A FAULT-TOLERANT COHERENCE PROTOCOL FOR DISTRIBUTED SHARED MEMORY SYSTEMS

By

PALLAVI K. RAMAM

Bachelor of Science
St. Francis College for Women
Hyderabad, India
1991

Master of Science
Indian Institute of Technology
Bombay, India
1993

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 1998

A FAULT-TOLERANT COHERENCE PROTOCOL FOR

DISTRIBUTED SHARED MEMORY SYSTEMS

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

D. E. Heisler

Blayne E. Mayfield

Wayne B. Powell

Dean of the Graduate College

PREFACE

Distributed Shared Memory (DSM) systems are becoming increasingly more significant as a result of being used more extensively in modern computing environments. DSM gives the illusion of shared memory on a loosely coupled system. In a scenario where systems are connected across a network, DSM coherence protocols should be able to scale well to larger networks. When real-time applications run on distributed systems, providing a high degree of reliability is an inherently error-prone environment is a formidable task. Regardless, fault tolerance, in terms of highly available data-access and uninterrupted service, should be provided. Recovery is the process of restoring a system to its normal operational state in the event of a failure. Reliability ensures the consistency of the data after recovery.

Existing DSM systems provide reliability by replicating data, either in stable storage or in the main memories of different processors. But these systems suspend the DSM service during recovery. In time-critical applications, providing *uninterrupted* DSM service to the greatest possible extent is a necessity and a challenge. Research has been reported in the literature on architectures with a single server and multiple clients. This thesis reports on investigations on finding a solution where the server is not a single point of failure, faster recovery is possible in the event of a failure, and increased throughput can be obtained during normal operation of the system.

It was found that better performance can be obtained by using the multi-server protocol when the user application exhibits locality of reference. The server was not a single point of failure and recovery from a single site failure was approximately 50% faster when 2 servers, instead of 1, were used.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my graduate advisor, Dr. Mansur H. Samadzadeh for his supervision, advise, and constructive guidance. I would especially like to thank him for motivating me and for giving me a sense of direction and focus when I most needed it. His insight and perspective on this research have given it its current shape. His exacting standards have improved the quality of this work. The sometimes torturous, yet satisfying process of this research has made me a better critic and researcher. I thank Dr. Samadzadeh for giving me this opportunity.

I would also like to thank Drs. Blayne E. Mayfield and G. E. Hedrick for serving on my committee. Working as a teaching assistant for them has increased my knowledge and given me a better perspective on the subject. I am also grateful to the Department for its generous financial support.

I thank Dr. David Koppelman, of the Department of Computer and Electrical Engineering at Louisiana State University, for making the Proteus simulator available for this research. His prompt and detailed replies to all my queries contributed greatly to my understanding of Proteus. His help expedited a laborious implementation process. I am amazed at, yet grateful for, his unparalleled generosity in his numerous offers to debug selected segments of my code, whenever I got myself into nasty trouble. I am especially grateful for his patience with me whenever problems arose.

I would like to take this opportunity to acknowledge the difference Venkat has made to my career. His insistence in not allowing me to start implementation until I understood all the issues paid off. The sound programming habits and the importance of design that he instilled in me have made me a better professional. I thank Shishir, Ram, Amith, Lee, Satish, and Kamalakar for hearing me out whenever I bounced ideas off of them. Crucial ideas and parts of this thesis gained shape after such sessions. I would like to remember my father, who would have loved to see me take the right direction. Finally, I thank my mother, who has been a constant source of encouragement and inspiration.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 4

2.1 Fault Tolerance and Recovery 4
2.2 Memory Coherence 6
2.3 Basic Model 8
2.3.1 Failure Model and Definitions 8
2.3.2 Functional Model of a DSM System 9
2.4 Related Work 11
2.4.1 Boundary-Restricted Coherence Protocol 11
2.4.2 Dynamic Boundary-Restricted Coherence Protocol 13
2.4.3 Reliable Mirage+ 14

III. RESEARCH GOALS AND SCOPE 17

3.1 Problem Formulation 17
3.2 A Possible Solution 19
3.3 Preliminary Idea 20
3.4 Research Scope and Limitations 22

IV. FAULT-TOLERANT PROTOCOL MODEL 24

4.1 System Architecture 24
4.2 Server Functionality 27
4.3 Client Functionality 28
4.4 Read and Write Operations 28
4.4.1 Read Operation 29
4.4.2 Write Operation 32
4.5 Migration 36
4.5.1 Migration Algorithm 36
4.5.2 Migration Process 37
4.5.3 Consensus Problem 41
4.5.4 Solution to the Consensus Problem 44
4.6 Recovery 46

Chapter Page

V. SIMULATION ENVIRONMENT 50

5.1 Why Simulate? 50
5.2 Simulation Environment 51
5.3 Configuration of the System in the Thesis 53

VI. IMPLEMENTATION ISSUES 55

6.1 Threads 55
6.2 Server, Client, and User States 58
6.2.1 Server States 58
6.2.2 Client States 61
6.2.3 User States 65
6.3 Synchronization 68
6.4 Signal Handling 69

VII. ASSUMPTIONS, INTERPRETATION AND OPTIMIZATIONS 74

7.1 Assumptions About the Simulated Environment 74
7.2 Differences in Interpretation 78
7.3 Optimizations 82

VIII. RESULTS 87

8.1 Basic Configuration 87
8.2 Experiments 91
8.2.1 Performance Analysis of Single Server Architecture versus 2 Server Architecture 91
8.2.2 Locality of Reference 96
8.2.3 Multiple Servers 98
8.2.4 Recovery 101
8.3 Limitations 105

IX. CONCLUSION AND FUTURE WORK 106

9.1 Conclusion 106
9.2 Advantages and Disadvantages of the Multi-Server Protocol 108
9.2.1 Advantages of the Multi-Server Protocol 108
9.2.2 Disadvantages of the Multi-Server Protocol 110
9.3 Future Work 111

REFERENCES 114
APPENDICES 119

Chapter Page

APPENDIX A - GLOSSARY 120

APPENDIX B - TRADEMARK INFORMATION 124

APPENDIX C - PROGRAM LISTING 125

LIST OF FIGURES

Figure Page

1. Out-of-Cluster Read Request........ 29
2. Out-of-Cluster Read Request, Reader in Cluster........ 31
3. Clock Site Invalidates In and Out Cluster Readers........ 32
4. Out-of-Cluster Write Request........ 34
5. Migration Process: Client Handing Over Clock Page........ 38
6. Server Hands Over Client to Another Server (Solution 1)........ 41
7. Server Hands Over Client to Another Server (Solution 2)........ 43
8. Server Hands Over Client to Another Server (Solution 3)........ 46
9. Client and Server Threads........ 56
10. Server State Diagram........ 59
11. Client State Diagram........ 62
12. User State Diagram........ 66
13. Change of Clock Site During Write Request........ 81
14. Out-Cluster-Reader Server Requests Write Page........ 84
15. Effect of 2 Servers on Average Request Latency........ 92
16. Effect of 2 Servers on Total Number of Requests........ 93

17. Effect of 2 Servers on Hit Percentage 94

18. Total Traffic Generated, Normalized to Single Server 95

19. Classification of Total Traffic Generated, as In-Cluster, Out-Cluster, and Migration Traffic 95

20. Effect of Locality of Reference on Total Number of Requests, Normalized to Single Server 97

21. Effect of Locality of Reference on Total Number of Migrations 97

22. Effect of Multiple Servers on Request Latency, Normalized to Single Server 99

23. Effect of Multiple Servers on Total Traffic Generated, Normalized to Single Server 100

24. Effect of Recovery Latency on Single Cluster of Different Configurations 104

LIST OF TABLES

Table Page

I. Basic System Configuration Parameters 92

II. Different System Configurations 96

CHAPTER I

INTRODUCTION

Multiple processor systems are being used more extensively nowadays and for quite some time studies have concentrated on increasing the performance and improving the reliability of such systems [Patterson and Hennessy 96]. A shared memory system on tightly coupled processors makes global physical memory equally accessible to all processors. Although shared memory provides ease of programming, it typically suffers from increased contention for a single bus, longer latencies in accessing the shared memory, and limited scalability [Protic et al. 96] [Kermarrec et al. 95] [Tanenbaum 95] [Mullender 93]. In the infancy of distributed computing, programs on machines that did not physically share memory ran in different address spaces, which led to the message passing model [Tanenbaum 95] [Singhal and Shivaratri 94].

A Distributed Shared Memory (DSM) system implements the shared memory model on a loosely coupled system. The words "shared memory" refer to the fact that the address space on different processors is shared. In a scenario where systems are increasingly connected across a network, DSM is a more attractive option than shared memory.

DSM provides a virtual address space shared among processors on loosely

coupled multi-processor systems. It gives the illusion of a shared memory system without the hardware bottlenecks and limited scalability. It hides the remote communication mechanism and the explicit data movement that must exist when processors connected across a network work together.

DSM coherence protocols should be able to scale well to larger networks [Theel and Fleisch 96a]. Although attractive, developing a DSM coherence protocol is not a trivial task and much research has focused on it [Nitzberg and Lo 91] [Lo 94] [Mohindra and Ramachandran 91] [Protic et al. 96]. Moreover, when critical or real-time applications run on distributed systems, a high degree of reliability is essential. Fault tolerance in terms of highly available data-access and *uninterrupted* DSM service is desirable. Addressing the issue of scalability, efforts to build large scale shared memory machines using small-scale to medium scale shared memory machines have been made [Erlichson et al. 96]. However, this work concluded that low latency networks may be required to achieve this goal.

DSM systems that address recoverability and consistency to provide reliable service have been developed. These systems provide reliable DSM by replicating data, either in stable storage [Koo and Toueg 87] [Wu and Fuchs 90] [Satyanarayanan et al. 94] [Feeley et al. 94] or in the main memories of different processors [Kermarrec et al. 95] [DeMatteis 96]. These mechanisms, to a large extent, guarantee that data can be recovered and is consistent before the system restarts. But they necessitate an interruption to the DSM service for recovery to take place. Systems which have tried to use a primary-backup approach to provide uninterrupted service, have had to dismiss the approach because of unacceptable overheads [DeMatteis 96].

Continuing in a different vein, most programs exhibit locality of reference. Even in a parallel programming model, the performance of a coherence protocol greatly depends on the data access patterns of the application running on the system [Eggers and Katz 88]. The pattern of data sharing is inherent in the application programs, and not caused by the underlying system architecture or the coherence protocol. The pattern of data access of applications was used as a motivation in this thesis work to design a DSM coherence protocol that can potentially increase throughput and, more importantly, reduce the recovery time in the event of a failure. This fault-tolerant protocol was simulated and its performance was compared with a simulation of an existing protocol.

The rest of this thesis is organized as follows. In Chapter II, a review of the literature is presented. Chapter III outlines the research goals and the scope of this research. Chapter IV describes the proposed fault-tolerant protocol model. An introduction to the simulation environment used for this thesis is given in Chapter V. Chapter VI describes a few key implementation issues. The underlying assumptions made during the implementation are outlined in Chapter VII. Chapter VIII describes the results and Chapter IX is the conclusion of this thesis.

CHAPTER II

LITERATURE REVIEW

2.1 Fault Tolerance and Recovery

A computing system consists of a number of hardware and software components that may potentially fail. Building a system with mechanisms that can avoid or tolerate failures is complicated. To make matters worse, a system usually does not always fail in the same way. This section introduces a few concepts about failure, fault tolerance, recovery, and reliability.

When the delivered service deviates from the specified service, a system *failure* occurs. The service delivered by a system is its behavior as perceived by its user(s); a "user" is another system that interacts with the former. An *error* is that part of the system which may lead to a failure. An error is caused by a *fault*. Therefore, *an error is a manifestation of a fault in the system and a failure is the effect of an error on the service* [Singhal and Shivaratri 94] [Avizienis and Laprie 86]. The time from the occurrence of an error to the resulting failure is called *error latency*. When an error causes a failure, it becomes effective. For example, a programming mistake is a fault that creates a *latent error* in the software. When the system executes the erroneous instructions with certain inputs, it causes a failure and the error becomes effective [Gray and Siewiorek 91].

Faulty behavior can be classified as a byzantine or a failstop failure [Schneider 90]. A *byzantine failure* is one where the faulty component can exhibit arbitrary and malicious behavior, with or without collusion with other faulty components. A *failstop failure* is one where the faulty component changes to a state that permits other components to detect the failure, and then stops.

Fault avoidance deals with prevention of fault-occurrence. *Fault tolerance* deals with how to provide service complying with the specification in spite of faults having occurred or occurring [Avizienis and Laprie 86]. In a fault-tolerant system, the standard specifications of a server should not only specify its failure-free semantics, but also its likely failure behavior (or failure semantics). As an example, error-correcting codes are used to provide fault tolerance.

A system is designed to be fault-tolerant in two ways [Cristian 91] [Singhal and Shivaratri 94]. A system may mask failures or it may present a well-defined failure behavior (or failure semantics) in the event of a failure. Adding redundancy, at both hardware and software levels, is a key approach to tolerate failures. This, however, is bound to increase the time and/or space overhead on the system during normal operation. The challenge lies in providing fault tolerance with minimal overhead during normal operation of a system.

The life of a system alternates between providing proper service and improper service. Proper service refers to delivering the specified service. Improper service is when the delivered service deviates from the specified service. According to Avizienis and Laprie [Avizienis and Laprie 86], reliability is "a measure of the continuous delivery of proper service (or equivalently, of the time to failure) from a reference initial instant".

