (subrequest). The request to cell A in Figure 12 is a top-level request and the others are nested requests, all of which belong to the same transaction.

4.3 Classification

Cells are classified into two categories. Cells in each category form the same kind of binding forces. The first kind of binding force results from those component entities of a cell which collectively realize one of the data types in the system, that is, the type manager (the cell manager) together with the objects of that type in that physical node forms a cell. Since operations on a particular type of object are possible only through the invocations to its type manager, the process initiated by the type manager to perform the well-defined operations on the objects also belongs to the same cell. Cells in this category are referred to as type cells. The cell manager serves as a guardian of its protected objects. Directory, file, printer, tape, etc., are typical types. Every typed object in a node is guarded by a type cell.

The second kind of binding force results from those component entities of a cell which harmoniously work together to provide a
certain distributed service as one of the DOS utilities. Cells in this category are called service cells. The cell manager serves as a service provider. A typical example of the service cell is a file transfer server.

4.4 Examples

To explain how a group of cells cooperatively perform services, the following example for file transfer is given. In this example, several cells are cooperatively performing the services of file transfer for users as shown below:

- file_transfer_cell
  It coordinates with other file_cells to perform the services of file transfer.

- file_cell
  It performs local read or write operations on files.

A request for transferring a file from X to Y is performed by the cooperation of one file_transfer_cell and two file_cells as shown in Figure 13. In this example, a user request is accepted by the file_transfer_cell at the requesting node, and the file_transfer_cell coordinates with two file_cells at source node X and at destination node Y, respectively. The functions performed in each cell can be defined as follows:

- File_transfer_cell:
1. send a request to the file Cell at X to read a source file.

   when an exception is raised: If it has been retried \( n \) times then quit else retry;

2. send a request to the file Cell at Y to write a destination file.

   when an exception is raised: If it has been retried \( n \) times then quit else retry;

- File Cell at source node:

1. read the source file and send its content back.

- File Cell at destination node:

1. write the destination file based on the file content.

The file transfer Cell in the example is a service cell which serves as a coordinator of its lower level operations: read and write on file Cells. The file Cells are type Cells which serve as guardians of their protected source and destination files.

4.5 Design Issues

Some design problems must be resolved when implementing the cell model. In the previous example, if there are more than one incoming request to the file Cell, a synchronization or concurrency control algorithm is required to schedule requests and to synchronize concurrent accesses to a shared file in the cell. If a cell such as the file transfer Cell crashes in the middle of the service due to node failures or other reasons, a recovery algorithm
Figure 13: An Example for File Transfer
is required to correctly restore the cell state. In order to facilitate communication between a client and a server cell (such as the file_transfer_cell and the file_cell in this case), communication primitives are required. In order to guarantee the "all or nothing" property, the coordination among the file_transfer_cell and the file_cells to correctly and consistently provide the service of file transfer for users is required. In addition, to prevent unauthorized users from illegally requesting services and accessing objects, some access control mechanisms need to be embedded in each cell. The aforementioned design problems can be summarized as follows:

- Synchronization or concurrency control problem:
  How to control concurrent accesses to shared objects in a cell; and how to achieve better concurrency without causing the domino effect?

- Recovery problem:
  How to correctly, economically, and rapidly restore the cell state after encountering failures?

- Communication problem:
  How to implement reliable communication among cooperating cells; and what primitives should be provided?

- Commitment problem:
  How to coordinate the committed actions among a group of cooperating cells to achieve the atomic property?

- Protection problem:
  How to protect the information from being illegally accessed or illegally utilized?
These problems are examined more closely in the following sections.

4.6 Synchronization or Concurrency Control

The following example demonstrates advantages and disadvantages of existing synchronization and concurrency control algorithms. We assume that there are two user requests, X and Y, for distributed services, both of which need to access a shared object M as shown in Figure 14. Both of the computations performed by requests X and Y are implemented in terms of nested atomic actions. Therefore, each forms a tree structure of computation as shown in the figure. Agent X2 intends to modify object M prior to agent Y1. The synchronization algorithm provided by the system is responsible for synchronizing and controlling the concurrent accesses of agents X2 and Y1 to object M. We have used this example to explain how to apply existing techniques to resolve the synchronization problem and to demonstrate the advantages and disadvantages of these techniques in the following.

TRADITIONAL SYNCHRONIZATION MECHANISMS

Direct applications of traditional synchronization mechanisms used in centralized systems such as semaphores, monitors [HOAR74], path expressions [HABE75], serializers [ATKI77], and selectors [LEIN82] to this environment may cause lost updates or intolerable
Figure 14: A Synchronization Problem
cascading rollbacks. The reason is that these mechanisms aim to resolve the problem of mutual exclusion and they do not take a group of cooperating nested actions into consideration. For example, in Figure 14, after agent X2 updates object M, traditional synchronization algorithms will allow agent Y1 to update object M. If request X is aborted later, the undoing of request X will also undo the update result of agent Y1. Therefore, the effect of the modification due to request Y1 is gone after the undoing and the problem of lost update may occur. In addition, if these algorithms are augmented to keep track of the dependency relationships among conflicting requests, the abortion of request X will cause the undoing of request Y. If several conflicting requests are concurrently accessing a shared object over a period of time, intolerable cascading rollbacks may occur.

LOCKING AND TIMESTAMPING

Concurrency control algorithms used in distributed database systems such as timestamping [SHAP77, REED78, BERN80], pessimistic locking [ESWA76, GRAY79, PAPA79], and optimistic methods [KUNG79, KUNG81, SCHL81] can be used in the environment. In the simple timestamping approach, all requests in the system are timestamped (Ts). Each object (X) in the system is associated with read and write timestamps (R-ts(X), and W-ts(X), respectively). The approach can be summarized as follows:

Read: if Ts < W-ts(X)
Then abort the request

else perform the request, and
\[ R-ts(X) \leftarrow \text{Max}(R-ts(X), Ts) \]

Write: if \( Ts < R-ts(X) \) or \( Ts < W-ts(X) \)

then abort the request

else perform the request, and
\[ W-ts(X) \leftarrow Ts \]

In other words, an object can be accessed only by timestamp order; otherwise they will be aborted and restarted later. This abortion becomes more serious in the environment supporting nested atomic actions since it has higher probability of aborting a request. Although the use of multiversion timestamping approach [REED78] may alleviate the restarting problem, the problem is not completely resolved. Moreover, it introduces another problem of manipulating multiple versions. In general, the distributed nature of our target system may cause intolerable request restarting if we adopt a traditional timestamping approach, especially when the system is heavily loaded and the contention rate of objects is high.

The locking method needs to acquire a lock before an agent can access an object. The locks an agent holds eventually will be released so that other agents may access the object. The grant or denial of a lock is based on the locking protocol.

The pessimistic locking is known as the two phase locking [ESWA76, GRAY79, MOSS81, LISK82]. The basic concept of this
approach is to delay other conflicting requests' accesses to the shared object until the computation of the previous access has been totally committed. This postponement of accesses avoids the possibility of lost updates and intolerable cascading aborts. In the example of Figure 14, agent X2 needs to acquire a lock of object M before accessing it. Once agent X2 finishes the update of object M, the lock will be retained until the total commitment of computation X. Therefore, when agent Y1 intends to acquire the lock of object M, the locking protocol will force agent Y1 to wait until the lock is released by X (that is, when computation X has committed). The two phase locking guarantees the requirement of serialization among conflicting requests. The delay of other conflicting requests to access a shared object avoids the domino effect. However, the degree of concurrency is restricted due to the retention of locks until the total commitment.

To improve the degree of concurrency in this environment, the optimistic method [KUNG79, KUNG81, SCHL81] can be applied. The basic concept of this approach is to allow other conflicting requests to access a shared object by keeping track of the dependency relationships among these conflicting requests, which access a shared object, as the commit condition. In the optimistic method, once agent X2 finishes the update of object M, the lock will be released. Therefore, agent Y1 is permitted to access object M. However, it also records a dependency relation X2 > Y1 (This
means that the commitment of Y1 depends on the commitment of X2) as the commit condition. Agent Y1 can only commit when agent X2 has committed. Whenever agent X2 is aborted, agent Y1 should also be aborted. Although the optimistic method may improve the degree of concurrency, it may cause intolerable cascading aborts if the likelihood of request conflict is high.

Many researchers have observed that the use of semantic knowledge can improve the degree of concurrency [ALLC83, ESWA76, GARC82, KUNG79, SHA83]. Garcia and Sha [GARC82, SHA83] consider the properties of the transaction while Spector and Schwartz [SCHW82, SPEC83] are concentrating on the semantics of operations in individual types. Several researchers have worked on an adaptive approach [BADA83, LIU84]. Badal proposed an adaptive concurrency control algorithm whereby it can switch between using short and long duration locks on low and high contention files. Liu, Sheth, and Singhal discussed an adaptive concurrency control strategy between single and multiple controllers by taking variations in topology and delay characteristics of the network into consideration.

In this dissertation, we propose an alternative solution to the concurrency control problem in a fault-tolerant network operating system environment supporting nested atomic actions. It combines the advantages of using an adaptive approach and semantic knowledge. The proposed algorithm is based on the concepts of timestamping,
type-specific locking, and an adaptive approach. It can resolve the request restarting problem caused by the timestamping methods, can provide better concurrency than the pessimistic locking method, and can avoid intolerable cascading aborts caused by the optimistic locking on high contention objects. In addition, interprocess cooperation within a transaction is made possible by using the algorithm. The proposed concurrency control algorithm is described in Chapter 5.

4.7 Recovery

Three aspects of the recovery problem need to be dealt with: restoring an object state, restoring a cell, and restarting a crashed node. The schemes to deal with computation recovery are usually classified into backward or forward recovery techniques [RAND79].

Backward recovery separates the problems of error diagnosis and damage assessment from those of how to continue to provide the specified service. Usually, it can be easily applied to a general purpose environment.

The forward recovery technique is to a much greater extent dependent on having identified the fault, or at least all its consequences. Error diagnosis and damage assessment become an important factor in the construction of forward recovery schemes. If the faults and their ensuing damage are simple, forward recovery
can be both simple and much more efficient than backward recovery. In a general purpose non-trivial system, these tasks are not very easy and are usually time consuming. Therefore, forward recovery mechanisms are generally used in a special purpose environment; while most recovery mechanisms in a general purpose environment are based on the concept of backward recovery. Moreover, if failures occur in the process of recovery, computation recovery becomes more complex if a forward recovery is used. The main reason is that the damage assessment is required in the forward recovery whenever a recovery is initiated. However, the procedure is quite trivial if a backward recovery is adopted. The undoing and redoing of the recovery process can simply resolve the problem of cyclic crashing mentioned above.

Exception handling is an example of the forward recovery. Typical examples of the backward recovery include recovery blocks, conversation, checkpointing, do-undo-redo logic, and so on.

Checkpointing is a backward recovery technique used very often in a centralized operating system to increase its recoverability. It can be implemented by snapshoting the system state periodically. The system can be restored to its previous state based on the checkpoint information when failures occur. Due to the decentralized nature of the target environment, it is very difficult to checkpoint every node in the system simultaneously. However, combining checkpointing and other techniques can still be used to
speed up the restart process and to provide recoverability in a distributed environment. The checkpoint interval is usually decided based on the average mean time before failure (MTBF) and the requirements of system reliability.

The concept of do-undo-redo logic has been very popular in reliable database systems [GRAY79]. The images of a record before and after an update are saved to guarantee the recoverability of a data record. Transaction journals, audit trails, undo/redo logs, and so on, are variations of this concept. The concept of do-undo-redo logic can be applied to a cell. The undo log records the before image of an accessing object whereas the redo log records the after image of that object. The implementation of the do-undo-redo logic can help us achieve the atomic property.

Our design goal is to devise a fast, correct, and reliable recovery algorithm which can deal with the aforementioned three aspects of recovery. This algorithm should be able to survive node crashes during its recovery procedure. The recovery algorithm is described in Chapter 5.

4.8 Communication

The purpose of intercell communication is to provide a high-level reliable and efficient communication mechanism in the cell model. The intercell communication can be described in terms of the client/server model as shown in Figure 15. In the model, two types
of communication can be used: asynchronous and synchronous. Message passing is a typical example of asynchronous communication. In this case, after a client sends a message (request) to a server, it is continuously pursuing other processing. The outcome of the request could be one of the following possibilities:

- the request is correctly received and performed by the server.
- the request is received by the server, but rejected due to protection or syntactical errors of the request format.
- the request is lost, and it never reaches the server.

In asynchronous communication, both the client and server can process computation simultaneously, so a higher degree of internal concurrency can be expected. However, the client must use other facilities or methods to check the result of the request.

The remote procedure call is a typical example of synchronous communication. It resembles an ordering procedure call: the invoking process is blocked until either a response message is received, or a failure exception is raised. When a remote procedure call is executed, a client process (the program which submits the remote procedure call) sends a request to the server process (the called program) and waits until its reply from the server has been received. This send_and_wait semantics establishes a simple and reliable communication between two processes. The problems related to reliable remote procedure calls have been discussed in [LAMP81b, NELS81, SHRI82].
In a remote procedure call, the client will wait until a reply from the server has been received. However, in a decentralized environment, waiting may be unnecessary if the result of the remote call is not immediately needed. A slight modification of this semantics may provide a better performance, which is described in Chapter 6.

4.9 Commitment

In the environment of cooperating cells, the computation performed by this group of cells forms a tree-like structure called the agent tree. The computation in each node of the agent tree is an atomic action. In order to guarantee the atomic property of the computation, a commit protocol is required to coordinate the commit actions of all agents participating in a request.

The most well known commit protocol is the two-phase commit protocol and its variations [GRAY79]. The traditional two-phase commit protocol can be applied to our environment to resolve this distributed commit problem in the following way: at prepare phase, when a top-level cell is ready to commit, it broadcasts an inquiry to all participating cells and asks "are you ready?". At commit phase, the top-level cell broadcasts a "commit" message to all participating cells if it receives positive acknowledgments from all participating cells in time; otherwise, it broadcasts an "abort" message to all participating cells to abort the whole computation.
Figure 15: The Client/Server Model
This commit protocol can guarantee the "all or nothing" property among this group of cooperating nested atomic actions in spite of node failures.

However, the requirement of "total commitment at once" in this protocol does not allow the commitment of the whole computation if one or several nodes, where some nested atomic actions were running, crashed after finishing their computations but before the total commitment. The commit action, therefore, will be delayed or aborted.

An alternative commit protocol which is more suitable for this tree-like structure and provides better performance is proposed in this study. The commit protocol guarantees the atomic property of the computation in spite of various failures such as node crashes or communication failures. The proposed commit protocol is described in Chapter 6.

