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.
Gherfal, Fawzi Fathi

ON THE RELIABILITY OF AN OBJECT BASED DISTRIBUTED SYSTEM

The Ohio State University

Ph.D. 1985

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark √.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages √
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) ________ lacking when material received, and not available from school or author.
12. Page(s) ________ seem to be missing in numbering only as text follows.
13. Two pages numbered ______. Text follows.
14. Curling and wrinkled pages _____
15. Dissertation contains pages with print at a slant, filmed as received ______
16. Other______________________________________________________________
   _________________________________________________________________
   _________________________________________________________________

University Microfilms International
ON THE RELIABILITY OF AN OBJECT BASED DISTRIBUTED SYSTEM

DISSERTATION

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

BY

Fawzi Fathi Gherfal, B.S., M.S.

* * * * *

The Ohio State University

1985

Reading Committee: 

Dr. Sandra Mamrak
Dr. Dennis Leinbaugh
Dr. Venkataraman Ashok
Dr. Ravi Sandhu

Approved By

Adviser

Department of Computer and Information Science
Dedicated to my parents Fathi and Khadija, my wife Huria, and my son Mohannad
Acknowledgments

I would like to thank my adviser, Professor Sandra Mamrak, for her constant encouragement, support, direction, and long hours of enlightening discussions. I am deeply indebted to her for her patience and understanding in the early part of this research, and for molding the ideas in this dissertation into the form they take now. I also would like to thank the members of my reading committee, Dr. Dennis Leinbaugh, Dr. V. Ashok, and Dr. Sandhu, for reading my dissertation and making useful comments and suggestions that led to the improvement of this dissertation. I would also like to thank the members of the Desperanto Project past and present for their enriching discussions and for their patience and understanding during my long presentations.

My deep gratitude goes to my wife and my son for their support and understanding and for making the difficult life of a graduate student more enjoyable. I would also like to thank my parents, my brother, and my sisters for their moral and financial support. I would especially like to thank my father who had foregone his education at early age so he could work to provide us with a good education to help us lead better lives.

I wish to express my deep appreciation to my dear friend Abdulrahim Beram for spending hours upon hours, sometimes though the night, helping me prepare the dissertation on time. He is responsible for the production of all the figures in the dissertation. To him I say, "A friend
in need is a friend indeed”. My thanks also go to Ahmed Elmagarmid for being a life long-friend and supporter.

I would like to acknowledge with gratitude the financial support provided by Al-fatah University in Tripoli, Libya, and the Libyan people throughout my education in the U.S.A..

Finally, my thanks go to the gang of the fourth floor past and present for the good times we shared.
Vita

EDUCATION and EXPERIENCE

May 10, 1954
Born, Tripoli, Libya

May 1977
B. S., Computer Science, University of Dayton
Dayton, Ohio

1977 - 1978
Graduate Teaching Assistant,
Computer Science Department
Al-fatah University, Tripoli, Libya

December 1980
M. S., Computer and Information,
Science, The Ohio State University,
Columbus, Ohio

1981 - 1985
Department of Computer and Information
Science, The Ohio State University,
Columbus, Ohio
FIELDS OF STUDY

Major Field: Computer and Information Science

Studies in Distributed Computer Systems:
    Professor Sandra Mamrak

Studies in Programming Languages:
    Professor N. Soundararajan

Studies in Computer Architecture:
    Professor Ming T. Liu

PUBLICATIONS


# Table of Contents

Acknowledgments iii  
Vita v  
Table of Contents viii  
List of Figures x  
1. Introduction 1  
  1.1. Problem Statement: Reliability Needs In The Distributed System of Interest 2  
    1.1.1. Recovery Needs in The Distributed System of Interest 4  
    1.1.2. Synchronization Needs In The Distributed System 5  
  1.2. Design Approach 5  
    1.2.1. Meeting Recovery Needs 6  
    1.2.2. Meeting Synchronization Needs 7  
  1.3. Software Structure and Design Goals 9  
  1.4. Contribution 10  
  1.5. Related Work 12  
  1.6. Plan of the Dissertation 16  
2. MODELING THE ENVIRONMENT 18  
  2.1. INTRODUCTION 18  
  2.2. ENVIRONMENT OF INTEREST 19  
  2.3. MODELING THE ENVIRONMENT 21  
    2.3.1. Modeling the Underlying System 21  
      2.3.1.1. Nodes 21  
      2.3.1.2. Communication Subnetwork 22  
    2.3.2. Modeling System Resources 23  
      2.3.2.1. Modules 24  
      2.3.2.2. Module Interface 25  
    2.3.3. Modeling Module Cooperation 26  
  2.4. MODELING FAILURE 28  
    2.4.1. Underlying System Failure 32  
    2.4.2. Module and Module Cooperation Failure 34  
  2.5. Effects of Failures 35  
    2.5.1. A Motivating Example 36  
    2.5.2. Summary of Effects and Reliability Requirements 38
8. Exception Handling

8.1. Exception Handling Mechanism— Components and Issues

8.1.1. Exception Declaration 198
8.1.2. Exception Detection, Signaling, and Propagation 200
8.1.3. Handlers 202
8.1.4. Control Transfer 206

8.2. Server/Client Model 208

8.3. Exception Handling as a Forward Recovery in the Environment of Interest

8.3.1. Characteristic of The Environment of Interest: 213

8.4. The Proposed Mechanism

8.4.1. Exception Declaration 216
8.4.2. Exception Detection, Signaling, and Propagation 217
8.4.3. Handlers 219
8.4.4. Transfer of Control 221

8.5. Implementation Of The Exceptions Handling Primitives 224

8.5.1. Module Interface Structure and Implementation 225
8.5.2. Primitives Implementation 230
8.5.3. Summary of Implementation 237

8.6. Summary 240

9. CONCLUSIONS AND FUTURE WORK 242

9.0.1. Summary of Contributions 249
9.0.2. Future Work 250

Bibliography 254
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A Set of Cooperating Modules</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Modules Interaction Implementing a Distributed Application</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Distributed Transaction Access Tree</td>
<td>30</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Distributed Computation Failure Due to a Node Failure</td>
<td>31</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Reliability System Support Architecture</td>
<td>46</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Module's Interaction Model</td>
<td>54</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Effects of Subtransaction Failure</td>
<td>67</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Bounded Buffer</td>
<td>74</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Node Status List</td>
<td>94</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Assignment List</td>
<td>96</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Active Watched Nodes List</td>
<td>97</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Module Interface Components</td>
<td>103</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Object Multi Versions</td>
<td>105</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Object Directory and Versions Descriptors</td>
<td>108</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Structure of Uncommitted Versions</td>
<td>110</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Version Dependencies</td>
<td>113</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Simple Recovery Line and Remote Access List Records</td>
<td>121</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Module's Operation and Object State Transition Diagram</td>
<td>130</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Resource Invariant Form</td>
<td>145</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Syntax Representation of a Selector of a Bounded Buffer</td>
<td>147</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Concurrent Access of Bounded Buffer</td>
<td>155</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Granularizer Structure</td>
<td>162</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Module's Operation Request Processing in the Scheduler</td>
<td>167</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Module Access: Submitting of Operations</td>
<td>173</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Forming Transaction Dependency Sets</td>
<td>177</td>
</tr>
<tr>
<td>Figure 26</td>
<td>Dependency Information Transfer During Validation</td>
<td>179</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Static Association</td>
<td>203</td>
</tr>
<tr>
<td>Figure</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>28</td>
<td>Dynamic Association</td>
<td>205</td>
</tr>
<tr>
<td>29</td>
<td>Client/Server Model</td>
<td>209</td>
</tr>
<tr>
<td>30</td>
<td>Framework of Server Module Activities</td>
<td>210</td>
</tr>
<tr>
<td>31</td>
<td>Declaration of Exceptions Signalled by a Stack</td>
<td>218</td>
</tr>
<tr>
<td></td>
<td>Module</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Exception Handlers Declaration</td>
<td>222</td>
</tr>
<tr>
<td>33</td>
<td>Implementation Relation of Module and Module Interface</td>
<td>227</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>Module’s Interface Code Sections</td>
<td>229</td>
</tr>
<tr>
<td>35</td>
<td>Implementation of Exception Detection</td>
<td>231</td>
</tr>
<tr>
<td>36</td>
<td>Implementation of Exceptions Procedure</td>
<td>234</td>
</tr>
<tr>
<td>37</td>
<td>Implementation of Module’s REQUIRE Section</td>
<td>238</td>
</tr>
</tbody>
</table>

xiii
Chapter 1
Introduction

A major advantage of a distributed system is its ability to share resources among heterogeneous, autonomous single systems. A general-purpose distributed system is built by connecting a number of heterogeneous computers (nodes) through a communication network. One important characteristic of a node is the notion of autonomous control. A node has full control over its resources and data. It can choose to share such resources with other nodes or withdraw them at any time.

A major challenge in developing the distributed system properties described above that allow for resource sharing is devising methods that allow local applications and databases to be created autonomously at their single nodes and later to be integrated into the distributed system as a part of distributed applications and distributed resources. These methods should allow for the construction of well-defined interfaces for such resources which are used to simplify the construction of the integrated distributed application and distributed resources out of local resources. Important issues that have to be addressed in constructing such interfaces are interface to local OS, data-type conversion, distributed access synchronization, reliability, resource naming, and access control. In this dissertation we are mainly interested in synchronization and system reliability.

Distributed systems are inherently reliable. Because of decentralized
control, separate components (nodes) of the system may fail independently, and hence, there is the potential for localizing and isolating such failures. However, the very same property of decentralized control, in addition to the physical separation which creates a weak link among nodes, may hinder reliability. If a distributed system counts on having all its components work correctly all the time, then it is likely that at least one component or a link fails at any given time, causing a total system failure.

The potential for achieving reliability in distributed systems cannot be realized without the proper software support. In this dissertation our goal is to design comprehensive software support in order to efficiently support reliable resource sharing and distributed processing in the environment described above. In particular, the support enables the system to efficiently preserve the system state consistency and to enhance system availability, in spite of failure and concurrent access of resources. Such an objective must be realized without unnecessarily restricting concurrency, wasting useful computation, and incurring intolerable penalties in cost, complexity, and performance.

1.1. Problem Statement: Reliability Needs In The Distributed System of Interest

The distributed system can be viewed as a number of resources provided by a number of heterogeneous atomic nodes. Each resource may have been constructed independently to be used in a single system and later interfaced to be used in the distributed system. Each resource has resource dependent synchronization and recovery constraints to preserve resource consistency constraints and control the way a resource is accessed. A number of resources may cooperate to construct a
distributed resource, distributed application, or a virtual system. An instance of resource cooperation is executed as a distributed transaction, and a number of transactions may be executing concurrently. Because a resource may be accessed concurrently by a number of transactions, it is possible that a transaction uses an intermediate inconsistent state of the resource that is produced by a previous concurrent transaction. In addition, a node failure or a link failure can cause the unavailability of a resource, resulting in the failure of a distributed transaction leaving the state of the resources it accessed inconsistent. Thus, multiple concurrent transactions and arbitrary system failure will cause inconsistency and system unavailability. Recovery from such failures and synchronizing distributed transactions to enhance availability and preserving consistency are the main issues in constructing reliable distributed systems.

Two types of recovery can be employed, backward recovery and forward recovery. Backward recovery preserves the system consistency by restoring the system to a previous consistent state; usually some useful work has to be abandoned in association with backward recovery. Forward recovery tries to force the system to a consistent state in spite of failure and the amount of useful work lost is much less. However, forward recovery needs application dependent recovery information to accomplish its task.

A number of distributed database concepts, such as atomicity and nested atomic transactions are used to treat recovery and synchronization in the database environment. Atomicity implies indivisibility and totality. Transactions are indivisible with respect to other running transactions and either normally terminates or not at all. Indivisibility is associated with synchronization while totality is associated with recovery.
Such concepts mentioned above can not be directly applied to our environment and need to be extended for the following reasons. In database systems, objects are simple uninterpreted row data which we term passive objects; such objects are manipulated through simple operations namely, Read and Write. Objects (resources) in the environment of interest are very complex and services provided (operations) are more complex. In addition, resources have application dependent synchronization and recovery needs to preserve the resource dependent consistency constraints. We will term such objects active objects. It is clear that the recovery and synchronization model used in database systems needs to be extended to capture the properties of the environment of interest.

Before we discuss our approach and review our model, we review next the recovery and synchronization needs of the environment.

1.1.1. Recovery Needs in The Distributed System of Interest

The recovery needs in the environment include the following: 1) Preserve the system state consistency that can be violated due to a number of possible failures; 2) Provide for permanence of effect; that is, because different components of the system may fail independently, and a component may have reported some requested results, which were relied on by the client, it is important that a later failure does not invalidate such results; 3) Save healthy work by eliminating orphans which are a duplicate request resulting from a failure that causes the abandoning of the original request causing the loss of healthy work and inconsistency; 4) Provide for error confinement that may be propagated from one component to another causing other components of the system to fail; 5) Enhance the system availability by providing for an alternative
component providing similar service, if a requested component becomes unavailable.

1.1.2. Synchronization Needs In The Distributed System

System inconsistency can be caused by the unsynchronized access to distributed resources by a number of concurrently running distributed computations. A mechanism to control concurrent access to shared distributed resources is needed to preserve the system state consistency. The synchronization need of the environment is two fold. The first is the need to preserve consistency constraints of individual resources. The second is to synchronize multiple concurrent transactions in such a way that each transaction preserves the consistency constraints of all the resources it accesses, and each transaction must be synchronized with respect to other running transactions so as to preserve the consistency of the total system (collection of resources).

1.2. Design Approach

To provide for the reliability and synchronization needs, we adapt an object based distributed processing reliability model that captures the properties of environment. We believe the model is a general one and can be applied to other environments. The model is based on the idea of the Recoverable Module to model a resource. A Recoverable Module is an abstract data object that is able to tolerate failures and manipulate exceptions and hence form a unit of recovery. A number of recoverable Modules cooperate to implement a reliable distributed application or reliable distributed resource, an instance of which is executed by a distributed transaction. A distributed transaction has the atomicity property (totality and indivisibility) and forms a unit of distributed
synchronization. However, it is not used as a unit of recovery as will be explained later.

1.2.1. Meeting Recovery Needs

To meet the recovery needs of environment, we use a recovery approach that is based on combining backward recovery techniques and forward recovery techniques to implement the recoverability property of the recoverable Module. Backward recovery in the form of atomic actions is used to preserve the individual Module's (resources) consistency and also used to preserve the system consistency in case of user abort of the transaction. Forward recovery is used in the form of an exception handling mechanism to inject semantic knowledge to aid in confining failure effects, to control error propagation, and to enhance availability. In addition, both techniques are used to implement the Recoverable Module to tolerate node failures.

A number of researchers [33, 38, 3] have studied the problem of reliability synchronization in distributed systems, most of whom have proposed reliability models that are based on atomic transaction as the unit of recovery and synchronization. We term these models transaction based recovery models. Distributed atomic transaction takes the system state from one consistent state to another. A failure causes the entire distributed transaction or subtransaction to be aborted regardless of the type of failure, whether it is a node failure, a link failure, or a failure causing the unavailability of the requested service. Such an approach does not take full advantage of the independent failure of a separate component executing under different control, where there is potential for isolating and localizing the effect of the failed component. In the distributed environment of interest a computation may be large and
complex. Thus, aborting such a computation may cause the loss of substantial useful work and therefore degrade performance. Our approach calls for the use of the Recoverable Module to tolerate node failure and to manipulate unavailability exceptions resulting from a link failure or software exception. Therefore, the distributed transaction accessing the Recoverable Module's operation is not forced to abort as a result of such failures and healthy computations may be saved. In addition, a Recoverable Module is able to eliminate orphans and provide for permanence of effect. However, the totality property of the atomic distributed transaction is still used to provide for user abort of transaction, and the abort of transactions due to synchronization problems.

1.2.2. Meeting Synchronization Needs

To meet the synchronization needs of the environment, a mechanism is needed to allow for the specification and preservation of the Module dependent synchronization constraint and to synchronize the multiple concurrent transactions.

The specification of Module dependent synchronization constraints allows for the specification of other synchronization semantic knowledge that can be used to increase the degree of concurrency. Synchronization semantic knowledge has been used by a number of researchers in database systems and general distributed systems to increase the degree of concurrency among transactions [14].

Schemes that are used in database systems, such as Locking and Time stamps, do not allow for semantic knowledge specification, enforce strong serializability, and do not preserve resource dependent consistency
Constraints. In addition, such schemes cause the delay of distributed transactions waiting to use the object until the object is released, thus restricting the degree of concurrency.

Concurrency can be improved by minimizing the delay caused by the exclusive access of the object by a distributed transaction, hence improving performance. A novel optimistic concurrency control mechanism for synchronizing distributed computations is proposed to minimize such a delay and achieve a high degree of concurrency. Optimistic methods rely on detecting transaction conflicts in accessing an object rather than on conflict avoidness as in the case with pessimistic methods and thus can increase the degree of concurrency. One disadvantage of the optimistic mechanism is that it cause a domino effect aborts of transaction that are interdependent (accessing a shared object). However, using the object oriented recovery proposed rather than the transaction oriented recovery model reduces the amount of abort since transactions are not forced to be aborts due to a failure. In addition, the use of synchronization semantic knowledge decreases the interdependencies among transactions and thus reduces the effects of domino effect aborts.

Our approach is to combine the use of the traditional synchronization mechanism that allows for semantic synchronization knowledge specification and an optimistic concurrency control to form the basis of the synchronization mechanism. An additional feature of our reliability model is the fact that the synchronization mechanism conforms to the object model philosophy rather than the transaction model in the sense that only the cooperating Modules accessed by a distributed transaction are involved in synchronizing the transaction with other concurrent
transactions No higher authority is needed at each site to perform the synchronization as in the case with transaction based models. We will term this property object based synchronization control.

1.3. Software Structure and Design Goals

The reliability software support is constructed in three layers, a recoverable component layer, a recoverable Module layer, and a cooperating Module layer. Each layer masks a number of failures and propagates unhandled failures as exceptions in a controlled fashion. The recoverable component layer provides for detecting, isolating, and recovering from component failure. Only unrecoverable failures are raised as an exception to the next layer. The recoverable Module layer provides the support to enable the Module to tolerate a node failure, to preserve consistency constraints of the Module, to confine failure effects, to control error propagation, and to enhance availability. The cooperating Module layer’s support provides for implementing the totality and indivisibility properties of the distributed transaction and for the reliable Module interaction among the cooperating Modules. A number of mechanisms are employed to provide for each layer’s support.

In designing the needed recovery and synchronization mechanisms, a great deal of emphasis is put on efficiency. It is our goal to build a reliable distributed system support for the environment of interest, without introducing intolerable penalties in complexity, cost, or performance degradation. It is not our goal, however, to provide continuous system operation through some type of component redundancy or data duplication. If a resource can not be accessed due to a failure or autonomy, then an alternative service from other Modules providing similar service is selected using the recovery semantic knowledge.
provided. Hence, our goal is to provide fault tolerance rather than fault masking on the resource level.

In designing these mechanisms to satisfy the recovery and synchronization needs in the environment of interest, our goal is to provide for the following:

High degree of concurrency of distributed computations,

Minimal waste of useful work due to a failure,

Minimal efficiency degradation incurred during normal operation and reasonable overhead during recovery from failure,

No introduction of any constraints by the mechanism used on the post facto building of distributed applications and resources out of existing resources, and

The provision of a mechanism for specifying and preserving resource dependent consistency constraints and recovery semantic knowledge.

1.4. Contribution

This dissertation provides unique contributions in three aspects. First, we provide comprehensive recovery and synchronization support for a unique distributed environment, allowing for the sharing of resources among autonomous and heterogeneous single systems. Second, we propose a general object based reliability model that captures the properties of the environment of interest. The model allows for the use
of object based recovery, the specification of both recovery, and synchronization semantic knowledge to achieve flexibility, efficient recovery, and a high degree of concurrency. Third, we propose a number of new recovery and synchronization mechanisms that implement the features of the reliability model proposed. The recovery mechanisms use both backward and forward recovery techniques to confine a failure's effect, save healthy work, and use recovery semantic knowledge to enhance availability. The synchronization mechanism uses an optimistic concurrency control technique and user specified semantic knowledge to achieve a high degree of concurrency. These mechanisms include:

A synchronization and concurrency control mechanism that is constructed by combining a traditional synchronization mechanism with an optimistic concurrency control mechanism, which uses semantic knowledge to increase the degree of concurrency of distributed computations (transactions). This mechanism preserves the consistency constraints of individual resources and the group of cooperating resources.

An object management mechanism that uses the multi-version objects technique, supports the optimistic concurrency control in achieving a high degree of concurrency of distributed computations, and provides for object restoration in case of a node failure or computation abort.

A Module's operation management mechanism that implements the atomic property of the Module's operations and the distributed computations.

A Module recovery mechanism that confines node failure
effects and restores the Module's state to a consistent state after a node failure.

An exception handling mechanism that allows the use of recovery semantic knowledge to aid in recovery, enhance availability, to control error and exceptions propagation, and to provide for reliable Module interaction.

1.5. Related Work

A number of researchers have addressed the reliability problem in a general purpose distributed system and have assumed similar environments [33, 38, 30, 43, 49, 48, 4, 3, 27]. Most of these proposals were an extension of the reliability techniques and concepts used in distributed database systems, such as atomicity, nested atomic transactions, and semantic knowledge [14]. While we use similar extensions of the same concepts, we believe that our approach differs in three ways: first, in the purpose or orientation of the design (i.e., system oriented, language oriented, database oriented); second, in the recovery model used; third the mechanisms used to implement the properties of the recovery model. Next, we compare and contrast our approach to each of the proposed approaches and models.

**Moss and Reed:** Moss [33] and Reed [38] can be considered the pioneers in the area; their approaches were direct extensions of database models. While they have assumed abstract objects as the units manipulated by the transactions, these objects are passive, do not have synchronization and recovery properties, and are simple row data. Moss and Reed's models do not have the facility to express synchronization and recovery semantics. In addition, both of the recovery models are transaction
based where the transaction is the unit of recovery and synchronization, and the totality property of the atomic transactions is used to recover from all types of failures, such as node failure, link failure, or software failure. To implement the required mechanisms, both approaches require the use of a higher authority to manipulate transactions at each site.

Our approach, on the other hand, differs in the following ways. First, we use an object based recovery model to recover from different failures, and the object dependent synchronization dictates the way distributed transactions are synchronized. Second, we use object based synchronization control where a high authority is not needed at each node to manipulate transactions. The cooperating objects collectively are able to manipulate and insure the atomicity property of the distributed transaction executing the object cooperation. Third, Moss uses a locking method to synchronize multiple concurrent transactions where Reed uses time stamps. Both methods are pessimistic and restrict the degree of concurrency unnecessarily especially when used in conjunction with our model. A number of the works presented next are based on Moss’s work and use transaction based recovery, in addition to locking based synchronization.

**Liskov:** The work of [30] in association with the Argus project is based on Moss’s work, and hence it assumes the same transaction based recovery and synchronization. Argus, however, is designed as a distributed user application language. It is an extension of CLU language and favors heavy data abstraction. The objects are assumed to be passive where there are no object dependent synchronization and recovery semantics properties that are specified as part of the object definition. The only semantic user is the distinction between the read
and write operations on the objects. Since Argus is based on Moss's work it supports transaction based recovery. It also uses locking to synchronize distributed transaction and thus unnecessarily limits the degree of concurrency.

Weihl and Allchin: Weihl [48, 49] and Allchin [3, 2, 4] separately proposed similar models and approaches to treat recovery and synchronization. Weihl's work is an extension of the work done in the Argus project. Both have object based models where objects are active and may have object dependent synchronization constraints and allow for synchronization semantic specification. In both models, synchronization is dictated by the synchronization properties of the individual objects. Weihl has initially used Serializers for object dependent synchronization, but later used multi-mode locking similar to Allchin's. Both implementations of atomic actions are based on Moss's work.

Our work is similar to Weihl's in that he allows objects to be constructed separately and then to be combined to construct larger distributed objects, much the same as constructing distributed resources using already defined resources in the environment of interest. Weihl's work is also similar to the Module cooperation model presented in Chapters 2 and 3. However, both Weihl's and Allchin's proposals differ from our work in three ways. First, their recovery approach is based on Moss and thus it is transaction based recovery. Second, we have used an optimistic concurrency control mechanism to synchronize the concurrent transaction where both of their proposals employ pessimistic approaches based on locking and hence can restrict concurrency. Third, we use Selectors to provide for object base synchronization which allows the use of high level language specification of object synchronization.
constraints where both have used a multi-model locking to specify and implement synchronization semantic knowledge. We believe a high level specification language is much more flexible, powerful, and easy to use. Allchin has indicated in his work that the invention of a specification method for specifying synchronization semantics is a considerable contribution. Allchin has proposed a suit of replication algorithms to enhance availability. Our approach is to provide for an exception handling mechanism to allow the object definer to specify a similar service in case a failure causes the unavailability of a requested service.

Spector: Spector & Schwartz [43, 42] have proposed construct for a building a reliable distributed system based on performing transactions on abstract objects. Their model allows for the specification of synchronization semantic knowledge in association with object definition. Such semantics are specified using operation dependency. Operations are classified as to whether they conflict or not. Object dependent synchronization is implemented using multi-mode locking similar to Allchin’s. Conflicting operation are delayed before entering the object.

Once again, while their approach is similar to ours in allowing the specification of object dependent synchronization semantics, they use transaction based recovery as opposed to our object based recovery. In addition, they have used a locking based mechanism to synchronize concurrent transactions which can restrict concurrency unnecessarily.

Lian: Concurrent to our work Richard Lian [27] has developed a similar recovery model; however, our synchronization approach differs considerably. While he recognizes the importance of synchronization semantic knowledge in his model, he did not suggest a mechanism to specify and implement such semantic knowledge. Lian’s work is also
based on Moss's and uses a locking based method to synchronize multiple concurrent transactions, and hence has the same disadvantages. However, he has suggested a commit protocol that takes advantage of the hierarchical structure of the distributed transaction. Lian has advocated the use of an exception handling mechanism to handle software failures in his model; however, he did not specify a mechanism to implement such a facility. We use an exception handling mechanism as a forward recovery technique to enhance availability and confine error propagation. We believe the design and implementation of such a mechanism for the environment of interest is not simple and needs to be addressed.

1.6. Plan of the Dissertation

In Chapter 2 we present a model for the distributed environment of interest for an error and failure free environment, including a model for possible failures and their effects on the reliability of distributed processing. The chapter also establishes the reliability needs of the environment. In Chapter 3 we present a reliability model and system structure for the reliability software support and specify the function and the services provided by each layer. In Chapter 4 we discuss the basic concepts relating to implementing the functions and services provided by each layer. A number of existing mechanisms to implement such services are examined along with their adaptability to the environment of interest. In addition, we discuss our strategy for providing such services. In Chapter 5 we discuss the recovery mechanisms needed to implement recoverable nodes and Recoverable Modules. These mechanisms include recoverable node management, the Module's object management, the module's operation management, and the Module's recovery management.
In Chapter 6 we present a synchronization mechanism to synchronize Module operation. The mechanism is an extension of a traditional synchronization mechanism (Selectors) which allows the use of semantic knowledge and preserves the Module's consistency constraints. In Chapter 7 we present an optimistic concurrency mechanism for synchronizing distributed transactions. This global concurrency control mechanism is coordinated with extended selectors of Chapter 6 to achieve a high degree of concurrency. Chapter 8 describes an exceptional handling mechanism. Chapter 9 summarizes the work in this dissertation and gives directions of future work.
Chapter 2

MODELING THE ENVIRONMENT

2.1. INTRODUCTION

One of the advantages of distributed systems is their ability to share resources among autonomous heterogeneous single systems. Each system may provide a number of resources to be used by other sites while it uses the resources provided by other sites. Resources must be interfaced to be used efficiently in the distributed system. New resources may be constructed out of a number of other resources provided at different sites. Due to physical separation and multi-independent control, a number of failures and synchronization problems may occur which effect the reliable cooperation of resource sharing among the different systems. Such failures may cause system inconsistency and unavailability.

In this chapter, the environment of interest is described and a model that captures the properties of the environment is proposed. In the model we assume a failure free environment. In a later section, we will investigate the effects of failure and unsynchronized access to distributed resources on the system reliability by introducing a failure model and identifying the possible events that can occur which may hinder the system reliability. Finally, we specify our design goals and strategy for providing reliable distributed processing in the environment of interest by specifying a reliability model that preserves the consistency of the system state and enhance availability.
2.2. ENVIRONMENT OF INTEREST

The environment of interest can be described as a collection of fully functioning, heterogeneous, autonomous, single systems that are connected through a communication network, either a long haul or local area network. Each system has its local OS and distributed function support which is provided as a guest layer on top of the existing local operating system. Each node may be completely available to accesses through the network or it is completely unavailable. Furthermore each node provides a number of services in the form of resource (data-base, OS resource, or computation). The most important properties of this environment can be summarized as follows:

1. **Autonomy**: A node is owned by individuals or organizations who control how the node is used and what resources are to be shared, and have control over when the resources are available. For example, it is possible that some data and computation may be completely private and not available for network use. While other resources might be presented only partially, it is also possible that only a specific mode of access is allowed, or accesses are only allowed at certain times. In addition, a node can refuse service to a remote request of a different node because of some local constraints, such as limited computer time resource and/or interference with local computing tasks.

2. **Post Hoc Integration of Resources**: Resources provided by a specific node may be constructed using the service of the resources provided by other nodes. Thus, local applications and databases may be created autonomously then integrated
at later time with other applications and data accessible in the distributed system in a post hoc fashion. This means it is possible to construct distributed applications and resources out of existing resources and databases under the existing operating systems.

3. Modularity: A resource provided in the distributed environment is described to its user in terms of its behavior without reference to the actual implementation of the resource. Thus, a user only sees the service provided by the resource and not how it is provided or implemented.

In order to support the cooperation of autonomous single systems so as to be able to share computation resources and to achieve common goals, a coherent set of protocols have to be developed. Such protocols must allow for the construction of a well-defined interface for local resources, which are used to simplify the later unplanned construction of an integrated distributed application and resources out of the existing local resources and databases. Such protocols also make it possible to achieve reliable resource sharing and reliable distributed processing.

Important issues that have to be addressed in developing these protocols are: interfacing to the local OS, data-type conversion, resource naming, access control, security, synchronization, and reliability. In this work we are mainly interested in the synchronization of distributed access to distributed resources and system reliability.
2.3. MODELING THE ENVIRONMENT

In this section, a model is presented for the distributed environment of interest described in the previous section. In the model we assume a failure free environment. This model is based on the concept of Module and Module Interface described in the Desperanto Project [31]. A Module model, a distributed object or resource, provides some services to other requesting Modules and itself may request the service of other Modules to accomplish its task.

A number of cooperating Modules form a sub-system or a distributed application. Each Module is interfaced into the distributed system through a well-defined interface called the Module Interface. In this section we specify, as part of the model, the properties of the underlying system which include nodes and a communication subnetwork. We then present a model for a shared resource in the form of a Module and a Module Interface and Module's cooperation.

2.3.1. Modeling the Underlying System

The underlying system is modeled as a collection of autonomous heterogeneous nodes that are connected by means of a communication network and communicate using message passing.

2.3.1.1. Nodes

Each node consists of one or more processors and one or more levels of memory. The nodes are heterogeneous, e.g., may contain different processors, come in different sizes and provide different capabilities, and may be connected to different external devices. Each node is autonomous in the sense that is described in the previous section. Nodes
may go down (crash) and up (recover) and new nodes may be added to the system or removed at any time.