In a DSM system, fault tolerance is provided by ensuring recoverability of data. *Recovery* refers to restoring a system to its normal operational state and is a complicated process. In other words, ensuring recovery of a system improves it reliability. A recoverable system must be able to survive any single-site failure. A reliable system also ensures the consistency of data after recovery.

2.2 Memory Coherence

A memory consistency model indicates how a programmer views the memory. The basic operations allowed on data are read and write. A memory consistency model restricts the values a read can return. A process running at a node or a site will expect a read to return the last value written. But in a DSM environment, a node may be reading stale data if the last write at another node has not yet propagated to this node due to communication latency. In a DSM system, there is no global notion of time that can provide a deterministic total ordering for all reads and writes [Lo 94]. To ensure that data at all nodes is coherent, a synchronization or control mechanism must be used. It is best elucidated in the words of Singhal, "a memory consistency model is a set of allowable memory access orderings and is enforced by the DSM system" [Singhal and Shivaratri 94]. "Coherence" is used as a general term for the semantics of memory operations and the word "consistency" refers to a specific kind of coherence.

Coherence semantics are guaranteed by a DSM server. They define the notion of correctness and *what* is guaranteed by a DSM system. Strict consistency, sequential consistency, processor consistency, weak consistency, and release consistency are a few

of the coherence semantics used in various DSM systems [Nitzberg and Lo 91]. The coherence semantics provided in the work in this thesis is sequential consistency. As defined by Lamport, a system is *sequentially consistent* if "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [Lamport 79].

A coherence protocol implements the coherence semantics; it defines *how* this notion of correctness is achieved. DSM systems allow concurrent access via data replication. Nodes or sites which access shared data have local copies of it. A coherence protocol keeps the replicas consistent. It ensures that all copies of the data have the same information and that nodes do not access stale data. A *coherence unit* is an abstract memory object that is guaranteed to be consistent.

Write Broadcast (WB) and Write Invalidate (WI) are two popular coherence protocols. A detailed comparison of WB and WI can be found in [Eggers and Katz 88].

- *Write Broadcast or Write Update*: A write to a shared data object causes all copies to be updated. Once a site has cached a copy of a shared object, that copy is never deleted. Thus, an arbitrary copy can be used for reading, but all copies must be modified as a part of a write operation.
- *Write Invalidate*: A write to a shared data object causes the invalidation of all copies except one before the write can proceed. Once invalidated, copies are no longer accessible.

2.3 Basic Model

This section introduces the basic model of Mirage+, a DSM system [Fleisch and Popek 89] [Theel and Fleisch 95] [Theel and Fleisch 96a] [Theel and Fleisch 96b]. In the following discussion it is assumed that sequential consistency is guaranteed for individual pages of a segment. Pages are chosen as coherence units because of their identical nature, small size, and the correspondence to the memory system supported by the processor's memory management unit. Providing robustness and consistency in a DSM system can be tackled using two approaches: a memory-oriented or an application-oriented approach [Juul and Fleisch 95]. In Mirage+, reliability is primarily examined from a memory-oriented perspective.

2.3.1 Failure Model and Definitions

A network environment has m sites, R_1, R_2, ..., R_m, where m is less than or equal to the total number of sites, all with independent failure rates, in the network. The sites are connected by communication links that may fail independently of each other. Each site may also fail independently of other sites. This can lead to the network being partitioned into subnets. A DSM service is made available to the sites in the model. The DSM system supports a paged segmentation scheme. It maintains one or more segments S_1, S_2, ..., S_s. A segment S_i, $1 \leq i \leq s$, consists of one or more pages P_1, P_2, ..., P_{n_j}.

A segment stores shared memory data. C is the set of all client sites and is a subset of the set of all sites $\{R_1, R_2, ..., R_m\}$. The number of client sites is n, where n $\leq$

m. S, a single site, is the server and belongs to the set $\{R_1, R_2, \ldots, R_m\}$. Only sites in C, the set of client sites, can send requests to the server S. The protocol specified [Theel and Fleisch 95] [Theel and Fleisch 96b] is to be used within a segment and the architecture is assumed to have a single non-replicated DSM server. Also, in the case of a server failure, a new server is selected [Theel and Fleisch 96b] using the standard election algorithms for distributed systems [Tanenbaum 95]. In the specified protocol [Theel and Fleisch 96b], no correlation between segments and sites is specified.

2.3.2 Functional Model of a DSM System

The main operations a DSM system must perform are the creation and destruction of segments, and read and write operations. Processes create shared memory segments by specifying the size of the segment, name, and access protection (read-only or read-write). Processes locate and map segments into their virtual memory address space. Once a segment is mapped, the shared memory behaves like conventional memory, the only difference being that changes to the underlying memory are visible to other local or remote processes sharing the segment [Fleisch et al. 94]. Processes can also share segments, that is, a process may have pages from more than one shared segment mapped into its memory.

A client reads (writes) a page if it is in local memory. If not, it submits a request to the server and blocks until it gets a result. The server services the requests sequentially by maintaining a request queue and directory information for pages in the network. The result sent back to the client contains the request data and a *mode* (explained below). If

the client gets the requested results, the operation is deemed successful. When the server cannot satisfy a request, it sends an error message to the client. Thus the client will not stay blocked forever if a request cannot be met. The primary aim is to provide high availability by servicing a majority of the requests successfully. The page received by the client is cached in the client's local memory.

The client uses the page according to the granted mode. The *mode* consists of a tuple: (read attribute, write attribute). If the client reads (writes) the data of a page cached with a mode having the local read (local write) attribute, then it is a *local read operation (local write operation)* carried out locally at the client site. If the page is not cached at all or cached with the global mode, the corresponding request has to be sent to the server, which, in turn, will perform a *global read operation (global write operation).*

It should be mentioned that the term 'library site' is used interchangeably with the term 'server' in the literature [Fleisch and Popek 89] [Theel and Fleisch 96a]. There is one library site per segment, and it is typically the site which creates the segment. The protocols specified are to be used within a segment and do not specify any kind of communication across segments, or between library sites of different segments. Moreover, no correlation between sites and segments is specified in the open literature.

The 'clock site' for a page is another distinguished site in the model. The clock site for a page has the most recent copy of a page. When a server gets a request for a page, it forwards the request to the corresponding clock site. The clock site, in turn, sends a reply to the requesting site. This decentralizes control, to some extent, from the server site to the clock site. However, it is possible that a clock and server are at the same site. Mirage+ [Fleisch and Popek 89] is implemented in the kernel of an existing

operating system. The memory coherence protocol used is Write Invalidate [Fleisch and Popek 89]. There may be multiple read copies of a page in the network simultaneously, but there may be only one write copy at any one time (single writer, multiple readers). So, if there are multiple readers of a page, one of the readers is a clock for that page. In the case of a writer of a page, the writer site is the clock site.

Mirage [Fleisch and Popek 89] attempts to control thrashing by using a clock mechanism. Readers or the current writer are given a time window (Δ) in which they are guaranteed to possess the page uninterruptedly. This time window provides some degree of control over processor locality. The clock site keeps track of this time window, which explains its name. When a clock site gets an invalidation message from the server, it checks to see if the time window for that page has expired. If not, it replies to the server with the amount of time the server must wait before resending the message. The clock handles a read/write request in different ways depending on whether it is a read clock or a write clock (refer to Reliable Mirage+ [DeMatteis96] for details).

2.4 Related Work

2.4.1 Boundary-Restricted Coherence Protocol

Theel and Fleisch [Theel and Fleisch 95] [Theel and Fleisch 96b] proposed a class of protocols called Boundary-Restricted (BR) coherence protocols which provide high data availability at low operation costs. These protocols also scale well. The motivation

for this class of protocols arises from the following goals for a protocol for a large, error-prone distributed environment.

1. *Limited Workload Dependability*: The number of copies of a page must, only to a certain degree, depend on the workload. This is to ensure that data availability and cost lie within specific boundaries. This is a limitation of both WB and WI coherence protocols.
2. *Lower Bound on the Number of Cached Copies*: Having only one copy of a page in the network should be avoided. A failure at that single site will lead to the unavailability of that page. This is a drawback of WI when there is only one write copy of a page in the system, where a write copy is a page in the write mode.
3. *Upper Bound on the Number of Cached Copies*: If all clients have a copy of a page, a global write operation will have to update all the copies. Since there are more error-prone components, the write operation may not be successful. This is a liability of the WB protocol when all the clients are readers.

The BR coherence protocol is defined by two parameters, BR(w, n): w the minimum number of copies of a page cached at client sites, and n the maximum number of client sites in the network. When a site writes to a page, there are w write copies of the page in the network. When a site reads from a page, the number of read copies varies from w+1 to n. Therefore, throughout the lifetime of a DSM page, the minimum number of copies available is w and the maximum is n.

Fault tolerance in terms of high availability of data is achieved by adding redundancy to the system. The higher the degree of redundancy, the more tolerant the system will be of malfunctioning components. Redundancy is introduced by increasing

the number of page copies or replicas cached at sites in the network. This adds a management overhead – multiple copies of a page must be kept up-to-date and clients must be prevented from reading outdated copies. This will increase costs.

The number of sites contacted by the DSM server during a read (write) operation is called a *read (write) operation cost*. *Data availability* is the probability that at any arbitrary point in time, data from a page is accessible. When updating all existing replicas, the number of up-to-date copies is constant and the availability stays the same. A write operation alters the number of copies, which will have a strong impact on the availability and costs of subsequent operations. There is a strong correlation between operation cost and data availability [Theel and Fleisch 96a] [Theel and Fleisch 96b]. Increasing data availability increases costs, while lowering the cost, in turn, will lower data availability.

2.4.2 Dynamic Boundary-Restricted Coherence Protocol

Theel and Fleisch [Theel and Fleisch 96a] also proposed a Dynamic Boundary-Restricted (DBR) coherence protocol class which was an extension to the 'static' BR coherence protocol class [Theel and Fleisch 95] [Theel and Fleisch 96b]. Its motivation was based on the facts that the workload of a DSM server (long-term workload) varies tremendously from application to application, and the workload submitted from a currently running application to the DSM server (short-term workload) is highly varied throughout the lifetime of the application. A DSM server using this protocol dynamically monitors the current workload and adjusts parameters to provide high data availability at

low costs. It does this by switching between instances of the Boundary-Restricted coherence protocol class described earlier in Section 2.4.1. These instances are ones where w, the number of write copies, is modified depending on the current workload. According to the authors, the DBR coherence protocol class outperforms the corresponding BR coherence protocol in terms of mean operations cost. However, the data availability provided by the two protocols is almost the same.

2.4.3 Reliable Mirage+

The stated goal of the BR and DBR protocols was to provide high data availability at low costs [Theel and Fleisch 96a]. There was one important issue that was not addressed by the authors. Though fault tolerance was one of the motivating factors for that research, *the server was a single point of failure*. The server was the only site that maintained a request queue and directory information for pages in the network. Moreover, when a client sent a request, it would block waiting for a reply. If a server could not service a request, it would send an error message to the client. But this would happen only when the server was functioning. If it was not, the client would obviously not be informed and would stay blocked forever. Also, if the only copy of a page was at a clock site and that site crashed, that page would be irrecoverable. Reliable Mirage+ [DeMatteis 96] ensures recovery from a single-site failstop failure.

DeMatteis [DeMatteis 96] proposed a way to recover from a server crash. Initially, having a 'shadow' server or a backup server was considered. All operations performed by the server had to be duplicated at the 'shadow' server site. This generated

too many additional messages and, according to DeMatteis [DeMatteis 96], caused a 30% overhead to the "normal operation of the preliminary system", which was dismissed as being too high.

DeMatteis then proposed an implementation of a reliable service called Reliable Mirage+ using the following key concepts (quoted from [DeMatteis 96]):

1. Processes retry when requesting shared pages after a designated amount of time.
2. Sequence numbers are used to eliminate duplicate requests.
3. Site failures are detected using existing AIX/TCF functionality.
4. State information is maintained throughout the sites that are accessing shared pages.
5. Recovery is possible by polling remaining sites to reconstruct the global state of the system.

Data availability was guaranteed by maintaining replicated copies of pages. The *trailer* site is the site that has the most recent invalid copy of the page. When a clock site invalidates its copy, it does not remove the page from memory. The read or write page is made a trailer page, an invalid copy which can be used during recovery. The trailer pages are given version numbers. During recovery, in case the clock site for a page fails, the site with the trailer page having the latest version number becomes the new clock site. Reliable Mirage+ recovers all lost data and makes sure that it is consistent (i.e., all pages have been recovered and all internal DSM system state information is accurate) before restarting the system.

Reliable Mirage+ is similar to recoverable DSM [Kermarrec et al. 95], which extends its coherence protocol to use replicated data as recovery data. Reliable Mirage+ gives details about how trailer information is used to recover the data. When a library site

fails, all sites must send information about pages in their local memories. The data is packed into bitmaps to reduce the number of messages that need to be sent [DeMatteis 96].

According to DeMatteis [DeMatteis 96], there are two methods of providing recoverable shared memory: eager recovery and lazy recovery (explained below). Once recovery starts, it is important that a backup for each page exists (or is created). The reason being that if a site which has the *only* copy of a page crashes or fails, then that page is irrevocably lost. A 'window of vulnerability' is from the time a failure occurs until every page has a backup copy at an alternate site. If a failure occurs during this time, the only copy of a page might be lost.