4.10 Protection

In order to protect against unauthorized accesses into the programs and objects in a cell, incorporation of access control mechanisms into a cell is necessary. The protection mechanism is based on the concept of the close environment, the least privilege principle, and the protected subsystem [BERS80, COOK79] [DENN76, LIND76, POPE74, RATT80, RATT81, SALT75]. A cell is a collection of programs and objects that is encapsulated so that
other executing programs cannot access them but can only invoke them through designated entry points. This tight control of access rights protects against unauthorized accesses, prevents information from inadvertently escaping, provides a small access domain, and limits the information flow between cells.

4.11 Summary

In this chapter, we have described the basic concepts of cell structure and its classification. A cell is a two-level process structure with the manager as its root node. The processes and objects of a cell must reside at the same physical node. There are two categories of cells: type and service cells. The type cell abstracts a type of object and implements the concept of data abstraction. The service cell abstracts a type of service and implements the concept of control abstraction. The type and service cells can cooperatively perform various services for users. The computation defined in this group of cells is implemented based on the concept of nested atomic actions.

Design issues of the cell model such as concurrency control, recovery, communication, commitment, and protection were also discussed. In addition, the applications of existing techniques to the cell model and their advantages and disadvantages were also addressed. The detailed algorithms of the proposed synchronization, recovery, communication, and commitment are described in the
following chapters.
This chapter describes the concurrency control and recovery algorithms which are proposed for the cell model. As mentioned in the previous chapter, existing algorithms such as two-phase locking and timestamping can be applied to the model. However, the requirement of serialization and limited use of the semantic knowledge restrict interprocess cooperation and may reduce the degree of concurrency. The optimistic method may improve the degree of concurrency in certain applications; however, it may cause unnecessary abortions of requests if the contention rate of objects is high. Some efforts have been made on the use of semantic knowledge. Taking advantage of the semantic knowledge about typed requests and transactions, a concurrency control algorithm is proposed in this chapter to improve its concurrency and to avoid causing intolerable cascading rollbacks. The algorithm also allows for interprocess cooperation within a transaction.

The recovery of a cell is based on the following concepts and facilities: checkpointing, do-undo-redo logic, an active request
log, and a robust recovery algorithm. The active request log is a special facility associated with each cell. The purpose of the active request log is to maintain a consistent view of request status among related cells in the system. The recovery algorithm can survive node crash in the middle of a recovery process.

Section 5.1 introduces the transaction model, describes terminology and basic assumptions, and presents the concurrency control algorithm. The algorithm is based on the concepts of timestamping, type-specific locking, and an adaptive approach. The semantic knowledge about typed requests and transactions is used. The deadlock problem is resolved based on detection. Analysis of this algorithm is presented in Section 5.2, in terms of correctness, deadlock, cascading rollbacks, performance, and robustness. Section 5.3 presents a set of recovery algorithms to undo aborted requests and to restore a crashed cell. They rely on the following concepts and facilities: checkpointing, do-undo-redo logic, and the active request log. The correctness and robustness of these algorithms are addressed in Section 5.4. Finally, a summary of this chapter is given in Section 5.5.
5.1 Concurrency Control Algorithm

Before introducing the proposed algorithm, the transaction model and some terminology and assumptions are presented in the following.

5.1.1 Transaction Model

As mentioned in the previous chapter, a user request in a distributed operating system can be performed by a group of cooperating cells as shown in Figure 16. A user request can be viewed as a transaction and subrequests to each cell which are called nested requests. Once a cell receives a request it creates an agent (a nested atomic action) in that cell to perform the service. The request and its corresponding agent (nested atomic action) have the same identifier. Therefore, a transaction may logically consist of more than one nested request (nested atomic action) which form a tree-like structure. The cell which receives the request from the user is called a top-level cell and its request is called a top-level request. Each request in the system is associated with one transaction only.
Figure 16: The Transaction Model
5.1.2 Terminology

Several terms and algorithms such as local commit, global commit, active request, active transaction, and compatible and conflicting requests are defined in this subsection.

A request (or nested atomic action) is said to be locally committed if the service it requested in a cell has been successfully performed. A request is said to be globally committed if the transaction associated with it has committed. When a top-level request (top-level nested atomic action) commits, the transaction commits. Figure 17 shows a life span of a request in a cell.

A request is said to be active in a cell if it has been received and validated by the cell and until the transaction, with which it is associated, has committed. A request is said to be active in an object from its access to the object until it has globally committed. A transaction is said to be active in a cell if any of its nested requests in that cell are still active. A transaction is said to be active in an object if any of its nested requests in that object are still active or have not globally committed. A transaction may be active in a cell, but inactive in other cells.

The relationships among requests in a resource-sharing environment can be classified as competitive and cooperative. Usually, requests in the same transaction cooperatively manipulate
Figure 17: The Life Span of a Request in a Cell
objects while requests in different transactions competitively access objects. The dependency relationship of computations among nested requests in a transaction is usually predictable at the design stage. However, the dependency relationship of computations among nested requests in different transactions is usually unpredictable and depends upon the current system loading.

There are two classes of requests defined in the system: compatible and conflicting requests. The compatible requests are those which are permitted to simultaneously access an object. For instance, read requests are compatible requests. The conflicting requests are those which are not permitted to simultaneously access an object such as read and write requests or write and write requests.

5.1.3 Assumptions

Before presenting the concurrency control algorithm, basic assumptions of the environment are described in the following:

- Loosely synchronized timestamps

Each node in the network maintains an eventcount, a monotonically increasing number, which is loosely synchronized among the nodes. A timestamp is a unique number which can be generated by concatenating the local eventcount with the pre-assigned node number.

- Timestamped requests

Each request in the system has a unique identifier which consists of two parts: (a) timestamp and (b) generation number. The generation number represents a hierarchical
relationship among a tree of nested requests in the transaction. The top-level request has a generation number zero. All requests in a transaction have the same timestamp and different generation numbers. The nested atomic action, created in each cell by a request, has the same identifier as the request.

- Timestamped objects

Each object in the system is associated with a group of timestamps inherited from the active requests accessing it. An object may have more than one timestamp if there is more than one compatible request (such as read requests) simultaneously accessing it. A timestamp of an object will be removed when a transaction, which has the same timestamp, has globally committed.

- Multiple versions

Each object may have multiple versions. Updates to an object are recorded in a new temporary version that becomes permanent only if the transaction has committed. In case the updates are to be undone (i.e., aborted), the temporary version is discarded.

5.1.4 Basic Concept

As mentioned above, a concurrency control algorithm is required in a cell to schedule incoming requests and synchronize concurrent accesses to a shared object. Let us look at an example as shown in Figure 14. Computations X and Y are implemented based on the concept of nested atomic actions as shown in the tree structures of the figure. In this example, nested atomic action X2 updates object M before nested atomic actions X1 and Y1. If X2 is aborted later for some reason, the undoing of X2 will result in the undoing of the effect of nested atomic actions X1 and Y1 on object M, and the
problem of lost updates will occur. In order to resolve this problem, one of the following approaches can be taken:

- pessimistic approach

In this approach, if a request is still active in an object, we prohibit any conflicting requests of other nested atomic actions from accessing the same object. As shown in the previous example, once X2 accesses object M, no conflicting requests from other nested atomic actions such as X1 and Y1 are allowed to access object M. However, X1 is permitted to access object M only if X2 has locally committed and the lock of object M from X2 has been propagated to X0 while Y1 is permitted to access object M only if the whole tree of computation X has globally committed. This is a conservative approach to guarantee the consistency of objects. It intends to resolve the consistency problem at the cost of degrading the degree of concurrency. The two phase locking is a typical example of this pessimistic approach.

- optimistic approach

In this approach, if there are active requests in an object, we allow other conflicting requests to access the same object as long as we keep track of their dependency relations as commit conditions. Once a request intends to commit, it can do if the request successfully satisfies its commit conditions; otherwise, it will abort. In the previous example, once X2 accesses object M, it allows conflicting requests from other nested atomic actions (such as X1 or Y1) to access object M as long as X1 or Y1 records the dependency relationship X2 > X1 (i.e., X1 can commit only when X2 has committed) or X2 > Y1 (i.e., Y1 can commit only when X2 has committed) as its commit condition. However, if Y aborts, computation X will be forced to be undone. This is an aggressive approach. It intends to improve the degree of concurrency but may cause an intolerable cascading aborts if many conflicting requests intend to simultaneously access a shared object. The optimistic concurrency control is an example.

- adaptive approach

In this approach, we allow a fixed number of conflicting requests to access a shared object in an overlap period.
If the number of conflicting requests which simultaneously access the object equals the predefined number, the concurrency control mechanism will be changed from the optimistic approach to the pessimistic approach. In the previous example, if we set the maximum number of conflicting requests to be 1, once X2 accesses the object, it allows one more conflicting request from other nested atomic action (such as Y1) to simultaneously access the object. Y1 will also record X2 > Y1 as its commit condition. If conflicting requests from other nested atomic actions (such as X1) come, they will be forced to wait until both X2 and Y commit and the lock, which X2 held, has been propagated to X0. However, if Y aborts, computation X will be forced to be undone. This is a hybrid approach between the pessimistic and optimistic approaches. Badal's adaptive method is an example of this approach [BADA83].

- adaptive approach based on semantic knowledge

In this approach, the requests from the first transaction are manipulated based on the optimistic approach, but the requests from other transactions are handled based on the pessimistic approach. In the previous example, once X2 accesses the object, it only allows other requests from transaction X (such as the request from X1) to access the object. The conflicting requests from other transactions (such as Y) must wait until X has committed; therefore, the problem of cascading aborts of transactions is eliminated. This is a hybrid approach between the pessimistic and optimistic approaches, using the semantic knowledge of transactions as a criterion.

Our concurrency control algorithm is based on the adaptive approach using semantic knowledge. In addition, we extend two modes of locks (read/sharable and write/exclusive) to type-specific locking as proposed by Schwartz and Spector [SCHW82, SPEC83]. The detailed algorithm is described in the following.
5.1.5 Algorithm

According to the proposed concurrency control algorithm, any access to an object can be divided into three primary steps:

- acquiring a lock

An agent should ask for a lock from its manager before performing any access to the object. A lock is granted according to the locking rules which are described later. Once the manager grants a lock to an agent, it should also record the timestamp of the agent into that of the object if the object does not have this timestamp.

- resource accessing based on mutual exclusion

Traditional synchronization algorithms such as monitors, serializers, or selectors can be used to resolve the problem of mutual exclusion and to allow the agents to physically access objects.

- releasing a lock

A lock held by a request is released once the request globally commits. The timestamps of the object with which the lock is associated will be appropriately updated.

The locks are managed distributedly by each cell. The modes of locks for each object are granted depending on the types of the requests permitted to operate on that object. For instance, in a "directory" object, we allow the following four typed requests to be performed: read, insert, delete, and modify. Therefore, there are four modes of locks for the directory object: read, insert, delete, and modify.

Each object might have more than one timestamp depending on the number of active transactions on that object. Once a transaction
commits, the lock of that object will be released, and the corresponding timestamp of that object will be removed.

The locking rules in the concurrency control algorithm can be summarized as follows:

1. A lock will be granted based on the following conditions:

   - For an object with a single timestamp the same as that of the incoming request, a lock with an appropriate mode will be granted for either a compatible or a conflicting request. However, a new commit condition will be created and saved in the stable storage for a conflicting request if it will access a temporary version. The dependency relationship will be recorded as $X > Y$, where $X$ is the timestamp of request which created the last temporary version and $Y$ is the timestamp of the current request.

   - Otherwise, a lock with an appropriate mode will be granted only for a compatible request.

2. Once a lock is granted for any request of a new transaction, its timestamp will be recorded and stored in the object.

3. Once a nested atomic action intends to commit and if it is a common ancestor of the two parties in the commit conditions created by itself or inherited from its descendant, those commit conditions will be checked. If it fails to meet these conditions, the nested atomic action will be aborted and the request IDs and commit conditions will be abandoned. However, if it satisfies the conditions, the nested atomic action will locally commit and the commit conditions just checked will be abandoned and will not be propagated to its immediate ancestor.

4. Once a nested atomic action commits, in addition to the result, its ID (called the commit ID), those commit IDs from its descendant, and the commit conditions including those created in the nested atomic action or inherited from its descendant (except those just being checked)
will be propagated to its immediate ancestor (its client).

5. Once a transaction commits (i.e., its nested requests or nested atomic actions globally commit), the timestamp of the objects associated with it will be removed and the temporary versions which it created will be changed into permanent versions.

In order to explain the locking rules, an example is shown in Figure 18. A transaction is cooperatively performed by three cells A, B, and C, as shown in Figure 18.a. Their relationships and the corresponding agent tree are shown in Figure 18.b. Agents (nested atomic actions) X11, X12, and X2 are located in cell C where they are subsequently updating object M as shown in Figure 19. We assume that agent X11 updates object M first, agent X12 next, and agent X2 last. According to the locking rules, the commit IDs and commit conditions at each step are also shown in Figure 19.

A commit protocol is proposed to coordinate the commit actions among this tree-like structure of computation [LIAN83a]. The commit protocol consists of two phases, local and global commitment. The local commitment describes the commit procedure for non-top-level nested atomic actions. When a non-top-level nested atomic action commits, the commit IDs and conditions as well as its result will be propagated to its immediate ancestor as described above. The locks it held are kept in the stable storage of that cell. The global commitment describes the commit procedure for a top-level nested atomic action. When a top-level nested atomic action commits (i.e., the transaction commits), a global commit message will be
periodically sent to its nested cells until the acknowledgments from the nested cells have been received. Once a nested cell receives a global commit message, the locks it held will be released and the timestamp of accessed objects will be updated. This continuing sending of global commit messages until confirmation resolves the problem of communication failures between them and node failures of nested cells after local commitment. Detailed discussions are described in the next chapter.

Requests are said to be compatible or conflicting depending upon a set of synchronization constraints specified in each cell. The constraints specify the compatibility of typed requests for an object. Any typed request which meets the synchronization constraints is viewed a compatible request on that object; otherwise, it is viewed as a conflicting request. The following is an example of synchronization constraints for an object which has only two typed requests: read and write. The synchronization constraints can be shown as follows:

\[ 0 \leq \text{read.cnt} \quad \text{and} \quad 0 = \text{write.cnt} \]

or

\[ 0 = \text{read.cnt} \quad \text{and} \quad 0 \leq \text{write.cnt} \leq 1 \]

Where \( \text{type_request.cnt} \) indicates the number of active type_requests in the object.

This set of synchronization constraints defines conflicting requests as follows:
Figure 18: An Example for Concurrency Control
Figure 19: Commit IDs and Commit Conditions
- if there is an active read in the object, all incoming write requests are viewed as conflicting requests.

- if there is an active write in the object, all incoming read and write requests are viewed as conflicting requests.

The characteristics of the concurrency control algorithm can be described in the following:

- The semantic knowledge about typed requests and transactions can be used; therefore the degree of concurrency may be improved.

- Cooperation of nested requests within a transaction is possible; therefore additional design flexibility is provided.

- No lock conversion is necessary.

5.1.6 Deadlock

In the proposed concurrency control algorithm, the deadlock problem may occur. It is resolved based on a deadlock detection approach.