A node has two types of memory, permanent memory and volatile memory. Permanent memory can survive a node crash; however, the contents of volatile memory are lost in a crash. An important property of the permanent memory is its reliability. All updates and accesses to the permanent memory can tolerate failure. It is assumed that every pair of nodes can communicate in both directions, directly or indirectly using message passing.

Each node interfaces to the distributed system through a kernel. All remote access and local service are coordinated through the kernel. The kernel, a guest layered on the top of the local operating system, is reduced to a portion of what is usually included in an operating system. The kernel provides basic functions, such as process management, interprocess communications, protection, and naming. Other traditional operating systems services are viewed as resources which are accessed through request messages.

2.3.1.2. Communication Subnetwork

Two nodes communicate only by sending messages over the communication subnetwork. A message is considered as a unit of communication, and no bound is placed on its size. Any node can transmit a message to another specific node. The network may be a local area network (using bus, ring, loop, etc.), or it can be a long haul network such as the ARPANET. The following assumptions are made about the communication network:

Messages are correctly delivered or not at all,
Messages may be lost,

Messages may not arrive in the order that they are sent from some sender,

A message may be duplicated arriving more than once,

A message may be arbitrarily delayed, and

A message may never be spontaneously generated.

2.3.2. Modeling System Resources

Using an object-oriented model [19], system resources are modeled as abstract objects called a Module. A Module is an entity that models physical and logical resources, encapsulates, and controls access to a system resource, such as a database, a device, or a computational resource. Access to the protected resource object is provided by a set of operations called Servers that can be called from outside the Module. The service provided by the Module is much more complex than Read and Write operations, usually assumed to be the type of requests in database systems. A Module is characterized by its behavior in response to the defined operations being performed.

The object oriented model is used in object-oriented programming languages systems, such as CLU [29] and ALPHARD [50]. It is an attractive approach in constructing reliable systems. The major benefit of object-oriented systems is that they limit the view of the internals of the object defined. The Module can be specified, understood, and used without reference to its implementation. In addition, the Module can be
implemented without reference to its eventual use. Such properties aid in limiting error propagation, enforce Module protection, and make the use and the implementation of the Module a simpler task to accomplish. Furthermore, by specifying a well-defined interface for the Module, constraints may be easily expressed in terms of the allowable uses of abstract operations on the Module.

2.3.2.1. Modules

Modules are written and implemented independently of any other Modules. A Module may be constructed using the service (operations) provided by other (remote or local) Modules that are independently written; in turn, a Module may provide service to another Module. Modules are local to the specific node. The Module does not share address space with any other Module. A node may support several Modules.

To communicate with other Modules, a Module uses a message passing facility, where messages received by a Module are the kind it expects. Modules may communicate in terms of abstract data types that are of interest to the Module. In other words a Module does not need to deal with the form of messages suitable to the subnetwork, e.g., bit strings, but it can send as parameters predefined or user defined data types, such as integers, arrays, and stacks.

A Module has a set of integrity constraints that dictate way the Module is accessed or modified to preserve the consistency of the Module. These constraints should not and must not be violated.

Internally, a Module consists of a set of objects (possibly empty) and processes that manipulate these objects. A process is an active execution
of a Server (operation) code. Server processes share the Module's objects directly, but sharing of objects across Modules is not allowed. A server process acts as an agent to a remote requester and executes the desired service on behalf of the requester.

2.3.2.2. Module Interface

To allow for the later unplanned use of a Module as part of a distributed application or subsystems, a semantically clean interface (Module Interface) is specified for the Module. Such an interface specification includes the service (operations) which the Module provides, the services which it requires, and the constraints and invariants under which these services are provided and required.

The Module Interface is tailored for the specific needs of the Module. The definer of the Module Interface must specify the synchronization constraints to enforce the Module invariants and the conversion routines that are used to convert the data-types (parameters) passed to the Module.

A distinction is made between programming-in-the-large and programming-in-the-small. A Module itself is thought to have been written by a programmer-in-the-small. When a Module is to be incorporated into the distributed system at a later time, the task of defining an interface for it is said to be done by a programmer-in-the-large. The programmer-in-the-large uses an appropriate specification language to specify the semantics of the Module Interface.

The Module Interface can be considered as a Module manager, which performs the following functions:
Access Control: To allow only authorized users to access the Module operations.

Data type Conversion: To convert data-types passed as parameters from a standard form to a form used by the Module and vice versa.

Interface Error Checking: To check the validity of remote requests to insure that they conform with the Module specification.

Scheduling of Remote Requests: To insure that if a request is valid, then the server process executing the request is scheduled and synchronized with other server processes in such a way that Module constraints are maintained.

2.3.3. Modeling Module Cooperation

As mentioned earlier a number of independent Modules may cooperate to implement a distributed sub-system, which can be thought of as a distributed application or a virtual machine. In providing some service a Module operation may request the service of a remote Module by requesting the execution of a particular operation that implements the desired service. In turn, the requested Module may request the service of yet a third Module in a nested fashion, thus extending the Module interaction to several levels of nesting. For example, we can consider Figures 1 and 2. A distributed application is performed by a set of cooperating Modules. A user requests the service of operation OP1 of Module M1. A server process is created to execute operation M1.OP1 on behalf of the user. M1.OP1 requests service of M2 and M3 by
Figure 1: A Set of Cooperating Modules
requesting the execution of M2.OP2 and M3.OP3. Two server processes are created to execute these operations. Similarly, Op2 of M2 requests the service of M5 and M6.

The user request to M1 is called a top-level request; all other requests are called nested requests. The server processes performing the respective operations form an access tree as shown in Figure 3. Each node of the tree corresponds to a requested execution of a Module operation by a server process. Each instance of an access tree is called a Distributed Transaction. Each remote request, which is a branch in the access tree, forms a subtransaction.

2.4. MODELING FAILURE

The model presented assumes failure free environment. A number of failures and exceptions may occur in this environment which may cause the system to behave unreliable. In this section, a model of these failures is presented, and their effects on the reliability of the system is indicated.

Events in the distributed system can be classified as desirable and undesirable events. Desirable events occur when the system performs according to its specifications; thus, in a reliable system only desirable events may occur. Undesirable events are defined as the divergence of the system from its specifications [37]. The undesirable events may be termed as failures which are divided into expected and unexpected failures. Expected failures are those that may occur due to the physical or logical property of a distributed component and can be anticipated in advance. Such failures include node failure, communication failures, media failure, and Module failure.
Figure 2: Modules Interaction Implementing a Distributed Application
Figure 3: Distributed Transaction Access Tree
Figure 4: Distributed Computation Failure Due to a Node Failure
Unexpected failures are those failures which cannot be anticipated in advance, such as design faults or failures due to a distributed component changing its bad detectable state to a bad undetectable state, for example, a garbled message changing to a valid message that cannot be detected using a checksum technique, or a disc controller returning the wrong page for a specific given address. The probability of unexpected failure must be reduced to an acceptable level by accounting for important failure modes.

Hardware failures on the component levels, such as failures of gates, buses, registers, memory, etc. and their properties, are not examined in this work. We are only interested in the failure properties of the underlying system, such as a communication, processor, and storage medium which may fail as a result of hardware failure. In addition, we will concentrate on the software aspects for achieving reliability of the distributed environment of interest.

2.4.1. Underlying System Failure

In this section we specify the failure properties of the underlying system components which include node failure, communication failure, and medium failure.

Any processor or volatile memory failure will cause the node to fail. When a node fails, it is said to have crashed, and when it is restarted it is said to have recovered. A node crash is equivalent to stopping the processor and resetting the processor and volatile memory to a standard state. When a crash occurs, all data stored in the system is inconsistent except data stored in the permanent memory. In addition, all Modules on the crashed node become temporarily unavailable. A node crash is
assumed to be transient, and the node recovers immediately after a crash. If a node crash lasts for a long period, however, the node is isolated from the other nodes until it recovers.

It is said that a communication network has failed if a message cannot be delivered properly to the target node due to a broken link or hardware problems. A link failure can cause network partitioning where the system is split into mutually isolated groups of nodes. In this model only one failure is assumed which is the inability to deliver a message by the network. Here we are making the assumption that the reliable message delivery is provided by the Transport layer on the top of the network. However, duplicate messages may be created by the sending Module due to time-outs or node failure, and such duplicates are treated in the higher layer's protocol. One might argue that many of the mechanisms that are used to provide reliable communications (no messages lost, no duplicates, messages arrive in order, and messages arrive intact) [46] might be necessarily constructed in the higher level protocols. This is called the End-to-end argument [41]. We feel that only some mechanism duplication is required in the upper layers.

A storage medium, such as disk storage, may fail or be corrupt due to decay. Medium failure will cause the loss or the inconsistency of the stored information. In addition, the information stored on the medium may be incomplete or incorrect due to incomplete write operation that is caused by a node failure.

Node, communication, and medium failures constitute the basic failures in our model. However, these failures may cause the failure of other components on a higher level, such as a Module's failure and a distributed application failure. Since ultimately we are interested in the
reliable processing of a distributed computation which constitutes the highest abstraction in our model, we feel it is appropriate to include Module and Module cooperation failures and exceptions as part of the failure model.

2.4.2. Module and Module Cooperation Failure

To its requester a Module fails if it becomes unavailable or if an undesirable result is returned as a response to the client. A Module becomes unavailable in the following cases:

- Due to autonomy constraints, such as removing the Module by its owner, or refusing service to the requester.
- Communication failure or a network partition.
- A node crash of the Module's home node; in such a case the Module state and volatile memory used by the Module will be lost.
- Failure of server process executing a requested service.
- If a Module can not correctly provide the requested service.

A Module may return an undesirable response or an exception if the requester violates the Module integrity constraints for requesting and providing the Module service, if invalid inputs are provided to the Module, or if an unrecoverable error is detected in the operation performing the service requested.

A number of clients may request the service of a given Module
concurrently. Such requests are executed by server processes on behalf of the remote requesters. These server processes must be synchronized in such a way as to preserve the integrity constraints specified by the Module. For example, if the Module is implementing a stack, then integrity constraints may be that no POP is allowed on an empty stack. Any request tempting to POP the stack while it is empty must be delayed until another request has inserted an element in the stack. A violation of Module constraints causes the Module to fail and to raise an exception.

A distributed application may not be able to proceed normally if one of the Modules involved in the Module cooperation implementing the distributed application or sub-system fails or if a user initiating the distributed applications aborts the distributed transaction executing the distributed application.

2.5. Effects of Failures

As stated previously, the distributed computation or subsystem is the highest abstraction in the model; thus, the distributed system may be viewed as a number of distributed computations or virtual machines that are accessing and manipulating distributed objects through some operations. By guaranteeing the reliable processing of these distributed computations, we guarantee that the distributed system is reliable. In this section we will examine the effects of different failures on the reliability of the system. This can be best demonstrated by the following example.
2.5.1. A Motivating Example

We can consider Figure A5. If a distributed computation is initiated on node n1 by accessing operation op1 of Module M1 (m1.op1), M1.op1 requests the service of Module M2 on node n2 by requesting M2.op2. M2 in turn requests the service of Module M3.op3 and M4.op4 on n3 and n4, respectively. Now the following scenario will take place: m3.op3 finishes, while m4.op4 is just started, and we can assume that m1.op1, m2.op2, m3.op3, and m4.op4 have made some changes to their respective objects as required. The following possible cases can now be considered:

If node n2 crashes, M2 will lose its state and requests to m3.op3, and m4.op4 will become Orphans which are healthy children that have done some work and can not be used, and more importantly the objects accessed by operations m2.op2, m3.op3, and m4.op4 are temporarily inconsistent. If no recovery is done to bring them to a consistent state, the system will assume an inconsistent state.

If the service of Module m4 cannot be provided (because of a link failure or the Module is unavailable), then the distributed transaction cannot proceed any further; it has to be aborted or m2 has to choose an alternative service that replaces the service of Module m4.

If Module m4.op4 cannot be completed because of some type of exception that is raised within m4.op4, this exception should be treated by M2 to prevent propagation of the error to other parts of the computation.
If the user requesting (initiating) the service of the top level Module \( m_1 \) decides to abort the request, then the effect of the entire transaction must be undone to preserve the consistency of the state of the system. This is also applied to any nested request (e.g., \( m_4.\text{op4} \) by \( m_2.\text{op2} \)).

Another problem that can cause access inconsistency (but not related to computation failure) is unsynchronized object access by distributed transactions. While a distributed transaction is in progress, the state of the objects accessed by the computation is temporarily inconsistent. To preserve the system state consistency, any other computation should not be allowed to access the inconsistent objects until the entire computation has successfully terminated and the state of these objects are made permanent. If a second computation \( c_2 \) uses the temporary inconsistent state produced by the first computation \( c_1 \), then a conflict is said to have occurred between \( c_1 \) and \( c_2 \).

If the distributed transaction has finished and an answer is provided to the user initiating the top-level-request, then a later crash of any node involved in the Module cooperation might cause the objects of those Modules to lose their latest state, and the answer given is not consistent with the internal state of the system. In general, if the system is requested to perform some action and responds that the action has been done, then the system state should reflect that action in spite of failure. This is called the permanence of effect problem [30].
The effects of failures and unsynchronized access of the distributed computation on the reliable processing in the distributed system can be divided into the following categories:

**System State Inconsistency:** A node failure may cause a distributed computation failure which can result in the system assuming an inconsistent or invalid state. Thus, a reliability mechanism should preserve system state consistency in spite of failure. This is achieved by providing a recovery mechanism that restores the system state to a consistent state in case of a failure.

In addition, the distributed transaction, of which the server process forms a subtransaction, must preserve the integrity constraints of all the Modules that it encompasses (access the services of), and each of these transactions must be synchronized with all other concurrent transactions that access a shared service or object in such a way as to preserve system consistency.

**System Availability:** A failure of a Module used by a distributed computation may prevent it from achieving its task. To deal with this type of failure, the reliability mechanism should provide for ways to restore the failed Module to a state from which the processing of the computation may be continued. If Module service cannot be provided due to a failure, then the mechanism must allow for ways to choose an alternative Module that provides the same
or similar service to continue the execution of the distributed computation.

**Loss of Healthy Work (orphans):** Failure can cause the loss of the healthy work that is performed on behalf of or as a part of a failed distributed transaction. The reliability mechanisms should provide for saving the healthy work of a requested Module in the case of the failure of the requester Module. Such work must be used after the recovery of the requester Module.

**External Consistency and Permanence of Effect:** The loss of the state of the objects manipulated by a distributed computation, which has reflected such a state to the external world and has terminated, will cause external inconsistency (state of the system does not reflect the state assumed by the outside world). A reliability mechanism for a distributed system should provide for permanence of effect to preserve external consistency.

**Error Confinement:** An error can be propagated to another system components causing the failure of such a component and the loss of more computation work. The system should allow for detecting and handling such errors to confine their effects to the failing component. For example, if a Module failure occurs due to node failure, it should not cause the abortion of the entire distributed transaction using the service of that Module. Instead, we should take advantage of the property of separate control of each node and recover the failed Module independently to allow for the continuation of transaction processing.
2.6. GOALS and STRATEGY

In designing the needed reliability mechanisms, much emphasis is put on efficiency and the adaptability to the environment of interest. It is our goal to build a reliable distributed system for the environment of interest, without introducing intolerable penalties in complexity, cost, or performance degradation.

2.6.1. Goals

The system software designed to address the reliability needs of the environment includes mechanisms to preserve system state consistency, concurrency control, and external consistency, as well as to provide for error localization and confinement and for user specified semantic knowledge to support recovery and synchronization. In designing this software our goal is to provide for:

- A high degree of concurrency of distributed computations,
- The minimal waste of useful work done to a failure,
- The minimal efficiency degradation that is incurred during normal operation, and reasonable overhead during recovery from a failure.
- Mechanisms which do not introduce any constraints on the post facto building of distributed applications out of existing Modules,
- The mechanism which incurs low cost in terms of storage used, and processing time, and
The use of semantic knowledge to enhance the degree of concurrency and achieve efficient recovery by enhancing availability.

2.6.2. Strategy

Our approach is to present a reliability model that is based on the concept of a recoverable model. The model is an extension of the model presented in this chapter. The model achieves the reliability properties desired to fulfill the reliability needs of the environment. A number of mechanism are then presented which implement the properties of the model and conform to our goals.

2.7. SUMMARY

In this chapter the properties and the characteristics of the distributed environment of interest have been described along with a model that captures such properties. The model is based on the concept of Module and Module Interface to model distributed system resources. A distributed application or subsystem is implemented by a number of cooperating Modules; an instance of this Module cooperation is executed by a distributed transaction. A number of failures can occur in the distributed system which cause system inconsistency and hinder the system unreliable. A model of failure is presented, and the effects of such failures on the functions of the distributed system are discussed. Finally the goals and strategy of achieving reliable distributed system processing are set forth.
Chapter 3
SYSTEM STRUCTURE

3.1. INTRODUCTION

In this chapter we present a reliability model that satisfies the reliability needs of the environment. The model is based on the concept of a Recoverable Module, a fault tolerant Module that is able to manipulate exceptions and recover from different failures, and thus form the unit of recovery in the system. A number of cooperating Recoverable Modules form a reliable distributed application or subsystem. An instance of this application or subsystem is executed by an atomic distributed transaction. The transaction forms the unit of synchronization and is used to restore the Module’s states of the Modules accessed by the transaction in case of transaction abort. The distributed system may be viewed as a large number of Recoverable Modules that cooperate and interact to implement a number of reliable distributed applications and subsystems.

This chapter presents a system structure and reliability model for constructing reliable distributed processing in the environment of interest that is based on a layered approach. Each layer provides some reliability services that support the construction of Recoverable Modules and a reliable Recoverable Modules interaction (Distributed transaction). This chapter describes those services at an abstract level.
3.2. RELIABILITY MODEL AND SYSTEM LAYERS

In the highest level of abstraction a distributed system is viewed as a number of cooperating distributed Recoverable Modules. Each of these Recoverable Modules performs a specific predefined function, where a single node may host a number of Recoverable Modules. A distributed application or resource may be constructed using the service of a number of cooperating Recoverable Modules, as indicated in the model of Chapter 2. A reliable system architecture that provides this abstraction in the environment of interest is constructed in three layers, the Component layer, the Recoverable Module layer, and the Recoverable Modules interaction layer as shown in Figure 6.

Each layer consists of abstract reliable components that are built using primitives provided by the lower layer, in addition to some reliability mechanisms provided in the same layer. The layer in turn provides some primitives in a well-defined interface to access its components. Reliable components of each layer attempt to mask all failures that can occur in that layer; however, non-recoverable failures are propagated to the higher layer as exceptions. In the remainder of this chapter, we identify the functions and the primitives provided by the components of each layer.

3.2.1. Component Layer

The component layer is the lowest level layer, consisting of the physical components of each site, which include the hardware processors, the storage medium used by the node, and the communication subnetwork that connects the different nodes.
This layer provides the software support responsible for detecting, analyzing, and recovering from all types of component failures. It insure that the components are restored to a consistent state and can be reliably used in case of failure. This layer provides a reliable interface through a number of reliable primitives on the top of which the Recoverable Modules of the next layer are built. This layer provides better behaved components and presents cleaner and more reliable primitives. This layer masks all failures that are recoverable at the layer (e.g., lost messages, crash and recovery of nodes, etc.) and raises as an exception the unrecoverable failures. As shown in Figure 6, the component layer includes the following components.

1. Recoverable Node (processor) Service: A service that allows for detecting the crash of other nodes and network partition and reconfiguration of the system to isolate the crashed node. If a node crashes (non-transient failure) or if it cannot be reached due to a long term network partition, all other nodes are informed of this fact, and any request directed to such a node are rejected by raising an exception. In addition, an exception is raised to all the Module's operations waiting for a service from a Module on the crashed or isolated node. Upon node recovery and restart, all other nodes are informed of the node recovery so that normal processing can be continued. In case of a transient node failure, the mechanism allows for restoring the state of all Modules that were active at the time of the node failure so that they may continue execution. It should be noted that in the case of non-transient node failure, the work done by the Module and all requested operation by the Module have to be aborted. Thus, useful work may be lost. Isolation of a crashed node aids in confining error propagation from the crashed node to other nodes.
2. Reliable Permanent Storage Service: A service that provides a stable store out of a physical disk store, which can survive a node crash with high probability and protect against read/write errors due to medium decay. The stable store mechanism also provides for an atomic write that will survive a node crash, and achieve the availability of data by replication. The information stored on stable store may be used to aid in recovery after a node failure. Hence, reliable permanent storage service provides for masking a medium failure with high probability.

3. Reliable Communication Service: A service that provides for a reliable message passing facility and masks out communication failures, such as lost messages, out of sequenced messages, duplicated messages, and garbled messages, is supported in this layer. This mechanism will allow for detecting and recovering from such failures. In addition, if a failure cannot be masked, such as a long term network partition, an exception is raised.

3.2.2. Recoverable Module Layer

Based on the primitives and the software support provided in the component layer, such as a stable store, recoverable nodes, and reliable communication, Recoverable Modules are constructed. A Recoverable Module, besides providing the normal service provided by the resource it encompasses, is designed to tolerate and manipulate failures and exceptions. The Recoverable Module incorporates some reliability mechanisms that enable it to achieve this task. This layer provides the support that enhances a regular Module defined in the model of Chapter 2 to be a fault tolerant recoverable one.

Recoverable Modules have the same structure as regular Modules.
Figure 5: Reliability System Support Architecture
described in the model. However, the Recoverable Module’s operations, executed by server processes, are defined to be atomic actions [24]. An atomic action has the totality and indivisibility properties, where either the operation is successfully executed or none of its effects take place. Thus, if a server operation is aborted before completion (due to node failure or client requesting operation abort), all objects of the Recoverable Module accessed by the server operation are restored to their original value. If the server operation successfully completes, then all the changes it made are made permanent (using reliable permanent storage), and any later failure of the node will not effect its results, thus achieving permanence of effect. In addition, atomicity implies that the atomic action is indivisible with respect to other concurrent atomic actions which means the an operation can not rely on a state produced by a different concurrent operation until the state is committed. Thus, atomic action operations form a unit of synchronization and recovery within the Module. In order to provide for atomic properties of the Module operations and enable the Recoverable Module to tolerate and manipulate failures, a number of services are provided by this layer. These services are discussed next.

1. Committing and Aborting: To implement the atomic property of Module operations, the Recoverable Module provides for committing and aborting the changes to the Module’s objects that are accessed by the Module’s operations. The commitment of the objects implies the permanence of the change made to the object since the last committed operation of the Module. The commitment of the atomic Module’s operations must be coordinated with the commitment of other atomic operations that are part of other Modules cooperating with the Module. In addition, the Recoverable Module provides for correctly restoring the
altered state of the Module’s objects in case an abort of a requested Server operation (action).

2. Module Recovery: The Recoverable Module is tolerant to node failure and interaction failure with other Recoverable Modules. In case of a node failure, and as part of node recovery, a consistent state of the Module must be restored to a state the Module existed in prior to the node failure. Using this restored state, the Module is able to continue providing the interrupted service. Such features enable the Recoverable Module to confine node failure and localize its effect. As a result, the distributed computation using the service of the Module does not have to be aborted, thus avoiding the loss of substantial healthy computation and resources and avoiding the creation of Orphans. If an exception has been raised as a result of the Module attempting the access of a service provided by another Module or due to the unavailability of the requested remote Module service (due to network partition, node failure, or autonomy), the Module is able to react to such an exception using the provided recovery semantic knowledge defined by the Module definer in the Module interface that is specified using an exception handling mechanism. In handling these exceptions, the Recoverable Module may choose an alternative service, and thus enhance availability through reconfiguration. This reconfiguration is specified by the Module definer and is application dependent. Therefore, the use of semantic knowledge is supported for Module recovery which can enhance the performance in the environment where recovery is assumed to be very application dependent.

3. Synchronization: A Recoverable Module provides for the synchronization of multiple requests of its service, such synchronization preserving the consistency of the Module objects and enforcing the
Module constraints specified by the Module definer. In addition, the mechanism uses some semantic knowledge provided by the Module definer to aid in achieving a higher degree of concurrency among requests which improves the overall performance of the system.

A synchronization mechanism to synchronize multiple concurrent operations of a Module must preserve the atomicity of each Module operation as indicated previously, preserve the local synchronization constraints of the Module, and must be coordinate with a global concurrency control mechanism to preserve the consistency constraints of a multiple cooperating Modules that are implementing specific distributed application (more on this in Section 3.2.3).

4. Interface Exception Detection and Handling: A recoverable Module is able to detect any attempted violation and misuse of its services that is defined through its interface and insures that any request conforms with the Module specification. It is also able to raise exceptions that form a response to such attempts. This is to prevent residual software or other errors from being propagated from one Module to another and to remain confined to the defective Module.

Therefore, the Recoverable Module is used to tolerate all kinds of failures, node failure, link failure, network partition, and an exception raised during Module interaction. The Recoverable Module is a unit of recovery in the distributed system that is used to tolerate all such failures. A distributed transaction, or subtransaction, accessing the Recoverable Module is not forced to abort to preserve system consistency. We term this recovery approach object based recovery as opposed to transaction based recovery, where the total distributed transaction or subtransaction is used as a unit of recovery. We believe that object
based recovery is much superior in that it provides for saving substantial useful work that otherwise would be abandoned.

3.2.3. Recoverable Module Cooperation Layer

This is the highest layer in the system structure. It provides the reliability service required to support the reliable cooperation among a number of Recoverable Modules that implement a distributed application. Such support includes the reliable Module interaction and the coordination of commitment and synchronization of the atomic Module's operations that are executed as part of the distributed transaction implementing the Module cooperation. The global commitment and synchronization services support the maintenance of the consistency constraints that encompass the set of cooperating Modules.

The Module interaction is transaction-based, where a distributed transaction is used to execute an instance of Module cooperation as was specified in the model of Chapter 2. A distributed transaction is described as a tree of server processes that is called the server access-tree shown in Figure 3. Each server process executing a requested Module's operation forms an atomic action termed atomic subtransaction; thus, each branch of the entire access-tree is an atomic subtransaction, and the access-tree forms nested atomic actions which are termed atomic distributed transactions. The top-level operation of the top level Module in the Module cooperation forms the top-level atomic action. Each remote request of the service of other Modules constitutes a nested atomic subaction.

In the nested atomic action model [33, 10, 11], each atomic action may be built by using other atomic actions called a subaction. When an
atomic subaction terminates successfully, its results remain tentative pending the outcome and the commitment of the parent (requester) atomic action. When the top level atomic action commits, then all other descendant atomic subactions must commit in a nested fashion.

The distributed transaction takes the system from a consistent state to another consistent state, and has totality and indivisibility properties. The totality property implies that the entire distributed transaction either successfully terminates or none of its effects on the state of the cooperating Modules takes place. The totality property is used to preserve the Module’s states accessed by the transaction, in case of user abort or synchronization problems.

To the user requesting the service of the Module that is provided by the top level Module’s operation (action), the total distributed transaction is seen to be atomic. If for some reason the user decides to abort the requested operation, the effects of all subtransactions, which are part of the distributed transaction, are nullified. In addition, if an atomic subaction is aborted by its requester, the effects of the subtree created by that subaction are also nullified.

The indivisibility property implies that if a number of distributed transactions, each executing an instance of a group on Modules cooperation, are executing concurrently, then a distributed transaction appears to be atomic to other distributed transactions. Tentative results and values produced by the transaction cannot be depended on by other concurrent transactions unless the transaction has been committed. A number of reliability services are provided by this layer to support the atomic property of the distributed transaction and reliable Module cooperation. These services are discussed next.
Global Commit and Abort: A distributed commit protocol is employed to coordinate the commitment of each atomic subaction, thus preserving the all-or-nothing property of the distributed transaction. In case of abort, a recovery and restoration mechanism is employed to coordinate the state restoration of all affected Modules.

Global Concurrency Control: A global concurrency control mechanism is used in coordination with a synchronization mechanism synchronizing concurrent execution of Recoverable Modules operations to control the concurrent execution of the distributed transactions in such a way as to preserve the atomic property of each distributed transaction, and thus preserve system consistency. The concurrency control mechanism employed provides for a high degree of concurrency, and thus improves performance.

Reliable Cooperating Modules Interaction: The interaction between two Recoverable Modules is based on the server/client model (Figure 7). Reliable communication is provided by using reliable communication primitives that allow for requesting remote Module service, waiting for a response of the request and aborting such a request if need be. The response may indicate the normal completion of the requested service, or it may be an exception that is returned as a response to the requester. Three types of exceptions may occur during the Modules' interaction: component exceptions which are raised by the component layer due to a component failure, such as the failure of the server node, network partition, and link failure; interface exceptions, that are raised if the request of the client violates the interface specification or that are due to the unavailability of the request Module due to
autonomy and are exceptions which are raised within the server Module. This third group of exceptions include hardware exceptions, such as division by zero, insufficient memory, or other residual software exceptions. In addition, they also include user-defined exceptions detected and raised within the Module; these exceptions may indicate normal but undesirable results which are handled differently by the client Module. An exception propagated from another requested Module must be handled by the server Module or propagated to be handled by the client in order to guarantee reliable interaction. The client may specify, either as part of its interface or within the client Module itself, a handler to react to the raised exceptions, thus providing for specifying recovery semantics that are used by the Recoverable Module.

An exception handling mechanism is employed as part of this layer's support to provide facilities to declare, raise, and handle exceptional conditions within the Module and as part of the Module interface.

3.3. Summary

The reliable distributed system support for the environment of interest is constructed in three layers. The first layer is a component layer that provides for reliable service for detecting and manipulating component failure and provides a reliable interface on the top of which is built the Recoverable Module's support of the next layer.

The Recoverable Module layer provides for constructing a Recoverable Module that is able to tolerate and manipulate failures and exceptions. The reliability services provided in this layer include a committing and aborting requested Module operation to preserve the totality property of the Module operations, to provide for restoring the Module state in case
Figure 6: Module's Interaction Model
of node failure in such a way as to confine the effects of the node failure, to synchronize the multiple requests to its services in order to preserve the Module consistency, and to treat exceptional conditions that rise during its interaction.

The Recoverable Module cooperation layer provides for reliable interaction of a Recoverable Module that implements a distributed application, distributed service, or a virtual machine. The reliability service provided by this layer includes a global concurrency control mechanism to synchronize concurrent distributed transaction in such a way as to preserve system state consistency and provide for a high degree of concurrency. It provides for global commit protocol to coordinate the commitment of each subtransaction that is a part of the distributed transaction in such a way as to preserve the all-or-nothing property of the distributed transaction. This layer, in addition, supports the reliable interaction of the multiple cooperation Module and provides support to treat exceptional conditions arising during such an interaction in order to enhance system availability.