Eager recovery ensures that every page has a backup before the server resumes functioning. The system is brought to an entirely stable (i.e., the system can recover from another failure) and consistent state (i.e., all internal DSM state information is accurate) before start-up. In this situation, the processes waiting for pages have a higher latency, but the window of vulnerability is smaller.

In *lazy recovery*, the server begins satisfying requests as soon as one copy of every page is retrieved from all sites. A user-level process is run in the background which is responsible for installing backups of every page. Meanwhile, the server starts functioning. This results in a lower latency for processes, but leads to a larger window of vulnerability.

CHAPTER III

RESEARCH GOALS AND SCOPE

3.1 Problem Formulation

As mentioned earlier, in Section 2.4.1 in Chapter II, the Boundary-Restricted coherence protocol attempts to overcome the limitations of Write Broadcast and Write Invalidate coherence protocols in the context of providing reliability. The Dynamic Boundary-Restricted protocol (Section 2.4.2) takes advantage of the dynamic changes in the workload during the lifetime of an application.

WB and WI use replication to allow more than one process to share the same data. Replication reduces read latency, but it increases write latency due to the added expense of invalidating or updating all remote copies of the data. The problem becomes more complicated when a coherence protocol must not only perform well in terms of latency, but is also used to provide recoverability. Ironically, the goal of the BR and DBR protocols (see Sections 2.4.1 and 2.4.2 for a brief description) is to provide higher data availability, but they do not equip a system to handle a server failure. Reliable Mirage+ (see Section 2.4.3) [DeMatteis 96] ensured that the system could recover from a single-site, failstop failure, be the site a server or a client.

The BR and DBR protocols and Reliable Mirage+ deal with a single server, multiple clients architecture. The problem of a client sleeping forever (in the event of the server going down) in the BR protocol class [Theel and Fleisch 96b] was resolved in Reliable Mirage+ [DeMatteis 96]. The client sets a time out after sending every request. But a malfunctioning server will eventually lead to almost zero data availability. The only data available to the client will be the pages cached locally at the client site.

Fault tolerance was added to the Reliable Mirage+ system by adding redundancy to the system at client sites. In an environment with a single server, fault tolerance is added by having more than one copy of every page in the system. The number of copies of a page is optimized to minimize cost. Since the server is the focal point and all clients send requests to the server, a single server is still a *bottleneck* if the number of clients and/or requests is large. Even though recovery from a server crash was possible in Reliable Mirage+, the server was still a *single point of failure*. The fact that a system could recover was good enough, but the *whole* system would still be down for a while until the system recovered. In other existing systems that provide reliable DSM, recovery entails either reading from stable storage or suspending the system until recovery is complete. At the risk of belaboring a point, it must be reiterated that in time-critical applications, providing uninterrupted DSM service, to the greatest possible extent, is a necessity and a challenge.

The research goals of this thesis work were to investigate the feasibility, design, develop a prototype and evaluate the performance of a DSM coherence protocol exhibiting the following behavior.

- The server is not a single point of failure.
- Recovery faster than that in Reliable Mirage+ is possible in the event of a failure.
- Increased throughput over Reliable Mirage+ is possible during normal operation of the system.

3.2 A Possible Solution

The goal of the prototype system, where the server is not a single point of failure, with faster recovery and higher throughput than Reliable Mirage+, may be achieved by adding redundancy to the system (probably by adding more servers) without considerably increasing the overhead. A possible solution could be the following. The servers could be made to service requests concurrently, each keeping track of what the rest of the servers are doing. This would reduce the bottleneck caused by a single server. Moreover, if one server goes down, another can take its place almost immediately ('almost' because it would take some time to detect a failure at a site).

This solution may have its disadvantages. If the servers must operate concurrently and serve as backups for each other, every operation performed by one server must either be multicast to all other servers or be duplicated at least at one other server. And, presuming that more than one server is capable of doing the same operation on the same object, a suitable synchronization mechanism is required. Also, if a server goes down, a client must keep track of which server(s) to contact. The location of the servers would not be transparent to the clients, thus increasing the burden on the client sites. Location transparency can probably be provided by partitioning the data by address, using a

mapping function to determine which server(s) to contact based on the address being referenced.

The feasibility of this solution has not been examined. Even if this solution were feasible, it would not reduce the recovery time. The whole system would have to be suspended during recovery. Maintaining location information about multiple servers and synchronization between the servers during normal operation would probably cause a substantial overhead to the system and the trade-off obtained might not be worthwhile.

3.3 Preliminary Idea

Most sequential programs exhibit a high degree of locality of reference [Denning 72]. The performance of parallel programs in a DSM system depends on the number of parallel processes and the frequency of updating shared data. Specifically, it depends on the amount of contention for modifiable data. If the shared data is read-only, the simple solution of allowing multiple readers will suffice. When the shared data is read-write or write, the coherence protocol used (say WB or WI) and the data access pattern of the application become significant. The data sharing pattern can affect the number of invalidation messages or update messages for WI or WB, respectively.

The size of a page, the unit of sharing, can have far reaching implications on performance. The argument is outlined in the following three cases.

1. Consider a small page size, say the smallest theoretically possible, of one word: Access to every *new* word (one that has not been referenced earlier) can potentially

cause a page fault. This page size does not exploit locality. However, the system may perform well when data sharing is minimal.

2. Consider a large page size: The system performance will improve when locality is high. The page may have been brought in because of a reference to a word in it. The chance that the referenced words, or words on the same page, are referenced again is high. Such references will not generally cause additional page faults (that would in turn generate network traffic) since the page in question is already in the memory.

3. There is a counter-argument to Case 2. A larger page size can result in false sharing. *False sharing* occurs when different processes request the same page, but need to access exclusive sections of the shared page [Hyde and Fleisch 96]. This can result in thrashing, greater network traffic, and degraded performance.

The concept of segmentation is based upon a logical view of memory, rather than a physical view of memory (as in paging). In the traditional sense, memory is logically split up into segments based on *functionality.* Data segment, stack segment, the address space of each processor, and instruction segment are examples of segments [Silberschatz and Galvin 94]. Now consider the following scenario. The total set of *shared* pages is partitioned into segments, each being a mutually exclusive subset of the total pages (this is in keeping with the assumption in Mirage [Fleisch and Popek 89]). Assume that the segments are created such that the parallel processes that are running on the system exhibit locality of reference towards the pages in a single segment for a specific time period. Here, segmentation is done depending on *locality* and not functionality. This is called *temporal locality*, which states that recently accessed words are likely to be accessed in the near future [Patterson and Hennessy 96].

Returning to the previous train of thought, assume that each segment is kept consistent by a specific server. In the time period when processes exhibit greater temporal locality towards accessing a specific segment, these processes send requests to the server maintaining this particular segment. This would provide autonomy to the servers, not only in terms of servicing any requests for pages in their respective segments, but also in terms of recovery. The bottleneck caused by a single server would also be removed.

3.4 Research Scope and Limitations

The solution to the problem is based on Reliable Mirage+. It is designed as an extension to Reliable Mirage+. One limitation of the solution is an assumption about the locality of reference of applications as explained in Section 3.3. The analysis of data sharing patterns of specific parallel applications is beyond the scope of this research. The degree or effect of false sharing (see Section 3.3 or refer to [Hyde and Fleisch 96]) is not explored. The results are purely based on a synthetic application.

Reliable Mirage+ is implemented in the kernel of an operating system [DeMatteis 96]. It examines reliability from a memory-oriented perspective. In this work, a multi-server system was simulated and compared with a single-server simulation of Reliable Mirage+. It is impossible to simulate a real system with 100% accuracy. Differences in assumptions and interpretations are bound to occur, especially when the only information about Reliable Mirage+ that is available is through articles in the open literature.

However, a simulation is a reasonable representation of a system, and allows one to evaluate its feasibility and performance.

OKLAHOMA STATE UNIVERSITY

CHAPTER IV

FAULT-TOLERANT PROTOCOL MODEL

The preliminary approach to solving the problem was introduced in Section 3.3. This chapter delves into the details of the system. Throughout this chapter, it must be kept in mind that this system is built on top of Reliable Mirage+ [DeMatteis 96]. The potential deviations and the underlying assumptions are discussed in Chapter VII.

4.1 System Architecture

The following items describe the architecture of the new system.

1. The network environment has m sites with independent failure rates. The sites are connected by communication links that may fail independently of each other and of the other sites. Sites are designated to be server sites and/or client sites. Since a server site may also act as a client site, the set of servers *may* be a subset of the set of all clients.
2. A database is distributed among the m sites and is accessible by the clients.
3. The network is partitioned into n *clusters*, where $n \leq m$. Each cluster consists of one or more *sites*.
4. Each cluster is monitored by an exclusive server.

5. Clients are assigned to specific clusters.
6. Each cluster would have *approximately* $\lfloor m/n \rfloor$ clients or 1/n of the total number of clients.
7. A server will service requests sent by the clients in its own cluster as well as by clients in other clusters.
8. A page is chosen to be the coherence unit. The set of all pages is partitioned into n disjoint subsets, where n is the total number of servers which is the same as the total number of partitions.
9. A subset of the pages may not necessarily comprise of consecutive pages. The method used to partition the pages will depend on the application and will affect the contents of each subset of the pages. Subsets capture a logical view of data, where client processes exhibit locality of reference with respect to the subsets. Such subsets are called *segments.* A page cannot belong to more than one segment.
10. Each server is responsible for the consistency and maintenance of the current and backup copies of a segment. A server is said to 'own' these pages. Every server may not necessarily be responsible for the *same number* of pages. The load on a server depends on the size of the segment it maintains and the number of clients it monitors. Hence, every server may not necessarily have the same load.
11. The network is partitioned into clusters on the assumption that the needs of the clients in a cluster lean towards the segment owned by that cluster's server for a certain period of time. This is analogous to the concept of 'locality of reference' [Patterson and Hennessy 96]. In particular, it is 'temporal locality'. This would imply that a client would belong to a certain cluster and would send all requests to the server in

that cluster for a period of time. This does not necessarily mean that a client cannot have access to data in a different segment during this time period. A request for data in a different segment is handled by its server, which contacts the relevant server to get the data.

12. A client cannot directly contact a server other than its own. When a client needs data from a different segment, it sends a request to its own server. Its server contacts the segment's server, gets the requested page, and forwards it to the requesting client. Hence, it is possible to access data from a different cluster, but it involves the overhead of a longer route and extra messages.
13. If a client repeatedly accesses pages in a segment different from that owned by its server, it 'migrates' to that cluster. This migration is initiated by its server, in cooperation with other servers. The term 'migration', as used in this thesis, does not allude to the conventional concept of a *process* being moved from one processor to another [Mullender 93] [Tanenbaum 95]. It only means that a client is sending its requests to a different server, i.e., it is a part of a new cluster. A cluster, it must be emphasized, is only a logical abstraction and not a physical category.
14. A server is responsible for maintaining backups of the pages it 'owns'. These backups are maintained at other sites in its cluster. If a server goes down, it is responsible for recovering the pages it owns. Each server is autonomous with respect to servicing requests for pages in its segment, servicing the needs of the clients in its cluster, and for being responsible for the recovery of its segment and clients.
15. A server maintains information about other server site locations and the set of pages owned by every other server. This information is minimal, that is, it will require very

OKLAHOMA STATE U

little memory. This information and the information about the set of clients and pages in its segment are maintained in nonvolatile storage.

The terms page, server, client, cluster, segment, ownership of pages, and migration, whenever used in the rest of this thesis, refer to their definitions in this section.

4.2 Server Functionality

A server maintains a list of pages in its segment. The pages in its segment are allocated statically, once in the beginning, for each run of an application. The segment configuration does not change during the lifetime of the application. The clients in a server's cluster, however, are not statically allocated. Clients move dynamically from one cluster to another depending on their data access patterns. A server maintains a dynamic list of the clients in its cluster. A server has a page table with the following information about each page: the current clock site for the page, the mode (read/write) of the clock, the trailer site, and the latest trailer version.

If the page does not belong to its segment, the server's page table will reflect the existence or nonexistence of the page in its cluster. However, a server *must* have information about all pages belonging to its segment. A server has a request queue containing messages sent by its clients and by other servers. A server also maintains information about other servers. It stores the location and state of every other server along with the set of pages in its segment. A server also maintains migration data. Say a client of server S_x made a request for a page in segment s_y, belonging to server S_y. Upon getting a reply from server S_y, server S_x increments the number of requests made by this

particular client to server S_y. A server also maintains a list of read and write pages that is has 'loaned' to other clusters. The information about pages loaned to other clusters is needed when sending invalidation messages for these pages. The concept of 'loaning a page' is explained in Section 4.4.

4.3 Client Functionality

The clients' functionality is the same as in Reliable Mirage+. A client maintains a page table with information about whether a page is in memory, and if so, the page mode (read/write). The page table also has a flag set if the client is a clock site for a page. A clock site has the latest trailer version number and the list of readers of that page, if the clock is a read clock. A client also maintains a request queue, which is a queue of messages sent to this client.

4.4 Read and Write Operations

The mode of operations within a cluster are the same as in Reliable Mirage+. The coherence protocol used is WI. However, the coherence protocol for a request that has to go out of the cluster to be serviced depends on whether it is a read or a write request. For an out-of-cluster read operation, WI is the protocol. For an out-of-cluster write operation, a modified version of WB is used.