Whenever a lock is denied, a test message will be initiated for deadlock detection if the timestamp of the request is less than any timestamp of the object which will be accessed by the request. The transaction of the request is called the initiating transaction. The message will be retransmitted periodically until the initiating request has been served or aborted.

Deadlock detection consists of initiation, detection, and resolution. Deadlock detection is initiated when a waiting request
in a cell has a timestamp less than any one of the objects which it
intends to access. The test message contains a list which contains
the timestamp of the initiating transaction. The detection
basically performs an edge chasing to detect if there is a cycle in
the wait-for graph among a group of transactions. It avoids the
construction of a global wait-for graph for deadlock detection. A
similar concept has been proposed in [Moss81]. The deadlock
detection method is described as follows:

1. check which transaction holds the object (if n
   transactions simultaneously hold the object such as
   transactions with compatible requests, associate a copy
   of the message with each transaction), add the
   transaction into the list in the test message, and go to
   (2).

2. propagate the test message to every tree node of the
   transaction and check if they are waiting in a cell for
   any object
   a. if yes: check if the timestamp of the object is
      smaller than the one of the initiating transaction.
      i. if yes: abolish the test message and exit.
      ii. otherwise: go to (3).
   b. otherwise: abolish the test message and exit.

3. check which transaction holds the object (if n
   transactions simultaneously hold the object, associate a
   copy of the message with each transaction)
   a. if the transaction is in the list: an inter-
      transaction deadlock occurs.
   b. otherwise: add the transaction into the list in the
      message and go to (2).
The reason for abolishing a test message in (2.a.i.) is to prevent it from initiating multiple test messages in the same deadlock cycle. Therefore, overhead is reduced. Figure 20 illustrates this aspect. In this example, transactions A, B, and C have the timestamps 10, 20, and 43, respectively. According to the deadlock detection method, two test messages will be initiated in this example: (1) by transaction A when it waits for object X, and (2) by transaction B when it waits for object Y. The first test message will trace the cycle back to A and will finally detect the deadlock. However, according to (2.a.i), the second message will be abolished once it reaches object Z.

Once a deadlock is detected, deadlock resolution is used to select a candidate transaction from the list and to abort it to break the cycle. The selection is based on the following criteria:

- If there is more than one deadlock cycle, select a most frequent occurrence of transaction among the lists. The purpose is to resolve deadlocks by aborting a minimum number of transactions.

- If there is no common transaction among lists, select the youngest transaction (with the largest timestamp) in each list, and abort this transactions.
Figure 20: An Example of Multiple Test Messages
5.2 Analysis of The Algorithm

The algorithm presented above is analyzed in terms of correctness, deadlock, cascading aborts, performance, and robustness in the following subsections.

5.2.1 Correctness

According to the proposed algorithm, a conflicting request which intends to access an object will be manipulated based on the pessimistic locking if it satisfies any of the following conditions:

- there are multiple active transactions on that object.
- there is a single active transaction on that object and the timestamp of the object is not the same as the one of the conflicting request.

In other words, only compatible requests from different transactions are allowed to access an object in an overlap period. Conflicting requests from other transactions will be forced to wait until the active transaction commits. Therefore, the consistency of objects among requests from different transactions can be maintained.

If there is a single active transaction on that object, any nested request of this transaction will be granted a lock with an appropriate mode. Dependency relationships among the requests from the same transaction which performed updates on the same object will be maintained by the commit conditions dynamically generated and
checked at their immediate ancestor. Since there is at least one common ancestor for any two nested requests within a transaction, the internal consistency will always be checked at an appropriate time and at an appropriate place. If the consistency has been violated, the related requests will be aborted. Therefore, the locking rules guarantee the consistency of objects within a transaction.

5.2.2 Deadlock

Hold-and-wait, no preemption, and exclusive control are necessary conditions for a deadlock [ISL080]. In the cell model, a transaction may consist of several nested requests to form a tree of nested actions. Each leaf of the tree represents a nested action performed by a typed request. Typed requests are only allowed to access specific typed objects. The algorithm allows nested requests in a transaction to cooperatively access the same object as long as they meet the requirement of mutual exclusion. If an object is held by a nested request of the transaction, other requests of the transaction which intend to access the object will eventually be allowed to perform. The condition of hold-and-wait will never hold for any nested action of the tree. Therefore, a cell-based system is free from intra-transaction deadlock.

According to the algorithm, we claim that the following statement is true:
"If there is an inter-transaction deadlock, a deadlock detection will be initiated."

[proof]

Assume that there is no test message being sent and an inter-transaction deadlock does exist in the system. Then, according to Holt [HOLT72], there exists a cyclic relation in the wait-for graph, such as

\[
p_1 \rightarrow r_1 \rightarrow p_2 \rightarrow r_2 \rightarrow p_3 \rightarrow \ldots \rightarrow p_n \rightarrow r_n \rightarrow p_1
\]

where \( p_i \) represents a transaction,
\[
r_j \text{ represents an object,}
\]
\[
 p_i \rightarrow r_j \text{ means } p_i \text{ is waiting for object } r_j,
\]
\[
r_i \rightarrow p_j \text{ means } r_i \text{ is held by } p_j.
\]

Since there is no test message being sent, according to the algorithm, there is no transaction in the cycle which is waiting for an object having a larger timestamp.

Hence,

\[
\text{ts}(p_1) > \text{Max}(\text{ts}(r_1)) \geq \text{ts}(p_2) > \text{Max}(\text{ts}(r_2)) \geq \ldots
\]
\[
\ldots \text{ts}(p_n) > \text{Max}(\text{ts}(r_n)) \geq \text{ts}(p_1)
\]

where \( \text{ts} \) denotes the timestamp, and 
\( \text{Max}(\text{ts}(r_i)) \) denotes the maximum of the timestamps of the object \( r_i \) if it has more than one timestamp.

It is a contradiction.

Therefore, if there is an inter-transaction deadlock, a deadlock detection will be initiated.
The deadlock detection algorithm uses test messages to chase the wait-for graph and to detect a cycle. Once a deadlock is detected, deadlock resolution will guarantee eventual system progress by selecting and aborting a transaction in the cycle.

5.2.3 Cascading Rollbacks

The concurrency control algorithm prohibits conflicting transactions from accessing the same object in an overlay period. Although it restricts the degree of concurrency, compared to the optimistic methods, it can completely avoid the possibility of intolerable cascading rollbacks when many conflicting transactions simultaneously access the same object. Failure to meet the commit conditions within a transaction will only cause at most the abortion of that particular transaction, and no other transaction will be affected. Therefore, it can avoid causing intolerable cascading aborts of transactions.

5.2.4 Performance

Selection of concurrency control algorithms is usually based on the following factors:

- degree of concurrency:

  If a concurrency control algorithm can provide a better degree of concurrency, the system performance can be improved.

- algorithmic overhead:
The overhead introduced by a concurrency control algorithm will also affect the system performance.

- other tangible/intangible benefits:

Other related problems, such as design flexibility, deadlock resolution, and the complexity of deadlock detection algorithms, should also be taken into consideration.

As mentioned above, the traditional timestamping approach is simple, but the problem of request restarting becomes very serious when the system is heavily loaded and the contention rate is high. Access sequences are very important in this approach, and designers may have difficulties in controlling them, especially in a distributed environment where the delay of inter-node communication may be varied.

The two-phase locking can avoid causing the domino effect. However, the delay of accesses from other conflicting requests degrades the concurrency. The proposed algorithm allows accesses from other conflicting requests within a transaction, instead of postponing them. Moreover, the type-specific locking used in the algorithm can take advantage of the semantic knowledge about typed requests. Therefore, it may provide a better degree of concurrency. The requirement of serialization in the two phase locking and the timestamping approach preclude interprocess cooperation even within a transaction. As shown in Figure 21, transaction X consists of three nested atomic actions, X0, X1, and X2. Nested atomic actions X1 and X2 cooperatively perform operations on object M (i.e., an
one-slot buffer). X1 performs a series of insert operations on object M while X2 performs a remove operation on object M only when X1 has performed an insert operation on it. Therefore, the remove operations issued by X2 are performed alternatively with the insert operations issued by X1 as shown in the figure. According to the two-phase locking and timestamping approaches, the requirement of serializability prohibits this kind of cooperation between nested atomic actions X1 and X2. However, the cooperation between X1 and X2 is made possible by using the proposed algorithm.

The optimistic approach can provide a higher degree of concurrency than does the proposed algorithm if the conflicting rate is low. However, in a general purpose environment, the conflicting rate among requests is hard to anticipate and the problem of intolerable cascading aborts may occur if it is high. This may be unacceptable for some applications.

Major overhead involved in the algorithm includes: (1) processing of locks and commit conditions, and (2) test messages for deadlock detection. Several attempts have been made to simplify the process of commit conditions in the concurrency control algorithm and to reduce the number of test messages in the deadlock detection method. The deadlock resolution also aims to sacrifice the least amount of computation.
Figure 2.1 An Example of Interprocess Cooperation
5.2.5 Robustness

The following are issues related to the robustness of a concurrency control algorithm in a distributed environment:

- **Degree of locality for concurrency control information**

  If the concurrency control information is available locally, it can reduce the potential problem due to the failures of remote sites or communication links.

- **Correctness of the algorithm when the concurrency control information is incomplete due to site or communication failures**

  If there are mechanisms embedded in the system to avoid improper concurrency control, to prevent it from possibly introducing a deadlock, or to eliminate other problems due to faulty or incomplete concurrency control information caused by site or communication failures, robustness can be improved.

The concurrency control information in the proposed algorithm includes the timestamps of an object, the timestamp of a transaction, and the synchronization constraints specified in a cell. The timestamp of a transaction is created once a transaction is received by the system. The timestamps of an object are updated whenever a request is accessing the object or a request which has accessed the object has globally committed. The synchronization constraints are maintained locally in a cell. The locks are locally maintained in stable storage for each cell and they are not propagated from a cell to another as described in the commit protocol. Therefore, the problem caused by site or communication failures can be reduced to a limited extent.
5.3 Recovery Algorithms

The recovery algorithms in the cell model need to resolve two problems:

- maintaining resource consistency while aborting or undoing a request

The algorithm must cooperatively work with the concurrency control algorithm to restore a correct object state when a request is aborted or undone.

- correctly restoring a cell state after its crash

The cells must work cooperatively with one another to correctly, rapidly, and economically restore a cell state should failures occur.

The recovery algorithms rely on the following concepts and facilities: (1) cell-wide checkpointing, (2) do-undo-redo logic, and (3) the active request log. These aspects are described in Subsections 5.3.1, 5.3.2, and 5.3.3, respectively. The recovery algorithms based on them are then presented in Subsection 5.3.4.

5.3.1 Checkpointing

As described before, the purpose of checkpointing is to improve recovery performance by reducing the amount of work required to construct a correct state upon restart. The methods for checkpointing can be characterized by two parameters: (1) checkpoint intervals, and (2) the scope of checkpointing. The checkpoint interval is classified into two types:
- fixed interval (time-driven):

The checkpoint is taken after every fixed period of time.

- variable interval (event-driven):

The checkpoint is taken only when an event occurs. The following are some possible events: having received a fixed number of requests, or having received a fixed number of particular typed requests.

The scope of checkpointing refers to what types of entity are recorded during checkpointing. There are three possibilities about the scope: process state only, object state only, and both process and object state. In addition, checkpointing can also be taken in terms of a request (transaction), a cell, a node, or the system.

In this chapter, we describe a cell checkpointing method based on the time-driven interval and with the scope of both processes and objects.

The cell state can be snapshoted periodically. The interval of checkpoints is usually dependent upon the average mean time before failures (MTBF) and reliability requirements. If the checkpointing is the only recovery mechanism adopted in the system, the longer a checkpoint interval is, the less overhead it introduces. However, the computations which are lost due to failures are greater.

In order to reduce the overhead involved, only the last two checkpointing records are kept by a cell. The purpose of keeping two checkpointing records is that at least one complete checkpointing record can be used in case failures occur in the middle of writing the last checkpoint.
5.3.2 Do-Undo-Redo Logic

The do-undo-redo logic requires that each cell should have the capability to undo a request when a request is aborted and to redo a request in case its result is lost. This concept provides a cell with adequate flexibility to maintain the consistency of its objects and the cell state. There are several ways to implement the concept of do-undo-redo logic. Notable among them are the shadow version and the recovery log approaches.

A shadow version can be viewed as a temporary version of an object. It is usually implemented in a volatile storage. Before a request intends to access an object, a temporary (shadow) version based on its permanent version is created in a volatile storage. The update will be performed on the shadow version. Once the update is complete and the request intends to globally commit, the shadow version will be changed into the permanent version. In this implementation, if failures occur in the middle of the update, the undo of the request is quite trivial.

Based on the concept of the recovery log, each cell associates with two logical recovery logs: undo and redo logs. The logical logs may be implemented in a physical log. The undo log records the request-related state before serving a request whereas the redo log records the request-related state after serving a request and before its commitment. The recovery logs are stored in the stable storage to survive various failures. Based on this implementation, when a
request intends to access an object, an undo log will be created in stable storage. The request will directly perform the updates on the object which is marked as a temporary version. Once the request intends to commit, a redo log will also be created on the stable storage. Finally, the temporary version will be changed into the permanent version if the request globally commits. If failures occur in the middle of this update, the request can be undone by simply restoring the information in the undo log.

In these two approaches, the overhead introduced by the recovery log is about two times that introduced by the shadow version. However, if the object is implemented in the volatile storage such as main memory, the shadow version approach is inadequate to correctly restore the state when failures occur. Therefore, the shadow version approach is usually adopted in the file or database systems where objects are implemented in the disk system.

In order to avoid keeping an infinite period of the recovery logs, the recovery logs that were taken before the second last checkpoint can be discarded.
5.3.3 Active Request Log

Each cell in the system has an active request log (ARL) which records the status of all active requests in the cell. The active request log is stored in a stable storage system which can survive various failures such as node crashes and medium corruption. Once an incoming request in a cell has been validated, a proper entry in the active request log will be created. The information recorded in each entry of the log includes three major parts:

- request status
  It contains request identifier, request image, request status (committed, aborted, in progress), reply, the locks it retains, committed IDs, and commit conditions.

- client information
  It contains client identifier.

- subrequests information
  It contains subrequests which has been issued by the request, their server identifiers, their status, and the replies from the servers if committed.

The active request log is designed to correctly record the request status and its relationships with related cells (its client or its servers) in stable storage to survive various failures. The purpose of the active request log is to maintain a consistent view of active requests among related cells in the system in spite of various failures.
5.3.4 Recovery Algorithms

As mentioned before, two recovery problems need to be addressed: how to maintain resource consistency while aborting or undoing a request, and how to correctly restore a cell state while encountering failures.