In summary, a number of atomic distributed transactions may be executing concurrently, each accessing a number of Recoverable Modules' objects through their atomic operations, and each is executing to respond to a user request, where a distributed transaction is an instance of the access tree formed by the Module cooperation. These concurrent distribute transactions must be synchronized and each must be tolerant to failure in such a way that system consistency is preserved. The consistency constraints encompassing all the Modules accessed by a transaction must also be preserved. The synchronization of requests by a Recoverable Module form an integrated part of the global concurrency
control mechanism that synchronizes the concurrent transaction with emphasis on a high degree of concurrency. Exceptions resulting from the interaction or exceptions raised within the interacting Modules must be handled in a uniform manner.
Chapter 4
SYSTEM LAYERS SUPPORT

4.1. INTRODUCTION

In this chapter we present the mechanisms supporting the reliability services provided by each layer that aid in the construction of the Recoverable Module, as well as providing a reliable Recoverable Module cooperation with the properties indicated in the previous chapter. To implement the reliability services provided in each layer, a number of existing mechanisms are reviewed, pointing out their advantages, disadvantages, and their applicability to the environment of interest. The approach to implement each of the reliability services provided is then presented. Such approaches are, in our opinion the most applicable to the environment of interest and conform to our stated goals set forth in Chapter 2 in terms of efficiency and performance. Throughout the chapter a number of design issues related to providing the proposed mechanisms are discussed. These issues include Recoverable Module recovery, synchronization and concurrency control, commitment protocol, and reliable Module interaction. In each subsection we specify which services can be adequately provided by an existing mechanism and which require new mechanisms to implement a particular recoverable service. The specific design of the new mechanisms constitute the remaining chapters of this thesis.
4.2. Component Layer's support

This layer consists of three reliable components that provide the reliability services of this layer. These services are a reliable communication component, a reliable storage component, and a recoverable node component. To provide for the reliability support for the first two components, we feel that existing mechanisms are adequate and can be adapted [24]; however, a mechanism is needed to implement a Recoverable Node service. Such a mechanism must allow for detecting a remote node crash and network partitioning, reconfiguring the system accordingly to isolate the crashed node. Upon node failure it must start a recovery procedure to restore all Recoverable Modules and allow the recovered node to rejoin the system. The mechanisms to implement the services of this layer are discussed next.

4.2.1. Reliable Communication Component

There are a number of existing communication protocols that provide for the reliable communication services supported in this component, such as protecting against lost messages, out of sequence messages, duplicated messages generated by the network, damaged messages, and provide for the best effort to deliver a message. The LINKS protocol is one example that provides such a service [17, 45]. In this presentation we assume that such services are provided and will not address them. It should be noted that this layer is not able to eliminate duplicate messages created by the requesting Module due to a failure and that a higher level layer must provide for such a facility.
4.2.2. Reliable Permanent Store Component

The services of this component are provided in the form of a stable store facility that defines a number of primitives to be used to reliably store and retrieve information in a storage device. Stable storage was introduced by Lampson [24] by converting unreliable disk storage into an ideal device by combining pairs of non-decay related disk pages into an abstract object called a stable page. Three operations are defined on the stable pages: Stableput, Stableget, and Cleanup. The Cleanup operation is to resolve differences between pages after a crash. These three operations are built using lower level operations, Carefulput and Carefulget which operate on the physical pages of disk storage. A Carefulget reads a page repeatedly until either the page is successfully read or some limit of read errors is reached. A Carefulput is performed by writing on a page immediately followed by a read. This is repeated as needed until the date is successfully read or an error limit is reached.

A Stableput operation is implemented by doing a Carefulput to one page and then to the other. The second Carefulput is not performed until the first one is completed successfully. This is done just in case a disk crash occurs in the middle of a Stableput. A Stableget does a Carefulget from one of the pair of pages which make up a stable page. If the Carefulget is unsuccessful, a Carefulget is done for the other page that makes up the stable page.

The Cleanup operation is performed after a crash. If both pages of a stable page are good and the data is the same, then Cleanup does nothing. If one page is good and the other is bad, then a copy of the good page replaces the bad page. If both are good but contain different data, then some criteria is used to decide which version of the data will
be placed on both pages. If both pages are found to be bad, then this is a considered catastrophic event and an error occurs. The details of the mechanism are found in [24]. We assume the use of stable storage to implement the services provided by the reliable storage component of this layer, and it is not discussed any further.

4.2.3. Recoverable Node Component

The services provided in this component include the detection of node failure, reconfiguring the system to isolate the failed node, and reentering the node to the system after it has recovered. To detect that a remote node has crashed, a watch dog mechanism is employed where each node is watched by a prespecified set of nodes. In case of a non-transient node failure of the node, the watching node is responsible for informing other nodes in the system to isolate the crashed node. Each node then is to take appropriate measures to recover from such a failure, if appropriate, and avoid using the services provided on the crashed node until it has recovered. Once a failed node recovers, it informs all other nodes of the fact, and each node will update its status accordingly so that it can be used and provide services to the recovering node. The details of the mechanism is presented in Chapter 5.

4.3. Recoverable Module Layer’s Support

The reliability services provided in this layer supports the construction of a Recoverable Module that is tolerant to failure and is able to manipulate exceptions. As described in the previous chapter, the provided services include:

1. Module recovery, which enables the Recoverable Module to
tolerate node failure, interaction failures and, the restoration of Module objects in case a Module's operation aborts,

2. Committing and aborting Module operations in order to preserve the atomic property of the Module's operations,

3. Synchronization of a Module operation to preserve Module consistency, and the increase of the degree of concurrency to enhance efficiency, and

4. Provision for detecting and handling interface exceptions.

Next we will discuss the design issues involved in providing each of the services of the layer, review some existing mechanisms that may be employed to implement the services, and propose novel approaches to implement the services when or where no existing mechanism is judged to be adequate.

4.3.1. Module Recovery

There are three related problems of recovery. 1) How is a Module state restored in case of a node failure so that it is possible to continue the execution of the interrupted operations and to confine the effects of this failure, therefore causing the minimal waste of useful work due to such failure? 2) How can a Recoverable Module using recovery semantic knowledge recover from interaction exception, such as the unavailability of the service of a remote Module, so that it is able to continue providing the requested service and thus support system availability? 3) How is the state of Module's objects restored when a Module's operation aborts in such a way as to preserve the atomic properties of the Module's operations?
In discussing the possible recovery mechanisms that can be employed by a Recoverable Module, we would like to distinguish the treatment of these three related problems. We will address each of these three problems, but first we will examine the possible recovery mechanism available.

Recovery Techniques

As part of recovery, the state of both Module objects and server processes have to be restored to a consistent state. Depending on the way a consistent state is restored, recovery mechanisms may be divided into two broad categories: Forward recovery and Backward recovery [37]. Forward recovery techniques use the present faulty state to arrive at some consistent state. In backward recovery, a prior consistent state from which a computation can restart is restored.

Forward recovery techniques are dependent on having identified the fault and all its consequences. Error diagnosis, damage assessment, and repair becomes an important part of the construction of the forward recovery technique and directly relates to the question of how to continue providing the specified service. Forward recovery requires a complete understanding of the particular application for which the Module may be used. Thus, generalized applications of forward recovery mechanisms are not quite feasible as indicated by [37]. Based on the types and assumptions about the faults that can occur, and the resulting damage, a forward recovery can be both simple and much more efficient than backward recovery. The most common technique for forward recovery is exception handling [26], where exceptional conditions are the anticipated error conditions in the system.
Backward recovery techniques require facilities for establishing recovery points which, after a failure, are used to reconstruct or restore the state of the system to the most recent state existing at the most recent recovery points prior to the failure. Thus, backward recovery usually involves undoing everything that has been done since the last recovery point. Backward recovery is relatively simple due to two facts; first, the question of damage assessment and repair are treated quite separately from those that continue to provide the service specified; secondly, the damage assessment takes virtually no account of the nature of the fault involved.

A number of techniques are used in backward recovery which include atomic actions and checkpointing. These techniques differ depending on the amount of system activities or computation that will be abandoned as part of recovery. For example, atomic actions, in case of failure, will restore the state of the objects and the process manipulating the object within the atomic action to their initial state which existed before entering the atomic action, thus abandoning all the activities performed within the atomic action. Techniques, such as careful replacement, multi-version techniques, and logs/audit trail, are used in implementing atomic action to restore the state of the objects (database) accessed within the atomic action.

Checkpointing is a backward recovery technique that is implemented by periodically saving the current state of the system, which includes the state of the objects and the state of the processes manipulating those objects. In case of failure the system is restarted from the last checkpoint. Thus, unlike atomic action the state is restored to the last checkpoint and less computation and activities need to abound.
In a centralized system, a checkpoint is established system wide by saving all processes and object statuses. System wide checkpointing has at least two disadvantages. First, the system has to slow considerably at the time of checkpointing. Second, in the distributed system it might not be feasible or practical to establish checkpoints across all system nodes.

In a transaction based system, a checkpoint can be established individually for each transaction, rather than having a system wide checkpoint where each transaction is checkpointed periodically. A transaction, in case of a failure, is backed to the last checkpoint, thus saving useful computation.

Our approach is to incorporate both forward and backward recovery techniques to recover from different types of possible failures. Unlike a simple transaction mechanism, where the different failures are treated the same way, that is, simply by aborting the transaction and thus causing unnecessary loss of computation, our approach is to treat such failures differently in such a way that healthy computation are saved and failure effects are confined. Next, we will discuss the possible approaches to treat each Recoverable Module's failures and present our solution.

**Recovery from Node Failure**

A number of approaches can be used to recover from node failure. We will discuss some of these techniques, examine their advantages and disadvantages, and indicate their adaptability to our environment, as well as introduce a new mechanism to handle node failure.

1. **Distributed Atomic Transaction**: This approach calls for the use of the entire distributed transactions as a recovery unit. In case of node
failure, where part of the distributed transaction has been executed, the state of the object of all the Modules accessed by the transaction will be restored to their initial value before such a transaction has started and the entire transaction is aborted. Since an atomic transaction takes the system state (state of objects of the system Modules) from one consistent state to another, the system state will then be restored to a consistent state. This approach has the disadvantage that all of the useful work done within the entire distributed transaction is aborted. In addition, the approach does not confine the node failure and localize its effects. This approach is most appropriate for a centralized data base system where transactions are small and a node failure causes the total failure of the system. In the environment of interest, a distributed transaction may access an arbitrary large number of objects of Recoverable Modules and thus abort the entire transaction which may cause substantial loss of work. In addition, different Recoverable Modules that are accessed by the transaction may fail independently, since they may reside on different nodes; thus, there is a chance to confine and isolate the affects of their failure.

2. **Atomic Transaction and Checkpointing:** In a transaction based system, each individual transaction may be checked-pointed individually by saving the state of the transaction objects and processes. In the distributed system of interest, a distributed transaction consists of a nesting of atomic subaction that may run on different nodes and may fail independently. Thus, one approach is to have every subaction issue its own checkpoints. In case of failure, only the affected subaction (the subaction executing a Module operation that resides at the failed node) is backed to its last checkpoint and restarted from that point. This approach has the advantage that the entire distributed transaction does
not have to be aborted, and thus useful work (computation) can be saved.

The main disadvantage, however, of this approach is that any useful work performed by a healthy child, which is a subtransaction initiated by the failed subtransaction after the last checkpoint, and its access-tree is lost, thus losing useful computations and wasting resources. For example, in reference to Figure 9, a failure of Module B will cause the access tree created by its remote request of operation op2 of Module C to be aborted.

A checkpoint can be placed anywhere in the subaction and can be established many times within the subaction. However, placement and frequency of checkpointing within a subaction affects efficiency. For example, in Figure 9 again, operation Op1 of Module B makes a remote request to operation op2 of Module C, then makes a request to op3 of Module D. If a checkpoint is established before both remote requests of B to C and D (ck1) and if a crash of B occurs after C responded but while B is waiting for D, then continuing the processing from the last checkpoint will cause a duplicate request to C, even though it has finished. This means the requests to C and D became orphans [34], and the useful computations performed by op2 and op3, along with their access trees, are lost. It should be noted that operations op2 and op3, if not aborted, will cause system inconsistency.

A reasonable approach is to establish a checkpoint before each remote request in order to limit the backup to only the in-progress remote request. There are two problems associated with this approach. First is the creation of orphans due to the rerequesting of the remote subaction. The second problem that affects efficiency is due to the frequency of
Figure 7: Effects of Subtransaction Failure
establishing checkpoints. A checkpoint involves substantial work, which includes saving the current value of the objects used by the subaction on a stable store as well as a transaction state. If a subaction invokes a large number of remote requests, then the high frequency of establishing checkpoints is very costly. This goes against our goal of incurring minimum overhead and cost during normal operation and a reasonable cost during recovery.

3. Recovery Lines and Remote Access Lists: This is a different approach which eliminates the problems mentioned above and provides better efficiency forms which are the basis of our recovery form node failure mechanism. The approach uses a simple recovery line and remote access list. A simple recovery line involves saving the request message from the requester. A remote list is a list of all remote requests and possibly responses that a subtransaction may perform. These lists are saved on a stable store. In case of failure, the recovery management routine of the Recoverable Module is responsible for aborting all in progress operations, restoring the Module objects, and using a recovery line and remote request list, and restores the Module's operation to the state existing before the crash so that interrupted processing can continue. Procedures for recovery from node failure and restart of a subaction will be presented in detail in Chapter 5. Establishing simple recovery lines and saving requests on remote request lists become part of the server operation manager of the Recoverable Modules. Maintaining and manipulating objects versions is part of object-management of the Recoverable Modules.

Recovery from Interaction Failure

A server process that is executed on behalf of a remote requester itself
requests the service of another remote Module; such a request may not be completed due to a remote component failure or an interface failure. Therefore, the distributed transaction may not progress any further. In a simple transaction system the entire distributed transaction may have to be aborted if such an event occurs, thus causing the loss of substantial computation power. Our approach is to use forward recovery by using exception handling to recover from such exceptions. An exception handling mechanism allows the Module definer to specify the recovery of semantic knowledge to handle such exceptions. The Module's definer may specify an alternative service that can be used (if available) which might exist on another available node. However, the original request and the access tree it created must be aborted. The exception handling mechanism provides for primitives that can be used to retry the same request, choose an alternative service, or raise an exception to the requester of the operation.

An exception handling mechanism will be proposed in Chapter 8, and further details are given in this chapter in Sections 4.3.4 and 4.4.3.

Recovery from User Abort

We assume that a subtransaction of the entire distributed transaction may only be aborted by the requester (which may be the requesting Module or the user initiating the distributed transaction) or by the system due to synchronization problems as will be explained in Section 4.4.1. In such cases objects of the cooperating Modules accessed by such a transaction or subtransaction must be restored to their initial state in such a way as to preserve the atomic property of the distributed transaction and the subtransactions. Each Recoverable Module provides for maintaining and managing its own objects, in such a way that it is
able to restore to a consistent state such objects in case of aborting a server operation accessing the objects.

A number of methods may be used in restoring the Recoverable Module objects to their initial value. Such methods include the following:

1. **Careful Replacement** [45]: In this method, two copies of the object are maintained while it is being updated, the current copy and a shadow copy. Updates are made to the current copy, and the shadow copy maintains the version before the update. Upon abort the shadow copy is used to restore the object. Of course, such a copy must be saved in stable store to be able to survive a node crash.

2. **Multiple Version** [38]: Updates to the object are performed on a new version which upon commitment becomes a permanent copy. In case of abort the new version is discarded; therefore, restoring the object is simply done by discarding the updates performed on the new version of the object. In this approach we also must save versions in stable store so that they are able to survive a node crash.

3. **Logs/Audit Trail** [15, 28, 16]: In this method all operations performed on an object are recorded on a log or audit trail. The log is used to support the restoring of the object in case of abort by undoing the logged operations. The log is also used to redo some of the operations that are part of committed transactions. In this approach logs and objects must be saved in stable store to be able to survive a node crash.

Our approach calls for using a combination of an extension of careful
replacement in the form of multiple versions and differential file techniques to maintain the objects of a Recoverable Module. A new version of the object is created for each Server operation instance that is accessing the object. The object version will be written on the permanent object when the distributed transaction accessing the version of the object commits; otherwise, the version is discarded. Since a Module object may be large, the multiple versions are then kept as differential files where any new updates to an object are recorded on a differential file. Periodically the differential file is merged into the object and so it is more efficient and less expensive to maintain the versions of the object. An implementing version as differential files has been used in [44] in implementing a multi-version approach similar to Reed's [38]. In our approach we assume that uncommitted versions of the object do not have to be saved on stable store; thus, it is more efficient than the existing techniques. The details of the object management of a Recoverable Module will be explained in Chapter 5.

4.3.2. Synchronization of Recoverable Module's Operations

Synchronization problem are twofold. First, how do we synchronize server processes executing a Module operation so as to preserve the Module consistency constraints and provide a high degree of concurrency? Second, how can distributed transactions that share access to a Module's object be synchronized to preserve the consistency of the Modules they encompass, preserve system consistency, and allow for a high degree of concurrency?

In this section we will discuss the synchronization of server processes executing Module operations. Such synchronization is performed by the Recoverable Module so as to preserve the Module consistency constraints.
It must be an integral part of the global consistency control mechanism which synchronizes the concurrent execution of distributed transaction in order so to preserve the consistency constraints of the collection of cooperation Module that the distributed transaction encompasses.

A synchronization mechanism, besides preserving the consistency constraints of the Recoverable Module, must allow for the specification of such constraints. The constraints information is provided by the Module definer, as well as semantic synchronization knowledge that may be used to increase the degree of concurrency of execution of Module operations, and ultimately increases the degree of concurrency of distributed transactions, thus enhancing performance. For example, the specified constraints may indicate what Module operations are compatible and can be executed concurrently without effecting consistency, whether strong serializability is required for correctness for the particular application, and if different implementation of the Module allows more concurrency. These points will be elaborated on further in this chapter in Section 4.4.1 on global concurrency control.

Traditional synchronization mechanisms, such as monitors [17], Path expression [8], event counters and sequencers [39], and selectors [25], are used to solve the mutual exclusion problem and synchronization of sharing OS resources. In spite of the fact that they are concerned with controlling concurrent access to share objects and allow for specifying the Module's constraints and semantic knowledge, they cannot be applied directly to solve the synchronization problem in the distributed environment of interest. The main reason is that they provide for controlling and maintaining the consistency of those resources they control (protect, in the sense of monitor), but do not coordinate and
maintain the consistency constraints of a group of Modules that are accessed by distributed transactions this can cause inconsistency due to lost update anomaly or inconsistent retrieval anomaly. For example, we can suppose transactions $T_1$ and $T_2$ require to insert and remove an item $X$ in a bounded buffer object, respectively. Using a traditional synchronization mechanism, the following interleaving of the transaction requests is acceptable (Figure 10). $T_1$ inserts $X$ in the buffer object, and before $T_1$ commits, $T_2$ removes $X$ from the buffer. However, if $T_1$ is later aborted before it commits, then $T_2$ would have used an inconsistent value of $X$ (was not meant to be inserted). This is called the inconsistent retrieval anomaly. A similar problem may occur if $T_1$ updates the object $X$ of Module $m_1$ (using server $S_1$). If $T_2$ is allowed to update $X$ and if $T_1$ is aborted, the $T_2$ update will be lost. This is called the lost update anomaly and may be permitted by a traditional synchronization mechanism such as Monitors. This problem occurs basically because the synchronization mechanism does not insure that $T_2$ may not proceed with removing $X$ before $T_1$ has normally terminated and has committed all the changes it made to the objects. In addition, it does not insure that the commitment (normal termination) of $T_2$ is dependent on the normal termination of $T_1$. Another problem with existing traditional synchronization mechanisms is that they are designed to maintain and control access to a permanent copy of the object. In our design they must be extended to maintain and control access to multiple versions of the object as proposed in Section 4.3.1.

In sum, it is clear that traditional synchronization mechanisms must be extended and combined with a global concurrency control mechanism to allow for the specification and maintenance of Module consistency constraints, in addition to preserving the consistency constraints of a
Figure 8: Bounded Buffer
number of cooperation Module, an instance of which is executed by a
distributed transaction. Our approach is to devise such a mechanism
which will be discussed further in Section 4.4.1, and the details of which
will be given in Chapter 7.

4.3.3. Committing and Aborting of Recoverable Module's Operations

The operation of the Recoverable Module is executed by server process
as an atomic action, and thus the effects of such an operation are either
successfully terminated or none of its effects on the Module objects take
place. The Recoverable Module's support provides for committing and
aborting Module operations. This simply can be done by writing the
uncommitted version used by the operation into the Module object in
case of commit, or discarding such version in case of abort. However,
the commitment and the aborting of the local Module operations has to
coordinate with other operations that are running as subtransactions of
the same distributed transaction. Existing mechanisms to implement a
commit protocol and their applicability to the environment of interest
will be discussed in Section 4.4.2 on global commit protocol.

4.3.4. Interface Exception Detecting and Handling

A Recoverable Module is able to detect and raise as an exception any
attempted violation or misuse of the service it provides. Such an
exception, for example, includes invalid parameters, unauthorized access,
or violation of Module constraints. In such cases the Module will reject
such a request.

An exception handling mechanism is provided as part of the
Recoverable Module layer support to enable the Module definer to specify
as part of the Recoverable Module interface the conditions under which a requester is violating or misusing the Module service. It also provides for detecting such violations and raising exceptions as a response to the request. This support is provided as part of a comprehensive exception handling mechanism to declare, raise, and handle exceptions that arise during the interaction of a group of cooperating Recoverable Modules that implement a distributed application. Such a mechanism will be presented in Chapter 8, and will be discussed further in Section 4.4.3 on Recoverable Module interaction.

4.4. Recoverable Module Cooperation Layer Support

In this section we discuss the needed mechanism to provide for the reliable Recoverable Module cooperation to support reliable distributed applications and reliable distributed subsystems. The section discusses the mechanisms to support the following reliability services provided by the layer:

Global concurrency control,

Global commit and abort protocol, and

Reliable Recoverable Module interaction.

In each subsection we will examine the adaptability of the existing mechanisms and point out their disadvantages and then propose our approach to providing such mechanisms to provide for reliability services of this layer.
4.4.1. Global Concurrency Control

In the model specified for the distributed environment of interest, a distributed transaction is defined to be an instance of Module cooperation at execution which may be processing a distributed computation. A distributed transaction is atomic to failure and to other concurrent transactions. This is required so as to avoid the lost update and inconsistent retrieval anomalies [15].

In addition to the traditional synchronization mechanisms, such as path expressions, sectors, etc., there are a number of existing approaches proposed with which to solve the synchronization problem. In this section we will examine the advantages and disadvantages of these mechanisms when applied to resolve the concurrency control and synchronization problem of the environment of interest.

The concurrency control mechanisms used in distributed database systems can be classified in two main categories based on synchronization primitives [6, 20]. These are locking based mechanisms and time stamp based mechanisms. Locking mechanisms include schemes, such as the basic two phase locking [15] scheme, primary copy two phase locking, majority consensus, and centralized two phase locking. Time stamp based mechanisms include schemes, such as the basic time stamp ordering scheme, multi-version time stamp ordering, and conservative T/O. It is not our intent at this point to explain the details of each of these schemes, but we will present the basic idea behind each of the two major classes so that it is possible to discuss their applicability in the environment of interest. The reader is referred to the excellent survey in [6].
Berstine and Goodmen have surveyed other concurrency control methods which do not fit the locking and time stamp ordering classification. These methods include the certifiers, which is an optimistic approach [5], where access requests of objects are processed on a first-come-first-served with no synchronization at all. Conflicting read-writ operations and write-write are never blocked or rejected, but only a conflict information summary is maintained by each database manager. Synchronization occurs when a transaction tries to terminate (or commit all its changes). Using the conflict summary information of each site, a transaction is certified (allowed to commit) if it does not conflict with other transactions. Because summary information is required from all sites even though they might have not participated in processing a transaction being certified, this approach can incur a lot of overhead and can be judged as being impractical in a distributed system [20]. Other optimistic mechanisms were presented for distributed database systems, for example, Kung's and Ceri's [23, 9]. These mechanisms are based on timeouts to detect conflicts and thus can abort transactions unnecessarily and we believe that they may be inefficient in our environment. These mechanisms will be discussed in detail in Chapter 8.

In Section 3.1.3.2. it was concluded that traditional synchronization mechanisms in their existing form cannot support preserving the consistency constraints of a group of cooperation Modules that are encompassed by a distributed transaction. It was also observed that traditional mechanisms have to be combined with a global concurrency control mechanism to solve the synchronization problem in the environment of interest.

The applicability of Locking and time-stamp mechanisms to the
environment will be discussed below; in turn, this discussion will be followed by a summary pointing out our new approach to solving the synchronization problem in the environment of interest.

**Locking Mechanisms:** Locking schemes [15, 12] are based on having a transaction lock objects to insure their unaccessability while in a temporary inconsistent state. In the simplest case each object has a unique lock which is held by at most one transaction at a time. When a transaction attempts to access, i.e., read or write an object, a lock request by a transaction is granted, if and only if the associated lock is not being held by any other transaction; otherwise, the request is delayed.

To improve concurrency, operations may be classified as read or write operations, and each object will have two locks, read lock and write lock. Before reading the object, a transaction must obtain a readlock. Before writing into the object, it must obtain a write lock on the object. A transaction must hold all locks until it terminates (commits). Obtaining locks is governed by two rules: (1) different transactions cannot simultaneously own conflicting locks; and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks.

The definition of a conflicting lock depends on the type of synchronization being performed: for rw (read/write) synchronization two locks conflict if (a) both are locks on the same data item, and (b) one is a deadlock and the other is a writelock; for ww (write/write) synchronization two locks conflict if (a) both are locks on the same data time, and (b) both are writelocks.

There are a number of drawbacks to applying lock techniques in the environment of interest which include the following:
Serial Processing of Concurrent Transactions: Because a transaction must retain a lock to the object until it completes, all concurrent transaction sharing access to that object must be delayed until the object is released; thus, concurrent transactions are processed almost serially. Serializing transaction processing can restrict the degree of concurrency and degrade performance.

Simple Objects Type and Operations. In most existing locking-based mechanisms, objects are assumed to be independent database entities directly associated with physical or logical storage units (pages, records, or files), and operations are assumed to be simple Read and Write operations. In the model for this environment, objects are assumed to be abstract and user defined, as well as the operations on these objects.

Module Consistency Constraints: The locking mechanisms do not provide for specifying and maintaining the Module imposed consistency constraints that are Module dependent, where the Module is able to schedule requested operations in such a way as to preserve the Module dependent constraints.

Interleaving of Compatible Operations: Compatible Module operations on shared objects may be interleaved in real time without causing conflicts (reads, for example, may be performed concurrently). Compatibility of the Module's operation are Module dependent and may be specified as part of the Module interface specification. This semantic knowledge can increase the degree of concurrency and enhance
performance. Locking based mechanisms do not provide for the specification of such semantic knowledge.

**Specification Versus Implementation Behavior:** In general, a Module can be implemented in many ways and still meet its externally specified behavior requirements. Particular implementation details may affect the amount of concurrency allowable. For example, we can consider a Module that maintains a directory object with insert and remove operations. Simultaneous inserts may be forbidden to preserve correct specification behavior. However, implementation details may allow two insertions which affect different directory entries to be done without conflict. More concurrency could be achieved if such knowledge is used in synchronizing concurrent requests of Module operations.

**Serializability Requirements:** A strong serializability [43, 3] of concurrent transactions as a condition of correctness, such as the one required by locking mechanisms, may not be required for all applications. For example, we can consider a mail system that accesses a number of Modules where each Module is implementing a user mailbox. The mailbox is implemented as a FIFO queue with operation insert and remove. We can assume that there are two concurrent transactions initiated by two different senders, s1 and s2, and each is to deliver a message to two different users, a and b. A schedule produced by strong serializability requires that after the execution of both transactions, the order of the message in mailbox a and b is identical: s1 followed by s2 or
s2 by s1. A closer look at this application reveals that the order of receiving messages is not important in this application, and thus such strong serializability is not required. Some application will except a weak serializability or serial behavior for different reasons such as performance [18, 32], availability [13], and simplicity [35].

In summary, lock-base mechanisms make minimal use of semantic knowledge about the objects and the operations used by the transactions and therefore prevent delay or abort transactions unnecessarily.

**Time Stamp Ordering Mechanisms**: Time stamp ordering [6] is a technique whereby a global (across all nodes) serialization order is selected before the initiation of a transaction, and the execution of the transaction is forced to follow the global ordering, thus guaranteeing serialization. The transaction is assigned a unique time stamp (TS) when first initiated, and when the transaction is to access an object, the following protocol is followed: in the case of a Read request, if TS of the transaction is less than the largest time stamp of a write request that has already been processed, the read request is then rejected, and the requesting transaction is aborted. Otherwise, the read request is processed. In the case of a Write request by the transaction, the request is rejected and the transaction is aborted if the TS is less than the largest time stamp of a transaction that requested a read or write request which is already processed. Otherwise, the write request is processed.

The multi-version time-stamp ordering mechanism introduced by Read [38] improves the degree of concurrency of T/O, by using multi versions for each object. Each version covers a specific span of time (time
stamp), so out-dated transactions can still be processed if they do not conflict with other more recent transaction requests.

Time stamp ordering has a number of drawbacks when applied to solve the concurrency control and synchronization problem in the environment of interest.

The first and most serious drawback is that T/O imposes a priori ordering on the processing of distributed transactions that have to be followed when processing the requests of these transactions by a Recoverable Module. Such global ordering may conflict with the synchronization constraints imposed by the Module. Going back to our bounded buffer example of Figure 8, we see that two transactions, T₁ and T₂, are initiated with time stamps TS₁, TS₂ where TS₁ < TS₂. If as part of T₁ a Module receives a request to remove an item from the buffer and if as part of T₂ the Module receives an insert request and if we assume the buffer is empty, the global ordering of the time stamp requires that T₁ must be performed before T₂. Thus, the remove request must be performed before the insert request. This violates the synchronization constraints of the Module that requires no remove operation to be performed while the buffer is empty. Therefore, time stamp ordering may not be suitable for the environment of interest.

Time stamps also share many of the drawbacks of locking schemes. They make minimum use of semantic knowledge, impose strong serializability requirements, and generally assume simple objects and operations.

**Extended Selectors:** It was shown that the existing concurrency
control and synchronization mechanisms at the present form cannot be directly used in solving the synchronization problem in the environment of interest. In this survey of the existing synchronization and concurrency mechanisms, Kohler [kohl81] has concluded that in their present form, none of these mechanisms can support concurrency control in a general object-based distributed environment. A useful mechanism that avoids the drawbacks of the existing mechanism and is suitable for the environment of interest must provide for the following: It must

- support arbitrary user defined objects structure and operations,
- support the specification and use of semantic knowledge provided by the Module definer to achieve a higher degree of concurrency,
- minimize the delay that is incurred by a transaction due to strong serializability requirement,
- allow for the specification and preserving of Module constraints, and
- allow for preserving the consistency constraints that encompass the consistency constraints of the group of cooperating Modules that implement a distributed application.

Kohler suggested that such mechanisms will probably be hybrid mechanisms based on the existing synchronization methods. Our intent is to design such mechanisms. The proposed mechanism combines the use of a traditional synchronization mechanism, namely, selectors [25], with a novel optimistic concurrency control mechanism. Selectors are
extended to reliable selectors which are fault tolerant (mainly survive a node crash). Selectors will provide for specifying semantic knowledge and Module's constraints, and synchronizing access to the Module's object so as to preserve the Module consistency constraints. The optimistic concurrency control coordinate with reliable selectors to guarantee that consistency constraints are preserved encompasses the group of cooperating Modules accessed by a transaction and multiple concurrent distributed transactions are synchronized to preserve system consistency. This mechanism incorporated the use of a form of a multi-version technique, unique system wide id (UID), and deadlock detection techniques. The details of the mechanism will be explained in Chapter 6.

4.4.2. Global Committing Protocol

Committing protocol involves determining how to commit a change to the Module's objects made by server operations as part of a distributed transaction in coordination with the commitment of changes to other Modules' objects accessed by the transaction.

A commit protocol is needed to cooperate the commitment of each individual subtraction that is part of the atomic distributed transaction in such a way as to preserve the atomic property of the entire distributed transaction. If a transaction has to be aborted, then all the objects it has modified must be restored to the state before the transaction began.

1. **Two Phase Commit**: The two phase commit protocol [15] has been widely used in many variations in data base systems. The basic idea behind the two-phase commit protocol is the following: when a
transaction has finished and is ready to commit, a first phase prepare message is sent by a centralized commit coordinator to all participating cohorts. All the cohorts prepare to commit and send back an ack message. In the second phase, if all participating cohorts respond that they are ready to commit, then a commit message is broadcast by the coordinator to the participants to commit; otherwise, the coordinator tells all the participants to abort.