4.4.1 Read Operation

A read request by a client for a page belonging to its server is the same as in Reliable Mirage+. The client, on a read page fault, sends a request to its server. The server forwards it to the clock site. If the clock is a reader, it forwards the page to the requesting client. If the clock is a writer, it downgrades to a reader and forwards the page to the requesting client.

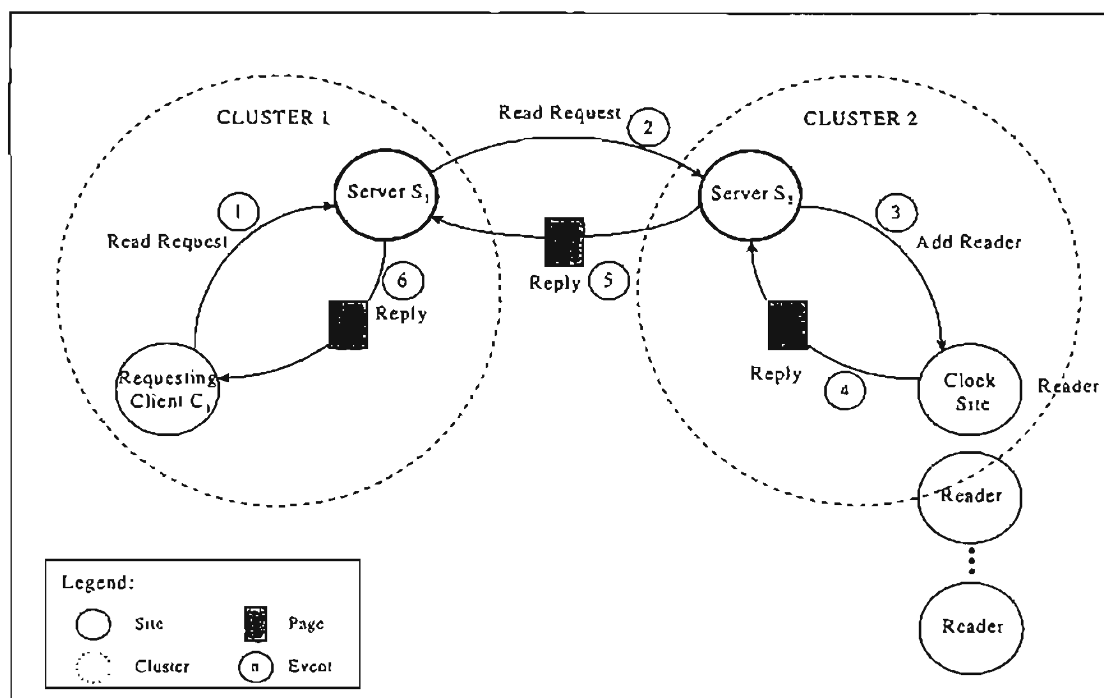

Figure 1. Out-of-Cluster Read Request

Figure 1 illustrates an out-of-cluster read request when the clock is a reader. The requesting client C_1 sends a read request to its server S_1. S_1 forwards the request to server S_2, the server which owns the page. S_2 forwards the request to the clock site. The clock

site, which is a read clock, adds S_1 as a reader and sends the reply to its server S_2. The clock site only needs to know the cluster to which it is giving the page. It does not need to know the specific client(s), C_1 in this case, in the cluster which will read from this page. S_2 adds the page number to the list of read pages given to S_1. S_2 forwards the reply to S_1. S_1 modifies its page table to indicate that the requesting client C_1 is a reader of this page. S_1 forwards the reply to the requesting client. The client C_1 can now read from its local copy of the page. For the case of an out-of-cluster read request when the clock is a writer, the clock downgrades to a reader before sending a reply to S_2. The clock site is said to have 'loaned' a read copy of the page to cluster 1 because, eventually, cluster 1 will have to invalidate the page. The page will never be converted to a trailer page at any site not in cluster 2.

Figure 2 shows an out-of-cluster read request when another client in the same cluster is already a reader of that page. The requesting client C_{11} sends the request to its server S_1. Even though the server knows it is a request for an out-of-cluster page, its page table indicates that one of its clients, C_{12}, is a reader of that page. It does not forward the request to the server owning the page, S_2. Instead, it sends a message to C_{12} telling it to become a 'pseudo read clock' for that page and to send a reply to C_{11}. C_{12} adds C_{11} as a reader of that page and forwards the page to C_{11}. The overhead of going across clusters is reduced by allowing read clocks in other clusters. C_{12} is a pseudo read clock because it does not have to do everything a clock site does. It only maintains a list of readers. It does not keep track of trailer versions. Even though C_{12} is a pseudo read clock, the clock site in cluster 2, C_2, is still the read clock for this page.

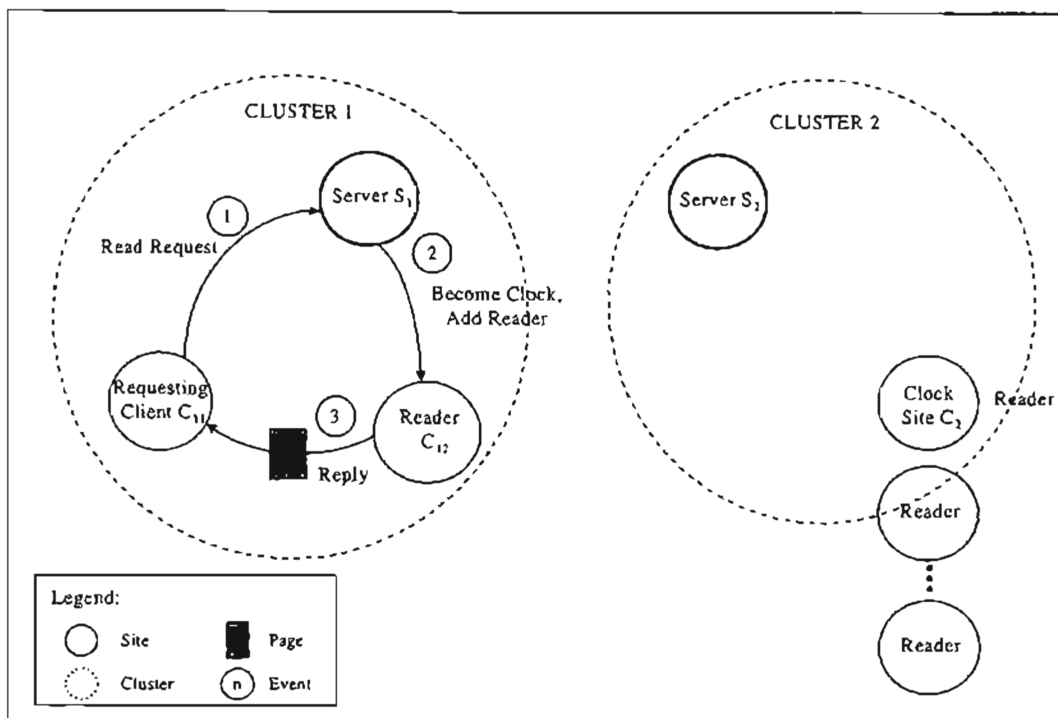

Figure 2. Out-of-Cluster Read Request, Reader in Cluster

Figure 3 shows how a clock site invalidates readers of a page, both within and outside its cluster. A clock site would typically invalidate its readers when servicing a write request. C_2 is a clock site for a page in cluster 2. C_2 has server S_1 as a reader and it also has clients in its cluster as readers of this page. S_1 has client C_{11} listed as a pseudo read clock for that page. C_{11}, in turn, has clients C_{12} and C_{13} as readers of this page. When the clock site C_2 wants to invalidate all readers of this page, it sends invalidation messages to all clients in its cluster. C_2 cannot directly send an invalidation message to server S_1. It sends an invalidation message to its server S_2. S_2 forwards the invalidation message to S_1 (S_2 kept track of the fact that S_1 was a reader of this page). S_1 removes C_{11} as a reader from its page table and forwards the invalidation message to C_{11}. C_{11}

OKLAHOMA STATE U

invalidates itself and sends invalidation messages to C_{12} and C_{13}. No acknowledgments are sent once the page is invalidated at various sites.

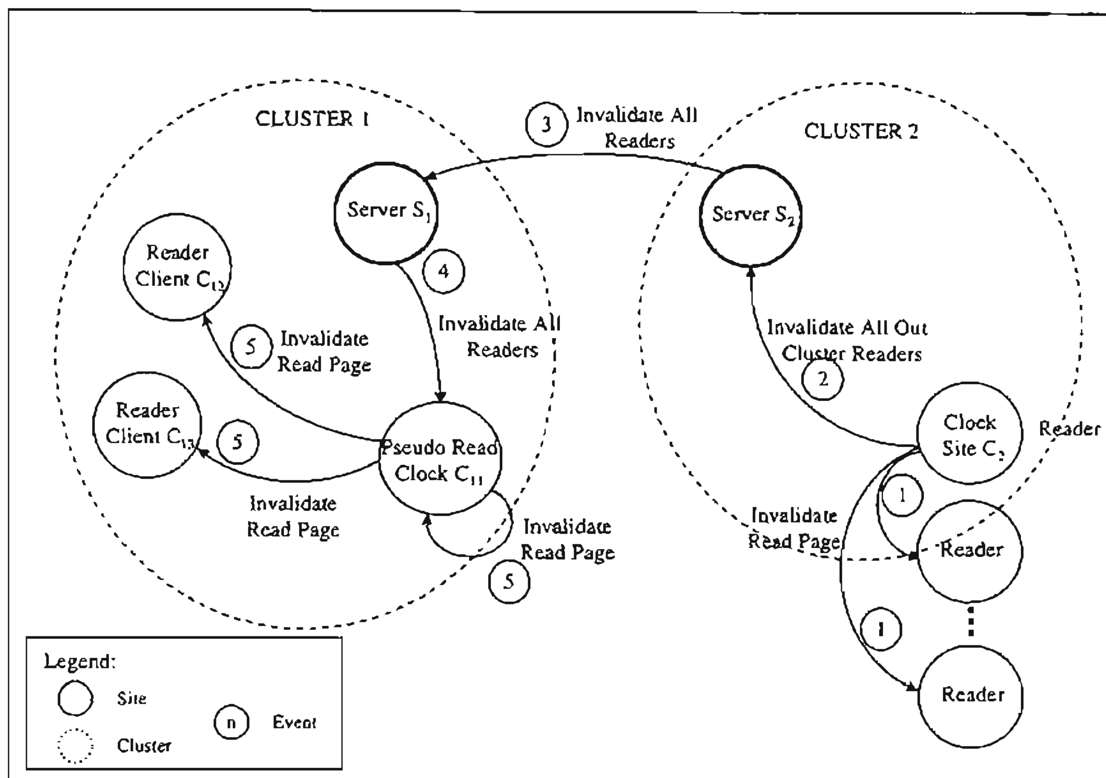

Figure 3. Clock Site Invalidates In and Out Cluster Readers

4.4.2 Write Operation

A write request by a client for a page belonging to its server is the same as in Reliable Mirage+. A WI protocol is used. The client, on a write page fault, sends a request to its server. The server forwards it to the clock site. If the clock is a reader, it invalidates all readers including itself. If the clock site is a writer, it invalidates itself. It becomes a trailer for that page because the page was presumably just written to. It

increments the trailer version, and sends the page along with the clock permission to the requesting client. The requesting client becomes the new clock. The new clock sends an acknowledgment to the server so that the server can send subsequent requests for this page to the correct clock site.

Write permission for a page carries the additional responsibility of a clock. The clock site keeps track of the latest trailer version and becomes a trailer page when it hands over the clock to another site. Therefore, in the case of an out-of-cluster write request, if the same WI protocol is followed, it would mean that a backup for a page in segment s_x may exist in cluster y if a client in cluster y was a writer of that page. This would defeat the purpose of giving autonomy to a server. A server may have to go to other clusters to recover data in its segment in case of a failure in its cluster.

Write Broadcast is an alternative option. But it comes with its disadvantages. A write to an out-of-cluster page would mean sending updates to all readers of the page, within and outside the cluster owning that page. This would be expensive if the number of readers or writers of that page was large. If instead there is only a single remote copy of a page, it is relatively inexpensive to perform updates by a remote store to the single copy.

The approach used depends on the pattern of data access. Replication is useful for data that is read more often than it is written to. Munin is a DSM system which uses *type-specific* coherence [Bennett et al. 90]. Instead of a single memory coherence mechanism for all shared data objects, it uses different mechanisms, each appropriate for a different class of shared objects. For instance, some of the shared data object types used in Munin are write-once, private, write-many, synchronization, migratory, and read-write.

This idea of providing different kinds of memory coherence, depending on the type of shared data, is used for out-of-cluster write requests. Again, it is not the functionality of data, but the locality of data that is used to define its 'type' in this context. An out-of-cluster read request uses WI as explained in Section 4.4.1. However, an out-of-cluster write request uses a combination of WI and WB.