RESTORING AN OBJECT STATE

To correctly restore an object state while aborting an update in case of failures or aborting a locally committed request, we need to perform the following two steps:

- to restore the information in the undo log, and
- to append an entry in the redo log to record the abortion of the request.

Once a request has globally committed, undoing the request is impossible. The only feasible solution is to invoke a compensation request which removes the effect of the previous request.

RESTORING A CELL STATE

Cell and request states play very important roles in the system recovery. Before introducing the recovery algorithm, the state transitions of cells and requests are described in the following.

Each cell has three possible states: normal, crash, and restart states as shown in Figure 22. A cell will be changed from a normal state into a crash state due to a node crash. Once the node has been repaired, it will enter the restart state and perform the recovery functions. Finally, when the correct state has been
restored, the node will reenter the normal state.

The possible state transition diagram of a request is shown in Figure 23. A request is created in a client cell and submitted to a server cell. Once the acknowledgment from the server has been received, the state of the request in the client is changed from "submit" to "in process". Once the service has been performed or aborted by the server and its reply has been sent back to the client, its state will be changed from "in progress" into "commit" (local commit) or "abort", depending on the case. Finally, when the transaction commits (the request globally commits), the proper entry in the active request log will be removed and the request will be forgotten. According to this state diagram, possible states of a request in a client cell include "submit", "in-progress", "commit", "abort", and "global commit". However, possible states of a request in a server cell include "in-progress", "commit", "abort", and "global commit".

To restore a correct cell state after a crash, the following steps should be performed:

1. restore the cell state based on the last checkpoint taken;
2. consolidate the active request log with those of its related cells,

The related cells include the clients of the active requests in the cell and the servers of the sub-requests submitted by the active requests in the cell. The purpose of consolidation is to do consistency checks between the cell's active request log and those of the
Reconfiguration

Normal \rightarrow Crash

Crash \rightarrow Repaired

Recovery/Reconfiguration complete

Crash again

Normal \rightarrow Crash again

Normal \rightarrow Crash \rightarrow Repaired \rightarrow Restart

Figure 22: The State Diagram of a Cell
Figure 23: The State Diagram of a Request
related cells. Whenever there is an inconsistency between the request status, the one with a higher priority will override the other. The changing of request status to "commit" or "abort" in an ARL will create an entry in the redo log. When an entry of an active request log is removed during consolidation from the status "commit", it will be handled as a global commit (release locks, change temporary versions into permanent versions, change timestamps of objects, etc.). However, when a subentry of an active request log is removed during consolidation from the status "submit", no other action needs to be taken. The priority of the status is stated as follows:

a. "abort" has a higher priority than "commit".

b. "commit" has a higher priority than "in-progress".

c. "in-progress" has a higher priority than "submit".

d. The missing entry has the highest priority among them.

3. reconstruct a cell based on the recovery logs and active request log;

a. based on the redo-log, restore in a chronological order the redo entries, which were created after the last checkpoint.

b. based on the active request log and undo log, restore in a chronological order the undo entries which were created after the last checkpoint for the active requests and did not locally commit.

4. re-process the active requests which do not locally commit.
5.4 Analysis of Recovery Algorithms

The recovery algorithms are analyzed in the following, in terms of correctness and robustness.

5.4.1 Correctness

To prove the correctness of the recovery algorithms, we need to show that the aforementioned algorithms can correctly restore the object and cell state.

Mutual exclusion of the concurrency control algorithm in the cell model only allows one of the conflicting requests to perform at one time; therefore, the undoing of a request can be done by simply restoring the before-image of the object in the undo log. The consistency of the object can be guaranteed since no other conflicting request is allowed to access it simultaneously.

Figure 24 shows five different types of requests, in terms of their life span and their relative sequence to the occurrences of checkpointing and crashing. They are:

- type-1 requests
  The requests have been received and performed before the last checkpoint.

- type-2 requests
  The requests have been received before the last checkpoint but finished after the checkpoint and before crash.

- type-3 requests
The requests have been received before the last checkpoint but are still in the middle of execution when a crash occurs.

- type-4 requests

The requests have been received after the last checkpoint and finished before crash.

- type-5 requests

The requests have been received after the last checkpoint but are still in the middle of execution when a crash occurs.

After performing step (1) of the recovery algorithm, the cell state will be restored to the last checkpoint and the after-images of the objects being accessed by the type-1 requests will then be correctly restored. After performing step (2) of the recovery algorithm, the status of objects in the cell is not changed. However, the consistent view of request status among related cells is maintained. After performing step (3.a) of the recovery algorithm, the after-images of the objects, which were created by the type-2 and type-4 requests, will be correctly restored. After performing step (3.b) of the recovery algorithm, the before-images of the objects, which were created after the checkpoint by the type-5 requests, will be correctly restored. After performing step (4) of the recovery algorithm, the active requests when a crash occurs such as the type-3 and type-5 requests will be reprocessed. Therefore, the recovery algorithm can correctly restore the cell state.

The cases examined below prove the correctness of the
Figure 24: The Types of Requests
consolidation procedure in step (2) of the aforementioned recovery algorithm.

1. A client crashes when it sends out a request but before receiving its acknowledgment from the server.

When a client crashed, the status of the request at the client was "submit" where its status at the server was uncertain. It might be a missing entry if the request was not correctly received by the server or "in-progress" if it was correctly received by the server. Based on the consolidation procedure, the status of the request in the client after restarting will be changed into a missing entry if the status in the server is a missing entry or "in-progress" if the status in the server is "in-progress".

2. A client crashes when its server sends back its reply to the client.

When a client crashed, the status of the request at the client was "in-progress" where its status at the server was either "commit" or "abort". Based on the consolidation procedure, the status of the request in the client after restarting will be changed into "commit" if its status in the server is "commit" and "abort" if its status in the server is "abort".

3. A nested cell crashes during a global commit.

When a nested cell crashed at its global commit, the status of the request at the nested cell was "commit". The status of the request in its client is a missing entry when the nested cell is restarting. Based on the consolidation procedure, the entry of the request will be removed and it will be handled as a global commit (release locks it holds, change temporary versions into permanent versions, etc.) when restarting.

Based on the aforementioned observations, correct status of requests can be restored during restarting.
5.4.2 Robustness

There are two basic approaches to providing recovery: storage and operation modes. The storage mode restores the system state based on the information recorded in stable storage. A significant advantage of this approach is that it provides the property of idempotency. In other words, multiple executions of the recovery procedure will not change the final results. The proposed recovery algorithm adopts this approach.

The operation mode restores the system state based on the execution of a predefined set of operations. This set of operations usually depends upon how many operations have been performed before the crash or upon the extent of the damages. It is usually time-consuming or even not feasible to acquire this information in a general-purpose distributed computing environment. The storage mode can be viewed as a special case of the operation mode.

If a cell crashes again in the middle of the recovery process due to node failures, it can be correctly restored by simply redoing the recovery according to the property of idempotency provided by the recovery algorithm.
5.5 Summary

In this chapter, we described concurrency control and recovery algorithms used in the cell model. The concurrency control algorithm is based on an adaptive approach using semantic knowledge. The request within a transaction is manipulated based on the optimistic method. Requests from different transactions are handled based on the pessimistic locking, and only compatible requests are allowed to perform in an overlap period. The consistency of objects among conflicting requests in a transaction is maintained by the commit conditions dynamically generated at each request. The concurrency control algorithm aims to provide a better degree of concurrency by using semantic knowledge, to allow interprocess cooperation within a transaction, and to avoid causing an intolerable cascading aborts of transactions.

The cell recovery relies upon the following concepts and facilities: checkpointing, do-undo-redo logic, an active request log, and a robust recovery algorithm. The use of stable storage is essential in this approach. The proposed recovery algorithm and facilities aim to correctly restore the cell state when failures occur. The property of idempotency provided by the recovery algorithm allows a cell to survive cyclic crashes in the process of recovery.

The communication and commit protocols which support the cell model are described in the following chapter.
Chapter 6

COMMUNICATION AND COMMIT PROTOCOLS

This chapter describes communication and commit protocols which are proposed for the cell model. As mentioned in Chapter 4, existing process interaction models, such as synchronous and asynchronous communication, can be applied to the cell model. However, in order to provide better reliability and performance, the semantics of intercell communication is based on the quasi-asynchronous communication. The two-phase commit protocol can be applied to the cell model to coordinate a group of cooperating cells to achieve the atomic property. However, crashes of participating cells after finishing their services but before global commit may delay or abort the committed actions, which is unnecessary. A commit protocol called Distributed Partial Commit Protocol (DPCP) is, therefore, proposed to remedy this shortcoming.

Intercell communication protocols and primitives are described in Section 6.1. Analysis of the intercell communication model is presented in Section 6.2, in terms of reliability and performance. A commit protocol, which takes advantage of the semantic knowledge about the tree-like structure in the nested atomic action
environment to improve performance, is proposed in Section 6.3. Section 6.4 contains a discussion of the performance and robustness of the commit protocol. Finally, a summary of this chapter is given in Section 6.5.

6.1 Intercell Communication

There are two major considerations in the design of intercell communication: reliability and performance.

When two cells communicate with each other, a communicating party is expecting to receive a message from its corresponding party. There are two possible outcomes: (1) the message arrives in time, and (2) the message never arrives or arrives too late. In the first case, the message may be in error or damaged due to communication failures. Some coding, verification, or retransmission methods can be used to correct some damages or errors in the message. However, in the second case, there may be several possible causes, such as:

- node crashed
- network partitioned
- message lost due to communication failures
- message arbitrarily delayed due to long routing, network congestion, deadlock, etc.

Therefore, intercell communication must be carefully designed to achieve fault tolerance as follows:
- the communication is reliably accomplished, or
- An exception is raised when failures occur.

The communication model and primitives adopted in the cell model are discussed below.

6.1.1 The Client/Server Model

The client/server model is used to model the behavior and interaction of intercell communication. In the model, a client sends a request to a server. The server will perform the service once it receives a request. Finally, the result will be sent back to the client. There are three possible semantics associated with this model:

1. asynchronous communication (send and immediately proceed)

   After a client sends a request to a server, the client will immediately proceed to process other tasks in spite of various possible outcomes of the request. The control flow between a client and a server for this type of communication is shown in Figure 25.

2. synchronous communication (send and wait for reply)

   A client sends a request to a server and the client can proceed further processing once it receives the reply from the server. The control flow between a client and a server for this type of communication is shown in Figure 26.

3. quasi-asynchronous communication (send and wait for acknowledgment)

   A client sends a request to a server, and when the server receives the request, it will check the validation of the request and sends appropriate (positive or negative) acknowledgment back to the client. Once the client
receives the acknowledgment from the server, it can proceed further processing. The control flow between a client and a server for this type of communication is shown in Figure 27.

Traditional message passing is a typical example of asynchronous communication, and the procedure call is a typical example of synchronous communication. Asynchronous communication is not very reliable because messages may be lost somewhere or may be garbled due to communication failures. The sender will never know the outcome unless it checks with the destination. Synchronous communication is quite reliable. However, tight synchronization between sending requests and receiving replies such as procedure call does not allow parallelism between the sender and receiver. In order to improve the parallelism, we have adopted the quasi-asynchronous communication as the semantic basis of intercell communication.

6.1.2 Communication Primitives

Two communication primitives are provided in intercell communication: invoke and wait. The invoke primitive sends a request to a server and waits until an acknowledgment from the server has been received or an exception has been raised. The detailed operation of the primitive is as follows:

1. record a subrequest entry in the active request log.
2. set a timeout counter.
Figure 25: The Control Flow of Asynchronous Communication
Figure 26: The Control Flow of Synchronous Communication
Figure 27: The Control Flow of Quasi-asynchronous Communication
3. send the request to the server.

4. wait for acknowledgment from the server:
   - if a positive acknowledgment has been received: return "normal" to the user.
   - if a negative acknowledgment has been received: return "interface exception" to the user.
   - if timeout occurs: retransmit the message to the server and if the situation still exists after several retries, notify the cell responsible for system configuration, and return "failure exception" to the user.

Two types of exceptions may be raised: interface and failure exceptions. An interface exception indicates that the request fails to pass the validation check in the server. Possible causes of this exception may include illegal/unauthorized requests, parameter errors, and so on. A failure exception indicates that the server fails to respond to the request. Possible causes of this exception may include cell failure (due to a node failure), isolated cell (due to a network partition or broken link), improper timeout settings, and so on.

The wait primitive will force the client to wait until either a reply has been received or an exception has been raised. The detailed operation of the primitive is shown as follows.

1. check the active request log
   - if its result has been received: return the reply to the user;
   - otherwise: wait for a while and recheck the active request log; if no result has been received then go
to 2.

2. an inquiry will be sent to the cell responsible for system configuration to check the status of the server:

   - if the server crashes: return "failure exception" to the user.
   
   - if the server is up: go to 3.

3. an inquiry will be sent to the server and a timeout counter will be set:

   - if the service has been committed or aborted: return the reply to the user.
   
   - if the service is still in progress: wait until a timeout occurs.
   
   - if a timeout occurs: notify the cell responsible for system configuration, and return "failure exception" to the user.

In intercell communication, a pair of invoke and wait primitives is semantically equivalent to a remote procedure call.

6.2 Analysis of The Intercell Communication Model

The intercell communication model is analyzed in terms of reliability and performance in Subsections 6.2.1 and 6.2.2, respectively.
6.2.1 Reliability

In cases of communication errors, node crashes, or network partitions, several abnormalities may occur during intercell communication. These problems are discussed in the following.

- communication errors

When communication errors occur, a client may retransmit the request to the server. The server will probably receive multiple copies of the same requests. The problem of duplicate requests may occur. Since the service performed by the server is not guaranteed to be idempotent, some mechanisms embedded into the cell to guarantee that only one request is performed are necessary. A duplicate request reject mechanism can be built into a cell based on the active request log. Each cell has an active request log which contains the status of all active requests in that cell. Each identifier of a new request will be checked against the log. If the same identifier already exists in the log, the request will be rejected; otherwise, a new entry will be added to the log.

- network partition

When a client and its server are isolated due to a network partition, a failure exception will be raised to the client. Therefore, proper actions can be taken based on its requirements.

- node failure

There are two possible abnormalities for node failure: (1) a client crashes after submitting a request and before receiving a reply, and (2) a server crashes after receiving a request and before accomplishing the service. These two cases are discussed below.

* client failure

If a client crashes after invoking a request and before receiving a reply from the server, the problem of orphan process may occur. In the intercell
communication model, once an agent in the server commits (only after the result and after-images of accessed objects have been recorded in the active request log and redo log), it will be terminated by the manager. Therefore, no orphan process will exist in the server.

* server failure

If a server crashes after receiving a request and before accomplishing the service, the problem of indefinite waiting may occur. Since we never know when the server will be repaired and recovered, the best solution to the situation is to let the client decide whether to wait or to abort the request. In intercell communication, when this problem occurs, an exception will be raised in the wait primitive and proper actions can be taken based on its needs. If an abortion has been requested by the client, the consistency between the client and server must be maintained. When the server is restarted, the consistency check in the active request log will abort the service performed at the server, therefore, consistency can be guaranteed.