Moss [33] has extended this protocol to coordinate the commitment of a nested transaction in a distributed system. Moss implements the commitment of a nested transaction as follows: when the entire nested transaction has finished and is ready to commit, a first phase prepare message is sent by the transaction manager of the top level transaction (coordinator) to all transaction managers where subtransactions, which are descendant of the top level transactions, are executed (participants). In the second phase, if all participating transaction managers respond that they are ready to commit, then a committee message is broadcast by the coordinator to the participants to commit; otherwise, the coordinator tells all the participants to abort.

In his approach, Moss requires that the entire transaction must be aborted if the node where a subtransaction is executing fails. This can cause a substantial loss of healthy computations performed by the nested transaction.

In addition, Moss assumes in this protocol that the top level transaction manager can directly communicate with all the participants to coordinate the commitment of the nested transaction. In our model of cooperating Recoverable Modules, we assume that cooperation is formed in a hierarchical fashion where, for the sake of modularity,
abstraction, and the post facto introductions of a Module (a Module specified and implemented without reference to its eventual use) to the distributed system, a requesting Module is unaware of the implementation of the service it requests from the other Modules. Thus, the direct communication of Modules at all levels of the tree with the top level Module might violate the principal of modularity and abstraction, and hence this approach is undesirable.

2. Hierarchical Commit: Our approach is to provide a hierarchical commit protocol that is based on a two-phased commit that is suitable for the hierarchical tree-like structure, representing the Module cooperation in the distributed environment of interest which supports abstraction and modularity. The commit protocol guarantees the atomic property of the distributed transaction. In addition, the commit protocol does not automatically abort the distributed transaction in case of a node failure or an interaction failure, as in the case in Moss’s approach, but the abortion of the distributed transaction is explicitly done by the requester using the exception handling mechanism. A node that has crashed is restored to its state before failure by the recovery mechanism, and thus there is no need for aborting the transactions that have been executed on the node (but have not been terminated). Interaction failures are propagated to the Modules as exceptions, and it is the requesting Module’s decision to abort the subtransaction it requests.

The commit protocol is implemented as part of the optimistic concurrency control mechanism presented in Chapter 7. Before a transaction can normally terminate (commit), a two-phase protocol is initiated. A validation phase is used to prepare a transaction to commit and to check if the transaction does not conflict with other concurrent
transactions. If validation is successful, then a commit phase is started to commit all the changes made by the transaction to the Module's objects it accessed.

4.4.3. Recoverable Module Interaction

To support the reliable interaction of a group of Recoverable Modules, two main issues have to be addressed. First, how is reliable communication provided for so that a Module can reliably request and receive the service of another Module? The second issue is how are exceptional conditions that arise during Module interaction treated? Such exceptions must be properly handled to guarantee the successful and normal progress of the Module cooperation.

The second issue is determining how to handle an exception that may result during Module interaction to guarantee a reliable interaction. Three types of exceptions may occur during the Module's interaction. They were discussed in Section 3.2.2 of Chapter 3 and include:

- component exceptions due to component failure of the remote node of the server Module, or due to network partition,
- interface exceptions due to the violation of some Module constraints or illegal access of the server Module, and
- Module exceptions which include user defined and system exceptions raised within a Module operation or a propagated exception from a requested remote Recoverable Module.

The use of exception handling in recovery and Module's interaction has a number of advantages which include:
System availability may be enhanced since the unavailable service may be replaced by another to continue the distributed transaction progress.

Semantic knowledge about the specific distributed application is incorporated into the recovery mechanism, which can enhance performance since the designer (definer) of the Module knows best which alternative service may be best suited to the application used and what are the desirable recovery steps to take. This is especially true in this environment where multiple different distributed applications are processed.

Using exception handling helps contain uncontrolled error propagation and provides a well-defined manner of propagating and handling exceptions among the cooperating Module.

An issue that is important in designing an exception handling mechanism is the transfer of control after the execution of a handler in response to an exception. There are three exception handling models which specify different transfers of control: the termination model, Continuation model, and Propagation model. Most exception handling mechanisms [179, 26, 29] are based on one of these models which may not be flexible and adequate in handling all exceptions that can be raised in Module interactions. We feel that all three models should be adopted by the exception handling mechanism to give the user more flexibility. We propose four primitives that can be used within the handler which will explicitly transfer control according to one of the three basic models. In addition, a Retry primitive is proposed that will allow a request to be retried (rerequested) for a fixed number of times.
Another issue that is important when designing an exception handling mechanism for the environment of interest is language-incompatibility. It is assumed that Modules are written in different languages; some of these languages provide their own exception handling mechanism. Thus, exceptions within the Module may be raised or handled using such a facility. Since the different languages provide an incompatible exception handling mechanism, such an exception may not be properly handled. The proposed exception handling mechanism forms a standard form. An exception raised within a Module is then converted to a standard exception and raised to the requesting Module. The exception is converted to an exception in the format of the language used by the requesting Module, if the Module provides a handler for it. The details of the proposed exception handling mechanism are given in Chapter 8.

4.5. Summary

In this chapter, the implementation of the different services provided by the three layers of the system are discussed. The applicability of a number of existing mechanisms to provide the suggested services are discussed, pointing out their advantages and disadvantages. Some new approaches are suggested, and the details of these approaches will be presented in the remaining chapters.

In the Recoverable Module layer, a number of services are provided to enable the Recoverable Module to tolerate node failure and interaction failure as well as to be able to restore the Module object in case of a Module's operation abort. In addition, we presented a Recoverable Module able to synchronize a concurrent Module's operation using semantic knowledge to increase the degree of concurrency and to provide for the atomicity of the Modules operations.
In the Module’s cooperation layer, a number of services are provided to guarantee the atomic property of the entire distributed transaction executing an instance of the Module’s cooperation. These services include global commit and abort and global transaction synchronization. In addition, the layer provides for services to treat exception conditions resulting during the Module’s interaction to support system availability confine error propagation and controls complexity.
Chapter 5
RECOVERY

In Chapter 4 two distinct recovery services were discussed. The first is provided by the recoverable node component of the component layer which provides for the detection, isolation, and recovery of a node crash in relation to other nodes that are part of the distributed system. The second is the recovery service provided by the recoverable Module which enables a regular Module to tolerate and manipulate failures and exceptions during Module interaction.

In this chapter we address the design and the implementation of both services. In Section 4.1 we discuss the design and implementation of the recoverable node component, and in the remainder of the chapter we discuss the design and implementation of the mechanisms to support a recoverable Module.

5.1. Recoverable Node Component

All nodes collectively cooperate to detect a failure of a participant node and reconfigure the system by isolating this node and enabling the node to rejoin the system when this node recovers. The recoverable node component of the component layer consists of three routines that are employed in coordination with other recoverable node components of other nodes to detect node failure, reconfigure the system, and restart a crashed node. The detection routine is responsible for detecting the
failure of other nodes in the system and informing the participant nodes of the fact. The reconfiguration routine takes the appropriate measures to isolate the crashed node so that no more requests (messages) are directed to the failed node and to reestablish communication with a failed node once it is up. In case of a node failure, the node restart routine is responsible for restarting the node and taking the appropriate steps to bring back the node.

A node may be in one of three states, UP, DOWN, or RECOVERING. Each node keeps status information of all other nodes in the system. A data structure that contains N cells (where N is the number of nodes in the system) is used to keep such information (STATUS LIST) as shown in Figure 11. Each entry of the list contains two fields, node id and node status.

**Detection of Node Failure** To detect the crash of a node, the concept of watch dog is used. Each node is watched by a specific number of nodes. One is the primary watching node and the others are secondary watching nodes. The secondary watching nodes are arranged in a specific order. If the primary watching node fails, then the watching responsibility is transferred to the first secondary node in the order. A watching node is responsible for detecting the failure of a watched node and informing all other nodes of the fact. To detect the failure of the watched node the watching node performs the following functions:

If it has some active communication with the watched node, then it is an indication that the watched node is still up.

If there is no active communication with the watched node, a priority control message is sent to the watched node after a
D: Down
U: UP

Figure 9: Node Status List
specified period of ideal communication. The priority message is processed immediately and acknowledged by the watched node to indicate that it is up.

If the watched node does not acknowledge the priority control message after a limited number of tries in a specific time range (the time range is longer than the time for a Module to recover from a transient failure), then the watched node is declared down, and all other nodes are informed of that fact.

If the watching node fails, then its watching node assumes its watching responsibility (Secondary watching node). Each node maintains a statically created list called an Assignment List (AL) which maintains the watching responsibility of each node as shown in Figure 12. In addition, each node maintains a list of actively watched nodes (AWN) shown in Figure 13. This list includes all the ids of the nodes watched by this node. The AL is saved on the stable storage. If any of the watched nodes in the AWN list fails, the AL list is used to identify all the nodes watched by the failed node, and the ids of all such nodes are inserted in the AWN to be watched by the node. However, before these ids are inserted in the AWN, the status list is checked to make sure that the nodes to be inserted are not down. If one is down, then the process is repeated for it by identifying the nodes it is responsible for watching and inserting them in the AWN list. When a watched node has failed and then recovered, its id is inserted back in the AWN list and all the nodes it is responsible for watching are deleted from the AWN list.

It should be noted that if both a watched node n1 and its watching node n2 fail simultaneously, the algorithm will still work correctly and both nodes will be isolated. For example, if n3 is watching n2, and n2
<table>
<thead>
<tr>
<th>Node</th>
<th>Watch Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>N2</td>
</tr>
<tr>
<td>N2</td>
<td>N3</td>
</tr>
<tr>
<td>N3</td>
<td>N4</td>
</tr>
</tbody>
</table>

Figure 10: Assignment List
Figure 11: Active Watched Nodes List
is watching n1, and if both n1 and n2 fail simultaneously, n3 will detect the failure of n2 inform other nodes to isolate n2, and remove n2 from its AWN list and insert n1 as well as the ids of other nodes watched by n2. Hence, n3 becomes the watching node of n1. Now n3 discovers that n1 has failed and informs all other nodes to isolate n1. Thus both nodes are isolated.

**System Reconfiguration** The system is reconfigured to isolate a crashed node or to add a node that has recovered. When a node is declared to have failed by its watching node, the watching node sends a control message to all participant nodes informing them of the fact. When the message is received, the reconfiguration routine of the recoverable node component is invoked, which performs the following functions:

- It updates the status list of the node to reflect the failure of the node.

- All Modules having an outstanding request to the failed Module are informed by raising the node-failure exception.

- Once the node is marked down in the status list, any request by a Module to access a service on the failed node will be rejected and an exception is raised to the requesting Module.

- If a message is received from a node that has recovered to add the node to the active nodes of the system, the reconfiguration routine will simply set the status entry associated with the node in the status list to UP so that normal communication may resume with the node.
**Restart of a Crashed Node** After a node has crashed, the recovery routine of the recoverable node component is invoked to restart the node. The recovery routine performs the following functions:

- Initializes the node.
- Restores the AL list of the node from stable storage.
- Initiates the recovery process of all Modules active before the failure.
- Requests the status list from the watching node. Once this is received, it restores the node's status list. If the watching node is down, then it request the information from its watching node.
- Changes the statue of the node to UP and sends a control message to all participant nodes informing them that the node is up.
- Continues normal operation, once recovery is completed.

**5.2. Recoverable Module**

As was specified in the model, a recoverable Module internally is constructed as a number of data objects which are manipulated or accessed through a set of Module operations. These objects are considered to be permanent objects. An object access may be requested by a number of concurrent server processes executing on behalf of remote requesters. Such processes are synchronized to guarantee consistency.
A recoverable Module is failure resistant, and it is able to recover from node failure, transaction abort (due to synchronization problems or user abort), or interaction failure. To achieve recoverability, both Module objects and server operation must be managed so that it is feasible and efficient to restore the Module state to a consistent state from which processing may continue in case of failure. Figure 14 displays a number of interacting components that are part of MI. The components are:

Server Operations Manager Component: This component is responsible for processing each server operation request as part of a distributed transaction. It supports the atomic property of each server operation by providing for processing and committing/aborting each server operation.

Object Manager Component: This component interacts with a server operation manager to coordinate the access of a Module’s objects. It is also responsible for maintaining a history of object states to increase the degree of concurrency, which will be explained later. The object manager component, in addition, provides for accessing, preparing, and committing objects accessed by a transaction in coordination with the server operation manager.

Operation Scheduler Component: This component uses semantic knowledge provided by a Module definer to synchronize requested server operations in such away as to preserve Module consistency constraints and increase concurrency among requested server operations. This component interacts with both the server operation manager and object manager to achieve its task.
Recovery Manager Component: In case of a Module failure due to node failure or interaction failure or in case of server operation abort, the recovery routine of this Module uses recovery semantic knowledge to initiate the recovery of both the Module’s operations and objects. Both the state of the Module’s operations and a Module objects will be restored to a consistent state from which normal processing may continue.

In the following sections the implementation of the object manager, server operation manager, and recovery manager are discussed; however, the implementation of the operation scheduler component is presented in Chapter 6. To recover from interaction failure the recoverable Module uses an exception handling mechanism that allows the specification of Module definer semantic knowledge that is used as part of recovery. The exception handling mechanism will be discussed in Chapter 7.

Our approach in using a recoverable Module to recover from failure is distinct in the following ways:

Unlike a simple transaction system where the entire distributed transaction or the local subtransaction is aborted in case of failure thus causing the loss of substantial computation, our approach is to use a recoverable Module to confine the effects of a local node failure or a Module interaction failure. In case of a node failure, the recoverable Module will restore the effected subtransactions to a state that existed before the failure from which execution may continue. In case of an interaction failure, the Module provides for specifying and choosing an alternative service if available to avoid aborting the entire transaction. As a result, a
subtransaction or the distributed transaction are only explicitly aborted due to a client request or due to synchronization problems that will be explained in the next chapter. Hence, substantial computation resources may be saved.

The object manager component and the server operation manager component are designed so that the goal of confining failure effects is easily and efficiently achieved. The object manager is designed to provide for a high degree of concurrency by using a multi-version approach to maintain the Module's objects.

The idea of a simple recovery line list and remote access list use as part of the server operation management. Unlike regular checkpointing, these lists are a form of checkpointing which are established infrequently, encompassed only on a server operation, and stored as minimal information on a stable store. Hence, the less stable store is used, the less overhead is incurred. The lists are used to confine effects of node failure.

5.3. OBJECT MANAGEMENT

The objects may be of different types and sizes and are Module dependent specified by the Module definer as part of the Module specification. For example, an object may be a stack structure, a bounded buffer, a directory or a database record.

The Object-version manager maintains the history of the states of the Module objects and identifies the distributed transactions that are
Figure 12: Module Interface Components
accessing or depending on a particular state of the object. The different object states result from applying different operations of the Module on the object as part of different distributed transactions. The scope of the history that is maintained by the object-version manager depends on the degree of concurrency that is allowed. Obviously, if only one state is maintained at any time, then the degree of concurrency among the transactions sharing the object is minimal (only one transaction at a time may access the object). This approach is used in the careful replacement technique. If multiple states of the object are maintained, then the degree of concurrency would increase among the transactions that are sharing the object. Maintaining the entire history might provide a very high degree of concurrency, but it requires a substantial amount of storage space, and hence the number of states that is remembered must be limited.

Our approach calls for maintaining a limited history of the states of the Module objects as multiple uncommitted versions (see Figure 15). The limited history includes the most recently committed state plus all succeeding changes which are not yet committed and which may be dependent on each other, as will be explained next. Each transaction would be accessing a different state, and because a transaction is not waiting for a previous transaction accessing a previous state to commit, the transaction is not delayed.

The multiple uncommitted version technique is an extension of the careful replacement technique, and also similar to Reed's multi-version technique. A new version of an object is created for each transaction accessing the object, and all operations executed as part of the transaction are performed on the new version. Each version contains
Figure 13: Object Multi Versions
information to identify the distributed transaction performing an operation on the specific version. The version remains tentative until it is finalized when the accessing transaction commits. If the transaction aborts, the tentative version is discarded.

A version $V_j$ accessed by a transaction $T_j$ may be created using the value of a previous version of the object $V_i$ which is accessed by a previous transaction $T_i$. In case $T_i$ has not yet committed, the current transaction $T_j$ is then made dependent on $T_i$, and $T_j$ cannot commit until $T_i$ commits, and $V_j$ cannot be committed until the $V_i$ is committed.

Besides maintaining the history of an object, the object-version manager provides for accessing the object version, prepares to commit the version, and commits the version.

The multiple uncommitted version organization supports a high degree of concurrency among distributed transactions in the system. By maintaining several uncommitted versions, we allow for a high degree of concurrency among conflicting server operations that are executed as subtransactions, which in turn will increase the degree of concurrency among conflicting distributed transactions. Such an organization lends itself naturally to the support of an optimistic concurrency control mechanism, which will be explained in Chapter 6 which covers synchronization and concurrency control. The scheme is also designed to support the use of semantic knowledge provided by the object definer that permits non-conflicting server operations to execute concurrently. The state history that is remembered is limited to the number of the uncommitted versions and thus this approach of the managing object's versions requires less storage than other approaches, such as Reed's [38] where conceptually the entire history must be remembered.
In the next subsection we present the structure of an object version, the primitive operation that the version manager provides, and the implementation of these operations.

5.3.1. Managing and Maintaining Multiple Versions

A transaction accessing an object of a given Module through a server operation will result in the creation of a new version of the object, and all accesses of the server operation are performed on this version (for reason of efficiency a version may only be created for transactions that update the object). Each object and its versions are represented by a set of identical data structures which are linked together to represent the total state of an existing uncommitted version structure explained next.

5.3.1.1. Single Uncommitted Version Structure

An entry of an object directory is associated with each object (see Figure 16). Each entry contains the object identifier, and a pointer to the descriptor of the first version of the object. Each descriptor entry consists of the following fields:

- **version identifier**: The version identifier consists of two parts, the object identifier and the transaction ID of the transaction accessing the version. Hence, $V_i = <O_j, T_r>$. 

- **Version status**: The version status indicates the current state of the version which may take one of the following values:

  1. **Accessed**: The version is being accessed by a server operation as part of the distributed transaction, and operation is still in progress.
Figure 14: Object Directory and Versions Descriptors
2. Ready: The version is already accessed by the transaction but not yet prepared to commit.

3. Prepared: The version is prepared as part of a two-phase commit protocol to commit the distributed transaction.

4. Committed: The version is already committed.

5. Discarded: The version is discarded as a result of aborting the transaction accessing the version. This may be due to a user abort or due to a synchronization problem.

NP: The NP is a pointer to the descriptor of next immediate version that is dependent on this version.

BP: The BP is a pointer to the descriptor of a sibling version which is a version that is not dependent on this version. The two server operations accessing the two versions are non-conflicting and can execute concurrently, and thus the two distributed transactions accessing the versions do not form a dependency relation.

CP: This is a pointer to the descriptor of a version on which the current version depends for commitment. The current version cannot be committed until the version it depends on commits.

SP: A pointer to the storage cell that contains the version value.
Figure 15: Structure of Uncommitted Versions
5.3.1.2. Multiple Uncommitted Version Structure

The multiple uncommitted versions are logically organized as multiple columns as shown in Figure 15. A column consists of a number of versions which are linked through the BP pointer. Except for the version in the head of the column, all other versions are created to be accessed by concurrently executing non-conflicting server operations, and are called non-dependent versions (the value of one is not used to create another). Discarding one version of a column does not affect other versions, and committing a version does not depend on the commitment of other versions in the same column.

The value of the head version is the result of merging the values of the versions in the column. The header version is first created when the column is created and is not assigned any value. When the status of all versions of the column become ready, then the head version is assigned the new value.

Each version in the column is created using a value of one or more versions in preceding columns, and thus this version is said to be dependent on those versions. If a version is dependent on more than one version, and if such versions span a number of preceding columns or are in the same column, the version is made dependent on the head version of the last column which has a version that the current version depends on. If a version is created using the value of only one version, then it is made dependent on that particular version only. For example, in Figure 18, version $V_3$ in column 3 is created using the values of both versions $V_1$ and $V_2$ of column 1. $V_3$ is then made dependent on the head version of column 1. If $V_4$ is created using the value of $V_5$, then it is made dependent on $V_5$ only. When version $V_i$ is made dependent
on version $V_j$, the CP pointer of version $V_i$ points to the descriptor of version $V_j$.

A version $V_i$ that is dependent on version $V_j$ cannot be committed (as a result of commit a request) unless version $V_j$ is committed, and if version $V_j$ is discarded then version $V_i$ must be discarded.

The number of versions in a column are limited to a predefined number which is chosen by the system designer or which can be adaptively changed depending on the status of the system to achieve better efficiency. A new column is created when each version of the previous column has changed its state to ready, prepared, or committed, and the creation of a new version has been requested by a conflicting server operation, or when the limit of the number of concurrent versions per column have been reached. A column is destroyed when all its versions have committed and been written on permanent store.

5.3.2. Primitives Supported by Version-Manager

A number of primitives are supported by the version manager to enable other components of the recoverable Module to interact with the version-manager and to create, maintain, and restore the Module object version. This primitive includes CREATE, LOOKUP, ACCESS, PREPARE, COMMIT, and DISCARD. We will discuss these operations in turn.

$\text{CREATE}(O_i,T_i)$: Given the transaction id $T_i$ and the object name $O_i$, the CREATE operation will determine the appropriate version reference number $V\#$ and then create a new version to be accessed by a particular server operation which is a subtransaction of a distributed
Figure 16: Version Dependencies
transaction $T_i$. If the number of versions in the last column does not exceed the predetermined number (N) and the requesting server-operator does not conflict with any In-progress currently running server operations, a version is created in the last (current) column. Otherwise, the process creating the version is blocked and made to wait in a simple FIFO queue until all the versions of the current column has either been committed, prepared, or made ready. At such a time a new column is created and a new version is created in the column for the transaction.

To create a new version in a column, it is first determined which previous version the current version is dependent on. This is accomplished by checking all previous columns starting with the last one and traversing the versions of the columns through the BP. If the version is not dependent on any other version, then the value of the head version of the first column is used to create the current version. If the version is dependent on only one previous version, then the value of the version is used to create the version. If there are more than one versions that the current version depends on, then the current version uses the value of the head version of the last column where a depended-on version exists, and the current version is made dependent on that head version.

The CP pointer is set to the version id of the previous version or head version which the current version depends on. The version state of the newly created version is set to I. The version id is set to $<O_i, T_i>$. The BP of the version that was created last in the column is set to the version id of the currently created version, and the storage pointer is set to the address of the physical location of the version.

**LOOKUP:** $(O_p, T_i)$ Given the transaction $T_i$ and the object name $O_p$, the
LOOKUP operation will determine the appropriate version reference number, and then look for a such version. If such a version exists, then the column address and version descriptor address are returned; otherwise, a null address will be returned indicating that the requested version does not exist.

ACCESS: \((O_i,T_j)\) This operation looks up the version using the LOOKUP operation. If such a version exists, it returns its address in memory or else it creates a new version using the CREATE operation.

PREPARE(V#): Given the version id, the prepare primitive will attempt to prepare the version to be committed which is the first phase of a two-phase commit protocol that will be explained later. This is performed as follows:

If the status of the version is already prepared, then the result of the primitive is set to indicate that the version is prepared. This occurs if the operation is restarted as part of the recovery that is explained in Section 5.3.

If the status of the current version is discarded, then a result is set to indicate the inability to prepare.

If the status is not prepared, then

1. the status of the previous version which this version depends on is checked.

   a. If the status is commit, then this version can be committed. The status of the version is set to prepared and the result of this primitive is set to indicate that the version is prepared.
b. If the status of the previous version is discarded, then this version status is set to discarded, and the status of all versions dependent on it are discarded. The result is set to indicate the inability to prepare.

c. If the status of the previous depended-on version is either dependent or prepared, then the processes requesting the prepare is made to wait until the status of the previous version is set to committed or discarded.

2. If this version is the last version in the column to prepare, then the status of the head version is also changed to Prepared.

A depended-on previous version cannot have an I status, otherwise, the current version would have not been created (See CREATE).

COMMIT(V#): This primitive is to change the status of a version to commit, after it has been prepared. The version can only be committed if its status is prepared; otherwise, an error is raised. To commit a version the following steps are taken:

The status of the version is changed to commit.

If the version is dependent on the head version of the first column (this is a special version and will be called a commit version), and it is not dependent on any other version, then
the value of the head version is written on the permanent object using the Stableput operation, and the value of this version is written on the head version. The first version which depends on this committing version is set to point to the head version of the first column.

It should be noted that the status of the head version of the first column should always be committed so that versions depending on it can be prepared and then committed. The head version of the first column represents the only committed version and has the last, committed value of the object, where the rest of the versions are either discarded, prepared, ready, or in progress. Since the commit version is the only committed version that is used to prepare and commit other versions, there are no other versions which are not dependent on it. Thus, the first column has only one version, namely, the commit version, and it is called the commit column.

DISCARD(V#): This is a primitive function to discard a particular version, which will cause all the versions dependent on this version also to be discarded. To discard a version the following steps are taken:

The status of the version is changed to DISCARD.

The status of all versions depending on this version is set to DISCARDED. This is done by following the NP of each version to the next version.
5.3.3. Performance Consideration

Unlike Reed's approach which does not provide for object restoration, all the versions (object history) has to be saved in a stable store in order to survive a node crash, and hence no version restoration is needed. Our approach is to save all in progress and ready versions in volatile memory; however, committed, prepared versions still have to be stored in a stable store. In case of a node crash, all in progress and the ready version will be lost, but the committed and prepared versions will survive and need not be restored. A method for the restoring versions in case of a node failure will be presented in this chapter in Section 5.5.2.1. We think our method will save substantially on the amount of a stable store that is used; however, it will require some extra processing time. There is a trade-off between the amount of a stable store used and the processing time lost in case of a failure.

5.4. SERVER OPERATION MANAGEMENT

For a Module to be recoverable, both Module objects and operations must be managed in such a way so that it is feasible and efficient to restore a Module state to a consistent state in case of a failure. In the previous section, Module object management was presented. In this section we will present a procedure for operation management that facilitates the restoration of a recoverable Module state in case of a node failure and that preserves the atomic property of Module's operations.

A number of approaches to recovery from a node failure have been discussed in Chapter 4, which include using

The entire distributed transaction as a unit of recovery.
Individual subtransaction and check-pointing.

A number of problems associated with such approaches were pointed out in Chapter 4. These problems include: loss of healthy subtransaction, creation of orphans, and inability to confine the effects of a node failure, thus introducing inefficiency.

To eliminate these problems, a different approach for recovery from node failure is adapted. The approach is based on using the concepts of a simple recovery line and a remote access list.

A simple recovery line involves saving the request message from the requester. A remote access list is a list of all remote requests issued as a result of the initial request and possible responses. Both lists are saved on a stable store. In case of a failure the recovery management routine of the recoverable Module is responsible for aborting all in progress operations, restoring the Module objects, and using a simple recovery line and a remote request list to restore the Module's operation to the state existing before the crash so that interrupted processing can continue. In this section the idea behind a simple recovery and a remote access list are discussed. Both are then used to process requested operations.

5.4.1. Simple Recovery Line and Remote Access List

A recovery line is a form of a checkpoint where the status of the transactions and the Module's objects accessed by the transactions are saved on a permanent store. A recovery line then can be used to reinstate a previous consistent state from which processing can continue.
A simple recovery line involves saving only the request to the Module from a requesting Module, some status information which indicates the status of the request, and a possible reply. As part of recovery, a simple recovery line is used to restart the execution of the Module’s operations and continue processing, in case of a failure. The problem of the loss of the work of a healthy child and the creation of orphans would still exist if a simple recovery line were to be used. The reason for this is that when the execution of a Module operations is restarted, it creates duplicate requests a to remote Module operation, which causes the creation of orphans. To solve this problem a remote access list (RAL) is used. A record of each remote request performed by the Module operation is kept on the RAL, in addition to some status information of the request. The RAL can then be used to detect if a duplicate request is initiated by a restarting operation. Hence, orphans will be avoided and work of healthy children can be saved.

5.4.1.1. SRLL and RAL Data Structures

Figure 19 shows the structure of both a simple recovery line record and an RAL record. The simple recovery line record includes the following fields:

**Original Request:** This field includes the original request that is received from a remote requesting Module operation which includes the requested operation and its parameters.

**Transaction ID:** The ID of the distributed transaction part of which is the requested operation execution or a subtransaction.

**TAR:** Time of arrival exists so that requests can be ordered by time of arrival.
<table>
<thead>
<tr>
<th>original request</th>
<th>status</th>
<th>T_id</th>
<th>TAR</th>
<th>reply</th>
</tr>
</thead>
</table>

**Simple recovery line record**

<table>
<thead>
<tr>
<th>request</th>
<th>request #</th>
<th>status</th>
<th>reply</th>
</tr>
</thead>
</table>

**Remote access list record**

Figure 17: Simple Recovery Line and Remote Access List Records
**Status:** The requested operation may be in a different status during the processing of the request. These statuses include:

1. **IR:** This status is in progress, or ready, which means the request has been submitted and has not been completed, or the request has been executed and a reply message has been sent to the requester but has not been prepared for commitment as part of a two phase commit.

2. **P:** The subtransaction (execution of the requested operation) has been prepared but not yet committed, and a has-prepared message was sent to the requester.

3. **IP:** The subtransaction is in the process of preparing. A request has been received to prepare but still is in the process of preparing.

4. **IC:** The subtransaction is in the process of committing and has received a request to commit but still is in the process of committing.

5. **C:** The subtransaction has committed and a message has been sent back to the requester.

6. **A:** The subtransaction has been aborted.

**Reply:** If a reply to the request has been computed, then it is saved in this field.

The RAL record has the following fields:
Request: The name of the requested remote access operation.

Transaction ID: The same as in a simple recovery line record.

Request #: a unique ID that identifies a particular remote request which is a part of a particular transaction. It is used to identify a duplicate request.

Status: The remote request can have the same status as a record in a simple recovery line list, I, R, IR, P, IC, and C, except they address the status of a remote requested operation.

Reply: This is the same as in the SRLL record.

Both lists must reside on a stable store to survive a node or media crash. In Section 5.5.2.2 we will demonstrate the use of both lists to recover from node failure.

A number of operations are provided to manipulate both the RAL and the SRLL. These operations are as follows:

1. SAVE.REQ(request, param., status, Tid, RAL/SRLL): This operation is an atomic operation used to save a record on either the SRLL or RAL lists. It uses the operation Stableput that is provided by the stable store abstract to atomically write the request on the requested list.

2. DELETE-REQ(request,Tid,RAL/SRLL): This operation deletes an entry from the RAL or SRLL list.

3. UPDATE-STATE(request,Tid,status,RAL/SRLL): This
operation is used to change the status of a specific entry in the list.

4. READ-REQ(request,Tid): This operation uses the Stableget operation provided by the stable store abstract to retrieve a request.

5.4.2. Request Processing in Error Free Environment

A request is processed in such a way so that it is possible to recover from a node failure and preserve the consistency constraints of the recoverable Module. The synchronization of multiple concurrent requests will be addressed in Chapter 6. Here we assume that a selector schedule is provided that will synchronize concurrent requests in accordance with the consistency constraints of the Module. Figure 12 shows the Module interface components of the recoverable Module which will interact in processing the requested operations. The procedure for processing a request goes through the following steps:

1. When a request is received by the server operation manager of the MI, a simple recovery line is established by saving the request on a simple recovery line list. This is done using the SAVE.REQ operation.

2. The authenticity of the request is checked, and if the request is legal, then it is passed to the selector scheduler. Otherwise, it is deleted from the SRLL and a reject message is sent to the requester.