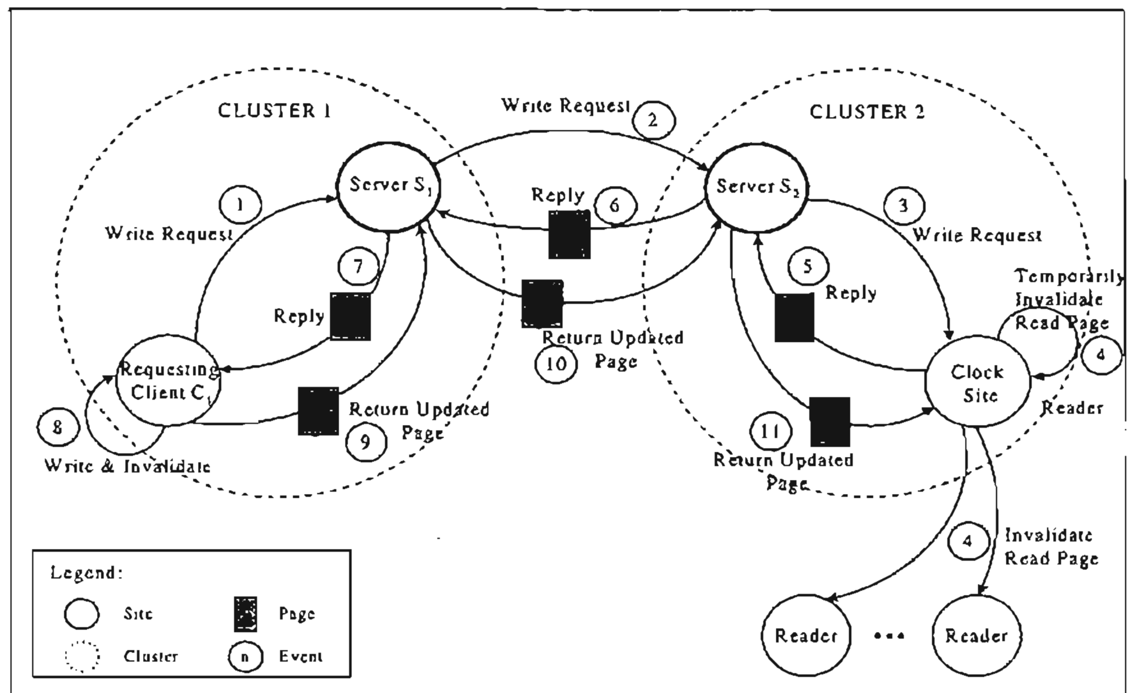

Figure 4. Out-of-Cluster Write Request

Figure 4 illustrates an out-of-cluster write request. The requesting client C_1 sends a write request to its server S_1. S_1 forwards the request to server S_2, the server which owns the page. S_1 forwards this request to S_2 even if clients in its cluster 1 are readers of this page. S_2 forwards the request to the clock site. If the clock site is a reader, it invalidates all existing readers. It invalidates itself 'temporarily' ensuring that a process

OKLAHOMA STATE UN

at its site cannot access the page. It does not remove the page from memory and is still a clock for this page. But it cannot read or write to this page temporarily. It sends a reply to it server S_2 'loaning' a write copy of this page to server S_1 for a specific period of time, say t. The clock site will not service further requests for this page until this time period expires. S_2 adds the page number to the list of write pages given to S_1. S_2 forwards the reply to S_1. S_1 modifies its page table to indicate that the requesting client C_{11} is a writer of this page. S_1 forwards the reply to C_1. C_1 writes to its local copy of the page and has to return the page once the time period t expires. Once this time period expires, C_1 invalidates the page and sends the updated page copy to S_1. S_1 forwards the page to S_2. S_2 forwards the page to the clock site C_2, which loads the page into its memory. Now C_2 can service further requests for this page.

It should be noted that by loaning an 'update copy' to a different cluster, S_2 does not lose control of the page. When C_2 sends the reply, it starts a timer. If it does not get the updated page within this time period, it regains control and sends an invalidation message to S_2, which gets forwarded to C_1 via S_1. Once C_2 has sent this invalidation message, it will not accept any subsequent message from S_1 returning the updated page. As is obvious, a write to an out-of-cluster page carries a very high overhead. Updating only the clock site and not multiple readers reduces the overhead, but it is still expected to be significant. To reduce this overhead, the concept of 'migration' is introduced into the protocol. Further details about migration are given in Section 4.5.

4.5 Migration

4.5.1 Migration Algorithm

In a multi-server environment, a client can request pages within and outside its cluster. If a client repeatedly accesses pages belonging to another cluster, it 'migrates' to that cluster. 'Migration' here means that the client will join the new cluster. It will then contact only the server in the new cluster for all its requests. Migration is done by its current server in cooperation with its 'to be' server.

Every server S maintains a two dimensional migration table of dimensions x (the number of clients in S's cluster) by y (the number of servers other than S). An entry in this table (client C_x, server S_y) is the number of requests client C_x has made for pages in server S_y's segment. Say a client C_1 made an out-of-cluster read or write request for a page belonging to server S_2. When S_1 gets the corresponding reply from S_2, it has to forward the reply to C_1. S_1 also increments the value for the entry (client C_1, server S_2) in its two dimensional migration table. At regular time periods, a server scans its migration table. If a client has made more than a fixed minimum number of requests, N, to any one other cluster, this client should migrate to that cluster. Basically, this client has made more than a specified threshold of requests to that cluster. It is possible that this client made more than this minimum threshold to *more than one* cluster. The cluster to which this client migrates is the cluster to which it made the maximum number of requests. If no maximum exists, a random cluster is chosen. No history of out-of-cluster requests is maintained to determine the client's temporal locality at this point. In case a client has

OKLAHOMA STATE U

made less than the threshold number of requests to another cluster, that value in the migration table is reset to 0 and the count of requests begins again in the next cycle.

4.5.2 Migration Process

Once a server S_1 determines that a client C_1 has to migrate to cluster 2, S_1 has to make sure that C_1 does not have any current or backup copies of pages in S_1's segment. S_1 sends a message m_1 to C_1 asking it to start the migration process (Figure 5). On receiving m_1, C_1 suspends its application process until the migration is complete. The application process should be suspended at a point when it is *not waiting for a reply to arrive*. This is because if the reply arrives *after* this client migrates, then this client might have a page, belonging to its old cluster, in a clock or trailer mode.

C_1 has to give up clock privileges for all pages for which it is a clock site. For every clock page in its memory, C_1 sends a message, m_2, to S_1. The message m_2 has the page, the latest trailer version number, and a read/write attribute. If C_1 is a read clock, it invalidates all readers as shown in Figure 3 (page 32, not shown in Figure 5). This is done to avoid inconsistencies in case one of the reader sites is also migrating. C_1 invalidates the page and becomes a trailer. This is done to deal with a situation where a site in cluster 1 goes down before client C_1 finishes migrating. In that event, the migration process is aborted and C_1 stays in cluster 1. C_1's trailer pages can then be used to recover any lost pages. Clock pages might have been in transit and it is important that these pages can be recovered if and when recovery begins.

OKLAHOMA STATE UI

When S_1 gets the 'hand over clock' message, m_2, it has to select a new clock. S_1 randomly selects a new clock site from its list of clients. A new clock site is selected for every page that is handed over. Say C_2 is to be the new clock site. S_1 sends a message m_3 to client C_2 telling it to become the new clock. Once C_2 installs the clock page, it sends an acknowledgment message m_4 to the server S_1.

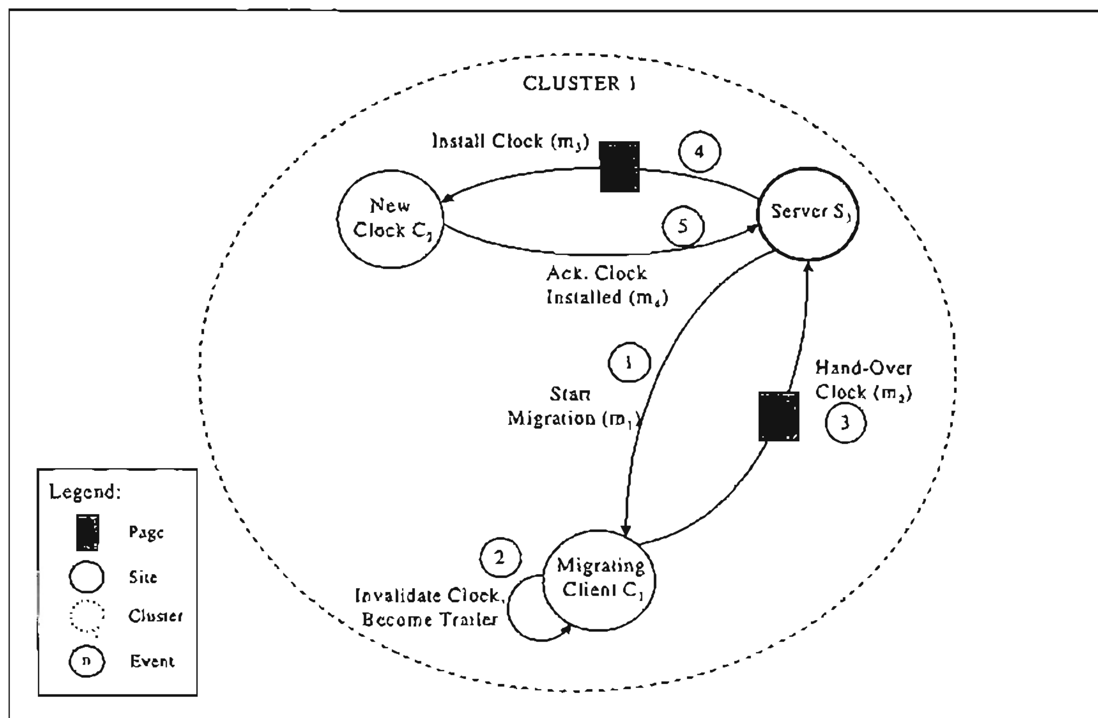

Figure 5. Migration Process: Client Handing Over Clock Page

When the clock is moving from one site to another, the server S_1 will not service any requests for that page. S_1 sets a flag in its page table, using which it ignores any subsequent requests for that page. So, from the time S_1 sends m_1 to the time it receives m_4, S_1 will not service any requests for that page. A client that sends a request for this page in this time period will eventually time out and resend the message.

OKLAHOMA STATE UNIVERSITY

The migrating client C_1 also invalidates all read pages (which are not clock sites) from its memory. It sends a list of these read pages to its server. Its server sends messages to all the corresponding read clocks telling them to remove the client C_1 from their list of readers. The server does not necessarily send one message for every page. It sends one message to each site that is a clock site for one or more of the pages for which C_1 is a reader. That site scans its read clocks and removes C_1 as a reader, if it is one.

If the migrating client C_1 has loaned any write-update pages to another cluster, it either waits until that page is returned, or it sends an invalidation message to that cluster when the time it was loaned for is over. It then has to hand-over this write clock to its server before it can migrate.

The migrating client C_1 also invalidates any out-of-cluster read pages that it may have in its memory. If it is a pseudo read clock for a page, it invalidates all the readers. It does not send a corresponding message to its server. C_1 also returns any out-of-cluster write-update pages it may have. Its server invalidates all out-of-cluster read pages, which reside at C_1, from its page table.

S_1 keeps a count of the number of pages for which C_1 is a clock site. Once S_1 receives an acknowledgment m_4 for every page for which C_1 was a clock, C_1 is ready to be 'handed over' to its new server S_2. Now, S_1 has to tell C_1 that S_2 is its new server. S_1 also has to tell S_2 to accept C_1 as a new client. The details of this hand-over process are given in Sections 4.5.3 and 4.5.4. Once this hand-over is completed, C_1 erases all trailer pages from its memory. These are pages belonging to cluster 1. C_1 should not have any pages in its memory and should not have any requests related to cluster 1 in its request queue when it migrates. C_1 then awakens its application process which had been

suspended during migration. Now C_1 has successfully migrated to cluster 2 and will send all is requests to server S_2.

It should be mentioned that when a client goes out of a cluster, it deletes all its trailer pages for that cluster. So, trailer pages that could be potential backups (if they have the latest trailer version) are lost. The system makes no attempt to ensure that new trailer pages are installed at various sites. This will require the overhead of re-installing trailer pages even if they are not the latest version. This is because only the clock site keeps track of the latest trailer version number and the server will have to coordinate with the clock site to re-install a trailer page at another site in its cluster.

If a trailer page is erased, the system will have to rely on the previous version during recovery. But, whenever a clock gives up clock privileges, it invalidates its copy and becomes a trailer page. Therefore, a subsequent write request for this page will cause the creation of a new trailer page. And, if that page is in a read mode in the system, there could be multiple readers, one of which will be used during recovery. Therefore, it is expected that the system *can* recover, and no pages will be irrevocably lost due to a client deleting its trailer pages.

From the above discussion, it can be seen that migration takes a heavy toll on the system. Migration itself was done to increase the throughput as a client was making expensive out-of-cluster requests. Intuitively, this protocol will perform well if the clients exhibit locality of reference towards pages in their respective segments.