6.2.2 Performance

The quasi-asynchronous communication adopted in the intercell communication provides a higher degree of internal parallelism than does the traditional procedure call. Also, its semantics allows a client to continue processing after receipt of an acknowledgment.

This higher degree of internal parallelism has a great impact on the response time of a user request. If intercell communication is implemented based on remote procedure calls, the response time of a user request may be longer than that of the quasi-asynchronous communication. Figure 28 is an example to illustrate this fact. In
this figure, a request is performed by a group of five cooperating cells: A, B, C, D, and E. The response time for the computation can be formalized as follows:

\[
\text{response time} = a + b + c + d + e + 8t \quad \ldots \ldots \text{(a)}
\]

where: \(a, b, c, d,\) and \(e\) denote processing time for nested atomic actions A, B, C, D, and E, respectively, and \(t\) denotes message exchange time for an intercell communication where we assume that the time needed for each message exchange between cells is the same.

Based on the semantic of the quasi-asynchronous communication, the response time will be approximately reduced to the following equation:

\[
\begin{align*}
\text{response time} &= a + b + d + 6t \\
&\quad \text{or} \\
&\quad a + b + e + 6t \\
&\quad \text{or} \\
&\quad a + c + 3t \quad \ldots \ldots \text{(b)}
\end{align*}
\]

In reality, the response time in the cell-based system by using the proposed intercell communication will be between the values in equation (a) and in equation (b), depending upon the structure of the agent tree and the computation semantics.
Figure 28: An Example for Calculating Response Time
6.3 Distributed Commitment

As mentioned in Chapter 4, a transaction may be performed by a group of cooperating cells. Each cell performs a part of the whole function and the computations performed in each cell are atomic. The cells may reside in the same physical node or may be scattered in different nodes. Thus, a distributed commit protocol to coordinate these cooperating cells is needed to achieve the atomic property in spite of node or communication failures.

The traditional two-phase commit protocol can be used to resolve this distributed commitment problem. However, as pointed out in Section 4.8, this protocol has the following disadvantages in our case:

- It does not take advantage of the semantic knowledge about the tree-like structure. With the structure, the preparation phase of the two-phase commit protocol is not necessary for the global commitment.

- It may postpone the commitment of a transaction or abort the transaction if some nested atomic actions have finished their computations but the nodes they reside in crashed after their local commitments.

In order to resolve these problems, a distributed commit protocol called Distributed Partial Commit Protocol (DPCP) is proposed. This commit protocol consists of two phases: local commit and global commit. Local commitment describes the commit procedure at a non-top-level cell. It intends to achieve the atomic property for a subtree of cooperating cells where the server is the root of
that subtree. Global commitment describes the commit procedure at a
top-level cell. It intends to maintain a consistent view of active
request logs among a group of cooperating cells. The local and
global commitments are described in Subsections 6.3.1 and 6.3.2,
respectively.

6.3.1 Local Commitment

Local commitment governs the commitment procedure between a
client and a server. When a server intends to commit or abort a
service, it should perform the following steps:

- append an entry to the redo log and update the request
  status in the active request log
- send the reply to its client
- terminate the agent

If the service is aborted and it contains subrequests to other
cells, the following additional steps should also be performed:

- update the subrequests' status
- periodically send abort messages to these cells to abort
  the committed subrequests until acknowledgments from the
  cells have been received.

Whenever a committed server receives an abort message from its
client, it needs to undo the service based on the undo log and to
remove the entry from its active request log.

Considering the example in Figure 29 and assume that Cell D
intends to commit. According to the local commit protocol, Cell D
needs to perform the following steps:

- append an entry in the redo log and change the status of request X11 in the active request log into "commit"
- send the reply back to Cell B (including the commit ID X11)
- expunge agent X11 in Cell D.

Assume that Cells D and E have locally committed and Cell B intends to abort for some reason. According to the local commit protocol, Cell B needs to perform the following steps:

- Append an entry in the redo log and change the status of request X1 into "abort" in the active request log.
- Send the reply back to Cell A (No commit ID will be included in this case.).
- Change the status of subrequests X11 and X12 into "abort".
- Expunge agent X1 in Cell B.
- Periodically send "abort" messages to Cells D and E to abort the committed services by subrequests X11 and X12 until acknowledgments from the cells have been received.

Once Cells D and E receive the "abort" message from Cell B, requests X11 and X12 will be undone and another redo entry will be appended into their respective redo logs.
Figure 29: An Example for Distributed Commitment
6.3.2 Global Commitment

Global commitment governs the commitment procedure in a top-level cell. When a top-level cell intends to commit (abort) a service, the following steps should be taken:

- append an entry to the redo log and update the request status in the active request log.
- send the reply to its client.
- terminate the agent.
- if the service contains subrequests to other cells, it should also perform the following:
  * periodically send global commit (abort) messages to its committed nested cells until acknowledgments from the cells have been received.
  * remove the request entry from the active request log if acknowledgment from its committed descendant has been received.

Once committed nested cells receive a global commit message from the top-level cell, it will broadcast this message to its descendants. The proper entry of the active request log in a cell will be removed only when the acknowledgments from its committed descendants have been received. The global commitment guarantees all or nothing service provided by nested cells and maintains the consistency of active request logs in all participating cells. Figure 30 shows possible commit actions in the proposed commit protocol.

The following is an example of a global commit. In Figure 29,
Figure 30: Possible Commit Actions for the DPCP
assume all nested cells B, C, D, and E have locally committed. When top-level cell A intends to commit, Cell A needs to perform the following steps:

- Append an entry in the redo log and change the status of request XO into "commit" in the active request log.
- Send the reply back to the user.
- Expunge agent XO in Cell A.
- Periodically send "global commit" messages to cells B and C until acknowledgments from the cells have been received.
- Remove the corresponding entry from the active request log if acknowledgments from Cell B and C have been received.

Once Cell B receives a global commit message from Cell A, it will periodically send a global commit message to Cells D and E until acknowledgments from the cells have been received. It will also remove the corresponding entry from the active request log if the acknowledgments from Cells D and E have been received.

If Cells C, D, or E receive a global commit message from their ancestors, they will remove the corresponding entry in their active request logs and will send back an acknowledgment to their ancestor.

The commit sequences in the tree-like structure is from the bottom to the top. Figure 31 is an example to illustrate possible commit sequences for a tree-like computation and its corresponding local and global commits.
Figure 31: Possible Commit Sequences for a Tree of Nested Atomic Actions
6.4 Analysis of Commit Protocol

The features of the distributed partial commit protocol are characterized as follows:

- The commit procedure is initiated by a lowest level server and finalized by a top level server.
- The protocol can accommodate dynamic restructuring of nested atomic actions due to node crashes, network partition, and even residual design faults.
- The protocol allows a partial commitment. The partial commitment has two meanings here. First, all agents participating in a job or request need not commit at the same time. Second, the commitment or abortion of a computation can be decided by the agent itself based on the results of its local and remote operations.

The commit protocol is analyzed in terms of robustness and performance in the following subsections.

6.4.1 Robustness

The robustness of the commit protocol is discussed in terms of communication errors, network partition, and node failures in the following.

- communication errors

Communication errors addressed here include messages lost or damaged. There are two possible abnormalities in distributed commitment: (1) loss of reply from the server to its client, and (2) loss of global commit or abort messages from higher level cells (clients) to lower level cells (servers). If the reply from the server is lost due to communication errors, the semantic in the wait primitive will automatically inquire the server. Therefore, the problem can be avoided. If the commit or
abort messages from higher level to lower level cells have been lost, the messages will be sent periodically until acknowledgments have been received. This also automatically resolves the problem.

- network partition

Network partition may prevent the messages (such as reply, global commit, abort, or even inquiry) from arriving at their destinations. From a client's viewpoint, a failure exception will be raised by the invoke or wait primitives; therefore, appropriate actions can be taken. In addition, the periodic sending of global commit and abort messages will guarantee the consistency of active request logs once the isolated party joins the system later.

- node failures

If a server fails before the commitment of a service, the aforementioned communication primitives will take care of it. We will ignore this situation. If a server fails after the commitment of a service but before its global commitment, as mentioned in the case of network partition, the periodical sending of global commit and abort messages will guarantee the consistency of active request logs once the crashed node is repaired and recovered later.

6.4.2 Performance

The performance of the commit protocol is now discussed in terms of the number of messages required and response time.

As demonstrated in the problem of the generals paradox [GRAY79], there is no shortest commit protocol for distributed commitment. However, the proposed commit protocol, in most cases, uses less messages than does the two-phase commit protocol in the nested atomic action environment.

In the following example, there is a group of n participating
parties. During the global commit, the number of messages used in the two-phase commit protocol is about $4(n-1)$ messages. However, in the proposed commit protocol, this number can be reduced to about $2(n-1)$ and the response time may be reduced for most cases, mainly due to the omission of the "prepare phase" from the two-phase commit protocol.

This commit protocol allows the commitment of a set of nested atomic actions if one or several nodes, where nested atomic actions reside, crashed after finishing their computations but before the global commitment. The undoing of the whole computation or the delay of the commitment are unnecessary. The DPCP allows the global commitment at this moment and lets cell managers maintain a consistent view of active requests. However, in the two-phase commit protocol, the top-level cell has to wait until all nodes are up. Therefore, response time may be reduced by using the proposed commit protocol.

6.5 Summary

This chapter described the communication and commit protocols which are proposed for the cell model. The semantic of the intercell communication is based on the quasi-asynchronous communication. Both reliability and performance are main concerns in intercell communication. Two communication primitives, invoke and wait, are used to provide reliable intercell communication.
The proposed commit protocol outperforms the two-phase commit protocol by taking advantage of the semantic knowledge of the tree structure. The protocol consists of two phases: local commit and global commit. Local commitment guarantees the atomic property for the subtree of cooperating cells when a server is the root of that subtree. And global commitment maintains a consistent view of active request logs among a group of cooperating cells. This protocol allows for dynamic restructuring and provides better performance in terms of the number of messages required and response time, as compared to the traditional two-phase commit protocol.

The architecture of a cell-based distributed operating systems is addressed in the following chapter.
Chapter 7

SYSTEM ARCHITECTURE AND PROTOCOL STRUCTURE

This chapter addresses how to construct a fault-tolerant distributed operating system (FTDOS) based on the cell model. Design goals, system structure, and various design issues of this distributed operating system are discussed. The protocol structure of this distributed operating system is also presented.

In Section 7.1, design goals of the system are described. The system structure is described in Section 7.2. Various design issues of this fault-tolerant distributed operating system such as naming, communication, protection, and fault tolerance are addressed in Sections 7.3, 7.4, 7.5, 7.6, respectively. The protocol structure of this cell-based distributed operating system is presented in Section 7.7. Finally, a summary of this chapter is given in Section 7.8.

7.1 Design Goal

The primary design goals of a cell-based fault-tolerant distributed operating system are three-fold:

- to support a reliable and resilient distributed computing environment which provides a uniform coherent view of
distributed resources and services for users,
- to meet user's performance constraints on both throughput and response time when a local or remote service is requested, and
- to design the system in a modular fashion so that addition, modification and installation of system functions can be easily performed.

To achieve the aforementioned goals, it is necessary to resolve the problems of distributedness and heterogeneity, to incorporate the features of fault tolerance and information protection, and to meet the requirements of extensibility and performance. An FTDOS structure based on the cell model is presented in the next section which achieves these goals.

7.2 System Structure

According to the cell model, a distributed operating system can be viewed as a collection of cells, each of which performs a set of predefined system functions. More than one cell may reside in a node. Figures 32 and 33 show physical and logical views of a cell-based DOS in a local computer network, respectively.

A user request is performed by a single cell or a group of cooperating cells which implements the concept of nested atomic actions. The service requested must be either completely done or not done at all in spite of various failures and faults. This group of cooperating cells is viewed as a virtual machine. Therefore, the DOS can be viewed as a set of virtual machines which provide a set
Figure 32: A Physical Structure of the Cell-based DOS
Figure 33: A Logical Structure of the Cell-based DOS
of predefined services for users.

A cell-based distributed operating system consists of a set of virtual machines, each of which is formed by one or more cells. Each cell consists of various entities (processes and objects). According to the system structure, the following design issues need to be resolved:

- **naming**
  
  how to name an entity (such as object) in the system? how and when to bind a name to a physical object?

- **synchronization**
  
  how to control concurrent accesses to shared objects in the system?

- **communication**
  
  how to communicate with one another among system entities (such as processes)?

- **protection**
  
  how to protect information from being illegally accessed during store and transmission?

- **fault tolerance**
  
  how to guarantee an acceptable system behavior when failures occur?

These design issues, except the synchronization problem, which can be resolved based on the concurrency control algorithm proposed in the cell model, are discussed in Sections 7.3, 7.4, 7.5, and 7.6, respectively.
7.3 Naming

There are two kinds of entities in a cell-based system: messages and cells. A request to a cell may contain several related message exchanges between a client and server cell while a cell may consist of several related entities (processes and objects). Messages can be identified according to request identifiers and message types. A good naming scheme for request identifiers should incorporate the semantic information about the structure of requests in a transaction. The following are possible implementations of request identifiers:

- timestamps plus the information about the structure of requests in a transaction:

  Figure 34.a shows an example of this implementation. Each transaction has a unique identifier: its timestamp. A request identifier consists of the timestamp of its transaction concatenated by its generation number in the transaction.

- concatenating timestamps:

  Figure 34.b shows an example of this implementation. A request identifier consists of the identifier of its immediate ancestor concatenated by a new timestamp.

Message types include the type of request, acknowledgment, reply, commit, abort, and inquiry, as explained in the following:

- request:

  a message sent by a client to a server for requesting a service

- acknowledgment:
Figure 34: Naming of Request Identifiers
a message sent by a server back to its client to indicate the validation of the request

- reply:
  a message sent by a server back to its client to indicate the success or failure of a request

- commit:
  a message initiated by a top-level cell to commit a transaction

- abort:
  a message initiated by a higher-level cell to abort some committed services at lower-level cells

- inquiry:
  a message sent by a cell to a related cell for checking the result or status of a request

A cell may consist of several entities (manager, agents, and protected objects). The identifier of a cell is the identifier of the manager in that cell. This unique identifier can be generated based on an eventcount in each node which observes a common protocol and uses stable storage to survive node crashes and to ensure its uniqueness. A scheme for generating a unique identifier, based on capability, is described in [WYLE79].

A distributed name server, responsible for mapping a high-level cell name into a unique system-wide identifier, is located at each node. The high-level names are given based on user convenience whereas the low-level names are used for addressing. The name server at each node maintains a fully replicated database which maps higher-level names into unique identifiers. Therefore, the update
of the database is necessary whenever a cell is created, terminated, or migrated from one node to another node. According to this two-level name space, only cell identifiers are maintained and known system-wide. The identifiers of intercell objects are only locally known and maintained by cell managers.