3. The selector scheduler will determine if the requested operation
can be executed in accordance with the Module consistency constraints and with its synchronization requirements. If and when the request satisfies such constraints, it is passed to the Module object version manager.

4. The object version manager will create a new version of the object for the request, if one does exists, and the requested operation is performed on such a version. The state of the version is set to I (in progress).

5. If, during the execution of the requested operation, the operation requests yet another remote operation that is provided by a remote Module, the remote access request is recorded in the RAL using SAVE.REQ and the state of the request is set to I.

6. This action is done for all remote access requests when a reply is received for a particular remote request. The result is recorded on the RAL entry and the status is changed to R. The server process executing the operation is reactivated, and the operation execution continues.

7. Once the executing of the operation is normally terminated, the selector scheduler is informed that the operation request is ready to exist, and the scheduler will update its status accordingly. The status of the version accessed by the transaction is set to R if the operation is the last one requested by the distributed transaction. (It is possible that more than one Module's operation are requested as part of the transaction. In that case the version should not become dependent until the last operation is executed.)
8. The status of the SRLL is updated to R and a reply is sent back to the requester.

It should be noted that the SRLL entry, all the RAL entries, and the version created for a requested operation executed by a subtransaction are not destroyed until the transaction is committed.

5.4.3. Performance Consideration

Unlike a regular checkpointing approach, the SRLL and RAL store minimal information on a stable store, are established infrequently, and encompass only a particular server operation, hence reducing the amount of the stable store used and the overhead associated with processing a request. However, extra processing still has to be done during recovery since every request that is in progress, or completed but not prepared, still has to be restarted, as explained next. Hence, there is a trade-off between the amount a stable store used and the associated overhead incurred during normal processing, and the amount of processing that has to be redone during recovery. To fulfill our design goal of incurring minimum overhead during normal operation and incurring reasonable overhead as part of recovery, we chose to reduce the overhead during normal processing and to use the approach of the SRLL and RAL.

In addition, unlike regular checkpointing approaches, the use of the SRLL confines the effects of a node failure and thus saves the healthy work performed by a requested subtransaction and eliminates duplicate requests that can cause orphans.
5.5. RECOVERY MANAGER

Part of the function of the recovery manager is to restore the states of both the Module's server operations and objects to a consistent state in case of a node failure or server operation abort. In case of a node failure, the state of both operation and object is restored to a state from which processing can continue after the failure. In case of a server operation abort, only the state of the object is restored to the state which existed before the execution of the server operation so that the effect of the server operation execution on the Module object is nullified.

5.5.1. Transaction Abort

The server operations of the recoverable Module are executed as atomic subtransactions. The server operation either normally completes and commits as part of the entire distributed transaction or it is aborted. The server operation may be aborted due to a client request or due to synchronization problems as will be explained in Chapter 6. In case a server operation is aborted, then all the remote requests it made must be aborted and all the Module's objects it accessed must be restored.

To abort a server operation, the server operations manager performs the following functions:

1. The status of the SRLL entry of the aborted server operation is set to A (abort).

2. The status of all the RAL entries associated with the server operation is changed to A.

3. Using the RAL a message to abort is sent to all remote
requested operations. The message is repeatedly sent until an ack. is received.

4. The object manager is then requested to discard the version used by the aborted operation. This is done using the DISCARD operation provided by the object manager.

5.5.2. Recovery From Node Failure

In case of a node failure, the state of the current requested operations of the Module (that are either In-progress or Ready) of the Module, and the Module object version accessed by such operations, must be correctly restored to a consistent state that existed before the node failure. The restored state of the Module is then used to continue execution of the Module operations. In this section a procedure for recovering the Module operations and object versions are presented.

A Module can be in one of three states, normal, crashed, or recovering. When a Module is created, it is in a normal state but will change to crashed if a node failure occurs. While the Module is recovering, the status is changed to recovering. We assume that in most cases a Module will remain in a crash state only temporarily and is recovered almost immediately.

5.5.2.1. Object Restoration

A node failure causes data and status stored on the volatile memory to be lost. Thus, all Module object versions would be destroyed, except for the versions that have a committed or prepared status. All the selector scheduler statuses that relate to requests in progress or ready request will also be lost.
In case of a node failure, only Ready and In-progress versions need to be restored. To accomplish this, all requested server operations that are in progress or that have been completed but not yet prepared will be restarted as explained next in Module operations status restoration. If such a process is performed correctly, it will result in recreating all the object versions that existed before the failure and thus restore the object versions.

5.5.2.2. Module Operations Status Restoration

The process of processing, preparing, and committing each requested Module's operation is presented as a state transition diagram shown in Figure 20. Each state in the diagram represents the compound state of the object version and the state of the requested operation in the SRLL entry. A state transition occurs when one of the appropriate operations specified is executed. To restore the Module status, the recovery component of the Module interface will process all entries of the SRLL in the order of the time of arrival. In processing each entry the status of the request is checked and the following is performed:

1. If the status is I or R (states 0, 1, and 2) then the requested operation is restarted (back to state 0) as follows:

   New server processes are created to run all such requests. The requests are submitted to the selector scheduler to be processed in the order of the time of arrival.

   Staring from state 0 in the state diagram, the processing of each request will continue as show in the state diagram and as was described in Section 5.4.2; however,
Figure 18: Module's Operation and Object State Transition Diagram
no new requests are processed until all the old requests are submitted to the selector scheduler.

When a recovering operation requests access to a remote operation, the RAL is checked, and

a. if the remote access request is in the RAL, then the status of the request is checked.

   i. if the status is IR, then the server process is made to wait for the response.

   ii. if the status is R (a reply is received for this request) or IP, then the reply is used and the execution of the operation continues.

If the request has not been recorded in the RAL, then it is recorded, and the server process is made to wait for a reply.

If the status is IP (states 3 and 4), then the status of the object accessed by the transaction is checked. If the status is P (state 4), that means the object and the remote accesses (subtransactions) have been prepared. The status of the SRLL entry is updated to P and a message is sent to the requester. Otherwise (state 3), the operation is restarted (back to state 0) as specified in step 1.

If the status of an entry in the SRLL is IC (states 6 and 7), then the subtransaction executing the operation
is committed as shown in Figure 18 and as was explained in Section 5.4.2.

If the status is C or P (states 8, 5), then nothing is done (back to the same state).

5.5.3. Correctness, Robustness, Performance Considerations

There are three points to consider in relation to the recovery algorithm presented. First, is the algorithm correct? That is, does it correctly restore the actual previous state? Second, how efficient is it? Third, is the algorithm robust and can it perform correctly in case of failure during recovery? These three points are discussed next.

Correctness: A requested server operation of a Module may be in one of the following states: In-progress, Ready, In-progress-of-preparing, Prepared, In-progress-of-committing, and Committed. Such a state is recorded on the simple recovery line entry of each request. Accordingly, the object version accessed by the operation is in one of the following states: In progress, Ready, Prepared, Committed, or Discarded. In case of a Module failure, both the operation and the object version states must be restored to the state which existed before failure. To demonstrate the correctness of the recovery, we show that the server operation and objects would be restored to the appropriate states after a failure. Figure 18 shows the possible states that the server operation (subtransaction) and the object accessed by the operation can assume after a node failure. This is displayed in the figure by dotted lines. These states are classified according to the way they are handled as follows:
Class 1: States 0, 1, and 2. As a result of a failure, the object versions will be destroyed, and since all in progress or ready requested operations are restarted in order of arrival (step 1 of the recovery algorithm), which is the same order they were processed in before failure, and no new requests are processed while recovering, the environment that existed before failure should then be recreated. The object versions which existed before failure will be restored for all in progress or ready requests.

Class 2: State 3. While a request to prepare the requested operation has been received and the simple recovery line entry state has been updated, but before the object is prepared, a failure occurs. In this case the object version is lost and has to be recreated. Since in step 2 of the recovery algorithm a request that has state IP is restarted in chronological order with all other ready and in progress requests, the object version for this request existed before failure should be restored. Once the request has finished, a prepared request will immediately follow to prepare the object version; hence, the status of the requested operation is restored to the one which existed previously to the failure.

Class 3: State 4. In this case the failure occurred after the version is prepared but before the SRLL is updated. Neither the object version nor the simple recovery line associated with the request are destroyed. Thus, the only step required is to finish preparing the request by updating the simple recovery line status to P and sending a hand-prepared message. This is done in step 2.
Class 4: States 5, 6, 7, and 8. In this case none of the versions or status information is lost. The only step required is in case of states 6 and 7 where a process is created to continue committing the request as provided by step 3 of the algorithm.

From the previous discussion it is clear that the status of the object versions and requested operations would be restored after a failure in all possible cases. Hence, the recovery algorithm is correct.

**Efficiency:** The main characteristics of the recovery algorithm are:

1. It uses checkpointing to limit the effect of failure. A failure of a Module (due to node failure) does not abort the entire distribution transaction.

2. It eliminates orphans, and thus the work of the healthy child is saved.

3. The amount of information saved in association with the checkpoint is small and the frequency of establishing checkpoints is low. Thus, a small overhead is incurred during normal operations.

4. Only the computation performed between the simple recovery line and establishing the last remote access entry has to be redone in case of failure.

An alternative to redoing the work is to save all Module
object versions on a stable store and use checkpoints periodically. In such a scheme, some of the work still has to be redone, mainly the computation since the last checkpoint. However, a more serious disadvantage is the fact that object versions have to be kept in stable storage and all update to such versions have to be performed on the stable storage. In addition, all system states that relate to a particular request must be saved in a stable store every time a checkpoint is established.

Two important characteristics of checkpointing that may affect efficiency are 1) the frequency of establishing a checkpoint and 2) the amount of information that is saved in association with a checkpoint. A checkpoint may be established periodically or can be event driven. It also may encompass a particular Module server operation, all Module server operations, or the total system (all Module). The type of information saved may be object status only, operation (system) status only, or both. Depending on the amount of information saved and the frequency of establishing a checkpoint, the system efficiency may suffer. Inevitably, establishing a checkpoint for the total system and saving both object and operation status that is performed very frequently are inhibiting, impractical, and inefficient.

Our approach is to use the Simple Recovery Line (SRL), and the Remote Access List (RAL) as a checkpointing method that is event driven, encompasses only server operations, and is established only twice. This checkpointing method avoids the overhead and inefficiency mentioned above.
Robustness: The question of robustness addresses whether the recovery algorithm still performs correctly in case of a node failure during recovery. In other words, it does not change the status of object and Module operation to a state from which recovery cannot be performed again. The recovery algorithm has the property of idempotency, which means that the multiple execution of the recovery algorithm will not change the final result. The state transition diagram of Figure 18 shows that in case of a failure a previous or the same state will be assumed and the recovery algorithm will take the operation and the version states from a restarting state assumed after a failure to the target state which existed before the failure. If, however, a failure occurs during recovery, the a state between the restarting state and the target state is assumed. The reexecution of the recovery algorithm will take the operation and the version states from the new restarting state to the old target state; hence, the recovery algorithm is robust in case of a node failure.

5.6. SUMMARY

In this chapter we have presented three mechanisms for managing multiple requested server operations and managing the associated object version accessed by such operations in a Module model and a recovery mechanism to recover from a node failure. Such mechanisms are designed to facilitate an efficient Module recovery in case of a node failure and provide for the atomic property of the recoverable Module's operations.
A recovery mechanism was introduced which efficiently recovers from a Module failure. The recovery mechanism was based on the use of a simple recovery line and a remote access list which are a form of checkpointing. It was shown that the recovery mechanism will correctly and efficiently restore the Module object and operation in case of a failure. The mechanism itself is idempotent and can perform correctly in the face of repeated failure during recovery. The object management, operation processing management, and recovery manager form a part of the Module interface component. The excessive use of a stable store is avoided to enhance efficiency.
Chapter 6

MODULE'S OPERATION SYNCHRONIZATION

A synchronization mechanism is used to prevent inconsistency, both for individual Modules and across a distributed transaction. In the latter case, the synchronization mechanism guarantees that the state produced by concurrent distributed transactions is semantically equivalent to the state produced if the transactions are executed serially. This property is called serializability and has been fundamental to most of the work in concurrency control [15, 12, 6, 36].

It was concluded in Chapter 4 that a synchronization mechanism suitable for this environment must provide for the following:

- the specification and preservation of individual Module constraints,
- the preservation of consistency constraints among distributed transactions, while allowing for a high degree of concurrency,
- support for a fairly arbitrary, user defined object structure and access operations, and
- support for the use of Module dependent semantic knowledge to achieve a high degree of concurrency.

In Chapter 4 we examined the applicability of a number of existing
mechanisms to solve the synchronization problems in this environment. It was concluded that existing synchronization and concurrency control mechanisms in their present form cannot be used directly to solve the synchronization problems in the distributed environment of interest.

A new synchronization mechanism for the environment of interest is proposed. The mechanism uses semantic knowledge provided by the Module definer, and a novel optimistic concurrency control mechanism to maintain consistency while achieving a high degree of concurrency. The mechanism is a 'hybrid', one based on combining a traditional synchronization mechanism, namely, selectors, to synchronize Module operations, and an optimistic concurrency control that uses the multi-version technique presented in Chapter 5 to synchronize concurrent transactions.

In this chapter we present selectors as a traditional synchronization mechanism that can be used by a recoverable Module interface to schedule concurrent Module operations. Selectors then are evaluated and extensions are proposed to achieve the required functionality desired in the environment of interest. Such extensions are presented in the form of Reliable selectors.

In Chapter 7 we present the optimistic concurrency control algorithm to synchronize distributed transactions, provide an algorithm to detect a commit deadlock that may occur during synchronization, provide a proof of correctness of the deadlock algorithm, and proof the serializability of the optimistic concurrency control mechanism.
6.1. SYNCHRONIZATION OF MODULE'S SERVER OPERATIONS

A synchronization mechanism for the Module's server operations provides for preserving the consistency the constraints of the recoverable Module and allows for the specification of such constraints. A simple example that illustrates consistency constraints of a Module is one of a Module implementing a bounded buffer, as in the producer/consumer problem. The Module provides two operations on the buffer object: insert an item into the buffer and remove an item from the buffer. A Module consistency constraint, or invariant, is that a remove operation cannot be performed if the buffer is empty. Thus, a transaction which issues a remove request has to be delayed until an insert is performed.

The mechanism, in addition, allows the Module definer to provide semantic synchronization knowledge that may be used to increase the degree of the concurrency of the execution of the Module operations, and ultimately increases the degree of the concurrency of the distributed transactions, thus enhancing performance. A number of researchers have observed that semantic knowledge can be used in increasing the degree of the concurrency in transaction processing [43, 2, 14, 12].

As was explained in Chapter 4 the semantic knowledge may be used to

specify interleaving of compatible operations,

specify specification versus implementation behavior of a Module's objects, and.

avoid strong serializability.
The traditional synchronization mechanism of selectors was chosen to synchronize a Module's operations and provide for the use of semantic knowledge. Selectors are chosen for extension over other traditional mechanisms because Selectors already provide for some of the added functionality that is needed. They provide a high level specification language, for example, which allows a Module definer to specify compatibility relations among the operations of the Module and the Module's consistency constraints. The compatibility relation specifications provide information about which operations can be executed concurrently (e.g., reads).

In Section 6.2, we discuss the properties of Selectors, and the way that Selectors can be used to synchronize multiple requests to the Module object. In Section 6.3, we propose some extensions to Selectors to the form of Reliable Selectors to enable them, among other things, to be fault tolerant.

6.2. Selectors

Selectors are introduced by Leinbaugh [25] and are used to synchronize the multiple concurrent requests of a particular resource. In this section a summary of the properties of Selectors is presented; for more details the reader is referred to [25]. A selector consists of the following four components to synchronize the Module server operations:

**The Resource Invariant Component:** The resource invariant defines what the resource is capable of. It is the condition that must remain true in order for the resource to continue to operate properly. This involves concurrency constraints and Module constraints. The concurrency
constraints specify the requests that can be serviced concurrently by the Module. The Module constraints are concerned with the state of a Module’s objects, what values it can assume, and what operations can be served when the Module’s object assumes a particular value.

The Selection Policy: Of those requests which can be serviced next, it may be desirable to select some requests over others to achieve efficiency either in the Module operation or efficiency in the response time of a particular type. The selection policy identifies the request to choose a particular request over others.

The Postponement Policy: A specific resource invariant and/or a specific selection policy may cause some requests to receive poor service. To solve this problem a modification to a simple selection policy is performed by identifying the requests that are responsible for others receiving poor service and temporarily ignoring them. The postpone policy identifies such requests upon their arrival and postpones them.

The expedite policy: Another modification to the simple selection policy to solve the poor service problem is to identify those requests that may starve and make sure they are serviced soon. The expedite policy is used to select requests to be serviced next to avoid starvation.
6.2.1. Selector Variables

Three variables are useful in constructing the resource invariant, and the postpone, the expedite, and selection policies.

**Set Counting Variables:** ACTIVE.requesttype, EXPEDITE.requesttype, WAITING.requesttype, and POSTPONED.requesttype, keep track of the numbers of requests of type requesttype, that are Active, Expedited, Waiting, and Postponed, respectively.

**History Variables:** These variables remember the history of the field values of requests sent to the resource. In particular, LAST-ACTIVE.requesttype.fieldname is the value of the field fieldname of the last request of type requesttype sent for service. LAST-ACTIVE.fieldname is the value of fieldname of the last request of any type sent for service. Similarly, NEXT-TO-LAST-ACTIVE retains the field value of the request sent for service before the last one sent for service.

**Resource State Variables:** Resource state variables are to reflect what the selector knows of the actual state of the resource. The state of the resource changes as a consequence of the operations performed upon it. Changes are made to state variables upon service of a request. However, the exact time of the charges to the resource itself is not known because there are delays in sending requests to the resource and receiving replies, because a request may be undone, and because within the performance of the operation itself there
will usually be a transitional period during which the resource state changes from the old value to new value. The value of each state variable is therefore kept as an uncertainty range of its possible values. This uncertainty reflects exactly the information the selector can know about the resource and greatly simplifies writing selectors that allow the concurrency of operations.

A condition in the resource invariant involving a state variable evaluates true only if it is true for every value in the state variable's uncertainty range. This is necessary because the invariant must always be true and the resource might be in any of the states reflected by the uncertainty range.

6.2.2. Preservation of Resource Invariant Upon Request Completions

An important characteristic of selectors as well as other schedulers is that the selector controls when a request begins service on a resource but does not control when a request completes service. This means that care must be taken so that the resource invariant will remain true regardless of request completions. To achieve this property, the resource invariant is restricted to the forms shown in Figure 21.

If a concurrency constraint is true while some requests are in service, then the constraint remains true when a request completes because the active counts will still be less than the bounding constant. A resource-limits expression can involve resource state variables and history variables. History variables only change values when a request begins service and consequently request completions leave them unchanged. Upon request completion, the uncertainty range of a resource state
Resource invariant
concurrency constraints and resource limits connected with AND and OR

concurrency control
active request name constant

resource limit
condition involving resource state variables and variables can use +, - and comparisons

Figure 19: Resource Invariant Form
variable decreases to a subset of its previous values. If a resource-limits expression evaluates true for a certain set SV of values within the uncertainty range of a resource state variable when a request begins service, the resource-limit will remain true for values that are in both the reduced uncertainty range and SV.

Only AND and OR operators are used to construct the resource invariant from individual terms. This structure results in the desirable behavior in that the resource invariant cannot be changed from true to false by making additional terms true. Consequently, the completion of a request cannot make the resource invariant false because the only effects of completion are to (possibly) change some concurrency constraints from false to true and (possibly) reduce uncertainty ranges resulting in evaluating resource-limits expressions over fewer values.

6.2.3. Selector Specification

The selector components are declared using a high level specification language. An illustrative syntax of Selectors is presented in Figure 22. The following are the four sections to selector specification:

Request Declaration Section: In this section the request types that a selector to receive are declared and the data structure needed to manipulate such requests are also declared.

State Variable Section: In this section all the state variables used in the selector are declared and initialized.

Processing Section: This section defines (with (PROCESSED BY) what resource operation each type of
REQUEST DECLARATION

REQUEST FIELDS
- type CHARACTER(1)
- item CHARACTER(99)

REQUEST TYPES
- insertitem HAS type = 'I'
- removeitem HAS type = 'R'

DECLARE STATE VARIABLES
- *items INITIALLY 0

PROCESSING
- insertitem PROCESSED BY insertroutine
  UPON SERVICE *items := *items + 1
- removeitem PROCESSED BY removeroutine
  UPON SERVICE *items := *items - 1

RESOURCE INVARIANT
- ACTIVE.insertitem <= 1 AND ACTIVE.removeitem <= 1
- 0 <= *items AND *items <= 10

Figure 20: Syntax Representation of a Selector of a Bounded Buffer
request performs and also defines what changes that operation causes the resource state variables (with UPON SERVICE).

Resource Invariant Section: In this section the resource invariant (resource constraints and concurrency constraints) are specified in terms of Selector variables.

6.2.4. Example Using Resource Invariance

The use of the resource invariant and the various kinds of variables are illustrated with the following example. A solution (Figure 20) to the Producer/Consumer problem using a 10 slot bounded buffer illustrates the use of the resource invariant and resource state variables.

We assume operations are written to allow an insertitem request and removeitem request to performed concurrently. Consequently, the resource invariant specifies that at most one insertitem and one removeitem request can be active in the resource at once. The state variable #items is used to keep track of the number of items in the resource buffer. Insertroutine adds another item to the buffer; therefore, the selector specifies that upon service #items is increased by one. Since the resource uses a 10 slot buffer, the resource limit 0 ≤ #items ≤ 10 is necessary to prevent over-filling or over-emptying the buffer.

This looks (and is) straightforward; however, this is only because uncertainty-range state variables are used. Solutions with other techniques, if they allow concurrency, require two variables, one to reflect the number of positions known to be to empty and the second to reflect
the number of positions known to be full. Extra care is then needed to test and update the proper variable at the proper times. This is unnecessary with selectors because the uncertainty-range variable \#items reflects exactly what is known of the resource state. Any request that could take \#item's uncertainty outside the range \([0,10]\) is forced to wait. For example, if no requests are active and there are 9 items in the buffer, then the uncertainty range is \([9,9]\) -- the possible values for \#items is only 9. A removeitem request can start, making the range \([8,9]\) and an insertitem request can start, making the range \([8,10]\). This means there are anywhere from 8 to 10 items now in the buffer. If the insertitem is finished first, the range is reduced to \([9,10]\). Another insertitem is not allowed to start because there may still be 10 items in the buffer. The range becomes \([9,9]\) when the remove completes making it permissible for another insert request then to start.

6.2.5. Deadlock and Starvation Properties of Selectors

The most basic properties of schedulers are determining whether use of a given scheduler results in the potential for deadlock or starvation of requests. In this section a number of definitions and theorems are presented to establish these properties in Selectors. The proof of the theorems are not presented; the interested reader is referenced to [25].

Definition: Request Deadlock. This is the situation where the selector/resource has reached such a state that regardless of future requests, regardless of what requests are correctly started, and regardless of the order in which requests complete execution, one or more requests which have entered the scheduler will never be executed. Such requests are said to be deadlocked. A scheduler has deadlock potential if some
possible sequences of events can result in one or more requests being deadlocked.

Definition: **Request Starvation.** A request is being starved in a deadlock potential free selector if it resides in the scheduler and under the current and possible future behavior of the scheduler the request will not be serviced. A scheduler has starvation potential if it is possible for requests to starve.

Two types of starvation are differentiated.

Definition: **Resource-limits Starvation.** A request starves even if the resource is idle because the resource invariant prevents it from being serviced.

Definition: **Request-abundance Starvation.** Request abundance starvation is cause by the continual arrival of (too many) other requests.

Arriving requests can prevent a request from receiving service in any of several ways. The concurrency constraints (because other requests are being serviced) may prevent a waiting request from being serviced. The selection policy may always select other requests. other requests may be continually expedited, or the request is postponed but its postpone condition does not evaluate false and new requests preventing others from being serviced.

Definition: **Resource-limits Blockage.** A request t has resource-limits blockage potential if there exists one idle resource state for which t cannot be serviced when it is the only request in the selector.
Definition: **Re-enterable Idle Resource State.** A resource has re-enterable idle resource states if, for every legal idle resource state reachable from the initial idle state, there is a sequence of requests which when executed take the resource back to its initial idle state.

**Theorem 1:** Resource-limits blockage is a necessary condition for deadlock and resource-limits starvation potential.

Theorem 1 establishes the root cause of the deadlock and resource-limits starvation potential.

**Theorem 2:** A selector with re-enterable idle resource states does not have deadlock potential if it does not include expedite or postponement policies.

Theorem 2 establishes that for most resources (those with only re-enterable states) only selectors with postpone or expedite polices need be examined for deadlock potential.

**Theorem 3:** A selector that is deadlock free and includes only the resource invariant (no selection, postpone, or expedite policies) remains deadlock free regardless of the selection policy introduce.

Theorem 3 demonstrates that a selection policy by itself cannot introduce deadlock.
6.3. Evaluation of Selectors

Are selectors in their present form appropriate to synchronize concurrent server operations of the recoverable Module in the environment of interest? We attempt to answer this question next, by examining two points. First, we determine whether selectors provide for the synchronization requirements specified in Section 6.1 which include:

- specification and preservation of a Module's consistency constraints.
- provide for using semantic knowledge to increase concurrency that includes the specification of a Module's operation compatibility, avoiding strong serializability, and taking advantage of inherited concurrency provided by a specific implementation.

Second, we examine how selectors perform in a failure prone environment like ours.

To evaluate whether Selectors provide for specifying consistency constraints and semantic knowledge, the following is pointed out.

1. Using selectors the Module's consistency constraints can be expressed in terms of resource limits as part of the resource invariant section. For example, if the consistency constraints of the bounded buffer are to prevent a removeitem operation to access the buffer when the buffer is empty, this is expressed by setting the uncertainty range of \( \#\text{items} \) to \([0,10]\). If a removeitem operation would decrement the low bound of the uncertainty range to 0, which is the case if the buffer is empty, then the removeitem operation is delayed.
2. The selectors also provide for specifying the compatibility relation of the Module's operations. This is done in terms of specifying concurrency constraints in the resource invariant section. In the bounded buffer example, it is specified that

\[ \text{ACTIVE.insert} \leq 1 \text{ AND } \text{ACTIVE.removeitem} \leq 1 \]

This indicates that the number of \text{ACTIVE.insertitem} operations cannot exceed one; thus, there is no compatibility among \text{insertitem} operations and similarly for \text{removeitem} operations. The "AND" indicates that both \text{insertitem} and \text{removeitem} operations are compatible. To make both operations incompatible, the concurrency constraint specification may then be:

\[ \text{ACTIVE.insertitem + ACTIVE.removeitem} < 1 \]

which indicates that the total number of active operations allowed to the resource is one, and thus a \text{removeitem} operation is incompatible with another \text{removeitem} operation and an \text{insertitem} operation is incompatible with another \text{insertitem} operation and both operations are incompatible with each other.

3. A selector does not provide the capability to schedule requests which access different and independent units of the object or resource. In other words, it does not provide for the added concurrency that may be supported by a particular implementation of the object or resource.

4. By allowing for the specification of the compatibility relations
among the Module's operations desired by the Module's definer, the selector provides for specifying the acceptable concurrent behavior of the operations that are part of different distributed transactions which when executed produce a desirable object state. Such a state may not be produced by a serial execution of the distributed transactions, and thus selectors can be used to avoid strong serializability. For instance, by specifying that the removeitem operation and the insertitem operation are compatible, this indicates that the execution of one does not affect the execution of the other and they can be exchanged in the history of the execution of the distributed transactions that they are part of. Consider Figure 23a where there are two bounded buffers maintained by two different Modules. Buffer 1 (B1) has one item (x) and buffer 2 (B2) has one item (y). We assume there are two transactions, T_a and T_b. T_a removes an item from B1 and inserts it in B2, and T_b removes an item from B2 and inserts it in B1, assuming that the last item inserted is the first to be removed. If the two transactions are executed serially T_aT_b, this gives the schedule R_1^aI_2^aR_2^bI_1^b producing the state of buffers shown in Figure 23b. However, because I_2^a and R_2^b are compatible they can be exchanged giving the schedule R_1^aR_2^bI_2^aI_1^b and producing the final state shown in Figure 23c. Such a schedule cannot be produced by a serial execution of T_a and T_b, and such a final state cannot be produced by a serial schedule. Hence, by specifying the compatibility relation, strong serializability can be avoided.

In evaluating how selectors perform in the failure prone environment such as the environment of interest, we point out the following:
Figure 21: Concurrent Access of Bounded Buffer
1. Since in the case of a node failure all the data in volatile memory is lost, as a result of a node failure all selector variables are lost and upon recovery they are reinitialized to the initial state. Thus, the selector state variables do not reflect the actual state of the object and cannot correctly synchronize requests to the object. Going back to the bounded buffer example, we can assume that there are eight items in the buffer; hence, the uncertainty range of the number of items (#items) is [8,8]. If at this time a node failure occurs, then upon recovery the uncertainty range is initialized to [0,0] reflecting an empty buffer where in fact there are eight items in the buffer.

2. When a request is aborted, the effect of the request on the state of the object must be nullified, and accordingly the resource state variables of the selector must reflect the new state of the object. However, Selectors in their present form would update the resource state variable, such as the uncertainty range, when the request is completed and not when it is committed at a later time. Thus, if the request is later aborted before it is committed, the resource state variables of the Selector do not reflect this fact. Referring back to the bounded buffer example, we can assume that the buffer is empty; hence, the uncertainty range of #items is [0,0]. If an insertitem request is allowed to proceed, then the range will be changed to [0,1]. When the insertitem operation completes but before it is committed, the range is updated to [1,1]. If a removeitem request is received, it is allowed to access the buffer since the range is [1,1]. However, if the
insertItem is aborted at a later time, this means that the removeItem operation has removed an item that was not meant to be inserted. At first glance this seems troublesome (and it is), but because of the way the object-version manager is designed, the remove operation would be using a version of the object that is dependent of the version used by the insertItem operation. Hence, this removeItem operation is made dependent on the insertItem operation, and thus aborting the insertItem operation would automatically abort the removeItem operation. However, the uncertainty range still has to be updated to reflect the state of the object in case of an abort to enable the Selector scheduler to correctly synchronize the concurrent Module’s operations.

From the previous discussion, it is clear that a selection policy is not needed to maintain the Module constraints or express semantic knowledge. A selection policy is only used to select certain requests over others out of the requests that can be served next. This is done to achieve some efficiency either in resource operation or in the response times of some requests. Expedite and postponed policies are used only to make sure that the selection policy is fair and does not cause poor service to some requests. Such policies cause the introduction of deadlock conditions, as stated by Theorems 2 and 3 of the previous subsection, and introduces some overhead. For reasons of simplicity as well as avoiding deadlock and unnecessary overhead, we feel that such efficiency is too costly and does not contribute to our goals of achieving a high degree of concurrency in the synchronizing Module server operations. We assume that requests are selected first-come-first-served, provided they satisfy the Module concurrency constraints as will be explained later.
We feel that History variables, such as LAST-ACTIVE.requesttype, are used to express selection, expedite, and postpone policies to solve a limited number of synchronization problems and thus are not needed in synchronizing the Module’s server operations.

In the next subsection a number of extensions and modifications to selectors are presented in the form or reliable selectors which are able to add the following functions to the already provided functionality of Selectors:

provide the capability to schedule requests which access different and independent units of the Module’s object;

provide the capability so that the selector’s resource state variables would reflect the actual state of the object in case of a request abort or in case of a node failure;

avoid unnecessary overhead caused by unneeded selection, expedite, and postponed policies.