OKLAHOMA STATE UNIVERSITY

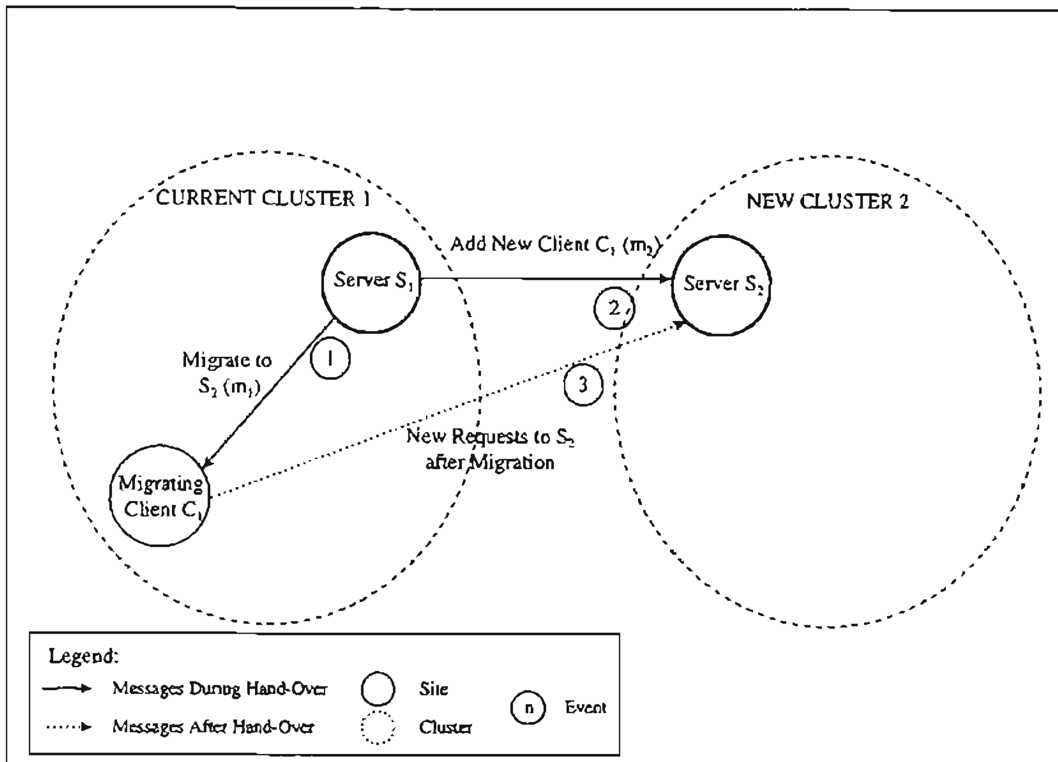

Figure 6. Server Hands Over Client to Another Server (Solution 1)

4.5.3 Consensus Problem

Once a client has handed over its clock pages to the server and the server has received acknowledgments from the new clocks for each of these pages, the client has to migrate from one cluster to another. In a failure-free environment, the solution is simple. Consider the protocol in Figure 6. Client C_1 needs to be 'handed over' to server S_2. It is server S_1's responsibility to ensure that the system is consistent after this hand-over is completed. S_1 sends a message to C_1 telling it to migrate to S_2 (m_1). S_1 also sends a message to S_2 telling it to add a new client to its cluster (m_2). S_1 now removes C_1 from its list of clients. On receiving m_1, C_1 will change the location of its server from S_1 to S_2.

OKLAHOMA STATE UNIVERSITY

On receiving m_2, S_2 will add C_1 to its list of clients and from then on will service requests from C_1. If m_2 is processed after m_1, C_1 might send a request to S_2 before S_2 gets m_2. This will not create a problem because S_2 can simply ignore the request until it gets m_2. C_1 will time out and resend the message until S_2 processes m_2. Once C_1 receives m_1 and S_2 receives m_2, the migration is completed. The client C_1 will send subsequent requests to server S_2 instead of server S_1.

The solution, unfortunately, is not so straightforward. This is because the system is prone to failure. Say S_1 sends m_1 and m_2. If S_2 fails before getting m_2 but C_1 gets m_1, C_1 will send its subsequent requests to S_2. But S_2 will never honor these requests because it never received m_2. Alternatively, say C_1 fails before getting m_1 but S_2 gets m_2. Once S_1 sends m_1 and m_2, it removes C_1 from its list of clients and its responsibility for the hand-over is over. Once C_1 recovers from its transient failure, it will try to send messages to its (what it thinks is current) server S_1. S_1 of course will ignore all such messages because C_1, according to S_1, is not in its cluster. C_1 cannot send messages to S_2 because it never got m_1. So now C_1 is a 'lost client'. No server in the system will accept messages from C_1. Cases where S_1 fails after sending m_1 but before sending m_2, or vice versa, will also lead to an inconsistent system.

Now, the solution that looked easy is far from easy. Let us consider another approach which uses acknowledgments. Consider Figure 7. Server S_1 sends m_1 and m_2 to client C_1 and server S_2, respectively. It waits for an acknowledgment to m_1 and m_2 from C_1 and S_2, respectively. Let us call these acknowledgments m_3 and m_4. Once S_1 gets these acknowledgments, it removes C_1 from its list of clients. But say S_1 sent m_1 and m_2. S_1 gets m_4 but fails before it gets m_3. Now, S_2 has C_1 in its list of clients. But when

OKLAHOMA STATE UNIVERSITY

S_1 recovers, it will still have C_1 in its list of clients, as it did not complete the hand-over. So now the system is inconsistent because both S_1 and S_2 have C_1 listed as their client. Let us modify the protocol slightly so that S_2 does not add C_1 to its cluster until it is sure S_1 has received m_4. That will require S_1 to send an acknowledgment for m_4, let us call it m_5. But, again, S_1 cannot be sure S_2 received m_5 as S_2 could have failed in the interim. This leads to an infinite loop of acknowledgments without a consensus between the sites.

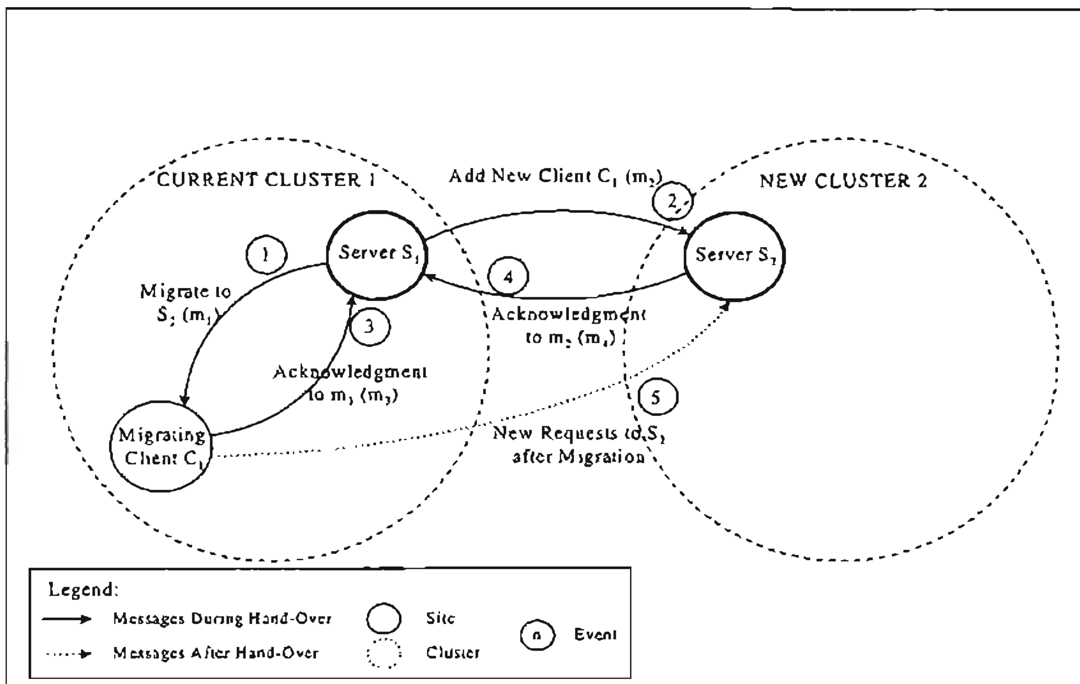

Figure 7. Server Hands Over Client to Another Server (Solution 2)

In the area of fault-tolerant computing, "the consensus problem is to form an agreement among the fault-free members of the resource population on a quantum of information in order to maintain the performance and integrity of the system" [Barborak et al. 93]. Such an agreement may be made regarding the configuration of the system, the

OKLAHOMA STATE UNIVERSITY

synchronization of its clocks, the contents of its communications, or any other value requiring global consistency.

4.5.4 Solution to the Consensus Problem

The consensus problem is not a recent problem. Distributed transactions, those that involve more than one server, face the consensus problem in a failure prone environment. The two-phase commit protocol is a solution [Coulouris et al. 94]. But this solution uses stable storage at every stage so that the system can recover from a failure at any stage of the consensus.

Fischer, Lynch, and Paterson proved that in a distributed system with an unbounded but finite message latency, there is *no* protocol that can guarantee consensus within a finite amount of time, even if a single process fails by stopping [Fischer et al. 85]. Turek and Shasha summarized the cases when it *is* possible to reach a consensus even in the presence of multiple failures [Turek and Shasha 92]. More optimistic assumptions on timing constraints within the network and among processors yield this solution.

A set of system parameters are identified, they are [Turek and Shasha 92] as follows.

1. *Processors can be either synchronous or asynchronous*: Processors are synchronous if and only if there exits a constant $s \geq 1$ such that for every $s+1$ steps taken by any processor, every other processor will have taken at least one step.

OKLAHOMA STATE UNIVERSITY

2. *Communication delay can be either bounded or unbounded*: Delay is bounded if and only if every message sent by a processor arrives at its destination within t real-time steps for some predetermined t.

3. *Messages can be either ordered or unordered*: Messages are ordered if and only if processor P_r receives message m_1 before message m_2 when P_1 sends m_1 to P_r at real time t_1, P_2 sends m_2 to P_r at real time t_2, and $t_1 < t_2$.

4. *Transmission mechanism can be either point-to-point or broadcast*: The transmission mechanism is point-to-point if a processor can send a message in an atomic step to at most one other processor. It is broadcast if a processor can send a message to all processors in an atomic step.

According to Turek and Shasha [Turek and Shasha 92], one of the cases where consensus is possible is when *messages are ordered and the transmission medium is broadcast.* This is a situation when processors are asynchronous and some of them may fail by stopping. However an atomic broadcast is possible. Applying this to the consensus problem for handing over of a client to another cluster gives a solution.

Consider Figure 8. It is assumed that S_1 can multicast m_1 and m_2 to C_1 and S_2, respectively, and removes client C_1 from its list of clients (both m_1 and m_2 are labeled 1). So, S_1 can fail before or after sending m_1 and m_2, but not in between. It is assumed that C_1 cannot fail from the time S_1 removes C_1 from its client list and the time S_2 adds C_1 to its client list. If S_2 fails before receiving m_2, S_1 no longer has C_1 as a client, C_1 has S_2 as its server but S_2 does not have C_1 as a client. The 'lost client' problem is solved by making C_1 intelligent. Once C_1 receives m_1, it keeps polling S_2 until it gets an acknowledgment. Even if S_2 went down before receiving m_2, it gets the polling message

OKLAHOMA STATE UNIVERSITY

once it comes back up. When S_2 gets the polling message m_3 from C_1, it adds C_1 to its list of clients and replies with message m_4. Now the migration of client C_1 from cluster 1 to cluster 2 is completed. The system is in a consistent state. Once C_1 gets m_4, it goes back to its normal state and can start sending its requests to S_2, its new server.

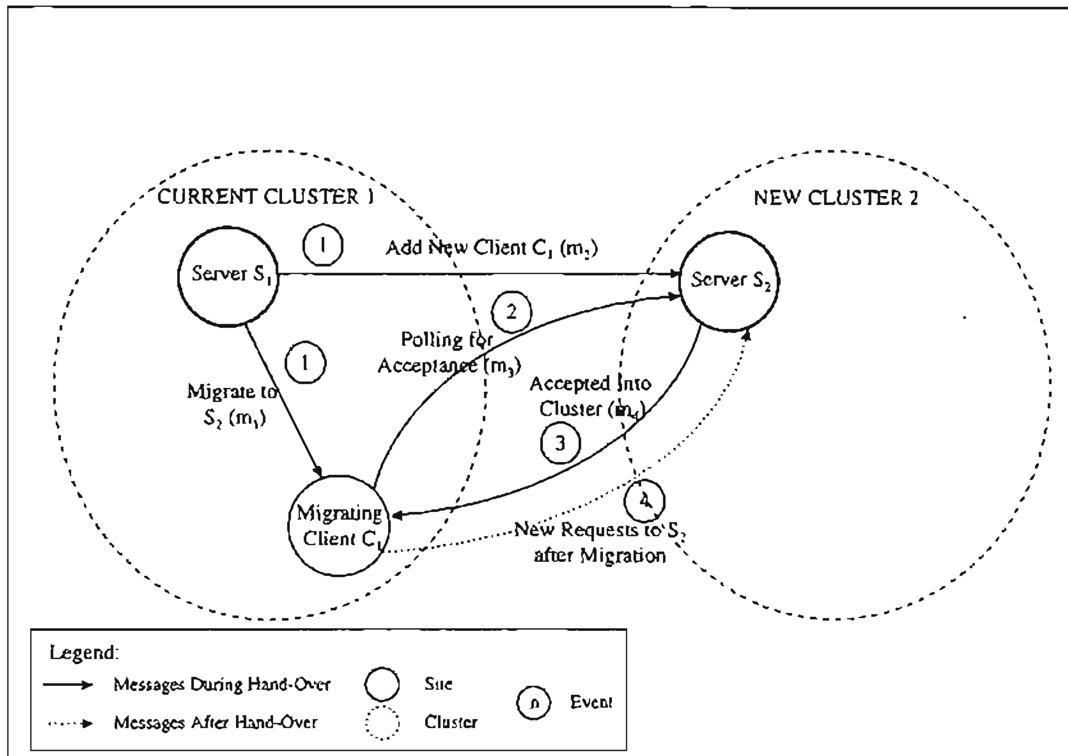

Figure 8. Server Hands Over Client to Another Server (Solution 3)

4.6 Recovery

Recovery is the process of restoring the system to its normal operational state in the event of a failure. When a site in the system goes down, the system must either wait for the site to come back up, in the case of a transient failure, or must continue without

OKLAHOMA STATE UNIVERSITY

that site, in the case of a permanent failure. In either case, the failed site may have pages in its memory that are now lost. The server has to recover these pages. The server must either try to obtain another copy of the page that might exist at another site, or it must 'rollback' to the latest available copy. Just recovering lost pages is not enough. The data in the system must also be in a consistent state. For example, if the server has a certain site listed as a clock site for a page, that site *must* be the clock site for the page. Or, if a clock has a site listed as a reader of the page, then that site *must* be a reader of that page.