7.4 Communication

Two levels of communication are provided in the system: cell and process. The intercell communication is based on the client/server model. The communication primitives, invoke and wait, are used to initiate an intercell operation and to guarantee the atomic property of an intercell operation as described in Chapter 6.

Interprocess communication is based on message passing. A special object, port, is used to provide communication between two processes. A port is a mailbox which can be viewed as a logical address of a process, and a process may have more than one port to exchange information with other processes.

The primitives to support interprocess communication may include the following:

- open (a port)
  to open a port for a process
- close (a port)
  to close an already opened port
- send (a message to a port)
to send a message to a port
- read (a message from a port)
to read a message from a port
- check (messages on a port)
to check if there is any message or how many messages are on a port

7.5 Protection

Information protection is mainly concerned with the problems involved in the protection of information being stored or being transmitted. The former, which includes unauthorized reads and writes, is concerned with distributed access control. The latter, the problem of protection during transmission, has been addressed largely through encryption. The studies in encryption have led to designs which not only protect data against a passive eavesdropper but also protect against fraudulent messages.

The access control problems in a cell-based distributed operating system rely on the following concepts:

- multi-level protection
- capability-based protection

In this capability-based protection system, the capability validation is required in a cell whenever a function is requested or an object is being accessed. In a group of cooperating cells, each request to a cell will be checked against its capabilities; this
forms multi-level protection lines. Encryption can also be applied to protect the capability being stolen during transmission.

To protect information from being illegally accessed during transmission, encryption can also be used.

7.6 Fault Tolerance

To protect against all possible faults in a non-trivial system is not feasible or, at least, is not economical. Therefore, our strategies for achieving fault tolerance are based on the following concepts:

- to survive detectable faults or failures
  to detect anticipated failures/faults such as node crash, communication failures, network partition, and some residual software design faults and to correctly restore system state
- to confine undetected faults or failures, and to prevent the propagation of their damage
  to design a system to have the property of error confinement, so that various errors and damages will be confined to a single cell

Based on the aforementioned concepts, the following fault tolerance techniques in the FTDOS have been developed:

1. detecting node failures or network partitions and reconfiguring the system
2. restarting from node crashes
3. surviving node crashes or network partitions
4. tolerating residual design faults
5. achieving damage confinement

The detailed concepts and operations are described in the following subsections.

7.6.1 Detecting Node Failures and Network Partitions, and Reconfigurating the System

The purpose of system reconfiguration after node crashes or network partitions is to improve efficiency of system operations and to guarantee acceptable system behavior.

As mentioned before, there is a cell in each node responsible for system configuration. This function could be performed by a dedicated cell or by a cell which also performs other functions. This cell maintains three sets of information in stable storage: (1) an eventcount $E$ and its pre-assigned node number, and (2) a list of reachable nodes. This list of reachable nodes contains a set of node numbers which can be reached directly or indirectly through the local node. Any node which cannot be reached by the current node due to a node crash or network partition will be removed from the list. If a new node joins the group, its node number will be added into the list.

Upon detecting a crashed, unreachable, or newly joined node, a node will send a reconfiguration message to all nodes in the system to start a system reconfiguration. There are three modes of reconfiguration: shrinking, growing, and mixed. The shrinking mode
indicates that either a node is crashed or the network is partitioned so that the number of reachable nodes is decreased. The growing mode indicates that either a new node joins the partition or several partitions have been merged so that the number of reachable nodes is increased. The mixed mode indicates a combination of shrinking and growing modes. The node which detects this change and initiates a system reconfiguration is called the initiating node. It is possible to have more than one initiating node which detect a configuration change. During the system reconfiguration, each node in the group will observe a reconfiguration protocol. The protocol is described as follows:

1. initiation phase:

Once a node detects a change in the system configuration, it will broadcast a reconfiguration message to all nodes it can reach and create new and lost node lists according to the configuration change. The new node list contains newly joined node numbers whereas the lost node list contains the disjointed node numbers.

2. consensus phase:

Once a node in the group receives a reconfiguration message from an initiating node, it will send an acknowledgment message back to the initiating node and enter the consensus phase. This message should contain its intended reconfiguration coordinator its eventcount, and new and lost node lists. The initiating node of the first reconfiguration message a node receives is usually selected as its intended coordinating node. If there are more than one reconfiguration message from different initiating nodes, only one intended coordinator can be selected by a node. This restriction will force only one coordinating node to be selected in a system reconfiguration although it may be initiated by more than one node. When the initiating node receives all
acknowledgment messages from every member node in the group, a reconfiguration coordinator will be selected according to the voting from the nodes in the group. If more than one node has the highest vote, the node with the lowest pre-assigned node number will be selected. The reconfiguration coordinator is the node with the highest votes and is usually the initiating node which first initiates the system reconfiguration (if there is more than one initiating node). The highest eventcount from member nodes will be selected as the reconfiguration version number. A final list of reachable nodes which reflects the recent configuration is constructed during this phase by the coordinator. It is generated based on new and lost node lists from group members according to the following rules:

- Let \( n_i \) denote the new node list from node \( i \) and \( l_j \) denote the lost node list from node \( j \). Then sets \( N \) and \( L \) at the initiation node can be defined as the union of all new node lists and lost node lists from member nodes, respectively.

\[
N = \bigcup_i n_i
\]

\[
L = \bigcup_i l_i
\]

- Let \( 0 \) be the original list of reachable nodes. Then the new list of reachable nodes can be generated according to the following:

\[
\text{new list} = 0 \cup N - L
\]

Where: \( \cup \): union operation
\( - \): difference operation

3. notification phase:

The initiating node will notify every node about the reconfiguration coordinator based on the voting it receives.
4. update phase:

The coordinating node will update the list of reachable nodes and the eventcount among new group members based on the new list of reachable nodes and the reconfiguration version number constructed at the consensus phase, respectively.

During a system reconfiguration, each member node temporarily halts all timestamp assignments (i.e., it temporarily halts processing all new requests until the reconfiguration is complete).

The following are two abnormal cases which the reconfiguration protocol should be able to deal with:

- multiple initiating nodes

A configuration change in the system may be simultaneously detected by more than one node. It may cause redundant reconfiguration processes.

- configuration change during the reconfiguration process

A node may crash, partitions may be merged, or a crashed node may be repaired and joins the group in the process of system reconfiguration.

The reconfiguration protocol can resolve the former problem (multiple initiating nodes) very easily. If a configuration change is initiated by more than one node, only one coordinator is selected to perform the configuration update; thus, it avoids causing redundant reconfiguration.

According to the reconfiguration protocol, consecutive configuration changes can be merged and performed at once. Therefore, the latter problem can be resolved.

In order to resolve the problem caused due to the failure of
the reconfiguration coordinator, all member nodes will periodically poll the initiation node, which it voted for, before the notification phase and poll the reconfiguration coordinator after the notification phase. If a node failure has been detected, it should initiate another reconfiguration message.

7.6.2 Restarting from Node Crash

To correctly recover from a node crash, the following restarting protocol should be followed:

1. initialize node.

2. restore the cell responsible for configuration (we need to restore the eventcount, the pre-assigned node number, and initialize the list of reachable nodes to contain the node itself only).

3. broadcast a reconfiguration message to other nodes to initiate a system reconfiguration according to the reconfiguration protocol described above.

4. initiate all cell recovery in the node once the reconfiguration is complete.

5. resume normal processing once the recovery is complete.

7.6.3 Surviving Node Crashes or Network Partitions

In order to survive node crashes or network partitions, the redundancy in the services or resources provided is necessary. Figure 35 shows that a service has been provided in different cells which are located in different nodes. The invoke primitive allows users to detect a node crash or network partition and to submit
another request to its backup cell. Similarly, the concept of recovery block can be easily incorporated into cells. Multiple copies of cells, which exist among different nodes, can be used to tolerate node crashes or a network partition as shown in Figure 36.

7.6.4 Tolerating Residual Software Design Faults

As described above, the recovery block can be easily incorporated into the design of a computation in a cell. It allows designers to specify multiple versions of software in an agent to detect and remove residual design faults. These software spare parts (additional versions) provide additional reliability and availability. Figure 37 shows an example which illustrates how to incorporate the recovery block into an agent so that it can survive residual design faults. This concept can be extended to several levels.

7.6.5 Achieving Damage Confinement

The capability-based protection and the independent address space of a cell provide a very good attribute of error confinement within a cell. The checking of information flow among cells is strictly enforced. This prevents unauthorized information from being transferred between different cells.
Invoke

Primary cell

Backup cell

Figure 35: Service Redundancy
Ensure «acceptance test»
By (invoke a computation at node X)
Else By (invoke a computation at node Y)
Else abort

Figure 36: Application of Recovery Block
Ensure «acceptance test»
By (algorithm 1)
Else By (algorithm 2)
Else abort

Figure 37: Application of Recovery Block
7.7 Protocol Structure

An extensible system is thought of as a system which can be easily adapted to a changing environment without disrupting its functioning. Basically, there are two kinds of environmental changes:

- due to performance requirement
  - including the replacement or addition of underlying component systems to improve the overall system performance
- due to functional requirement
  - including the addition of new system functions and the changing of existing functions

The system must be structured in a modular fashion so that these environmental changes can be performed easily and the design of the system need not be changed. In order to achieve this extensibility, the cell-based distributed operating system is designed based on the following concepts: (1) layered approach, and (2) recursive structuring.

The protocol structure of the system consists of three layers: (1) interprocess communication (IPC) layer, (2) service support layer, and (3) service layer as shown in Figure 38. The IPC layer supports the abstractions of ports and messages for interprocess communication. Both of the transport and session services in the ISO's open system interconnection (OSI) can be provided. The service support layer supports the abstraction of cells which is the
fundamental framework of our distributed operating system. The service layer supports the abstraction of reliable virtual machines which provide various reliable services to the users.

Detailed functions and structures of each layer are addressed in the following subsections.

7.7.1 Service Layer

The service layer supports the abstraction of reliable virtual machines. It provides reliable services to the users. All services have the atomic property: either completely performed or not performed at all in spite of various failures.

A group of cooperative cells forms a reliable virtual machine. The services provided by the virtual machine actually are those functions defined in the top-level cell of the group. Figure 39 shows an example illustrating the relationship between a reliable virtual machine and cells. Each node in the figure indicates a cell and the dashed circle represents a reliable virtual machine. Each computation defined in the virtual machine is an atomic action. Therefore, each computation (service) defined in the highest-level virtual machine represents a tree of nested atomic actions, called the agent tree.

The granularity of the virtual machines can be very flexible. It could be designed as small as a one node tree such as a single cell or a typed object, or as large as a tree-like structure which
Figure 38: Protocol Structure
Figure 39: A Reliable Virtual Machine and Cells
consists of a large number of cells located in different computers.

7.7.2 Service Support Layer

The service support layer supports the abstraction of cells and inter-cell operations (primitives). It is used to implement the concept of nested atomic actions in a distributed operating system environment.

This layer needs to support the concurrency control (synchronization) algorithm, recovery algorithm, communication primitives, commit protocol, and protection mechanisms based on the underlying interprocess communication.

The concurrency control algorithm is based on timestamping, type-specific locking, and an adaptive approach. Requests within the same transaction are manipulated based on the optimistic method whereas requests in different transactions are handled based on the pessimistic locking. The cell recovery is performed based on the do-undo-redo paradigm. A special facility called active request log plays an important role in the recovery and commitment. Intercell communication relies upon invoke and wait primitives. The commit protocol, which consists of local and global commit, can accommodate dynamic restructuring and provide better performance than the traditional two-phase commit protocol. Capability based protection prevents the information or function from being illegally accessed or utilized.
7.7.3 IPC Layer

The IPC layer supports the abstractions of ports and messages. Messages are exchanged between network addresses, viewed as ports on processes. A port is a special object and conceptually is a mailbox or a queue. A process may have more than one port, and ports in a process may have different priorities. Figure 40 shows an example of interprocess communication based on ports.

In the cell model, each cell manager has at least one port for external requests. The transport service can be provided based on send, receive and check primitives. It can be used to provide session services by opening and closing dedicated ports between processes.

7.7.4 Request Scenario

In order to explain a request scenario in different layers, an example is shown in Figure 41. A user program submit a system call to the distributed operating system for requesting a distributed service, e.g., file transfer. This request will be sent to a top-level cell which is responsible for file transfer. The top-level cell will coordinate with other cells at source node X and destination node Y to perform the file transfer function.

This distributed service at the application layer can be viewed as in Figure 42. In this example, a distributed service is requested by a user and being sent to a black box, a distributed
Figure 40: Interprocess Communication Based on Ports
Figure 41: An Example for Request Scenario
operating system. Once it either successfully finishes or unfortunately aborts the service, a reply will be sent back to the user.

This distributed service at the service layer can be viewed as in Figure 43. The service is submitted to a top-level cell which coordinates with other related cells to cooperatively perform the service for the user. Once the service is done, its reply will be sent back to the user by the top-level cell.

The distributed service at the service support layer can be viewed as in Figure 44. The service is performed by a group of cooperating processes which access related objects and communicate with one another based on interprocess communication.

The distributed service at the interprocess communication layer can be viewed as in Figure 45. The service is performed by a series of interactions between the interprocess communication facilities at nodes X and Y which cooperatively manipulate interprocess communications among related ports.

7.8 Summary

This chapter described design goals of a fault-tolerant distributed operating system and how to use the cell model to design a fault-tolerant distributed operating system. Various design issues such as naming, synchronization, communication, protection, and fault tolerance were addressed. The protocol structure of this
Figure 42: The View at the Application Layer
Figure 43: The View at the Service Layer
Figure 44: The View at the Service Support Layer
Figure 45: The View at the IPC Layer
cell-based distributed operating system was also presented.

Robustness, extensibility, and performance are main concerns of this distributed operating system. According to the layered approach and recursive structuring, a distributed operating system is designed in terms of cells. A group of cooperating cells which implement the concept of nested atomic actions forms a reliable virtual machine. This reliable virtual machine provides various distributed services for users.

Each entity in the system has a unique identifier, and multilevel naming can be constructed on top of the unique system-wide identifiers. The synchronization, communication, and protection issues on cells are based on the mechanisms provided in the cell model. Interprocess communication is based on the abstractions of ports and messages. The reconfiguration and restarting protocols were proposed to survive node crashes and network partitions. Recovery blocks were used to survive node crashes, network partitions, and residual software design faults.

The protocol structure was divided into three layers: service, service support, and interprocess communication layers, from the top to the bottom. Applications can be developed on top of the DOS. The performance issue of this distributed operating system is addressed in the next chapter.
Chapter 8

PERFORMANCE ISSUES

In the previous chapters, we have proposed the cell model as the fundamental framework to construct a fault-tolerant distributed operating system. The mechanisms for supporting the cell model have been discussed. Design issues of this cell-based distributed operating system such as naming, synchronization, communication, protection, and fault tolerance also have been addressed. Performance issue is discussed in this chapter.