6.4. Reliable Selectors

A number of modifications and additions are made to make selectors usable by the recoverable Modules in a reliable distributed system for the environment of interest.
6.4.1. State Variables

To solve the problem of Selectors not reflecting the state of the object in case of a node failure, an associated stable state variable will be used for every state variable. Stable state variables can survive a node crash and thus can maintain their values. The stable state variable will be updated when the version accessed by a server operation changes its state to committed; thus, the stable state variables reflect the committed state of the object.

6.4.2. Updating State Variables

To solve the problem of selectors not reflecting the state of the object in case of a request abort, a modification to the PROCESSING section of the selector specification is needed. The value of a state variable is kept in the uncertainty range. Such a range will increase when the request accesses the object and will decrease when it completes (version state in Ready). The update to the certainty range is specified by the object definer in the PROCESSING section of the selector. However, a request can be aborted after it has completed; thus, the state variable must be updated by reversing the effect of the completed request on the state variable. This is specified by the object definer in the PROCESSING SECTION of the selector. The statement UPON ABORT ( body ) is added to the PROCESSING SECTION. When a request is aborted, the body is executed to reverse the effects of the request. For instance, in the example of a bounded buffer the following is added to the processing section of the Selector specification:

```
PROCESSING
    Remove item PROCESSED by Remove-item
UPON SERVICE
    #items := #items - 1
```
6.4.3. Accessing Different Units of the Object Concurrently

Selectors do not allow for the access of different units of the object simultaneously by different requests as was discussed in Section 6.2. Having this property, the scheduler can allow for a high degree of concurrency and thus better performance. However, the performance can be degraded if the implementation of this property will introduce a high level of overhead, as in the case of using a lock for individual units [40, 22].

Different objects may have a different granularity of the independent units that make up the objects; thus, it is important to provide for allowing the object definer to specify the granularity of these independent units (since the object definer is familiar with the object implementation).

To provide for this property, we introduce the concept of Granularizer to selectors. A Granularizer represents the state of the object units from two perspectives. It represents the status of the unit in respect to whether it is busy, is currently accessed by an operation, or is free. The Granularizer also represents the semantic state of the unit in respect to whether the unit can be accessed by a specific operation. For example, an empty unit in the bounded buffer cannot be accessed by a remove operation. The status information maintained by the Granularizer is used to access different units of the objects simultaneously.

The Granularizer is a structure with some operations (SEARCH,
CHECK, SET, and RESET) on the structure. The granularizer structure consists of a number of maps representing the state of the units of the object as shown in Figure 24. In the primary map, each unit is represented by a cell and each cell is addressed the same way as the object unit it represents. The cell may assume two values: busy or free. The value of a specific cell in the primary map will be busy if the unit it represents is being accessed; otherwise, it will be free. The primary map can be considered as a selector counting variable since it represents some state of the object. However, unlike state variables, the primary map does not have an uncertainty range since it reflects whether the object unit is currently accessed or is free. The primary map will not have a corresponding stable state variable associated with it. Stable state variables are not needed since upon recovery the primary map should reflect a state of the object where no requests are using the object, which is the initial state. The reason is that all inprogress requests that were using the object, which the primary map is supposed to be reflecting, will be automatically aborted, and their versions discarded.

If a requested operation specifies the address of a specific unit of the object, the cell representing the specific unit addressed by the request is checked using the CHECK operation on the primary map. If the unit is busy, the value of the CHECK operation is false; otherwise, the cell is set to a busy state and the result value of the CHECK operation returned is true.

If the request does not specify a specific unit address, the Granularizer operation selects a unit and returns that unit’s address which will be used by the request. To be able to perform a correct selection, the
Figure 22:  Granularizer Structure
Granularizer must represent some semantics about the object. For example, if the object is a bounded buffer of 10 units, then selecting a free unit (not currently used) to perform a remove operation might not be sufficient to perform a correct remove operation. The selected unit must also have a value to be removed. To be able to do that, the Granularizer not only reveals the states of the object in relation to the concurrent server operation currently accessing the units of the object, but also will reveal whether a specific operation can be performed on the unit.

In representing the semantics of the object the Granularizer structure will have multi-maps, called GRN.operation-type, in addition to the primary map (see Figure 22). Each GRN.operation-type map represents the units of the object that can be accessed by the operation-type, and is used to acquire an address of a unit of the object that can be used by the operation-type. An address of a unit of the object J is returned if GRN.operation-type.J is not currently used by another operation and if it has a value that can be used by the operation-type. In the bounded buffer example, a Remove operation that does not specify a unit address can enter the resource if a unit of the object is not currently being used by any other (Inprogress) operation, and the unit also has a value that can be removed. The scheduler (selector) will determine such a unit using the GRN.Remove map and the primary map. A free (not being used) unit that has a value can either be in a Committed, Prepared, or Ready state, but not in Inprogress or Discard states. For efficiency reasons, the scheduler will select a unit in the following order: Committed, Prepared, then Ready. The GRN.operation-type map reflects such states.
The GRN.operation-type map reflects the state of the object and thus it is a state variable. As with other state variables, it is updated in the PROCESSING section of the selector when the request is completed, for example:

```
PROCESSING
  Remove item PROCESSED by Remove-item

UPON SERVICE
  # items := # items - 1
  GRN.INSERT.J = (R,ON)
```

This is to indicate that upon the completion of the remove operation, the GRN.INSERT.J is updated to indicate that the particular unit is now usable by the INSERT operation. J is the address of the specific unit, R is the status of the map (Ready), and "ON" indicates that it is usable by the operation-type. The GRN.operation-type version does not have to keep an uncertainty range since it is not used as part of the resource invariants specifications.

Like state variables, the GRN.operation-type map has an associate stable GRN.operation-type which survives a scheduler (due to node crash) crash and reflects the committed state of the object.

The structure of the the Granularizer specified by the object definer in a state variable declaration as follows:

```
DECLARE STATE VARIABLES   #items INITIALLY 0
  GRANULARIZER OF SIZE (#units) UNITS
  GRN.request-type1
  GRN.request-type2
```

When the request enters the object, the Granularizer structure is
updated to reflect that an operation is currently using a specific unit. When the request completes (exists the object) then all GRN.operation-type versions affected by the request will be updated to reflect the change in the value of a specific unit J.

There are a number of advantages to using the Granularizer. In the case when a request provides the address of the unit to be used (data base record), the use of the Granularizer requires little overhead as compared with another method that uses locking where the object definer has to define the granularity of the locks and to associate each lock with an object unit and to define different locks for each unit (read lock, write lock) [21]. The Granularizer in conjunction with the object multi-version technique is used to increase the degree of concurrency by allowing the concurrent processing of operations that do not effect each other, and by allowing the concurrent processing of operations that use different object units.

6.4.4. Request Processing in the Reliable Selector

Figure 25 shows the flow of a request through the selector. Once a request is received, the object limits conditions are tested. If the conditions are satisfied, then the request proceeds to the next step; otherwise, the request joins the waiting holder. The next step is to test the concurrent constraints of the operation in order to test if the operation requested can run concurrently with other active requests, regardless of the object unit it requests. If the concurrent constraints are satisfied, then the request proceeds to the resource. Otherwise, it will go to the Granularizer to test if the request is accessing an unused unit of the object (currently not used by another request). If the request provides as a parameter the address of the unit, then the test is
done by checking the primary map only. This is performed using a 
CHECK operation on the primary map. If the unit is busy, the request 
joins the waiting holder; otherwise, the cell is set to busy using a SET 
operation and the request proceeds to the object. If, however, the 
request does not provide the address, then the Granularizer chooses an 
appropriate unit which is not currently used by an active request and 
which semantically can be accessed by the specific request. This is done 
by performing a SEARCH operation on the GRN.operation-type map to 
locate a cell in the map which represents a unit that can be used by the 
operation (operation-type). If a cell could not be located, then the 
request enters the waiting holder. If a cell is located, then the 
corresponding cell in the primary map is checked using a CHECK 
operation. If such a map is free then the request is allowed to access 
the allocated object unit; otherwise, it joins the waiting holder.

Once in the waiting holder, requests are ordered by their time of 
arrival, on a first-in-first-out basis. A change of the conditions of the 
constraints can be caused by the completion of a request (return from 
the object). The selector will then examine the requests in the waiting 
holder to determine the first request that can be served (satisfies object 
limits, concurrent constraints, and concurrent unit access) and will send 
it to the object.

6.4.5. Deadlock and Starvation Revisited

Selectors can cause deadlock and starvation as was discussed in Section 
6.3. However, since a first-come-first-serve selection policy is used in 
Reliable Selectors, and no expedite or postpone policies are used, then by 
Theorem 2 of subsection 6.1.5 deadlock is eliminated from Reliable 
Selectors.
Figure 23: Module's Operation Request Processing in the Scheduler
Two types of starvation can occur in the selectors' request-abundance and resource-limit.

Request-abundance starvation is caused by the continual arrival of some request types that can be executed concurrently. In reliable selectors, this is avoided by putting a limit on the number of concurrent operations that can be executed at one time. This is done by the Module's version manager by putting a limit on the number of versions that can be created in one column (see Chapter 5, Section 5.3). Once this limit is reached all other requests that can execute concurrently join the waiting holder, where requests are processed by the time of arrival as will be explained next, and thus starvation due to request-abundance is eliminated in Reliable Selectors.

In resource-limit starvation, a request may starve because the resource state is not updated by another request to allow this request to be served. For example, a Remove operation will starve if the empty bounded buffer is not accessed by an Insert-item operation. To avoid this resource-limit starvation, a time-out mechanism is used. If the request is not served by a specific time and the resource remains idle for that period of time, then the request is aborted.

6.5. Interaction Between Object-version Manager and Reliable Selectors

It is our goal to separate the implementation of the selector from the implementation of the object-version manager, so that each will stand as a separate unit. This is done so that selectors can be specified and constructed separately at the same time the object is constructed or at a later time as the object is introduced in the distributed environment.
This also allows for modifying the synchronization policies of the object without affecting the object.

The Selector and Object Manager interact through defined interface as explained next. We will concentrate on how the selector reacts in the interaction, and not on how the OM performs its operation.

1. When the selector chooses a request to send to the object, the selector will update all appropriate state variables and selector variables, including the primary map and the GRN.operation-type maps. The request is then sent to the object.

2. When a request has been completed, a message is sent to the selector indicating this fact. The state variable, primary map, GRN.operation-type maps, and scheduler variable are updated accordingly. It should be noted that state variables are kept in the uncertainty range. When the request enters the object, the uncertainty range increases, and when it finishes it decreases.

3. At a later time the version used by a completed request will change its state to Committed. A message is sent to the selector indicating that fact. The selector, upon receiving the message, will update the stable-state variables that will reflect the state of the Committed versions of the object.

4. If, however, the object version changes its state to abort (because the request is aborted), then a message is sent to the selector for that request. The selector will perform the UPON ABORT statement of that specific request type. This will reverse the effect of the request on the state variables.
5. Due to the design of the OM, there is a limit to the number of concurrent requests that can access the object. Thus, the OM will send a message to the selector requesting it to stop sending further requests; therefore, the selector will be reactivated. When the OM is ready to receive new requests, it will inform the selector which will be reactivated. While the selector is reactivated, all new requests will join the waiting holder. Once the selector is activated, it will select requests to be served from the waiting holder in the first-come-first-served fashion provided they comply with the constraints. The reactivation of the selector helps eliminate starvation caused by Request-abundance as was explained earlier.

6.6. SUMMARY

This chapter presents a synchronization mechanism for synchronizing multiple concurrent requests by the Recoverable Module. The mechanism uses extended Selectors in the form of Reliable selectors. The mechanism is combined with a novel optimistic concurrency control mechanism to synchronize concurrent distrusted transactions presented in the following chapter.

A review of Selectors is presented, and then Selectors are evaluated to examine their applicability to the environment of interest. Some extensions and modifications to Selectors are presented in the form of Reliable Selectors. Reliable Selectors are failure resistant and provide for specifying consistency constraints of the recoverable Module and allow for the uses of semantic knowledge to achieve a high degree of concurrency.
We have adopted an optimistic concurrency control scheme to implement concurrency control among concurrent transactions. Since optimistic schemes are based on detecting rather than preventing conflicts between transactions, they have the potential for increased concurrency and minimized transaction delay. The scheme is used in cooperation with reliable selectors to guarantee that consistency constraints are preserved among a group of cooperating Modules accessed by a given transaction. Basically, distributed transactions proceed freely in the system, synchronized only by the constraints on the individual Modules themselves. Correct concurrency control occurs when a transaction tries to commit as a part of a two-phase commit protocol. If the transaction does not conflict with any other, it is allowed to commit. Otherwise, one of the conflicting transactions is aborted and restarted. In the optimistic approach it is assumed that conflicts are rare and so only a few transactions have to aborted. We believe that such an assumption is valid in our environment.

Several optimistic schemes have been presented [23, 9, 7]. These are not suited to our needs for one or more reasons. First, they have been developed for database rather than object based distributed systems. Second, global validation and conflict detection may be based on timeout mechanisms which tend to unnecessarily abort transactions that are not involved in a conflict cycle. Third, summary conflict information may be
required from all sites, even though they might not have participated in processing a transaction, thus incurring unnecessary overhead and rendering them impractical for use in distributed environments.

7.1. An Optimistic Concurrency Control Mechanism

The proposed mechanism incorporates the use of the multi-version technique of the Module object presented in the last chapter, a unique system id (uid), and a conflict detection technique. There are three steps in the execution and completion of a distributed transaction: 1) Module access, which requires creation and management of Module object versions, 2) validation of correct concurrency control, and 3) committing of the transaction. Each of these steps is specified in detail below.

7.1.0.1. Module Access

A distributed transaction is initiated by a top level request and is assigned a unique system wide id (uid). The subsequent sequence of Module requests form the Module cooperation or access tree. At each node in the tree, transaction requests are submitted to the reliable selector on a FIFO basis, and then are scheduled according to the selector scheduler's constraints (see figure 26). A transaction t2 may become dependent upon another transaction t1 if it accesses an object which has been changed by a yet uncommitted t1.

A new version of the Module object, with the version number equivalent to the transaction's uid, is created for each accessing transaction if a version does not already exist for it as was explained in Chapter 5. Once an object version is created for a transaction, it is used by all Module operations requested as a part of that transaction.
Figure 24: Module Access: Submitting of Operations
When the requested operation is complete, results are propagated up to the requester. However, the version is not written to the object until the requesting transaction commits. When results are propagated to the initiating Module and the requested operations of the Module have terminated, the validation phase of the transaction starts.

7.1.0.2. Validation

The primary purpose of the validation step is to prepare the transaction to commit as part of a two phase commit protocol. In particular, this involves changing the state of the object versions accessed by the transaction from R, ready, to P, prepared to commit. In addition, the step requires the exchange of validation information among cooperating Module interfaces accessed by a given transaction in order to enable the transaction to determine whether it is in conflict with another transaction(s) to assure correct concurrency control.

When a transaction finishes accessing all the objects it requested, the Module interface of the initiating Module initiates a prepare to commit message which is sent successively down the Module cooperation tree. When a Module interface receives a prepare to commit request, it asks the object version manager to prepare the version. At that time the local version is prepared and a has-prepared message is propagated back up the cooperation tree, as long as all lower nodes have already issued has-prepared messages. This scheme requires that the latest committed version in an object's version history must always exist so that it can be decided whether or not to commit a subsequent version.

In this scheme a cyclical commit wait, resulting in a commit deadlock, can occur. For example, if t1 is made dependent on transaction t2 in
accessing m1 and cannot commit until t2 commits, and if t2 is made dependent on t1 in accessing m2 and cannot commit until t1 commits, then a wait to commit cycle is formed. Since the validation step verifies that a transaction is not in conflict with another transaction, that is to say it is not dependent upon another transaction that is directly or indirectly dependent upon it, validation reduces to detecting if the transaction is indeed involved in a commit deadlock. If a commit deadlock is detected by the validation step, then the transaction is aborted. An informal description of the algorithm and the detection of a commit deadlock is presented, and then a formal specification of the algorithm and its proof of correctness are presented.

7.1.0.3. Committing

When the initiating Module interface receives has-prepared messages from the MI's of all Modules it requested, it sends a commit message to the immediate MI's, which in turn send it down throughout the cooperation tree. Each Module interface receiving the message requests that its object version manager commit as was described in Chapter 5.

7.2. Commit Deadlock Detection and Proof of Correctness

7.2.1. Commit Deadlock Detection

In this section an informal procedure for detecting a commit deadlock for a given transaction is presented. As was mentioned in the previous section, the function of the validation step of the proposed mechanism is to prepare each participant Module for commitment. However, a Module may not be able to prepare because a commit deadlock may have occurred. Thus, unless the commit deadlock is detected, the validation process may not terminate.
During the validation step some transaction dependency information is exchanged among the participants. Such information will allow a transaction or a group of transactions to detect whether they are in a commit deadlock.

The object manager of a Module attempts to maintain access to the Module object by a uid order (time stamp order). If, however, an out of order request (has a uid smaller than the uid of the last transaction to access the object) is received, the request is not rejected. Rather, it is allowed to access the object and dependency information is maintained.

Membership in two sets is maintained by the object-version manager for each transaction accessing a Module object version: the waiting-on set (ON\_\_O\_\_n) and the waited-on set (WD\_\_O\_\_n). The waiting-on set contains the uid's of all the transactions that T\_i are directly or indirectly waiting on to commit before it can prepare, and which have uid's larger than the uid of T\_i. T\_i should not be waiting on this latter set, according to the time-stamp ordering. The Waited-on set contains the UID's of all transactions that are directly or indirectly waiting on T\_i to commit before they can prepare, and which have UID's smaller than the UID of T\_i. In other words, the transactions are waiting on T\_i but they are not supposed to according to time-stamp ordering. For example, in Figure 27, T\_5 is made dependent on T\_7. Thus, T\_7 is a member of WN\_5 and T\_5 is a member of WN\_7.

The WN\_i and WD\_i are formed as follows:

1) At the initial access of the Module's object by the transaction T\_i, T\_i is inserted in WD\_j, and T\_j in WN\_i, the waiting-on set of T\_i, if T\_i is made dependent on T\_j directly or indirectly in accessing the object (T\_iO\_n
Figure 25: Forming Transaction Dependency Sets
---* TjO_n, and globally Ti ---* Tj), and if the uid of Ti is smaller than the uid of Tj (i<j).

2) During the validation step, validation information is exchanged in three phases: upward, downward, and waiting-to-prepare as shown in Figure 28.

**Upward:** As a result of receiving a prepare to commit message, the object-version manager sends both the waiting-on and the waited-on sets to the requesting Module's MI which in turn will update its sets and send the sets to its parent Module's MI. The process continues until the top level Module MI receives the sets of its children. The union of those sets will produce the global waiting-on set and the waited-on set.

**Downward:** The top level MI will send its updated sets to the MI of the Modules it accesses, which in turn update their sets and send them to the MI's of the Modules they requested. This process continues until all participant MI's have global waiting-on and waited-on sets.

**Waiting-to-prepare:** At any time while a version is waiting to be prepared, new transactions may be added to the waiting-on and waited-on sets. In this case the uid of such transactions are sent upward and then downward as explained before so that these transactions join the global waiting-on and waited-on sets.
Figure 26: Dependency Information Transfer During Validation
3) Whenever the \( W_{N_i}O_n \) or \( W_{D_i}O_n \) of a given object version is updated (a new element is inserted in the set), the following is performed. Let \( T_k \) be the element that is inserted in \( W_{D_i}O_n \). If \( T_iO_n \rightarrow^* T_jO_n \) and \( k < j \), then \( T_k \) is also inserted in \( WD_j \). Let \( T_k \) be the element that is inserted in \( W_{N_i}O_n \). If \( T_jO_n \rightarrow^* T_iO_n \) and \( j < k \) then \( T_k \) is inserted in \( WN_jO_n \). For example, in Figure 25, if \( T_3 \) is inserted in \( WN_5 \), then it is also inserted in \( WN_7 \).

If at any time during the validation step a transaction is a member of both the waiting-on and waited-on sets, then the transaction is involved in a commit deadlock. Otherwise, it is not.

7.2.2. Proof of Correctness

In this section we present the detailed algorithm to detect a commit deadlock and the proof of its correctness.

A.1 Terminology

Each transaction \( T_i \) is assigned a transaction unique id \( UID \) at the initiating site. Such UIDs form a partial order; if \( T_i \) is initiated before \( T_j \), then \( UID(T_i) < UID(T_j) \). For simplicity we will refer to \( i \) of \( T_i \) as the unique id of \( T_i \) and thus \( T_i < T_j \) if \( i < j \) and \( i,j \in N \).

If \( T_j \) is accessing an object \( O_n \) using an operation \( op_n \), it may use the value of the version \( V_i \) accessed by \( T_i \) to create its own version \( V_j \). \( V_i \) must have R or P status, which means \( T_i \) has finished accessing \( V_i \) but has not committed yet. Thus \( T_j \) depends on the commitment of \( T_i \) in accessing \( O_n \) which is denoted as

\[
T_jO_n \rightarrow^* T_iO_n
\]
and \( T_j \rightarrow T_i \) globally if
\[
\exists O_n \in O \quad T_j O_n \rightarrow T_i O_n
\]

\( \rightarrow^* \) is the transitive closure of the relation
\[
\rightarrow \quad \text{and}
\]
\( T_j \rightarrow^* T_i \) if \( T_j \rightarrow T_i \) or
\[
\exists T_m \in T \setminus T_j \rightarrow^* T_m \quad \text{and}
\quad T_m \rightarrow^* T_i
\]
A transaction \( T_i \) is said to be in a COMMIT-DEADLOCK if
\[
\exists T_j, T_i \in T \quad T_i \rightarrow^* T_j
\]
\[
\text{and} \quad T_j \rightarrow^* T_i \quad \text{for } i,j \in N.
\]

A.2 Commit Deadlock Detection Algorithm

For each Module object version \( V_i \) accessed by a transaction \( T_i \), two sets of transactions are maintained: the waited-on set \( (WD_j O_n) \) and the waiting-on set \( (WN_i O_n) \). The two sets are constructed as follows:

1) Initial Access:

At the initial access of the object \( O_n \) by \( T_i \),
\[
T_i \text{ is inserted in } WD_j O_n \quad \text{and} \quad T_j \text{ is inserted in } WN_i O_n
\]
if \( T_i O_n \rightarrow^* T_j O_n \) and \( i < j \).

2) Validation Step:

At the validation step of \( T_i \), and during the upward phase, the
global waiting-on set $WD_i$ and the global waited-on set $WN_i$ of $T_i$ are constructed such that

$$WD_i \equiv WD_i O_n$$ where $O_n \in S_i$ (set of objects accessed by $T_i$).

and similarly

$$WN_i \equiv WN_i O_n$$

During the downward phase, each local set assumes the value of the global set so

$$WD_i O_n \equiv WD_i$$ and $$WN_i O_n \equiv WN_i$$

This is done in the following way

\textbf{upward phase:}

Let $WD_i O_n^k$ be the $WD_i O_n$ of $T_i$ in level $k$ of the access tree of the transaction $T_i$. Then,

$$WD_i O_n^{k-1} \equiv WD_i O_n^{k-1} \cup WD_i O_p^k \cup WD_i O_m^k$$

and

$$WN_i O_n^{k-1} \equiv WN_i O_n^{k-1} \cup WN_i O_p^k \cup WN_i O_m^k$$

where $O_n$ is an object of Module $M_n$ accessed by $T_i$ in level $k-1$ and $O_p$, $O_m$ are objects of Modules $M_p$, and $M_m$ are part of the Modules requested by Module $M_n$.

Thus $WD_i O_n^0 \equiv WD_i O_n$ and $O_n \in S_i$. 
and \( W_{D_i} \subseteq W_{D_i}O_n^0 \)

**downward phase:**

\( W_{D_i} \) is then sent to all participant MIs, and the \( W_{D_i}O_n \) of every Module interface is updated such that,
\[
W_{D_i}O_n \equiv W_{D_i}O_n \cup W_{D_i}
\]
similarly for \( W_{N_i}O_n \).

**waiting-to-prepare phase:**

While the transaction \( T_i \) is waiting to prepare at \( O_n \) after the upward phase, an element may be inserted in \( W_{N_i}O_n \) or \( W_{D_i}O_n \) as follows,

let \( W_{D_i}O_n^0 \) be \( W_{D_i}O_n \) after the upward phase and before committing, and

let \( SW_{i}O_n \equiv W_{D_i}O_n \cap W_{D_i}O_n^0 \).

If \( SW_{i}O_n \neq \emptyset \) then

1. \( W_{D_i} \equiv W_{D_i}O_n \cup W_{D_i} \)

   by sending a message upward to the top level MI as in the upward phase. then

2. \( W_{D_i}O_n \equiv W_{D_i} \cup W_{D_i}O_n \) where \( O_n \notin S_i \).

The updated global set

\( W_{D_i} \) is sent from the top level MI to all participants as in the downward phase, similarly for \( W_{N_i} \).

3 **Set Update:**
At the Module level any time a WD$_i$O$_n$ or
WN$_i$O$_n$ of $T_i$ is updated, a new element is inserted, and the
following is performed. Let $T_k$ be the element that is
inserted in WD$_i$O$_n$. Then
if $T_i$O$_n$ ---* $T_j$O$_n$ and $k<j$, then
$T_k$ is inserted in WD$_j$O$_n$. Let $T_k$ be the element that
is inserted in WN$_i$O$_n$ if $T_j$O$_n$ ---* $T_i$O$_n$ and
$j<k$; then $T_k$ is inserted in WN$_j$O$_n$.

$T_i$ is in a COMMIT-DEADLOCK if

1) WD$_i$O$_n$ $\cap$ WN$_i$O$_n$ $\neq$ $\phi$
    which means $T_i$ is waiting on some transaction
    $T_j$, and $T_j$ is waiting on $T_i$.
2) $T_i$ $\in$ WD$_i$O$_n$ or $T_i$ $\in$ WN$_i$O$_n$,
    which means $T_i$ is in a cycle waiting on itself to commit.

A.3 Proof of Correctness

In this section a proof for the correctness of the COMMIT-
DEADLOCK procedure is presented. The dependency relation between
transactions can be represented as a dynamic waiting graph (DWG) that
is constructed and checked for cycles. If a cycle is found, then a
COMMIT-DEADLOCK exists, and one of the involved transaction must
be aborted to break the deadlock.
A Dynamic Waiting Graph (DWG) is a graph \( <V,E> \) where \( V \) is the set of vertices representing \( T \), the set of transactions, and \( E \) is the set of edges, where \( <ij> \) is an edge if and only if,

1) At initial accessing time of \( O_n \) by \( T_i \),

\[ T_i O_n \rightarrow^* T_j O_n \text{ and } i<j \text{ or } T_j \text{ is inserted in } W N_i \]

or \( T_i \) is inserted in \( W D_j \)

2) At any time during validation phase,

if \( T_i \in W D_k O_n \text{ and } T_k O_n \rightarrow^* T_j O_n \text{ and } i<j \)

then \( T_i \) is inserted in \( W D_j O_n \) and an edge \( <ij> \) is formed.

If \( T_j \in W N_k O_n \text{ and } T_i O_n \rightarrow^* T_k O_n \text{ and } i<j \)

then \( T_j \) is inserted in \( W N_i O_n \) and an edge \( <ij> \) is formed.

It should be noted that only one set, \( W D_i O_n \) or \( W N_i O_n \), is needed to form a cycle in the DWG graph. Thus, only one set needs to be maintained to detect a COMMIT-DEADLOCK. However, using both sets expedites the detection of a deadlock, if one exists, and hence the number of dependencies that might form before the detection of a deadlock is reduced, as well as the number of transactions that have to be aborted in case of commit deadlock, is reduced. The proof of correctness is done using only the set \( W N_i \) and can be done similarly using \( W D_i \).

**Proposition 1:** Let \( S_{ij} \) be a set transactions such that

\[ S_{ij} = (T_k \mid T_i \rightarrow^* T_k, T_k \rightarrow^* T_j \cup T_i \cup T_j, \]
and let \( T_n = \text{Max}(S_{i,j}) \mid T_n \in S_{i,j} \) and \( \forall T_m \in S_{i,j}, m < n \).

If \( T_m = \text{Max}(S_{i,j}) \) and \( T_i \rightarrow T_j \) and \( T_p \in WN_j \) and \( (T_m < T_p \text{ or } p = m) \), then \( T_p \in WN_j \) and there is a path \( T_i \) to \( T_j \) in the DWG.

**Proof:** \( T_i \rightarrow T_j \) implies that

\[
T_i \rightarrow T_m \rightarrow T_n \rightarrow \ldots \rightarrow T_k \rightarrow T_j
\]

which implies that

\[
T_iO_p \rightarrow T_mO_n \rightarrow T_nO_j \rightarrow \ldots \rightarrow T_kO_s \rightarrow T_jO_l
\]

(1)

Let \( T_p \in WN_jO_l \) and thus \( j < p \).
Since at set-update step, 

\( T_n \) is inserted in \( WN_j \) if \( T_n \in WN_j \) and

\[
T_i O_n \iff T_j O_n \quad \text{and} \quad i < n
\] (2)

then from (1),(2) since \( T_k O_l \iff T_j O_l \) and \( k < p \) then

\( T_p \) is

inserted in \( WN_k \) and there is an edge \( <T_k,T_j> \).

Further, since now \( T_p \in WN_k \) and \( l < p \) then \( T_p \)
is inserted in \( WN_l \) and the edge \( <T_p,T_k> \) is formed.

For all \( T_n O_x \iff T_m O_x \) in (1)

if \( T_p \in WN_m \) and \( n < p \) then \( T_p \in WN_n \) and there
is an edge

\( <T_n,T_m> \). Thus since \( T_m < T_p \) \( (T_m = \max(S_{i,j})) \), then

\( T_p \) will propagate up to \( WN_n \) and there is

a path \( T_i \) to \( T_j \) in \( DWG \).

**Proposition 2:** Let \( T_j = \max(S_{i,j}) \) and \( T_i \iff T_j \),

then \( T_j \in WN_i \) and there is a path \( T_i \) to \( T_j \) in \( DWG \).

**Proof:** Since \( T_k O_l \iff T_j O_l \) and \( k < j \) in

(1) then

\( T_j \in WN_k \) and there is an edge \( <k,j> \).

Since \( T_j \in WN_k \), \( T_j = \max(S_{i,j}) \), and \( T_i \iff T_k \)
then by

proposition 1, \( T_j \in WN_i \) and there is a path \( T_i \) to \( T_k \).
Thus there is a path \((T_i, T_j)\) and \(T_j \in WN_i\).

**Theorem 1:** The system is COMMIT-DEADLOCK free if and only if the DWG is acyclic.

**Proof:**

(1) The sufficient condition.

If no COMMIT-DEADLOCK exists then by definition

\[ \forall T_k \in T; \text{ if } T_k \longrightarrow^* T_i \text{ Then it is not the case that } \]

\[ T_i \longrightarrow^* T_k. \quad (1) \]

Let DWG be cyclic then \( \exists T_i \mid T_i \notin WN_i O_n \),

which implies that,

\[ \exists T_k \mid T_i \notin WN_k O_n \text{ and } T_i O_n \longrightarrow^* T_k O_n, \quad k<i \quad (2) \]

\[ T_i \notin WN_k O_n \text{ implies that, } \]

\[ \exists O_n \in S_i \mid T_k O_n \longrightarrow^* T_i O_n \text{ and } k<i \quad (3) \]

or \( \exists O_n \in S_i, T_j \in T \quad T_i \notin WN_j O_n \text{ and } \]

\[ T_k O_n \longrightarrow^* T_j O_n \text{ and } k<i, i<j. \]

Thus \( T_k \longrightarrow^* T_j \text{ and } T_j \longrightarrow^* T_i \text{ or } \]

\[ T_k \longrightarrow^* T_i. \quad (4) \]
From (2) and (3) $T_k O_n \longrightarrow^* T_i O_n$

$T_i O_n \longrightarrow^* T_k O_n, \; k<i$

which implies $T_k \longrightarrow^* T_i$, and $T_i \longrightarrow^* T_k$, $k<i$. This is in contradiction with (1), and similarly from (2), (4).