Inconsistencies may also arise if a message was lost when a site went down. Also, all copies of a page must have the same information. If a client in a cluster has a permanent failure, then its server must no longer have that client listed as its client. If the server site had a permanent failure, a new server has to be elected. Coordination between all the sites is required to elect a new server, to recover lost pages, and to ensure consistency. Recovery deals with making sure that there is at least one copy of every page. Making sure that the system is consistent after recovery is an important part of bringing the system back to normalcy. This is called reliability.

Reliable Mirage+ [DeMatteis 96] made sure that there was at least one backup for every page. These backups are not stored at a single site. If that is done, then the failure of that one backup site would mean that all the backups are lost. Rather, Reliable Mirage+ decentralizes the backup storage. Backups are stored in the main memories of various sites in the system. The trailer pages created whenever a clock gives up clock privileges, serve as backup copies in the system. Reliable Mirage+ assumes that failures are permanent. If the server fails, standard election algorithms are used to elect a 'coordinator', which also becomes the new server. The system gets into a recovery mode

OKLAHOMA STATE UNIVERSITY

and stops servicing requests until recovery is completed. The throughput of the system *during* recovery is zero. In Reliable Mirage+, it is assumed that a site can detect the failure of another site in the cluster (refer to [DeMatteis 96] for details).

In a multi-server environment, autonomy is given to servers during recovery. *If a site goes down, only that single cluster goes into recovery mode.* The rest of the system functions normally. Note that it is possible for two clusters to go into recovery mode in a single-site failure. This happens when the server at a failed site is in one cluster and the client at that failed site is in another cluster. Since every server stores its backup copies only at sites in its cluster (if one of its clients migrates, it erases its backup copies for this cluster), no communication outside the cluster is needed for recovery.

The recovery process within a cluster is the same as in Reliable Mirage+. All sites in the cluster send information about pages in their memory to their server. The server analyzes this information and sends messages to clients to become new clocks or to install more backup pages. The rest of the clusters continue functioning, though not at 100% efficiency. Any client in another cluster that requires a page from this cluster will have to wait until this cluster recovers. Since it will set a time out anyway and resend its request until it gets a reply, no new adaptation or regulation is required.

In a multi-server environment, reliability takes on a new dimension. Inconsistencies can occur across clusters. When a cluster goes into recovery mode, it may have loaned its pages to other clusters. Those clusters may continue to read from or write to these pages assuming they are the current pages. Those clusters may expect that they are reading current data or that a write to one of these pages will be reflected in the global database.

To ensure consistency across clusters, it is assumed that a site can detect a failure of another site in the system, both within and outside its cluster. In other words, it is assumed that a server can detect the failure of every other server. And, when a server starts the recovery process (it may not be the failed site), it informs all other servers about its status. When a server knows that another server is recovering or has failed, it *invalidates* all pages from that segment in its cluster. So, any client that had a page from that cluster can no longer access the page. The client will have to resend the request, which, of course, will not be serviced until that cluster recovers.

The crucial aspect of this multi-server architecture is that the whole system is not suspended during recovery. Clients in other clusters will still be able to function, as long as they do not need data from the recovering cluster. Intuitively, throughput should increase because of multiple servers. But the trade-off may be nullified if the overhead of migration or requests out of clusters is too high. The potential benefits of this architecture rely on the data access patterns of the clients and on the autonomy given to the servers.

OKLAHOMA STATE UNIVERSITY

CHAPTER V

SIMULATION ENVIRONMENT

In this thesis, Reliable Mirage+ [DeMatteis 96] was simulated and compared with a simulation of the multi-server fault-tolerant protocol. Reliable Mirage+ was originally developed in an environment having twelve IBM PS/2s connected by a 10Mbps Ethernet [DeMatteis 96]. A program-driven DSM simulator has also been developed [Shah 97]. It did a comparative study of WI, WB, and BR coherence protocols using simulation.

5.1 Why Simulate?

Two alternative approaches, other than simulation, can be used to evaluate a protocol. They are analytical modeling and building a real system. Analytical modeling of multiprocessor systems is difficult because of their complexity. Theoretical models for the verification of cache coherence protocols exist [Pong and Dubois 95]. Pong and Dubois' model uses a state machine approach and can be used to verify cache coherence protocol correctness and data consistency at an early design stage. But it does not facilitate the evaluation of protocol performance.

Building a real system, on the other hand, is problematic and is limited by hardware constraints. Simulation frees the evaluation of a protocol from hardware

constraints, it gives insight into the scalability of a protocol beyond the limits imposed by a real machine.

5.2 Simulation Environment

Proteus is a simulator for MIMD multiprocessors [Brewer et al. 91] [Brewer and Dellarocas 92]. It is an execution driven simulator which multiplexes multiple threads on a single host processor. The simulation consists of a simulator engine and user applications. The architecture of the simulation engine is flexible and can be configured to represent a target architecture. The user application code is directly executed on the host processor.

Proteus simulates the events that take place in a parallel machine at the level of individual machine instructions, bus or network accesses, interrupt requests, etc. The parallel application must be written in a simple superset of the C programming language and a set of supported simulator calls. Proteus provides libraries and constructs for message passing, thread management, memory management, data collection, etc. [Brewer and Dellarocas 92].

Proteus is accurate and fast because it uses 'augmentation' to measure the execution time of user applications [Brewer 92]. Traditional multiprocessor simulators simulate the machine cycle by cycle, interpreting each instruction. This generally results in high overhead. Proteus, on the other hand, augments the user code with additional instructions that track the cycles used by the code. The augmented code calculates the number of cycles required to execute the original unaugmented user code. The

OKLAHOMA STATE UNIVERSITY

augmenting is done in units of basic blocks, where a basic block is a group of instructions that must be executed as an *atomic* unit. The simulator clock is incremented once per block instead of once per instruction. Cycle counting or augmentation allows the execution time of the simulation engine (which simulates the underlying architecture) to be independent of the execution time of the user application. Augmentation allows the execution of the application to be interleaved with the simulation of the architecture without compromising on accuracy or speed (for details about verification of cycle counting and validation of the simulator refer to [Brewer 92] [Brewer et al. 91]).

The independent processor nodes in the simulated MIMD multiprocessor are connected by an interconnection medium. The interconnection network can be a bus, a direct network, or an indirect network. A *direct network* directly connects two processing nodes by point-to-point links. The 'distance' between two processing nodes is variable and is equal to the number of individual point-to-point links that must be traversed for data to travel from one node to the other. Communication locality can be exploited when direct networks are used. *Indirect networks* do not connect any two processor nodes directly. Instead, they use a number of internal switching stages that automatically route a packet to its destination. Here, the 'distance' between any pair of nodes is the same and is equal to the number of internal stages plus one. As is obvious, indirect networks do not exploit communication locality.

Proteus uses analytical formulas to calculate the expected latency of a packet traveling through the network [Brewer and Dellarocas 92] [Agarwal 91]. This calculation also takes the network congestion into consideration. Therefore, Proteus mimics a real network closely by taking the dynamic load on the network into account.

OKLAHOMA STATE UNIVERSITY

Shared memory as well as message passing architectures can be simulated using Proteus. Caches may or may not be used. The target architecture can be specified using an easy to use interface. The type of the interconnection network, the number of processors, existence of caches, etc. can be specified. The user program can use thread management routines, message passing routines, and interrupt handlers which are a part of the libraries provided by Proteus to simulate a complete parallel application.

The version of Proteus that was initially used for this research was Proteus version L3.10, which runs on a Sun SPARC system running SunOS 4.1.3 or Solaris 2.5. Subsequently, Proteus was upgraded to version L3.12, and then to version L3.13, as the upgrades had vital bug fixes. Proteus was originally developed primarily by Brewer and Dellarocas at Massachusetts Institute of Technology [Brewer et al. 91]. The Sun version used for this thesis has been developed at Louisiana State University by Koppelman [Koppelman 97a].

OKLAHOMA STATE UNIVERSITY

5.3 Configuration of the System in the Thesis

The base architecture of the environment in which Reliable Mirage+ [DeMatteis 96] and the multi-server protocol were simulated was the same. The only difference was in the memory coherence protocol implemented. The memory coherence protocol lies at a level in between the architecture and the user application. It may be considered to be at the operating system or kernel level. The WI protocol in Reliable Mirage+ and the hybrid multi-server protocol do not exploit communication locality between processors. In other words, the distance between any pair of processors is the same. The locality of reference,

which is exploited in the multi-server protocol, is exhibited in a cluster. A cluster, in this context, is only a logical abstraction and not a physical one. Designing the underlying network so that every cluster exhibits communication locality within the nodes in the cluster will, intuitively, not be able to reap the potential benefits of a client migrating from one cluster to another. The multi-server protocol exploits the locality of reference at the software level and not the communication locality at the hardware level. Therefore, the interconnection network in the simulation was an indirect network, where the communication latency between any pair of processors will be the same when the network load is the same.

The simulated system was a system of loosely coupled processors connected via an indirect network. Therefore, there was no shared memory. A message passing architecture was simulated. A message was sent using a 'send inter-processor interrupt' primitive that is provided by the Proteus library. No caches were used in the simulation. The overhead of cache block replacement strategies, or the cache hit or miss rates might cloud the performance of the single and multi-server protocols. Every processor is assumed to have enough local main memory to accommodate all the pages in the distributed database. This is assumed so that page replacement algorithms need not be simulated, which may affect the performance of the coherence protocol used. Threads are created on various processors whenever necessary. Details about the actual implementation of a client and a server, the signal handling mechanism, synchronization, etc., are given in Chapter VI.

OKLAHOMA STATE UNIVERSITY

CHAPTER VI

IMPLEMENTATION ISSUES

This chapter gives details about a few key implementation issues that arise when using Proteus [Koppelman 97a]. This chapter does not give details about the specific data structures used in the simulation. It highlights how constructs provided by the Proteus library can be used to design the simulated system. It also gives an overall conceptual view of the synchronization, signal handling, and client-server interaction in the simulated system.

6.1 Threads

The programming model provided by Proteus is a simple thread-based run-time system that uses a library of thread and memory management procedures. A user application is simulated as a parallel application by executing it as threads on various processor nodes. A client site will have a user 'process' and a client 'process' at that site. One thread is created for a client and one for a user. A server site will have a single thread for a server. When both a server and a client site are at the same processor, there will be three threads at that site as shown in Figure 9. Conceptually, the user threads are

OKLAHOMA STATE UNIVERSITY

partially at the user level and partially at the kernel level. The client and server threads are at the kernel level.

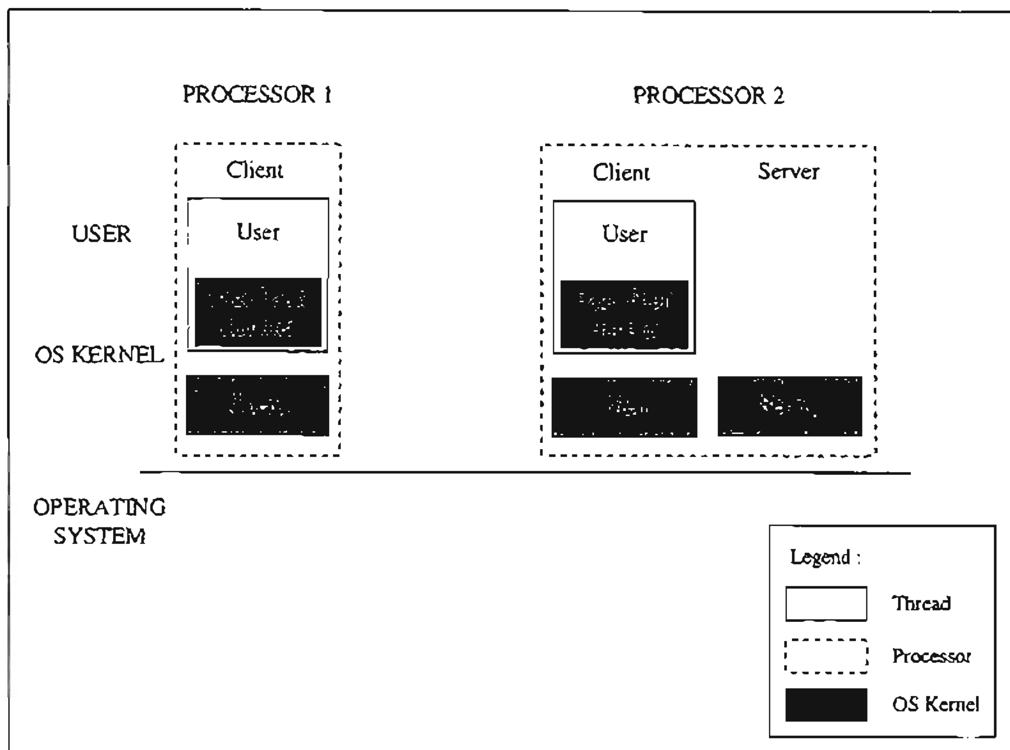

Figure 9. Client and Server Threads

The user thread represents the application process running at that processor. It reads from and writes to the pages in the distributed database. When a page is not in the local memory of a processor, a page fault occurs. When a page fault occurs, the page fault handler kicks in at the kernel level. It sends a request to the server in that cluster and puts the user thread to sleep. The request sent is a message that causes an inter-processor interrupt at the destination processor. Different types of interrupts can be defined using Proteus library routines. What happens to the message at the destination

site depends on the interrupt that occurs. When a message is sent to a server, the interrupt at the server site inserts the message in the server's request queue at that site. Similarly, when a message is sent to a client, the interrupt at the client site inserts the message in the client's request queue at that site.