Section 8.1 analyzes the overhead incurred in a cell-based system. Overhead can be classified into time and space overheads. Since a distributed operating system provides a set of predefined service for users upon request, the performance can be measured in terms of the performance of each service. Section 8.2 proposes two parameters to measure the system performance: response time of a system call and storage overhead associated with a system call, which are used as the basis of performance measurement in this chapter. Approaches which can be taken to improve the system performance are presented in Section 8.3. They include computation structuring, architectural support, and algorithm enhancement.
Finally, a summary of this chapter is given in Section 8.4.

8.1 System Overhead

In a cell-based system, a computation (system call) is structured into a tree-like structure. The major overhead incurred from the implementation of nested atomic actions includes the following:

- Processing time for establishing recovery points
  It includes the time to record various recovery logs (undo, redo, and active request logs) and to perform checkpointing.

- Stable storage for storing recovery logs
  It includes the storage to store the undo, redo, active request logs, and checkpoint records.

- Exchanging messages for coordinating nested atomic actions
  It includes the time to submit requests, to wait for replies, and to coordinate commitments.

The overhead can be categorized into time and space, and they usually depend upon the following design factors:

- structure of the agent tree
- implementation of nested atomic actions
- checkpoint intervals

In general, the overhead in processing time depends upon the structure of nested atomic actions. The greater the number of nested atomic actions (or number of nested levels) is, the more
overhead in processing time is introduced. The amount of storage overhead is also slightly affected by the structure. The algorithms and protocols used to implement the concept of nested atomic actions also have impacts on the overhead incurred. The more frequently the checkpoint is taken, the greater overhead in processing time is introduced.

8.2 System Performance

There are a variety of parameters which can be used to measure system performance. Since a cell-based distributed operating system is intended to provide services for each system call from users, its performance can be simply measured in terms of the following parameters:

- response time of each system call
- storage overhead associated with each system call

Response time of a system call can be defined as the duration from when a system call is issued to when either its reply has been received or its service has been confirmed. The storage overhead associated with each system call usually refers to the additional storage required in the processing of the service. It usually refers to the stable storage space required for storing recovery logs for a system call.
8.3 Performance Enhancement

As mentioned above, the system performance can be measured in terms of response time and storage overhead associated with each system call. In some applications such as real-time systems, performance is an important design goal. Careful analyses and calculations of various design parameters to meet the requirement of system performance are an essential step during the design process. The following are several approaches which can be taken to enhance the system performance:

- properly structuring a computation
- enhancing the algorithms or protocols for supporting the cell model
- architectural support for the cell model

These approaches are addressed in Subsections 8.3.1, 8.3.2, and 8.3.3, respectively.

8.3.1 Computation Structuring

A computation (system call) can be implemented by a group of nested atomic actions. By properly structuring the tree of nested atomic actions, response time of this system call can be improved. In this subsection, the impact of structuring of this agent tree on the system performance is investigated.

For the purpose of simplicity, it is assumed that the agent tree of the system call is a complete binary tree and the time and
space overhead associated with each nested atomic action are the same. At first, several symbols are defined:

\[ \begin{align*}
1 & : \text{number of nested levels} \\
n & : \text{number of nested atomic actions} \\
a & : \text{time overhead per nested atomic action} \\
s & : \text{space overhead per nested atomic action} \\
Q & : \text{total processing time for the computation} \\
t & : \text{transmission time for an intercell message}
\end{align*} \]

For a complete binary tree,

\[ n = 2 - 1 \quad \text{or} \quad \log_2(n+1) \]

If all computation in a system call can be concurrently processed, the response time of a system call can be defined as:

\[ \text{response time} = \text{processing time} + \ \\
\text{message exchange time} \]

\[ = 1 \cdot (Q/n + a) + 3t \cdot (1 - 1) \]

\[ = (Q/n + a + 3t) \cdot \log_2(n+1) - 3t \]

In this example, we assume that only three messages are required to invoke a nested atomic action: one for the request message, one for the acknowledgment from the server, and one for the reply message. The time needed for exchanging intercell messages is assumed to be the same and the processing and waiting times for message validation at a server are negligible. The storage overhead
can be shown as follows:

\[
\text{storage overhead} = n*s
\]

In reality, the assumption that every step of the computation can be concurrently processed is not feasible. The aforementioned assumption can be refined as follows: only \( r \) percentage of computation can be concurrently processed. The equations for processing and response time can be changed into:

\[
\text{processing time} = (1-r)*Q + 1*(r*Q/n+a)
\]

\[
\text{response time} = (r*Q/n+a+3t)*\log(n+1) + \frac{(1-r)*Q-3t}{2} \tag{a}
\]

We assume that \( b = (a*n)/Q \). It indicates the average overhead ratio per nested atomic action. Figure 46 shows the relationship between response time and the number of nested atomic actions with different combinations of \( b \) and \( r \) values when \( t \) is negligible. According to the figure, response time may be improved a lot if \( r \) is large in the computation. When the value of \( t \) becomes significant, however, the response time may increase as number \( n \) increases.
Figure 46: Response Time and Other Parameters
8.3.2 Algorithm Enhancement

The enhancement of the algorithms and protocols for supporting the cell model such as synchronization (concurrency control), recovery, communication, and commitment may improve the total performance.

The response time of a system call will be affected by the following factors:

- **Processing time:**
  It includes processing time for normal functions and processing time for overhead incurred due to nested atomic actions.

- **Message exchange time:**
  It includes the time required for exchanging messages among nested atomic actions during commitment.

- **Waiting time:**
  It includes the waiting time for accessing an object or the waiting time for being served.

- **Degree of parallelism:**
  Some processing time, message exchange time, and waiting time in a transaction may be overlapped to reduce response time depending upon the degree of parallelism.

Figure 47 shows how these algorithms affect the response time of a system call. These topics are addressed as follows:

- **Concurrency Control Algorithm:**
  A good concurrency control algorithm can reduce the waiting time for accessing a shared object and increase the degree of parallelism.
- Recovery Algorithm:

A good recovery algorithm can speed up the failure recovery process and may possibly reduce the overhead associated with each nested atomic action for both time and space.

- Communication Protocols:

A good intercell communication protocol can increase the degree of parallelism without sacrificing its robustness. Therefore, response time of a system call can be improved.

- Commit Protocols:

A good commit protocol can reduce the number of message exchanges required in a commitment and can avoid delaying the commit actions if delay is not necessary.

8.3.3 Architectural Support

Another approach to improving system performance is to reduce "a" or "t" values in equation (a); in other words, to reduce the time overhead associated with each nested atomic action and intercell message exchange. Design of an operating system kernel with architectural support can help achieve this goal and reduce the overhead incurred.

The design goal of this kernel is to support the mechanisms of the cell model. The kernel of the operating system consists of codes that are intensively and commonly used by all software at higher levels. It can be used to support higher level mechanisms, thereby reducing the complexity of designing and implementing the nested atomic action.
Figure 47: Response Time and the Algorithms
Recent dramatic progress in VLSI and firmware technology has given us a bright prospect for implementing the kernel. The architecture based on the kernel is shown in Figure 48. On top of a bare machine, a DOS kernel, consisting of a set of primitives and data structures, forms an extended machine to support high level functions required in the cell model. DOS kernels, which reside at each node, communicate with one another through the communication network.

In addition to common functions supported by traditional OS kernels (nucleuses) such as interrupt handling, low-level synchronization and memory management, this kernel should at least provide the following primitives:

- process management
  It includes creating, deleting, modifying, and scheduling processes. At least the following two levels of process scheduling should be supported: cell and process.

- process communication and coordination
  It includes interprocess communication primitives based on messages and events.

- capability management
  It includes creating, deleting, modifying, transferring and reading capabilities.

- log management
  It includes various primitives to manipulate recovery logs.

The architectural support for the above primitives can
Figure 48: The Architecture Based on the DOS Kernel
facilitate message exchanges, process management, communication, cooperation and swapping, domain management, and log processing.

8.4 Summary

This chapter discussed performance issues of a cell-based distributed operating system. The overhead incurred due to the implementation of nested atomic actions also was addressed. The response time of a system call and its associated overhead in stable storage are used as system parameters to measure the performance.

Several approaches have been proposed to improve the system performance. They include computation structuring, algorithm enhancement, and architectural support. The approach to improving system performance based on computation structuring is usually dependent upon application characteristics. The approach to enhancing algorithms relies on providing better underlying mechanisms and protocols to support the cell model. They include concurrency control, recovery, communication, commitment, and protection mechanisms. Effective architectural support can also provide a significant enhancement on system performance.
Chapter 9

SUMMARY AND FUTURE WORK

We have presented in the previous chapters the basic concept of the cell model, the underlying mechanisms for supporting the model, and the architecture of a cell-based distributed operating system. Various design issues such as naming, concurrency control, communication, protection, performance, and fault tolerance were also addressed. In this chapter, significant features of the cell model, the structure of a cell-based distributed operating system, and the underlying mechanisms for supporting the model are summarized. A simplified version of the model, which has been used to design communication link software to connect a DEC-20 and an Amdahl 470/V8 at the Ohio State University, is also described. Finally, areas for future research are suggested.

9.1 The Cell Model and its Applications

The rapidly decreasing cost of processors in the past decade has removed the need of concentrating computing power in a centralized location and has given an economic incentive for distributed processing. This new flexibility in system design
enables the functions of a complex system to be divided physically as well as logically. The trend toward the development of distributed systems is becoming more and more obvious, especially in the automation of service and manufacturing industries. The high economic cost of system failure in these applications, however, has created a significant demand for highly reliable computing systems [ANDE81, BART78, KATZ77, RAND79, RENN80]. This demand has spurred the search for a greater understanding of fault tolerance issues in distributed systems.

In general, a fault-tolerant distributed operating system should meet the following requirements:

- to deal with hardware failures as well as residual software faults,
- to have the property of error confinement to confine the damages due to undetected faults, and
- to perform dynamic restructuring to cope with partial failures.

Most existing techniques only meet part of the aforementioned requirements. For example, the recovery block and conversation are used to deal with residual software faults. However, the structures provided by these techniques are static. The N-version programming may be used to deal with hardware and software faults. However, the maximum allowable range used in a check of inexact voting may be difficult to determine and may change between executions of a version. Also, it cannot deal with partial failures. The concept
of atomic actions provides a good conceptual framework for error detection and recovery. However, the degree of recoverability and concurrency is limited. The nested atomic action has been proposed to remedy this problem. The model proposed by Moss to implement the concept of nested atomic actions is based on the concept of transactions proposed by database designers and aims to deal with hardware failures and to provide dynamic restructuring in case of partial failures.

Main result of this research is to propose the cell model which aims to meet the aforementioned requirements and proposes the underlying mechanisms for supporting the model. The cell model, which combines three basic concepts, modular decomposition, nested atomic actions, and exception handling, is proposed as a conceptual framework to construct a fault-tolerant distributed operating system. Conceptually, the nested transaction model can be viewed as a special case of the cell model when the number of cells in a node is limited to one and no type cell is supported. Figure 49 shows the differences between these two models in terms of the structure.

The characteristics of the cell model are as follows:

- Fault tolerance features:

  The survivability on detectable faults/failures and the error confinement on undetected software faults which a cell-based system inherits can improve the system's robustness.

- Use of semantic knowledge:
Since a cell is structured based on the abstractions (the services it provides or the data type it supports), the semantic knowledge about the function and data type can be fully utilized to strictly enforce protection, to improve concurrency, and to provide modularity.

- Migration of software:

The cell structure allows us to migrate software easily from one node to another. Two functions need to be performed during the migration: (1) update the location dependent information about the cell in the name server, and (2) migrate the cell.

- Design flexibility:

Designers have the flexibility to structure the software based on its computation characteristics and environment. The model can be used to construct a distributed operating system on top of existing local operating systems or to construct a global operating system from scratch.

According to the cell model, a fault-tolerant distributed operating system can be decomposed into a group of cooperating cells. This modular decomposition intends to decompose a complex system into several small and manageable subsystems according to their computation characteristics. Each cell performs a set of predefined services. The services defined in each cell are atomic: either successfully performed or not performed at all in spite of failures. A user request may be performed by a single cell or by a group of cooperating cells, with one of them serving as a coordinator (top-level cell). The former case implements the concept of atomic actions whereas the later case implements the concept of nested atomic actions.

Each cell aims to have the following fault tolerance aspects:
- the capability to tolerate anticipated failures/faults such as node failures, medium failures, and communication failures, and some unanticipated failures/faults such as failures due to residual design faults, and
- the property of error confinement.

In order to support the cell model, the following underlying mechanisms are required:

- concurrency control algorithm
to control concurrent accesses to shared objects in a cell and to guarantee the correctness and consistency of objects with an aim to improve the degree of concurrency without causing the domino effect.
- recovery algorithm
to correctly restore an object or a cell state when failures occur.
- communication protocol
to reliably and efficiently perform intercell communication.
- commit protocol
to achieve the atomic property among a group of cooperating cells.
- protection mechanism
to prevent unauthorized access to an object or function within a cell and to confine damages within a cell.

Figure 50 shows how the aforementioned mechanisms support the requirements of fault tolerance in the cell model.

According to the cell model, one of the cells in each node is responsible for maintaining system configuration. This cell maintains several data items in stable storage such as eventcount,
Figure 49: Transactions and Cells
Fault tolerance

Survivability on detectable faults/failures

Error confinement on undetected software faults

Nested atomic actions

Redundancy

Protection

Concurrency control

Recovery

Communication

Commitment

Figure 50: Fault Tolerance Features of the Cell Model
pre-assigned node number, and a list of reachable nodes. This cell also will detect node crashes and network partitions, and perform system reconfiguration.

There are two approaches to implementing a distributed operating system: (1) the guest layer approach, and (2) the global operating system approach. In the first approach, a distributed operating system is constructed on top of the local operating system residing in each node and provides distributed services for users. This approach can take advantage of existing software, thereby greatly reducing the amount of workload on software development. However, the heterogeneity of its underlying incompatible subsystems usually increases the complexity of problems and also imposes some limitations on system design. According to the cell model, the software architecture based on this approach can be shown in Figure 51.

In the second approach, a distributed operating system is constructed on top of bare machines. The global operating system approach requires a great deal of development effort, but provides an opportunity for designers to implement a homogeneous kernel which will facilitate the overall software development. According to this approach, a cell-based distributed operating system can be constructed based on the following layers from the bottom to the top: DOS kernel, IPC layer, Service Support Layer, and Service Layer as shown in Figure 52.
Figure 51: The Guest Layer Approach Based on Cells
Figure 52: The Global Operating System Approach Based on Cells
9.2 Underlying Mechanisms of the Cell Model

The underlying mechanisms required for supporting the cell model include concurrency control, recovery, communication, commitment, and protection. In the previous chapters, we have discussed some existing and novel algorithms to implement the mechanisms. In summary, they include the following:

- Concurrency control:

  The algorithms include basic timestamping, multi-version timestamping, pessimistic locking, optimistic methods, and adaptive approaches (based on semantic knowledge).