(2) The necessary condition: if a deadlock exists then DWG is cyclic.

Assuming that there is a COMMIT-DEADLOCK, then by definition

$T_j, \; T_i \in T,$

$T_i \longrightarrow^* T_j, \; T_j \longrightarrow^* T_i$ and $i<j$

$T_i \longrightarrow^* T_j$ implies that

$T_i \longrightarrow T_k$

$T_k \longrightarrow T_m$

$T_o \longrightarrow T_p$

$T_p \longrightarrow T_j$

which implies,

$T_i O_n \longrightarrow T_k O_n$

$T_k O_m \longrightarrow T_m O_m$
\[ T_0 O_i \rightarrow T_p O_i \]
\[ T_p O_i \rightarrow T_j O_i \]  \hspace{0.5cm} (A)

\[ T_j \rightarrow * T_i \text{ implies} \]
\[ T_j \rightarrow T_a \]
\[ T_a \rightarrow T_b \]
\[ T_g \rightarrow T_h \]
\[ T_h \rightarrow T_i \]

which implies that

\[ T_j O_n \rightarrow T_a O_n \]
\[ T_a O_n \rightarrow T_b O_n \]
\[ T_g O_i \rightarrow T_h O_i \]
\[ T_h O_k \rightarrow T_i O_k \]  \hspace{0.5cm} (B)

Let \( T_m = \text{Max}(S_{ij}) \) in (A) and \( T_b = \text{Max}(S_{ij}) \) in (B) and
b < m. Since $T_i \rightarrow T_m$ in (A) and $T_m = \text{Max}(S_{i,m})$
then by proposition 2 there is a path from $T_i$ to $T_m$ in $\text{DWG}$
and $T_m \in WN_j$.

At the validation step of $T_i$, $T_m$ is inserted in $\text{WN}_jO_k$
in (B). Since $T_m \in WN_jO_k$ in (B) and $b < m$ and
$T_j \rightarrow T_i$, then by proposition 1 there is a path
$T_i$ to $T_j$ in $\text{DWG}$ and $T_m \in WN_jO_n$.

At validation step of $T_j$, $T_m$ is inserted in $\text{WN}_jO_1$ in
(A). Since $T_m \rightarrow T_j$ and $T_m = \text{Max}(S_{m,j})$ and
$T_m \in WN_j$, then by proposition 1 there is a path $T_m$ to
$T_j$ and $T_m \in WN_m$.

Thus, there is a path $T_m$ to $T_j$, $T_j$ to $T_i$ and $T_i$
to $T_m$ which is a cycle in $\text{DWG}$. Q.E.D

7.3. Serializability and Consistency

A schedule produced by the proposed synchronization mechanism must
be consistent to preserve system consistency. A consistent schedule can
be defined as follows:

Let $S$ be a schedule $T_1, T_2, ..., T_n$. Let $T_iO_n < T_jO_n$ if transaction
$T_i$ performs an operation $O_{p_1}$ on object $O_n$ in some step in $S$ and
transaction $T_j$ performs operation $O_{p_2}$ on object $O_n$ at a later time, and
$O_{p_1}$ and $O_{p_2}$ are incompatible. Let $<^*$ be the transitive closer of $<$
then the schedule $S$ is consistent if $<^*$ is a partial order:

$T_1 <^* T_2 <^* T_3 <^* ... <^* T_n$
A serializable schedule is consistent, since it is equivalent to a serial schedule. They are equivalent in the sense that they transform a given initial state of the system (collection of Modules objects) to an equivalent final state. "Equivalent final state" is deliberately used here to indicate that the final state may be logically or semantically (as interpreted by the user) equivalent rather than being physically the same. The mailbox example in Section 4.4.1 show an instance of two schedules that are semantically, but not physically, equivalent. Therefore, the definition of serializability becomes more general and encompasses more schedules that are consistent but that may not be considered serializable because they do not produce the same state produced by a serial schedule.

Schedules produced by the proposed synchronization mechanism are serializable and thus consistent (or legal). We present a formal proof of consistency. Informally, from the definition of consistency, a schedule S is inconsistent if \( T_i \prec T_j \) and \( T_j \prec T_i \). In the proposed mechanism this is equivalent to saying \( T_i \rightarrow^* T_j \) and \( T_j \rightarrow^* T_i \), but this cannot occur since it will be detected as a commit deadlock during the validation step of either \( T_i \) or \( T_j \), and \( T_i \) or \( T_j \) will be aborted.

7.3.1. Proof of Serializability

The consistency of a schedule S produced by the proposed optimistic synchronization mechanism (OSM) can be tested by constructing and examining a directed dependency graph (DDG) for cycles.

A directed dependency graph (DDG) for schedule S produced by OSM is a graph \( <V,E> \) where \( V \) is the set of transactions and \( E \) is the set of edges, where \( <i,j> \) is an edge if and only if \( T_i \prec T_j \) in accessing some object \( O_n \).
Lemma 1: A DDG for a consistent schedule $S_1$ is acyclic.

Proof: Let DDG of $S_1$ be cyclic. Then $T_i, T_j \in T$

$T_i <^* T_j$ and $T_j <^* T_i$ but this is a
contradiction with the definition of consistency.

Theorem 2: A schedule $S$ is produced by OSM iff The DDG of $S$ is
acyclic.

Necessary Condition:

We prove that if $S$ is produced by OSM then the DDG of $S$ is
acyclic. Let DDG of $S$ be cyclic. Then

$T_i <^* T_j$ and $T_j <^* T_i$ which implies that

$T_i \rightarrow^* T_j$ and $T_j \rightarrow^* T_i$

but by theorem 1 this schedule cannot be produced by OSM
since it will be detected as a commit deadlock and either
$T_i$ or $T_j$ will be aborted.

Sufficient Condition:

We prove that if the DDG of $S$ is acyclic then $S$ is produced by
OSM. If DDG of $S$ is acyclic, then it is not the case that
$T_i <^* T_j$ and $T_j <^* T_i$ which implies that it is not
the case that $T_i \rightarrow^* T_j \text{ and } T_j \rightarrow^* T_i$

for $\forall T_i, T_j \in T$. Such a schedule is permitted by OSM,
and thus $S$ is produced by OSM.

7.4. Summary

The optimistic concurrency control mechanism takes advantage of the
multi-version technique of managing the Module's object to increase
concurrency among distributed transactions. Distributed transactions are
allowed to access the Module's objects synchronized only by the Module's
object limits and concurrency constraints. However, a transaction is not
committed until it is verified that it does not conflict with any other
concurrent transaction. It is possible that two transactions may be
deadlocked waiting for each other to commit. A commit-deadlock
detection algorithm and proof of correctness of the algorithm are
presented. The serializability property of the mechanism is discussed and
a proof that the mechanism produces serializable schedules is presented.
Chapter 8
Exception Handling

Reliable processing in the environment of interest is supported by a number of fault-tolerance measures. Forward recovery and backward recovery techniques can be used to recover from a number of different failures as was explained in Chapter 4. Backward recovery is based on restoring the state of the system (or Module) to a correct state that existed before the fault has occurred. Using the restored state the system attempts to continue processing. Forward recovery is based on identifying the fault, confining the effects of the fault, assessing the damage, and recovering from the fault. Backward recovery is usually used to recover from unknown faults and errors. In forward recovery errors are anticipated in advance and recovery measures are designed for it.

In the environment of interest both techniques are used to complement each other. Backward recovery is used in the form of the atomic recoverable Module's operations that can restore the recoverable Module state to a correct state in case of failure. Similarly, a distributed atomic transaction is used to restore a distributed system state (collection of Modules accessed by the transaction) to a correct state in case of failure.

Forward recovery is used in the environment of interest by the recoverable Module to recover from interaction failure, residual software errors, interface failure, and handle user declared exceptions. Interaction
failure is caused by the unavailability of a remote requested Module's operation caused by a permanent remote node failure, a link failure, or a withdrawal of the requested service. Residual software errors are raised within the Module operations due to incorrect input, insufficient processing resources, or programming errors. Interface failures include invalid requests or invalid parameters.

The exception handling mechanism allows the Module definer or system designer to use semantics knowledge to recover from different failures in accordance with the application needs; hence, recovery can be performed more efficiently.

There are a number of advantages to using an exception handling mechanism besides the use of semantic knowledge in recovery. The exception handling mechanism may be used to enhance system availability. A distributed transaction is not aborted due to unavailability of a particular requested service caused by a failure; instead using exception handling facilities as an alternative service may be allocated and the transaction may continue execution.

Due to the post integration of a Module into a distributed subsystem, some of the results provided by the newly integrated Module may not be desirable by the requesting Module. By the use of an Exception Handling mechanism, such results can be detected and eliminated, and only desirable results are allowed.

Most of all, an exception handling mechanism is a software construction tool that aids in constructing modular and well-structured software. By identifying exceptional conditions within a program and handling them outside the main body of the program, such a program becomes clearer,
modular, and readable, hence controlling the complexity of the software structure.

A number of exception handling mechanisms are proposed for a single system environment, such as CLU, ADA, Levin Proposal, and others [29, 179, 26]. These mechanisms are not primarily designed for a distributed environment such as the environment of interest. In this chapter we propose a new model for handling an exception that we believe is appropriate for the needs of the environment. Unlike other mechanisms, the proposed mechanism allows all possible transfers of control after handling an exception, which include the continuation of the execution of the Module raising the exception, the reexecution of the Module's operation, the return to the requesting Module, and the return to the invoker of the requesting Module. These control flow transfer alternatives are explicitly specified as part of handling an exception as will be explained later.

In Section 8.1 we introduce the components of an exception handling mechanism, and address the issues involved in designing such components. In Section 8.2 we use a server/client model of Module interaction to present the possible failures within a Module and as a result of Module interaction. In the subsequent sections we introduce the proposed exception handling mechanism as well as the implementation of the mechanism's primitives and present an example of using the mechanism.
8.1. Exception Handling Mechanism—Components and Issues

An exception handling mechanism provides the facility for detecting errors and abnormal activities, confines the damage that may be caused by signalling an exception condition in its proper context, and provides for error recovery and fault treatment by evoking a predetermined code to treat the exceptional conditions. Each Module is responsible for checking and identifying exceptional cases and informing the client Module of such exceptions. Conditions that do not conform to normal behavior are called EXCEPTIONS of that Module, and the Module is the SIGNALER of such exceptions. Recovery measures to handle such exceptions are provided in the form of a predefined program code called HANDLER. Once a Handler is executed, then a normal execution may be resumed, or it may be altered by the exception's flow control. The main four components of an exception handling mechanism are exception declaration, exception detection and signalling, exception handling, and transfer of control. These components are in turn discussed next.

8.1.1. Exception Declaration

Two types of exceptions may be associated with a software component. One is a standard set of implicit system exceptions, such as address out of range, division by zero, node failure, link failure, and process failure, and the other programmer defined exceptional conditions, such as invalid input parameters, unavailable resource, etc.

Programmer defined exceptions are given names that are identifiers associated with the exceptional conditions and are declared by the programmer. Implicit system exceptions are given system-defined names that are permanently associated with the system exception and need not be declared (e.g., ZERO-DIVIDE, LINK-FAILURE).
Exceptions may be declared as part of the server Module or the client Module. Declaring exceptions in the server Module is based on the view that the programmer of the server Module is aware of all possible exceptions that can occur, and thus these exceptions must be declared as part of the server Module's behavior. In this case, if an exception is signaled during the invocation of a server Module's operation, then it is necessary to make the client aware of such an exception so that appropriate measures can be taken by the client Module. This is found in CLU [29] and in Levin's proposal [26].

An alternative approach is to declare exceptions as part of the client Module. This is based on the view that different client Modules are interested in different exceptions; thus, a requiring procedure should declare the exception that it is interested in. ADA [179] uses this latter approach.

A SIGNALS clause is used to declare that a server Module's operation may signal a specific exception name. For example, a stack Module is providing three operations, POP, PUSH, and EMPTY. As part of the declaration of this Module's operations, the following is declared:

```
STACK PROVIDES:
  OPERATION POP(S: STACK):ITEM
  SIGNALS STACK-UNDERFLOW
```

Formal parameters may be associated with each exception name declared to provide for the capability to transfer values from the environment where the exception is signalled to the environment where the exception is handled. Without this capability, a large number of Handlers which
are similar, but which do not provide the same functionality, must be provided to handle each unique circumstances under which the same exception is signalled, or a Handler may be forced to make use of global variables, if provided, to determine the proper environment under which the exception is signalled. This may not even be possible in an environment such as the one of interest, where the server and client Modules might reside on two separate sites. In the later case, the amount of coupling between the server and the client Modules is increased.

8.1.2. Exception Detection, Signaling, and Propagation

During normal execution of a Module's operation, an exception may be detected by the system or by examining the programmer-defined exceptional conditions that evaluate to true. Signaling an exception is to draw attention to the event of the detection of an exception. Exceptions are signaled in two ways: implicitly by the system if a system-defined exception has occurred, or explicitly through the use of a SIGNAL statement to signal a programmer-defined exception.

An exception handling mechanism is often embedded in and integrated with a programming language which is used to write or implement the software component. The features and constructs of such a language can be used to examine the occurrence of an exception. For example, the IF statement can be used to detect that a stack underflow condition has occurred by testing if the top of the stack is 0 as follows:

\[
\text{IF TOP}=0 \text{ THEN SIGNAL(STACK-UNDERFLOW)}
\]

Signaling an exception has a semantic similar to calling a procedure.
The procedure that is called in case of an exception is the Handler that provides for treating the exception. Similarly, signaling an exception has syntax similar to calling a procedure. It is done by specifying the exception identifier and then specifying a parameter list of the actual parameters that are passed to the Handler. Such parameters must have the correct types which are expected by the Handler. An example is shown below:

\[
\text{IF } X < \text{MAY THEN}
\]
\[
\text{SIGNAL OUT-OF-BOUND}(X)
\]

The contest in which an exception is detected may not be the best context in which it should be handled, thus an exception is given a name and is passed to the context where it can be processed. This action is called an exception propagation. The actions of detecting an exception and processing it may be performed in two distinct Modules. Signaling an exception may cause the flow of information from one Module to another. It is possible that the recipient of an exception handles it by signaling another exception to its invoker.

Each exception to be handled by a client must be expressed in terms of the client Module's abstraction, and cannot simply restate the condition signalled by the original server Module detecting the exception. The implicit propagation of an unhandled exception will violate the Module abstraction by revealing information about the implementation of the Module. Using the stack example, we can demonstrate this fact. A POP operation of a Module implementing a stack can have an underflow exception. However, if the stack is implemented using an array and then if the exception SUBSCRIPT-OUT-OF-RANGE is propagated to the client Module of the POP operation, this reveals to the client Module
the fact that the stack is implemented using an array. Furthermore, if the client Module is to handle such an exception, it must know the details of the implementation of the server Module's operations, thus violating the Module abstraction and information hiding. An exception handling mechanism should not support implicit exception propagation but forces the programmer to explicitly rename the exception and propagate it.

8.1.3. Handlers

When an exception is raised within a software component, an associated software fragment called a Handler is invoked. The Handler provides measures to deal with the signalled exception. It may examine some past parameters associated with the exception to accomplish its task. As indicated previously a Handler can be thought of as a procedure that is invoked when an exception is signalled.

Different Handlers may be associated with one exception in different contexts; thus, an exception must be explicitly associated with the appropriate Handler for a particular context. Two approaches may be used: STATIC association and DYNAMIC association.

In STATIC association, Handlers are associated with exceptions in a particular context using the static structure of the software component. A Handler H is designated for invocation in case of raising an exception E by attaching an ON clause containing the designation of Handler H for exception E to some closed construct of the software component (e.g., block, loop, procedure, etc.). If the exception is signalled within the closed construct, then the Handler designated for the exception in the ON clause outside the construct is used. Figure 29 shows how an ON
BEGIN;
  PUSII(S,X);
  POP(S)
BEGIN;
  PUSH(S,Y)
  POP(S)
END;
ON STACK-UNDER-FLOW = H1
  .
  .
END;
ON STACK-UNDER-FLOW = H2

Figure 27: Static Association
clause can be used for static a Handler-exception association. In the figure H1 is associated with the STACK-UNDER-FLOW exception, if the exception is signalled by the POP operation in the innermost BEGIN block. H2 is used if the exception is signalled by the first POP operation in the outer most BEGIN block. Static association is used in ADA and CLU.

DYNAMIC association of a handler and an exception for a particular context is determined dynamically by the flow control of the program execution. An ENABLE statement is used within the software code to create a handling context for a particular handler and associates the handler with an exception. The DISABLE statement terminates the context and the association of the handler and the exception. Dynamic association is used in the PL/1 programming language.

An example of creating and destroying a handler's context and handler-exception association is displayed in Figure 30. In the figure showing nesting of contexts, the first ENABLE (E2=H2) creates the context for H2 and associates it with exception E2. The second ENABLE creates the context for H4 and H3 and associates them with exceptions E2 and E3, respectively. It should be Noted that any occurrence of exception E2 will be handled by H4 rather than H2 after this point. The first DISABLE statement destroys the last context and the second destroys the first context and the respective associations.

A handler may be provided by the client Module to handle an exception signalled by the server Module. A group of handlers may also be associated with one exception. These handlers are provided by all the potential client Modules that "use" the server Module's operations. When an exception is signalled in the server Module, all related handlers
Figure 28: Dynamic Association
provided by the potential clients are given a chance to respond and to try to treat the exception. When the exception is treated, the Module raising the exception is given a chance to continue execution if possible. This approach is used in Levin's proposal.

A universal handler may be provided at the server Module to handle a given exception. This handler is invoked no matter which Module is requesting the server Module's operation raising the exception.

8.1.4. Control Transfer

After the Handler of an exception is successfully completed, the control flow may be transferred to different points depending on the exception handling model adapted; these models include the Resumption model, Termination model, Retry model, and Propagation model. These models are explained in turn next.

1. **Resumption Model**: The handler is executed, and then the execution is resumed for the Module's operation that raised the exception when it left off. Hence, the control flow returns to the server Module's operation. This model is adapted by Levin in his proposal and the exception handling mechanism is used by PL/1. The approach is based on the idea that handlers may eliminate the abnormal condition caused by the exception, and the server Module's operation raising the exception is given a chance to continue processing.

2. **Termination Model**: The handler is executed, the server Module's operation is terminated, and a substitute result provided by the handler is returned to the regular Module's
operation. Hence, control is returned to the client Module. This model is adapted by CLU, and is based on the idea that a handler provides an alternative service or algorithm to the server Module's operation.

3. **Retry Model**: Upon completion of the handler execution, the providing operation is started over from the beginning. This may be appropriate if some actions are taken by the handler to correct the cause of the exception, but the operation must be restarted to insure correct results. For example, if an input parameter is invalid and has caused an exception and some damage and the results of the Module are no longer usable if the operation is continued, and if modifying the input parameter can eliminate the exception, then the input parameter is modified and execution of the operation is restated. This mode also can be used if the service is rerequested in a different manner.

4. **Propagation Model**: Upon the completion of the handler, a successful completion message is sent to the invoker of the client Module. In this model the handler provides an alternative service to the service provided by the client Module receiving the exception. This model is appropriate when the exception in the server Module necessitates that the client Module take different action than it was designed for in order to successfully provide service to its requestor. When the handler completes, the server as well as the client Module are terminated, and control is transferred to the invoker of the client Module.
8.2. Server/Client Model

As was explained in Chapter 3, a number of recoverable Modules may interact and cooperate to implement a distributed application or a distribute subsystem. A number of exceptions and failures may be encountered during Module's interaction.

Figure 31 displays a model for the interaction between two Modules. A client Module requests some service of the server Module. Upon completion of the request, the server sends back a response. Such a response may indicate a normal completion of the request or may be in the form of an exception (abnormal completion). A service may not be completed due to permanent failure of the server node or due to a link failure.

The framework of the activity of the server Module is displayed in Figure 32. The server Module receives a request for service of one of the operation it provides. The request may be parameterized. In providing the requested service, the server Module may request the service of another Module.

A number of exceptions or failures can rise during normal Module cooperation activity. These exceptions are categorized as follows:

1. Interface Exception: If the request is submitted with invalid parameters or the request is not authentic, then the request is rejected and an Interface exception is signalled. This may happen because the client is trying to use a resource that it is not permitted to use.
Figure 29: Client/Server Model
Figure 30: Framework of Server Module Activities

INE: Interface Excp.
RNA: Return to Normal Activity
NRS: Normal Response
IE: Interaction Exception
RQ: Request
AE: Abnormal Exception
EC: Exception Handling or Recovery Component
AT: Acceptability test
EX: Exception Response
FE: Failure Exception
LE: Local Exception
AE: Acceptibility Exception
2. Interaction Exception: If the server Module is not available due to a server node failure or a link failure or unavailability of the requested Module due to withdrawal of the resource, then an Interaction exception is signaled.

3. Abnormal Results Exception: A reply message may be received from a remote server Module; however, such a response may be an exception that is detected by the server Module's operation, and thus is considered as an abnormal response. The exception may be system-defined exceptions, such as insufficient memory, division by zero, etc., or programmer defined exceptions. It is possible that such exceptions do not constitute a failure; however, these conditions are isolated to be treated separately for reasons such as program readability, and controlling complexity.

4. Failure Exception: If a server Module fails because it cannot tolerate a failure or exception it detected and no measures are provided to treat the exception is specified in the server Module, then a failure exception is signalled. It should be noted that an abnormal exception and a failure exception are similar. The only difference is that failure exception is signalled because an exception could not be handled in the server Module.

5. Acceptability Exception: Due to the post integration of the Module into a distributed subsystem, a normal result may be considered by the client as an exception. In such case an acceptability exception is signalled by the client.
During normal activity of a Module's operation, if the Module operation receives an exception due to invoking a remote Module or itself detects an exception, then a forward recovery component of the Module is invoked.

8.3. Exception Handling as a Forward Recovery in the Environment of Interest

In the environment of interest an exception handling mechanism is used to specify semantic knowledge provided by the Module Interface designer to recover from the possible exceptions and failures that can occur during Module interaction. It permits Modules to communicate detection and handling of failures and exceptions among themselves. The mechanism is designed to encourage the preservation of the abstraction and information hiding, that each Module provides.

A number of characteristics of the environment of interest strongly influence the design and implementation of the exception handling mechanism. In the remainder of this chapter we describe the client/server model on which our work is based and examine these characteristics and what restriction they put on the design of the mechanism. The components of the mechanism are presented with the appropriate adapted features in Section 8.4. Implementation of the components of the mechanism is presented in Section 8.5.
8.3.1. Characteristic of The Environment of Interest:

Four characteristics of the environment strongly influence the design of the exception handling mechanism: Modularity, incorporation of pre-existing Modules, incompatibility of the Module’s languages, and physical separation of the of the Module’s sites. These are discussed in turn next.

Modularity: In the environment of interest a number of Modules cooperate to implement a distributed application or a subsystem. Each Module implements a procedural or data abstraction and provides to or requires from other Modules. Each data abstraction provided by the Module can only be accessed through its operations which are defined in the Module interface specification. Hence, a Module provides for information hiding which to hide implementation details from its requesters.

These properties of the Module influence the design of the exception handling mechanism in the following ways: Exceptions propagated from a server Module must be expressed in the context of the client Module. The Module’s operations as independent software units may signal exceptions and handle such exceptions within their own context without informing the client Modules, or propagating the exceptions.

Incorporation of pre-existing Modules: A Module may be created as a stand alone, and at later time the Module may be incorporated with other Modules to build a subsystem or a distributed application. It is possible that such a Module is created to run on a single system or it may already be active as part of another distributed subsystem providing and requiring service from other Modules.
Introducing a Module to be used in a distributed system forces the Module to face and handle some distributed exceptions such as node failure, a link failure, etc. Being a member of a number of subsystems simultaneously, it is possible that some of the results provided by the Module are undesirable or considered exceptions by some client Modules in a particular subsystem. In other words, the Module may provide universal service where only part of the results of the Module may be desired by a client Module, and some results must be isolated or treated as exceptions. In both cases the Module must be modified or rewritten to handle such exceptions.

The property of incorporating pre-existing Modules, influence the design in the following way. Since it is the function of the MI to interface the Module to the distributed system, such exceptions are then declared and handled as part of the Module interface in order to avoid the cost of rewriting or modifying the Module.

**Module's Language Incompatibility:** Modules that cooperate to implement a distributed application may be written independently using different programming languages. The different languages may support different exception handling mechanisms which is used by the Module to signal and handle exceptions within the Module or to propagate the exceptions to the requester.

Due to the Module's language incompatibility, an exception signalled by a server Module using the exception handling mechanism of the language cannot be properly propagated and then handled by the client Module using a different exception handling mechanism. Exception handling mechanisms may be incompatible syntactically in the way exceptions are expressed, signalled, and handled. They may also be incompatible
semantically in the functionality provided by the exception handling mechanism, such as the type of control flow mode adapted, and handler-exception association.

This property influence the design of the mechanism in the following way. The exception handling mechanism used in this environment must be designed to support standard form exception handling features where exceptions signalled by a particular server Module using the Module's language exception handling mechanism is converted to an exception of the provided standard form. Handled in context, then, of the standard form, the exception is propagated to the client Module in the form of the exception handling mechanism supported by the language of the client Module.

**Physical Separation:** Different cooperating Modules may be running on different nodes that are physically separated and that communicate through a communication network. According to the Module's cooperation model presented in Chapter 3, a server process or an agent is created by the MI at the site of the server Module to invoke the required service of the Module on behalf of the client Module. Hence, exceptions signalled by the Module's operations must be handled by the invoking agent, which may provide handlers to such exceptions, and in turn propagates such exceptions to the client remote Module.

This implies that an exception handling mechanism for the environment must support locating and invoking the agent's handlers on the server Module's site, and support propagation of such exceptions to the client Module.
8.4. The Proposed Mechanism

The main goal of the mechanism is to support exception detection and handling among cooperation Modules implementing a specific distributed application to facilitate forward recovery. We assume that detecting and propagating exceptions within a Module operation code is done using the exception handling mechanism facility provided by the host language. For the purpose of discussion, exceptions are divided into two types, local exceptions and distributed exceptions. Local exceptions are the ones that are signalled within the Module's operation code, such as abnormal response exceptions. Distributed exceptions are the ones that are detected and signaled in the Module interface, such as interface exceptions and interaction exceptions.

8.4.1. Exception Declaration

A Module is considered to be an abstract unit where the possible normal and abnormal (exceptional) behavior are declared as part of the Module interface. Exceptions are declared in the specification section of the Module's interface as well as the possible operations provided by the Modules and operations required by the Module's operations. Figure 33 illustrates the declaration of possible exceptions signalled by a Module implementing a stack abstraction. The SIGNALS statement come after each Module's operation and has the following general form:

```
SIGNALS exception-name(param-list) [(condition)]
```

Exception-name is the identifier to name the exception that is provided by the interface programmer. A list of associated parameters are declared as part of the exception. Condition is an optional logical expression which is expressed in terms of the input and/or the output of
the Module operation. This specifies a condition for signaling the exception. This optional condition is used to detect programmer-defined distributed exceptions. If the condition is not specified, then it is assumed that the exception is signaled within the Module's operation code. Implicit system exceptions have predefined system names and need not be declared.

8.4.2. Exception Detection, Signaling, and Propagation

As illustrated in figure 30, a number of exception types may be detected and signaled by a server Module. These exceptions include: interface exceptions, failure exceptions, abnormal Module operation exceptions, acceptability exceptions, and interaction exceptions. Exceptions may be detected within the Module's operation code (local exceptions) or detected by the Module's interface (distributed exceptions). System defined failures are detected and signaled by the system.

Local exceptions may be detected within the Module's operation code using the language constructs and the exception handling facility embedded within the host language, such as PL/1, CLU, Ada, etc. Such exceptions may be handled within the Module's operation code itself or it is propagated to the Module interface. In case the exception is propagated, the MI intercepts the exception. If the exception is declared in the Module interface as an expected exception from the Module operation, then the exception is transformed into a distributed exception by signaling a distributed exception. If the exception is not declared in the Module interface, then it is signaled as a failure exception to the client Module. Thus, exceptions will only be propagated in the proper context which is the context of the client Module.
Figure 31: Declaration of Exceptions Signalled by a Stack Module

```
... PROVIDES
  PUSH(S,X)
  SIGNALS(STACK.OVERFLOW)

  POP(S)
  SIGNALS(STACK.UNDERFLOW)
...
```
Distributed exceptions detected in the Module interface are detected by examining the logical expression associated with the exception name identifier declared in the Module interface specification. The logical expression which is expressed in terms of the input and output of the Module's operation are evaluated before and after the invocation of the Module's operations. If the logical expression evaluates to true, then the corresponding exception is signaled. However, a SIGNAL statement can be used to explicitly signal an exception in a handler in the Module interface as will be explained in the subsection on handlers. The SIGNAL statement has the following form:

\[
\text{SIGNAL(exception-name(param-list)}
\]

where exception-name is a declared programmer-defined exception identifier. The values of the parameters in the param-list associated with the exception may be all passed back to the invoker Module as part of signaling the exception.

8.4.3. Handlers

Handlers are provided by the invoker of the Module's operation raising an exception to recover from or to treat such exception. The handler provides for specifying semantic knowledge to achieve recovery from failure or an exception.

In associating a particular exception with a handler, we adopt the static association approach where the invoker of the Module's operation provides a handler of the Module's declared exceptions. The handler is syntactically part of the declared required operations in the Module's interface specification as will be explained later.
Since the Module interface of a server Module invokes the server Module's operations on behalf of a remote Module, it is considered an invoker that may provide handlers for exceptions signalled within the Module's operations. The handler can be one of the following cases: A UNIVERSAL handler is a handler that is provided in the MI of the server Module for a particular distributed or local exception. It is used to handle these exceptions regardless of which remote Module is invoking the Module's operation. A REMOTE MI handler is provided in the client's MI as part of the Module interface specification. A REMOTE handler is provided within the code of the requiring operation in the client Module.

In seeking a handler for a signalled exception the following steps are taken:

1. If the exception is signalled within the code of the Module's operation, then a Handler is sought in the code of the Module operation using the local exception handling mechanism embedded in the language. If the handler is not provided in the operation and is meant to be propagated to the invoker, then a universal handler in the MI of the invoked Module is sought. If the handler is not provided in the MI, then the exception is propagated to the client Module, provided that it is a declared exception in the Module interface. Otherwise, a failure exception is signalled to the client Module.

2. If the exception is signalled in the MI, then a Universal Handler is sought in the MI; otherwise, it is propagated to the invoker, provided that the exception is declared in the MI specification.
3. When an exception is received at the invoking Module, a handler is sought in the MI. A handler in the MI may propagate down the exception to be handled by the invoking operation exception handling mechanism.

**Handler Specification:** We adapt the handler specification features of CLU. An EXCEPT-WHEN statement is used to specify a handler to handle a given exception. The general form of the EXCEPT-WHEN is shown in Figure 34.

All the handler to expected exceptions associated with the invocation of a remote operation (operation-name) come after the declaration of the required Module's operation in the MI specification. The EXCEPT statement specifies which handlers are associated with which expected exception-name. If, when a signaled exception is received, the exception name is equivalent to an exception name in a WHEN statement, then the associated handler is invoked to handle the exception. If a condition is associated with the WHEN statement, then that condition is evaluated. If it evaluates to true, then the associated handler is executed. If the signaled exception is not specified in any of the WHEN statements or if no specified condition evaluates to true, then the OTHER-WISE clause is executed. The OTHER-WISE clause may be used to propagate a failure exception.