- Recovery:

  The mechanisms include the do-undo-redo paradigm, checkpointing, and other recovery facilities such as active request logs.

- Communication:

  The mechanisms available include traditional message passing, remote procedure call, and invoke/wait primitives.

- Commitment:

  The commit protocols available include the two-phase commit protocol and the proposed distributed partial commit protocol.

- Protection:

  The mechanisms mainly rely on capability-based protection, access control, and encryption.
9.3 Experiment

In this section, a simplified version of the cell model, which has been used to construct communication link software to connect a DEC-20 and an Amdahl 470/V8 at the Ohio State University, is described in terms of system functions, implementation, and results.

The purpose of this experiment is stated as follows:

- to obtain initial experiences on the implementation of the cell model
- to analyze the performance issues
- to demonstrate the feasibility of the model

9.3.1 System functions

The communication link system aims to connect a DEC-20 which runs the TOPS20 operating system and an Amdahl 470/V8 which runs MVS/JES2 as shown in Figure 53. The internode communication will observe the protocol of the minicomputer link facility (IBM JES2 compatible commands) at the Amdahl. According to this requirement, the DEC-20 has the provision to communicate with any campus computer which is connected to the Amdahl and observes the same protocol.

The functions provided by the link system include the following:

- Exchanging files between the DEC-20 and the Amdahl 470, the IBM 4341, and all of a hundred or so other computers on campus which now use the minicomputer link facility.
Figure 53: The System Configuration
- Printing of DEC-20 files on any of the IRCC printers, including provisions for special forms, etc. It also supports the output produced by DEC-20 text formatting facilities such as RUNOFF, SCRIBE, etc.

- Submission of batch jobs to the Amdahl from the DEC-20, with default routing of the output back to user's DEC-20 terminal. Users may check on the status of jobs currently running or cancel them. Held output may be selectively retrieved.

- Routing of output to the DEC-20 from jobs submitted elsewhere, such as TSO, WYLBUR, CMS, cards, minicomputers, etc.

- Automatic notification of job completion via screen messages and mail.

- Management of Amdahl disk data sets from the DEC-20.

- Access to other DEC-20 facilities from inside the link program such as MM, EMACS, and other DEC-20 commands.

- Availability even when the Amdahl is down. Jobs and commands are automatically queued up by the DEC-20 and sent as soon as the Amdahl comes up.

- On-line help information available on various subjects and commands while using the link.

The fault tolerance aspects of the link system include the following:

- automatic reconfiguration:

The system should be able to detect the Amdahl crash or the broken link and to automatically queue up all outgoing requests. The requests will be automatically sent out and processed as normal once the remote node or the link is repaired and up again.

- automatic restarting:

When the DEC-20 is repaired and up again after crash, the link system will automatically restore its state.
The software architecture and the underlying mechanisms of this implementation are described in the next subsection.

9.3.2 Implementation

In the implementation, the minicomputer link facility (and the MVS/JES2) at the Amdahl is viewed as a server which performs the service needed to be performed in the Amdahl. The software architecture of the link system can be simplified as shown in Figure 54. The link system can be viewed as consisting of an MLF (minicomputer link facility) server at the Amdahl, Queue and Link cells at the DEC-20, and a mail server at the DEC-20.

According to this architecture, a request from users will be received at the Queue cell. An appropriate authorization check is enforced at both cells. When a Queue cell finishes its internal task and before its commitment, subrequests will be submitted to the Link cell for link services. Once an acknowledgment from the Link cell has been received, it can proceed to process other requests. When a link service has been performed at the Amdahl, a reply will be sent back to the Link cell, and finally, it will reach the Queue cell. At this moment, the Queue cell will send a subrequest to the mail server to notify users of the completion of the service. Once the mailing service has been performed, the user request will commit.

In this implementation, the Queue cell serves as a top-level
Figure 54: The Simplified Architecture of the Link System
cell and the Link cell is responsible for maintaining the system configuration. The underlying mechanisms in the cells are described in the following:

- concurrency control:

Users requests are performed based on first come first served in a cell. No priority has been assigned to different requests. Since there is only one shared logical object: a service queue in the Queue cell, the concurrency control becomes quite simple in this case. However, there may be many active requests in a cell at the same time.

- recovery:

The do-undo-redo paradigm is implemented based on the shadow version approach. In order to improve the concurrency, the concept of compensation is also used to undo a request. No cell checkpoint is implemented.

- communication:

The intercell communication is based on the send-and-wait-for-acknowledgment semantic. The Queue and Link cells support multiple ports with different priorities.

- commitment:

The MLF is not constructed based on a cell. In this implementation, if a request is successfully performed, a reply will be sent back to the link cell; otherwise, no reply will be sent from the MLF. To remedy this problem, a timeout mechanism is incorporated into the Queue cell to abort this request should the service fail to succeed at the MLF server. The typical commit sequences are shown in Figure 55.

- protection:

The checking of the invocation to each cell is strictly enforced. Unauthorized requests are rejected upon reception.
Figure 55: Typical Commit Sequences of Link Request
9.3.3 Results

The first version of the link system has been running since May 1983 at the Ohio State University. Although there are some restrictions on the system due to the inter-node communication protocol, the fault tolerance aspects of automatic configuration and restarting have been quite successful. The overhead introduced by this simplified version of the model is acceptable in the environment, and the following are some reasons:

- Low speed:

  The speed of the link is relatively low (maximum allowable speed is 9600 baud due to the speed limit of the communication ports on both machines) compared to the speed of disk accesses. The response time of most requests is dominated by inter-node communication.

- Simple environment:

  The implementation environment is quite simple; therefore, the underlying mechanisms for supporting the model are simple and easy to manage.

9.4 Areas for Future Work

In this dissertation, the basic concept of the cell model and its underlying mechanisms have been proposed and discussed. Some existing and novel algorithms and protocols can be used to support the model. Various design issues to construct a fault-tolerant distributed operating system based on the cell model have also been addressed. However, there are some additional areas of interest
that are worthy of further investigation. The following extensions of this work should give us a more complete picture and information on the implementation of nested atomic actions:

- Implementation of a distributed operating system based on the cell model

In this research, only a simplified version of the model has been used to construct a communication link software system. It provides us with some initial experience on implementing the cell model. However, a full scale implementation of a cell-based distributed operating system will allow us to further investigate various issues and aspects of the cell model. The prototyping system should yield some additional refinement to the design and would be a most fruitful extension of this work.

- Programming language or tool support

The issues about using programming languages and tools for supporting the cell model has not been addressed in the research. However, we anticipate that the design and implementation of a programming language or programming tool for supporting the cell model can significantly reduce development efforts and can facilitate the construction of a cell-based software system.

- Performance issues

Performance has been an important design consideration in the implementation of nested atomic actions. In this research, several approaches have been proposed to improve the system performance such as architectural support, better underlying mechanisms, and computation structuring. The last approach is usually application dependent. Further research on the first two approaches may extend the application of the cell model to real-time environments.
REFERENCES

[ALLC83] Allchin, J. E., and McKendry, M. S.
Synchronization and Recovery of Actions.
2nd ACM SIGACT-SIGOPS Symposium on Principles of
Distributed Computing, August, 1983.

[ANDE76] Anderson, T., and Kerr, R.
Recovery Block in Action: A System Supporting High
Reliability,
Proceedings of 2nd International Conference on

Fault Tolerance: Principles and Practice.

[ATKI77] Atkinson, R. and Hewitt, C.
Synchronization in Actor Systems.
4th SIGPLAN-SIGACT Symp. on Principle of Programming

On the implementation of N-Version Programming for
Software Fault-Tolerance During Program Execution.

[AVIZ78] Avizienis, A.
Fault-Tolerance: The Survival Attribute of Digital
Systems.

[BADA83] Badal, D. Z., and McElyea, W.
A Robust Adaptive Concurrency Control for Distributed
Databases.
Submitted for Publication, October, 1983.

[BART78] Bartlett, J. F.
A 'NonStop' Operating System.
Proceedings of Eleventh Hawaii International
Timestamp Based Algorithms for Concurrency Control.
Proceedings of sixth international conf. Very Large
Data Bases, October, 1980.

[BERS80] Berstis, V.
Security and Protection of Data in the IBM System/38.
Proceedings of seventh Annual Symposium on Computer

[BRIN73] Brinch Hansen P.
Operating System Principles.

[CHEN78] Chen, L.
Improving Software Reliability by N-Version
Programming.
Technical Report ENG-7843, Computer Science
Department, UCLA, 1978.

[COOK79] Cook, D.
In Support of Domain Structure for Operating Systems.
Proceedings of seventh Symposium on Operating Systems

[DENN76] Denning, P. J.
Fault Tolerant Operating Systems.

[ENSL78] Enslow, P. H.
What is a Distributed Data Processing System?

[ESWA76] Eswaran, K. P., Gray, J. N., Lorie, I. L., and
Traiger, I. L.
The Notions of Consistency and Predicate Locks in a
Database System.

[GARC82] Garcia-molina, H.
Using Semantic Knowledge for Transaction Processing
in a Distributed Database.
Technical Report 285, Department of Electrical
Engineering and Computer Science, Princeton
University, June, 1982.
[GRAY79] Gray J. N.
Notes on Data Base Operating Systems.
Springer-Verlag, Berlin, 1979, pages 393-481.

[HABE75] Habermann, A. N.
Path Expressions.
Technical Report, Department of Computer Science,
Carnegie-Mellon University, Pittsburgh, Penn,
June, 1975.

[HOAR74] Hoare, C. A.
Monitors: An Operating System Structuring Concept.

A Program Structure for Error Detection and Recovery.
Springer-Verlag, Berlin, 1974,.

[JONE78] Jones, A. K.
The Object Model: A Conceptual Tool for Structuring
Software.

[KATZ77] Katzman, J. A.

[KIM82] Kim, K. H.
Approaches to Mechanization of the Conversation
Scheme based on Monitors.
IEEE Trans. Software Engineering SE-8(3):189-197,
May, 1982.

An Optimality Theory of Concurrency Control for
Databases.
Proceedings of the 1979 ACM SIGMOD Conference, May,
1979.

[KUNG81] Kung, H. T. and Robinson, J. T.
On optimistic methods for concurrency control.

[LAMP79] Lampson, B. W. and Sturgis, H. E.
Crash Recovery in a Distributed Storage System.
Technical Report, Computer Science Lab., Xerox Palo
[LAMP81a] Lampson, B. W.
Atomic Transactions.

[LAMP81b] Lampson, B. W.
Remote Procedure Calls.

[LEIN82] Leinbaugh, D. W.
High Level Description and Synthesis of Resource
Schedulers.
Proceedings of ACM Annual Conference: 169-175,
October, 1982.

[LIAN75] Lian R. C., and Wang, H. L.
System Software Algorithms in AMICAL Air Reservation
System.
Proceedings of International Computer Symposium

[LIAN83a] Lian, R. C., and Liu, M. T.
Cells: An Approach to the Design of a Fault-tolerant
Distributed Operating System.
Proceedings of Third Symposium on Reliability in
Distributed Software and Database Systems

[LIAN83b] Lian, R. C., and Liu, M. T.
Design of a Fault-tolerant Distributed Operating
System based on Nested Atomic Actions.
Proceedings of AIAA Computers in Aerospace IV

[LIND76] Linden, T. A.
Operating System Structures to Support Security and
Reliable Software.

[LISK82] Liskov, B.
Guardians and Actions: Linguistic Support for Robust
Distributed Programs.
Proceedings of the Ninth ACM SIGACT-SIGPLAN Symposium
on the Principles of Programming Languages :7-19,
[LIU82] Liu, M. T., Tsay, D., and Lian, R. C.  
Design of a Network Operating System for the  
Distributed Double-Loop Computer Network (DDLCN).  

[LIU84] Liu, M. T., Sheth, A. P., and Singhal, A.  
An Adaptive Concurrency Control Strategy for  
Distributed Database Systems.  
Proc. of International Conference on Data Engineering  
April, 1984.  
to appear.

[LOME77] Lomet, D. B.  
Process Structuring, Synchronization, and Recovery  
using Atomic Actions.  

[MCKE83] McKendry, M. S., Allchin, J. E., and Thibault, W. C.  
Architecture for a Global Operating System.  

[MOSS81] Moss, J. E. B.  
Nested Transactions: An Approach to Reliable  
Distributed Computing.  
PhD thesis, Department of Electrical Engineering and  

[MOSS82] Moss, J. E. B.  
Nested Transactions and Reliable Distributed  
Computing.  
Proceedings of Second Symposium on Reliability in  
Distributed Software and Database Systems :33-39,  

[NELS81] Nelson, B. J.  
Remote Procedure Call.  
Technical Report, Department of Computer Science,  

[PAPA79] Papadimitriou, C. H.  
Serializability of Concurrent Updates.  

[POPE74] Popek, G. J.  
Protection Structures.  
[RAND75] Randel, B.
System Structure for Software Fault Tolerance.

[RAND79] Randel, B.
Reliable Computing System.

[RATT80] Rattner, J. and Cox, G.
Object-Based Computer Architecture.

[RATT81] Rattner, J. and Lattin, W. W.
Ada Determines Architecture of 32-bit Microprocessor.

[REED78] Reed, D.
Naming and Synchronization in a Decentralized Computer System.

[REED79] Reed, D.
Implementing Atomic Actions on Decentralized Data.

[RENN80] Rennels, D. L.
Distributed Fault-tolerant Computer Systems.

Multiprocess Recovery Using Conversations.

[SALT75] Saltzer, J. H., and Schroeder, M. D.
The Protection of Information in Computer Systems.

[SCHL81] Schlageter, E.
Optimistic Methods for Concurrency Control in Distributed Database Systems.
Schwarz, P. M., and Spector, A. Z.
Synchronizing Shared Abstract Types.

Sha, L., Jensen, D., Rashid, R., and Northcutt, J. D.
Distributed Co-operating Processes and Transactions.

Shapiro, R. M., and Millstein, R. E.
Reliability and Fault Recovery in Distributed Processing.

Shrivastava, S. K., and Banatre, J-P.
Reliable Resource Allocation Between Unreliable Processes.

Shrivastava, S. K., and Panzieri, F.
The Design of a Reliable Remote Procedure Call Mechanism.

Spector, A., and Schwarz, P. M.
Transactions: A Construct for Reliable Distributed Computing.

Svobodova, L.
Resilient Distributed Computing.

Tsay, D.
MIKE: A Network Operating System for The Distributed Double-Loop Computer Network.
PhD thesis, Department of Computer and Information Science, Ohio State University, June, 1981.

Tsay, D. P., and Liu, M. T.
MIKE: A Network Operating System for The Distributed Double-Loop Computer Network (DDLCO).
Wyleczuk, R. H.
Timestamps and Capability-based Protection in a Distributed Computer Facility.