8.4.4. Transfer of Control

After the execution of a handler, flow control may be transferred according to one of the four models, Continuation, Termination, Propagation, and Retrying. These models were discussed in Subsection 8.1.4. Most exception handling mechanisms adapt one of the four
INVOKED | OPERATION(param-list) | EXCEPT-WHEN(exception-name1 ! condition)

    H1 handler-body

EXCEPT-WHEN(exception-name2)

    H2 handler-body

OTHER-WISE

    H3 handler-body

**Figure 32:** Exception Handlers Declaration
models. For most flexibility in handling exceptions in the environment of interest, all these models are adapted. Flow control is transferred explicitly using one of the following statements within the handler code, RESUME, RETRY, REPLACE, RETURN:

1. **RESUME**: When a RESUME statement is used, the execution of the interrupted Module operation raising the exception is continued. This model assumes that the handler has corrected the condition causing the exception and execution to continue. For example, the exception may be insufficient memory and the handler may invoke a Garbage Collection routine. Once the handler is successfully executed, then the execution of the interrupted Module may continue.

2. **RETRY**: A RETRY statement may be used in the body of the Handler to start the interrupted operation over. This will cause the Termination of the previous invocation of the operation and the reinvokation of the operation. This model assumes that some of the provided input to the Module may have been incorrect, and such input is replaced in the handler by a valid input, and the operation must be restarted.

3. **REPLACE**: The REPLACE statement is used in the handler to capture the functionality of the termination model, which is to terminate the Module's operation and return control to the client Module's operation after the execution of the handler. This is appropriate when the handler has provided alternative service or completed the service the server Module started. Thus, the handler has replaced the requested Module's operation.
4. RETURN: The SIGNAL statement is used to propagate the exception to the invoker of the client Module. The handler replaces the service of the server and the client Module’s operations. Both of the server and client Module’s operations are terminated, and control is transferred to the invoker of the client Module. This model is appropriate when an exception is converted from the context where it is signalled to the context of the invoking Module. For example, if the invoked operation is a POP operation of a stack Module which signal the exception OVER-FLOW, a handler provided by the invoker Module, which is converting a infix expression to a post fix expression, may signal a different exception to the invoker of the conversion Module, such as ILLEGAL-EXPRESSION.

8.5. Implementation Of The Exceptions Handling Primitives

In this section we discuss the implementation of the proposed exception handling mechanism. We attempt to answer the questions of how to implement the detection of exceptions, the signaling of exceptions, the allocation of the associated handler for the exception, the execution of the handler, and transferring control according the specified control transfer primitive.

There are a number of factors to consider when implementing the primitives of the proposed exception handling mechanism. First, Local exceptions are signalled using the Module’s language exception handling mechanism in the server Module, and may need to be handled in the client Module’s operation using a different exception handling mechanism. Thus, such exceptions have to be intercepted and converted to a

---
distributed exception, then converted back to an exception in the client’s operation code. This supports signaling exceptions from one environment to another. Second, because of the physical separation of cooperating Modules, exceptions and handlers need to be communicated between the server and the client Modules using the provided communication primitives, such as SEND and RECEIVE. Third, modification and rewriting of the Module’s operation is at best avoided, and hence distributed exceptions are declared and detected, and may be handled in the Module interface without changing the Module’s operation code.

Besides allowing for signaling local exceptions in the Modules’ operation code in addition to handling and converting the local exceptions in the MI, the mechanism allows for declaring, detecting, signaling, and handling distributed exceptions in the MI. In the following section we discuss the structure and the implementation of the MI that supports implementation of the functions of the exception handling primitives.

8.5.1. Module Interface Structure and Implementation

Figure 35 shows the implementation relation between the Module and its MI. An MI process is created to execute the MI code. Upon receiving a request for service for a Module’s operation, the MI process creates a server process. The server process is responsible for executing part of the MI code that relates to examining and detecting distributed programmer defined exceptions. In addition, the server process is responsible for the invocation and execution of the requested operation. The MI code as it relates to exception detection and operation invocation is generated from the MI specification and compiled and linked with the Module’s operation. This code is generated in the same language used
to write the Module's operation to allow for the invocation of the Module's operation from the MI using a PROCEDURE CALL.

The MI code executed by the server process consists of four sections as shown in Figure 36, the Provide section, the Require section, the Exceptions section, and the Operations section. Each section may contain one or more procedures which are executed as an atomic procedure that is either executed successfully to completion or not at all. The function of these procedures in each section are explained as follows:

1. **Provide Section**: This section contains a number of procedures, each procedure being responsible for accessing a provided operation. The function of this procedure is to examine the authenticity of the received request, to examine the exceptions conditions before the operation invocation, to examine exceptional conditions after operation invocation, and to send results to the requester.

2. **Require Section**: This section contains a number of procedures, each for invoking remote required operations that are required by the Module's operation. These procedures are called by the Module's operation to perform a remote access to a desired remote operation. Each procedure is responsible for sending the request to the server Module, receiving a reply, performing an acceptability test of the results, and providing handlers for the possible exceptions signalled by the server Module's operation. In addition, it is responsible for invoking such handlers and implementing the control transfer control primitives.
Figure 33: Implementation Relation of Module and Module Interface
3. Exceptions Section: In case a local exception is signalled within the Module's operation code within a distributed Handler in the Provide section, or in a handler in the Require section, the EXCEPTIONS procedure in this section is invoked. The function of the procedure is to communicate the exception that is signalled to either a Universal handler in the EXCEPTIONS section or to a remote client Module. In case an exception is communicated to a remote client, the procedure waits for a reply from the remote client, and either returns control to the operation that signalled the exception, or terminates the operation if the remote client chooses not to continue the requested operation.

The implementing procedure for requesting remote operations and the procedure for providing local operations in two separate procedures are used to avoid possible recursion which may not be supported by the programming language used to implement the Module interface code. For example, if the PROVIDE procedure invokes a Module's operation, and the Module's operation requests a remote operation by invoking the REQUIRE procedure and then if both REQUIRE and PROVIDE were implemented as one procedure, then recursion is needed. The same recursion can occur if the EXCEPTION procedure was part of the PROVIDE procedure (e.g., PROVIDE invokes an operation which signals an exception which invokes the EXCEPTIONS procedure).
OPR1: PROCEDURE (ParmList);

SEND_REQ: SEND(TO: PROVIDER, Operation, ParmList);
RECEIVE_RES: RECEIVE(FROM: PROVIDER, ParmList, action);
IF (Action = 'NORMAL_RETURN') THEN RETURN;
ELSE DO;
IF WHEN_COND = TRUE THEN
CALL HANDLER1 (ParmList, Postcontrol)
ELSE IF WHEN_COND = TRUE THEN
CALL HANDLER2 (ParmList, Postcontrol)
ELSE CALL OTHERWISE (ParmList, Postcontrol);
END;
IF Postcontrol = 'RETRY' THEN DO;
SEND(TO: PROVIDER, 'TERMINATE');
GO TO SEND_REQ; END;
IF Postcontrol = 'REPLACE' THEN DO;
SEND(TO: PROVIDER, 'TERMINATE');
RETURN; END;
IF Postcontrol = 'RETURN' THEN DO;
SEND(TO: REQUIRER, ParmList, 'NORMAL_RETURN');
SEND(TO: PROVIDER, 'TERMINATE');
TERMINATE_OPERATION; END;
IF Postcontrol = 'RESUME' THEN DO;
SEND(TO: PROVIDER, 'RESUME');
GO TO receive_res; END;

HANDLER1: PROC(ParmList, Postcontrol);
- /* fix up done, want to continue provider */
- Postcontrol = 'RESUME';
RETURN; END;

HANDLER2: PROC(ParmList, Postcontrol);
- /* this take the place of the provider */
- Postcontrol = 'REPLACE';
RETURN; END;

OTHERWISE: PROC(ParmList, Postcontrol);
SIGNAL(FAILURE_EXCEPTION);
RETURN('UNHANDLER_EXCEPTION');
END;

Figure 34: Module's Interface Code Sections
8.5.2. Primitives Implementation

In view of the implementation of the MI's PROVIDE, REQUIRE, and EXCEPTIONS sections, how are the exception handling primitives implemented? In this subsection we discuss the implementation of exceptions detection, exception signaling, handlers, and control transfer primitives (RESUME, RETRY, RETURN, and REPLACE).

Exceptions Detection: Exception are detected by either satisfying the declared exception condition, or by the execution of an explicitly specified SIGNAL statement. Exceptions conditions may involve either Input or Output arguments or both. The signaling of an exception may be done within the Module’s operation using a local signal statement or in the MI using the SIGNAL statement. Figure 37 shows the implementation of the detection of possible exceptions which are explained as follows:

1. Exceptions conditions associated with an operation invocation are checked by the PROVIDE procedure before and after the invocation of the associated Module’s operation in the provide section of the MI. If the condition is satisfied, then the exception-name associated with the condition is signaled using a SIGNAL statement.

2. If an explicit SIGNAL statement is used in the MI to signal an exception, then no special action is taken.

3. If the exception is signalled in the Module’s operation using a local language signal statement, a handler for that exception is provided, as required by the host (local) language, in the PROVIDE section as part of the PROVIDE procedure. Since
Figure 35: Implementation of Exception Detection
the PROVIDE procedure is the invoker of the operation when the exception is then signalled, an associated provided handler that is provided by the invoker is sought and invoked. The handler contains no code except for a SIGNAL statement to signal a distributed exception that corresponds to the signalled local exception. Thus, local exceptions are trapped and converted to distributed exceptions that are communicated either to the remote client or to a local Universal handler provided in the EXCEPTIONS section.

4. An acceptability exception may be signalled by the REQUIRE procedure in the client MI if the received normal results do not conform to the normal behavior of the client Module. This is done by testing the exception condition provided in each WHEN statement in Figure 35.

**Signaling an Exception:** The SIGNAL statement is used to signal exceptions. The SIGNAL statement is implemented as a PROCEDURE-CALL to the EXCEPTIONS procedure in the EXCEPTIONS section. The reason behind implementing the SIGNAL statement as a PROCEDURE-CALL so that once the EXCEPTIONS procedure communicates the exception and the decision is made to continue the execution of the operation is that a simple RETURN statement from the EXCEPTIONS procedure will then bring control back to the operation that signalled the exception.

The statement:

```
SIGNAL(EXCEPTION-NAME (param-list))
```
is implemented as

CALL EXCEPTIONS(EXCEPTION-NAME,param-list)

The EXCEPTIONS procedure is responsible for locating the appropriate handler as follows:

1. If the handler sought is a Universal handler provided by the server MI, then this handler is provided in the EXCEPTIONS section and is invoked.

2. If no Universal (provided by MI) handler is provided, then an exception message is sent to the client and waits for an answer from the client. The possible answer is either to CONTINUE or TERMINATE the required operation.

   a. if the answer is to CONTINUE then a simple RETURN statement is used to return control to the statement which comes after the SIGNAL statement in the code section or the handler that signalled the exception in the PROVIDE section, which in turn will return control back to the operation that signalled the exception.

   b. if the answer is to TERMINATE, then the provided operation is ABORTED.

Figure 38 displays the implementation of the EXCEPTIONS procedure. It should be note that to be able to continue the execution of the operation, the local exception handling mechanism must provide for the continuation model. If that is not the case, then the SIGNAL statement can be used within the Module's operation code to signal a distributed exception.
EXCEPTIONS:

/* check if a universal handler is provided */
IF ExceptionName = LOCAL_ExceptionName1 THEN
  LOCAL_HANDLER1 (ParmList, REPLY)
ELSE IF ExceptionName = LOCAL_ExceptionName2 THEN
  LOCAL_HANDLER2 (ParmList, REPLY)
ELSE
  BEGIN
    SEND (ExceptionName, ParamList)
    RECEIVE (REPLY)
    IF REPLY = 'RESUME' THEN
      RETURN
    TERMINATE
  END

Figure 36: Implementation of Exceptions Procedure
Seeking Handlers: Once an exception is signalled a handler is sought. One of the possible handlers may be located which includes: the UNIVERSAL handler, the REMOTE MI handler, the remote RO handlers. The code of the UNIVERSAL handler is included as part of the EXCEPTIONS procedure in the server MI and is invoked by the EXCEPTIONS procedure in case of an exception. The code of the REMOTE MI handlers are included in the procedures that are part of the REQUIRING section of the client Module and are associated with the appropriate required operation 39. The REMOTE RO handler is provided in the requiring operation of the client Module, and it is provided in anticipation of an exception to be handled within the operation code. To be able to invoke a REMOTE RO handler, a corresponding REMOTE MI handler must be specified for the exception that is to be handled by the REMOTE RO handler. The REMOTE MI handler will propagate down the exception by signaling the exception in the handler code. Thus, the exception is converted from a distributed exception to a local exception.

When seeking a handler for a particular signalled exception, the following function are performed.

1. When a local or distributed exception is signalled in the server Module, the EXCEPTIONS procedure is ultimately invoked. The EXCEPTIONS procedure determines if there is a corresponding UNIVERSAL handler for the exception. If these handlers are located, then they are invoked. Otherwise, an exception message is sent to the client.
2. The exception message is received by a REQUIRE procedure in the REQUIRE section that initiated the request. A REMOTE MI handler is sought for the exception by examining if the exception-name is specified to be handled by the client's operation. If a handler is located, then it is invoked. Otherwise, the handler provided in the OTHERWISE section is invoked. If the handler is provided to trap an exception to be handled by a REMOTE RO handler within the requiring operation code, then a signal statement provided by the exception handling mechanism of the host language is used to signal the exception and thus invoke the local handler.

3. When a normal response is received by the REQUIRE procedure, an acceptability test is performed in and explicit exception condition is specified as part of the WHEN statement.

4. After a handler is executed, control is transferred back to the PROVIDE procedure in the PROVIDE section (except when the exception is propagated again) which decides where to transfer control according to the transfer of the control primitive.

**Transfer Control Primitive:** Transfer of control primitives indicates to which procedure the control is transferred after the execution of the handler. The control transfer primitives are: REPLACE, RESUME, RETRY, and RETURN. The primitives are implemented as shown in Figure 39. The post CONTROL variable is assigned a value indicating
the type of control transfer to be performed, and the MESSAGE variable is assigned the value of the message to be transferred back to the code receiving the control. Depending on the type of control passed back to the REQUIRE procedure, the REQUIRE procedure may send a message to CONTINUE the requested operation, RETRY the requested operation, TERMINATE the requested operation and CONTINUE the client's operation, the TERMINATE server's and client's operations, and transfer control back to the invoker of the client's operation. In Figure 39 RECEIVE is a label of the receive statement in the REQUIRE procedure of the requested operation. In Figure 39 the AFTER-RECEIVE is the label of the statement after the RECEIVE statement to allow for the continuation of the local client's operation.

8.5.3. Summary of Implementation

To summarize the implementation of the exception handling features, a scenario of requesting a server Module's operation by a client operation is presented.

1. To request a remote server's operation, the Module's operation invokes the appropriate REQUIRE procedure in the REQUIRE section.

2. The REQUIRE procedure, using the SEND primitive, sends a request to the MI of the server Module and then executes a RECEIVE primitive to receive the reply.

3. The MI of the server Module invokes the appropriate PROVIDE procedure in the PROCEDURE section which
Next Req: RECEIVE (OPERATION, Pramlist)
    ON (LOCAL_EXCEPTION): H
    IF OPERATION = opr1 THEN
    BEGIN
      IF WHEN_CONDITION = True THEN
        SIGNAL (Exception_Name)
        ...
        CALL OPR1 (Pramlist)
    IF WHEN_CONDITION = True THEN
      SIGNAL (Exception_Name)
      ...
    H1: SIGNAL (LOCAL_EXCEPTION)

Figure 37: Implementation of Module's REQUIRE Section
examines all the possible declared exception conditions that are associated with the requested operation. If an exception is detected, then it is signaled. Otherwise, the requested Module's operation is invoked using a PROCEDURE-CALL. The exception conditions are examined again after the invocation, and if an exception is detected, then a SIGNAL statement is used. Handlers are provided for exceptions that are signaled within the operation code, and are associated with the PROCEDURE-CALL. These handlers contain a SIGNAL statement to signal the distributed exception that corresponds to the local exception signalled. If no exception is detected, then a reply is sent back to the requesting client's Module's REQUIRE procedure.

4. When an exception is signaled, the EXCEPTIONS procedure is invoked which tries to allocate a UNIVERSAL handler for the exception. If none is provided, then an exception message is sent back to the server MI using a SEND primitive and a RECEIVE message is executed to receive a reply. The reply may be to continue or terminate. In case of continue, a simple RETURN is used to return control to the signaler of the exception. In case of terminate, the requested operation is aborted.

5. The REQUIRE procedure in the client's MI receives the reply. If the reply indicates a normal response, then control is returned to the client's operation code. If the response is an exception, then a handler with the same exception name is sought. If it is found, then it is invoked. Otherwise, the OTHERWISE section handler is used.
6. A handler in the REQUIRE procedure may signal another exception using the host language to propagate the exception down to the operation code or use a SIGNAL statement to propagate the exception to the invoker of the client's operation. The handler terminates with one of the possible control transfer primitives to transfer control to the desired location.

8.6. Summary

In this chapter we have presented forward recovery measures in the form of an exception handling mechanism to complement backward recovery measures presented in the form of an atomic Module's operation and an atomic distributed transaction. The design of an exception handling in the environment of interest is influenced by a number of factors which include Modularity, host language incompatibility, physical separation, and post integration of the Modules into distributed sub-systems.

The mechanism is designed with more emphasis on controlling abnormal activities among Modules, rather than within a Module. The mechanism provides for declaring, detecting, and handling exceptions in the MI interface with minimum modification or rewriting of the Module's operation code. Exceptions are communicated from the server operation detecting an exception to the client Module handling the exception. Exceptions signalled within a Module's operation using the host language exceptions facility are converted to distributed exceptions to be handled by a provided distributed handler, or it is handled by a REMOTE handler provided by the client's operation. The proposed exception handling mechanism provides for all models for control transfer. Flow
control may be explicitly transferred to different locations using the primitives RESUME, RETRY, REPLACE, and RETURN within a provided handler. Finally, the implementation of the mechanism primitives were presented.
Chapter 9
CONCLUSIONS AND FUTURE WORK

We have presented a model and system structure to efficiently support reliable resource sharing among heterogeneous nodes in a distributed system. A number of mechanisms were designed to accomplish this task. In the following paragraphs we summarize the results of each chapter.

Chapter 2: To be able to study the effects of failure on resource sharing in the environment, a model of the environment of interest was presented. The model is based on the concept of a Module which is a data abstraction to capture the properties of a resource in the environment of interest. A number of Modules cooperate to implement a distributed application or a distributed resource. A number of distributed applications may run concurrently in the distributed environment. A failure model then was presented to capture the physical capabilities of the distributed components (nodes, links, storage, etc.) and to define their possible behavior. As a result, the effects of the faulty behavior of such components on the distributed application processing were identified, and the reliability needs of the environment were established. It was concluded that the reliability needs of the environment include: preserving the consistency of the individual Module's consistency constraints, and preserving the system state consistency, which may be violated due to a failure or the unsynchronized processing of distributed applications. In addition, the reliability needs included enhancing availability and minimizing the loss...
of healthy work lost due to a failure. A number of design goals to
efficiently provide for the recovery and synchronization needs were
presented. The most important design goals were: 1) to increase the
degree of concurrency among distributed computations through the use of
semantic knowledge, and an optimistic concurrency control mechanism, 2)
to provide for efficient recovery and enhance availability through the use
of recovery semantic knowledge, 3) to minimize overhead during normal
activity and reasonable overhead during recovery.

We feel that the distributed environment model and the reliability
model are realistic and capture the properties of the distributed
component as they exist in practice. We feel that our design goals and
recovery needs stated in this chapter were met.

Chapter 3: In this chapter we presented a comprehensive reliability
model that captures the reliability needs of the environment established
in Chapter 2 and a system structure. The reliability model is an
extension of the system model presented in the Chapter 2, and is based
on the concept of the Recoverable Module that is an extension of a
regular Module. The Recoverable Module is able to tolerate failures and
manipulate exceptions. It provides for specifying and using semantics to
synchronize the concurrent Module’s operations and to preserve the
Module’s consistency constraints. The Recoverable Module, a unit of
recovery, tolerates node and interaction failures and uses semantics to
enhance availability. A number of Recoverable Modules cooperate to
implement a reliable computation that is modeled as a distributed
transaction. The distributed transaction is atomic and is a unit of
synchronization, but unlike other models it is not a unit of recovery, in
the sense that a failure does not force the total distributed transaction to
abort. Multiple concurrent transactions are synchronized so that each transaction preserves the consistency constraints of each Module it encompasses and preserves the system state consistency.

The system reliability support was structured in three layers. The component layer which includes the reliability support to provide recoverable system components (recoverable node, recoverable storage, etc.) that models the reliable system components masks most failures and exceptions from the next layer. The Recoverable Module layer supports the construction of the Recoverable Modules. The Reliable Distributed Transaction layer provides the software support to aid in the reliable interaction of the Recoverable Modules and to support the atomicity and synchronization of distributed transactions.

We feel that the object based recovery model is much more efficient than transaction based models in providing for the construction of reliable distributed systems similar to the environment of interest. The Recoverable Module is a unit of recovery; thus, the effects of failure can be localized, eliminating orphans, saving healthy work, and not necessitating the aborting of the entire distributed transaction. In the model, Modules are defined as active objects, which unlike passive objects, assumed in the transaction based models, allows for defining and preserving the object's dependent consistency constraints. In addition, application dependent synchronization and recovery semantics can be specified to increase concurrency and achieve efficient recovery. The Model allows for the use of forward recovery in the form of an exception handling mechanism to deal with interaction failure, whereas transaction based models are based on backward recovery only and are not able to handle interaction exceptions to enhance availability.
Chapter 4: In this chapter a number of existing mechanisms were examined to determine their applicability to provide for the reliability properties specified in the model. It was concluded that using distributed atomic transactions or subtransactions as a unit of recovery or the combination of checkpointing and atomic subtransactions in Module recovery from node failure will not confine the effects of a node failure, causing the loss of healthy computations and the creation of orphans. A technique was proposed to support the Module recovery from a node failure that is based on the use of a simple recovery line and remote access lists to restore the Module's operations state, and the use of a multi-version technique to maintain and restore the Module's objects. The Module object management technique is also used in supporting the atomic properties of the Module's operations.

In providing for the synchronization needs of the environment specified in the model, it was concluded that techniques used in database systems and transaction based models, such as locking, timestamping, optimistic concurrency control, or traditional synchronization mechanisms, were inadequate for the following reasons. Locking techniques as they exist do not enforce the Module consistency constraints, do not allow for semantic knowledge specification to increase concurrency, enforce strong serializability as a condition of correctness, and unnecessarily delay the processing of transactions accessing a shared Module's object. Timestamps share the same disadvantages of locking besides enforcing a global ordering of transaction synchronization which might violate Module consistency constraints. The existing optimistic concurrency control mechanism, while avoiding unnecessarily delaying transaction executions, shares all other disadvantages of locking, in addition to the fact that the existing optimistic methods are based on time outs and unnecessarily
abort some distributed transactions. The traditional synchronization mechanisms (i.e., Monitors) cause the occurrence of the lost update and of retrieval anomalies. A novel synchronization mechanism that is based on combining reliable selectors and an optimistic concurrency control method is proposed. We believe that such a mechanism is most appropriate for providing for the synchronization needs and for avoiding all the disadvantages of other mechanisms.

To support the totality of the distributed atomic transactions, it was concluded that a nested commit protocol, as was proposed by Moss [33] and used by others [30, 27, 43], violates the Module’s modularity properties and enforces the use of a higher authority (other than the involved Modules in the transaction). The proposed commit protocol is part of the optimistic concurrency control mechanism and avoids such disadvantages.

An exception handling mechanism is used as a forward recovery technique for the Module’s recovery from an interaction failure and provides for the controlled Module’s interaction among the cooperation recoverable Modules.

Chapter 5: In this chapter a mechanism for supporting a recoverable node service was presented. The mechanism detects and isolates a non-transient crashed node, and supports node recovery. The mechanism is based on the watchdog technique to detect non-transient node failures.

To support the Module’s recovery from a node crash, three mechanisms were presented, the Module’s object management, the Module’s operation management, and a Recovery Management mechanism. The three mechanisms combined provide for localizing the effects of a node crash.
by restoring the Module's state to a consistent state existing before the crash from which the interrupted Module's operations may be continued. The object management mechanism uses a multi-version technique to support recovery and to support the optimistic concurrency control mechanism presented in Chapter 7. We feel that the three mechanisms are simple, efficient, and constitute a substantial part of the contribution of this work.

**Chapters 6 & 7:** In these two chapters a comprehensive synchronization mechanism to support the synchronization need specified in the model is presented. The mechanism is based on combining an extended traditional synchronization mechanism, namely, selectors with an optimistic concurrency control mechanism. Reliable selectors are used to synchronize the Module's operations and are used to specify the Module constraints and synchronization semantic knowledge to increase the degree of concurrency. Reliable Selectors are an extension of regular selectors that tolerate node failure and take advantage of the possible concurrency created by a specific implementation of the Module's objects. The Reliable selectors mechanism is coordinated with the global concurrency control mechanism. The Reliable Selectors provide for a specification based approach that captures the Module's synchronization semantics. We believe that such an approach is very powerful. Such an opinion is shared by [3], indicating that the discovery of a specification based approach is a considerable contribution.

In Chapter 7 a global concurrency control mechanism to synchronize distributed transactions was presented. The mechanism is based on an optimistic approach where transactions accessing the Module's objects are not synchronized except by the Module's scheduler to enforce the Module
consistency constraints. This approach eliminates the unnecessary transaction delays which are associated with the use of locking. The mechanism is coordinated with reliable selectors to satisfy the synchronization needs of the environment.

Unlike other existing optimistic concurrency control mechanisms used in database systems [9] which are based on time-outs to detect conflicting transactions and thus aborting a transaction unnecessarily, the proposed mechanism will only abort a transaction if it is involved in a conflict with other transactions. Aborting a transaction unnecessarily is very undesirable when an optimistic mechanism is used. An aborted transaction may cause a domino effect, causing the substantial loss of computations. In the proposed model we believe that the domino effect is substantially reduced for the following reasons: 1) conflicts are reduced by using semantic knowledge to increase concurrency among the Module's operations, and by putting a limit on the number of Module's operations that are dependent on each other; 2) transactions are not aborted due to node crash or interaction failures; 3) transactions are not unnecessarily aborted due to synchronization problems.

The synchronization mechanism conforms to the Object model rather than the transaction model in the sense that only the cooperating Modules accessed by a distributed transaction are involved in synchronizing the transaction with other concurrent transactions and no higher authority at each site is involved as in the case with the transaction managers in transaction based models. We believe that the synchronization mechanisms achieve our goal of improving efficiency by supporting a high degree of concurrency.

Chapter 8: In this chapter an exception handling mechanism to
support the Module's forward recovery and control interaction among cooperation Modules was presented. The design of the mechanism was influenced by a number of characteristics of the model and the distributed environment of interest. Such factors include Modularity, physical separation of the cooperating Modules, and incompatible programming languages used to implement the Module's operations. The mechanism is different from existing exception handling mechanisms in that it supports communicating, controlling, and handling exceptions among cooperation Modules that are physically separated. In addition, the mechanism adapts a transfer of control model that include all the proposed models (Termination, Resumption, Propagation) in addition to a proposed Retry model.

9.0.1. Summary of Contributions

In this dissertation we have presented a comprehensive design for software support to permit reliable resource sharing in the environment of interest. We have proposed a general reliability model that is based on active objects (Recoverable Modules), where the object is the unit of recovery using both backward and forward recovery. Only the group of Modules accessed by a transaction is responsible for the global synchronization and consistency of each Module, and no higher authority is required. A number of new mechanisms were proposed to support distributed processing in the model which included: a Module recovery and restoration mechanism that uses simple recovery line remote access lists, a version management technique, and a mechanism for detecting, isolating, and recovering from node crash. We have presented a comprehensive synchronization mechanism that provides for user specified semantics and that preserves global system state consistency. We have
presented an exception handling mechanism that captures the four models of control transfer and allows for specifying, detecting, and handling exception raising during Module interaction, hence supporting forward recovery.

9.0.2. Future Work

This work is by no means complete. There are a number of extensions and interesting areas for future investigation. Moreover, the mechanisms provided can be improved with experience gained in the area. We list below some areas of future research.

Experience and Implementation

While we have addressed a number of implementation issues throughout this work, a number of implementation questions still need to be addressed. The question of how to move versions of a Module's object between the permanent and volatile memory needs to be considered in more detail with stress on efficiency. In addition, versions are maintained as differential files. The procedure for merging a number of versions to one version the value of which is to be used to create another version needs to addressed.

In presenting reliable selectors, we have only addressed the specification of synchronization semantic knowledge, and we did not provide specific methods for implementing selectors, for example, how to represent the compatibility relation of the Module's operations, and how to examine and maintain the Module's synchronization constraints. Such questions need to addressed in detail.
In addressing the object version management, we have not addressed how to determine which version in a previous column the newly created version depends on. This must be done using the semantic synchronization knowledge specified in the selector to determine which previous operation the current operation conflicts with and thus use the previous operation's version to create the new version.

**Resource Replication:** A Module may be replicated over a number of nodes. The purpose of replication in the distributed system is to enhance availability and improve response time. However, this is done at the expense of restricting the degree of concurrency to achieve consistency. We proposed using alternative predefined Modules that are providing similar services rather than replication to enhance availability. This is performed using an exception handling mechanism. We feel that it is simpler and more appropriate to build replication on the top of our system so as to separate the issues of replication from the other problems of recovery and synchronization. This opinion is shared by Moss [33]. A number of issues have to be addressed which include naming the replicated object, maintaining the consistency of the different copies in case of update and across failure, specifying who controls the replication algorithm, and determining if it is possible to use a remote broadcast facility to update all copies of the replicated object. The interested reader is referenced to [3] for more details.

**Non-atomic Modules:** We have made the assumption that all modules accessed by the distributed transaction are maintained as permanent objects and thus all the Module's operations are atomic and the distributed transaction is atomic to maintain the consistency of the Module's objects. The question is, if such an assumption is relaxed and Modules may not maintain permanent objects, must such a Module
participate in supporting Module recovery, transaction management, and synchronization to guarantee the atomicity of the total distributed transaction? How may the mechanisms presented for recovery and synchronization be modified so as to guarantee that consistency of the Module's objects accessed by a non-atomic Module's request in the Modules cooperation access tree is still reserved?

In addition to the atomic Module assumption, we have assumed that all Module cooperation forms a tree-like hierarchy, called an access tree, where a child Module does not request the service of an ancestor Module, thus avoiding recursive access. In addition, ancestors are assumed to not share access to an object with their dependents concurrently. If such a mode of operations is desirable, then the object version management and Module's operation synchronization need to be modified.

Language Consideration: We have presented some language features such as specification of synchronization semantics, specification of Module recovery through the exception handling mechanism, and specification of Module interface that specifies the operations provided and required by a specific Module. However, our language features need substantial improvement.

Proof of Correctness: We have presented a proof of correctness for the optimistic global concurrency control algorithm; however, we have not formally proved as being correct our object management, operation management, recovery management of Chapter 5, and Module operation synchronization (Selectors) of Chapter 7. We have only argued the correctness of these mechanisms but offered no proofs. This an area of future work.
Bibliography