The server and client threads are essentially infinite loops. When the site at which they are running goes down, these threads exit. The server and client threads implement the coherence protocol (see Chapter IV, page 24). The server's thread pulls out a request from the server's request queue, services that request, pulls out the next request, and so on. If the request queue is empty, the server thread waits for the next message to arrive. Since busy waiting generally wastes time without accomplishing anything, the server thread is put to sleep for a short while. When it wakes up, it re-checks the server queue for any incoming messages. The server may forward a request to a client, say a clock site. This causes an inter-processor interrupt at the clock site's processor. The message is put into the client's request queue at that site. The client thread, like the server thread, pulls out a request from the client's request queue, services it, pulls out the next request, and so on. The client thread sleeps for a short period of time if the request queue is empty. A reply sent to the requesting site is inserted into the request queue of the client at that site. When that client services the reply, it 'signals' to the user that a reply has arrived. The user wakes up and can proceed by reading from or writing to the local copy of the page. The signal handling mechanism requires careful synchronization and is explained in Section 6.4.

OKLAHOMA STATE UNIVERSITY

6.2 Server, Client, and User States

The distributed environment that is simulated is complex because of the various events that may be happening at a client, a user, or a server site. For example, a user might be running or suspended waiting for a reply. A client might be servicing a request, asleep waiting for a message to arrive, migrating to a different cluster, etc. When a client site is migrating, the user thread at that site is suspended until the migration is completed. A 'migration thread' is created at the migrating client which hands over all the clock privileges to its server as illustrated in Figure 5 (page 38). Therefore, the migration thread will be sending information to its server and parallely the client thread at that site will be servicing any messages that may arrive. The server site has another thread running during normal operation that periodically checks if any of its clients should migrate to other clusters. Parallely, the server thread would be servicing any requests that arrive. The glue that holds this complex system together is a state variable defined for every user, client, and server.

It must be mentioned that messages are serviced atomically. This means that a server thread or a client thread cannot exit or abort servicing a request midway in the simulation.

6.2.1 Server States

Figure 10 is a state diagram of all the possible states and the transitions a server thread can have. When a server thread is first created, it has the state 'Server OK'. This

OKLAHOMA STATE UNIVERSITY

is when the server is servicing requests normally. In this state, it pulls out requests from the request queue, services them, and goes to sleep for a short period of time if the request queue is empty. Even though the server thread goes to sleep, that state is not added to the state diagram because it is local only to the server thread's infinite loop and is not used anywhere else. At the end of the simulation, the server thread moves from the 'Server OK' state to the 'Server Never Started' state and exits.

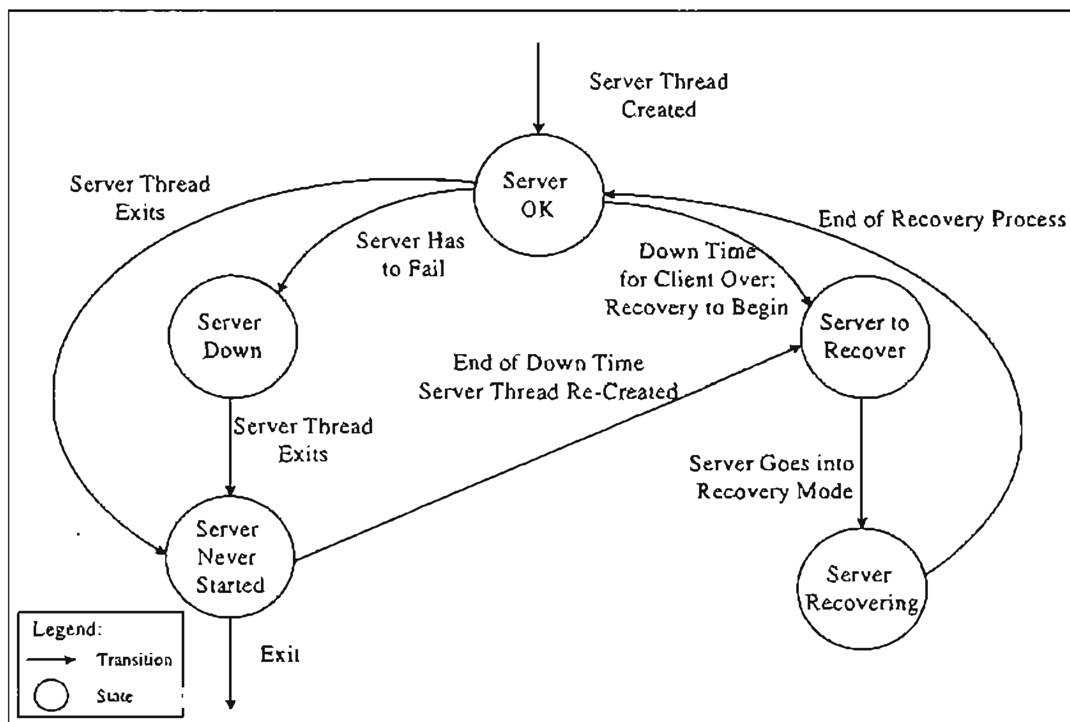

Figure 10. Server State Diagram

Messages are serviced atomically. This means that the server thread has to finish servicing a message once it starts servicing it. It cannot change to a state where the system is doing something else in the middle of a message. For example, it cannot go down or start the recovery process while servicing a message. When a server site is to go

down, the server thread must exit and all memory at that site will be erased. In the simulation of a failure, the server state is first changed to 'Server Down' when it is known that the server site must go down. The server thread might have been servicing a request at this point. Before the server thread pulls out the next request from the request queue, it sees that the server state is 'Server Down', and moves to the 'Server Never Started' state. The server thread exits since that site is down. This state implies that there is no *active* server thread at that site.

All failures in the simulation are transient failures. When a site goes down, it stays down for some 'down time' and then comes back up. It is not removed from the cluster, as in Reliable Mirage+ [DeMatteis 96]. Once a server thread is in the 'Server Down' state, it stays down for the down time. At the end of the down time, the server thread is re-created (it exited in the 'Server Never Started' state) and then moves to the 'Server to Recover' state. This is the state when the server thread must stop servicing requests and the system must get into recovery mode. 'System' here refers to that particular cluster. Another case when the server state changes to the 'Server to Recover' state, from the 'Server OK' state, is when a client at another site in the cluster goes down. While the client is down, the server will keep servicing requests normally.

Once the down time for the client is over, the server moves to the 'Server to Recover' state. This is a cue for the server thread to stop processing requests. When the server thread sees this state before processing the next request, it moves to the 'Server Recovering' state. At this point a recovery thread is created that handles the recovery process. The server thread sleeps until recovery is completed. This ensures that no normal requests are serviced during recovery. Once the recovery process is completed

OKLAHOMA STATE UNIVERSITY

and all data consistency checks have been made, the system can go back to normalcy. The server then moves to the 'Server OK' state after which the server thread starts servicing requests again.

6.2.2 Client States

Figure 11 is a state diagram of all the possible states and transitions a client thread can have. When a client thread is first created, it has the state 'Client OK'. This is when the client is servicing requests normally. In this state, it pulls out requests from the request queue, services them, and goes to sleep for a short period of time if the request queue is empty. Even though the client thread goes to sleep, that state is not added to the state diagram because it is local only to the client thread's infinite loop and is not used anywhere else. At the end of the simulation, the client thread moves from the 'Client OK' state to the 'Client Never Started' state and exits.

Messages are serviced atomically. This means that the client thread has to complete servicing a message successfully. It cannot change to a state where the system is doing something else in the middle of a message. For example, it cannot go down or participate in the recovery process while servicing a message. When a client site is to go down, the client thread must exit and all memory at that site should be erased. In the simulation of a failure, the client state is first changed to 'Client Down' when it is known that the client site must go down. The client thread might have been servicing a request at this point. Before the client thread pulls out the next request from the request queue, it sees that the client state is 'Client Down', and moves to the 'Client Never Started' state.

OKLAHOMA STATE UNIVERSITY

Figure 11. Client State Diagram

OKLAHOMA STATE UNIVERSITY

The client thread exits since that site is down. This state implies that there is no *active* client thread at that site.

Client failures are also transient failures. When a site goes down, it stays down for some 'down time' and then comes back up. It is not removed from the cluster, as in Reliable Mirage+ [DeMaiteis 96]. Once a client thread is in the 'Client Down' state, it stays down for the down time. At the end of the down time, the client thread is re-created (it exited in the 'Client Never Started' state) and then moves to the 'Client to Recover' state. This is the state when the client thread must stop servicing requests and the client must get into recovery mode. Here, an assumption is made that the server detects that this failed client is up again, and then starts the recovery process.

Another case, when the client state changes to the 'Client to Recover' state from the 'Client OK' state, is when another site in the cluster goes down and the server notifies all clients to start the recovery process. The 'Client to Recover' state is an indication for the client thread to stop processing requests. When the client thread sees this state before processing the next request, it moves to the 'Client Recovering' state. At this point a recovery thread is created that handles the client's recovery process. The client thread sleeps until recovery is completed. This ensures that no normal requests are serviced during recovery. Once the recovery process is completed and all data consistency checks have been made, the server sends a message to all its clients telling them to function normally. This changes the client state to 'Client OK' and the client thread starts servicing requests again.

When a server determines that a client should migrate out of its cluster, it informs the client to start the migration process. This changes the state of the client from 'Client

OKLAHOMA STATE UNIVERSITY

OK' to 'Client to Migrate'. When the user thread sees this client state, it stops sending further requests until migration is completed. Once the user thread stops sending further requests, the client state changes to 'Client Migrating'. A migrating thread is created at the client site which hands over all clocks residing at this client site and invalidates all read pages as explained in Section 4.5.2 (page 37). Once the server knows that the client is ready to migrate, it broadcasts a message to the client telling it to migrate to the new cluster, and to the new server telling it to accept this client (Figure 8, page 46). The client changes the location of its server to its new server and moves to the 'Client Migrated Polling' state. At this point a 'polling thread' is created that keeps polling the new server for acceptance. A reply from the new server changes the client state back to the 'Client OK' state. At this point the client thread resumes servicing requests and the user thread resumes execution of the application.

A client site may go down while migrating. In such an event, the client state changes to 'Client Down' from the 'Client to Migrate' or the 'Client Migrating' states (marked as state 'X' in Figure 11). An assumption was made that the client cannot go down once it is polling the new server for acceptance, until it is accepted (Section 4.5.4, page 45). The migrating client does not belong to *any* cluster at this point. Therefore the client state *cannot* change from the 'Client Migrated Polling' state to the 'Client Down' state.

However, the server can start the recovery process, due to the failure of another site in its cluster, while a client is migrating. In such an event, the migration is aborted. This is reflected in the state diagram in the transitions of the 'Client to Migrate' or the 'Client Migrating' states to one of 'Client to Recover' or 'Client Recovering' states.

OKLAHOMA STATE UNIVERSITY

When the client is in the 'Client Migrated Polling' state, it has not yet been accepted by its new server. It does not belong to *any* cluster at this point. Therefore, it cannot move to any of the recovery states from that state.

6.2.3 User States

Figure 12 is a state diagram of all possible states and transitions of user threads. A user thread changes its state depending on the state of its client. The client state is visible to the user since the client state variable is located at the same processor as the user and represents the same processing node. When a user thread is first created at a client site, it has the state 'User Running'. This is when the client site is functioning normally. The client thread is in the state 'Client OK'. The user thread is executing the application and reading from and writing to pages. When a page fault occurs, the page fault handler (which is executed by the user thread) sends a request to the server. The user thread goes to sleep for a certain period of time. It moves to the 'User Suspended' state and waits for a reply. The arrival of a reply is signaled to the user, which then moves from the 'User Suspended' state back to the 'User Running' state and accesses the page which just arrived. However, if a reply does not arrive within a certain time period, the user thread wakes up (without an external signal like an inter-processor interrupt or message) and moves to the 'User Running' state from the 'User Suspended' state. The user thread will then have to resend the earlier request.

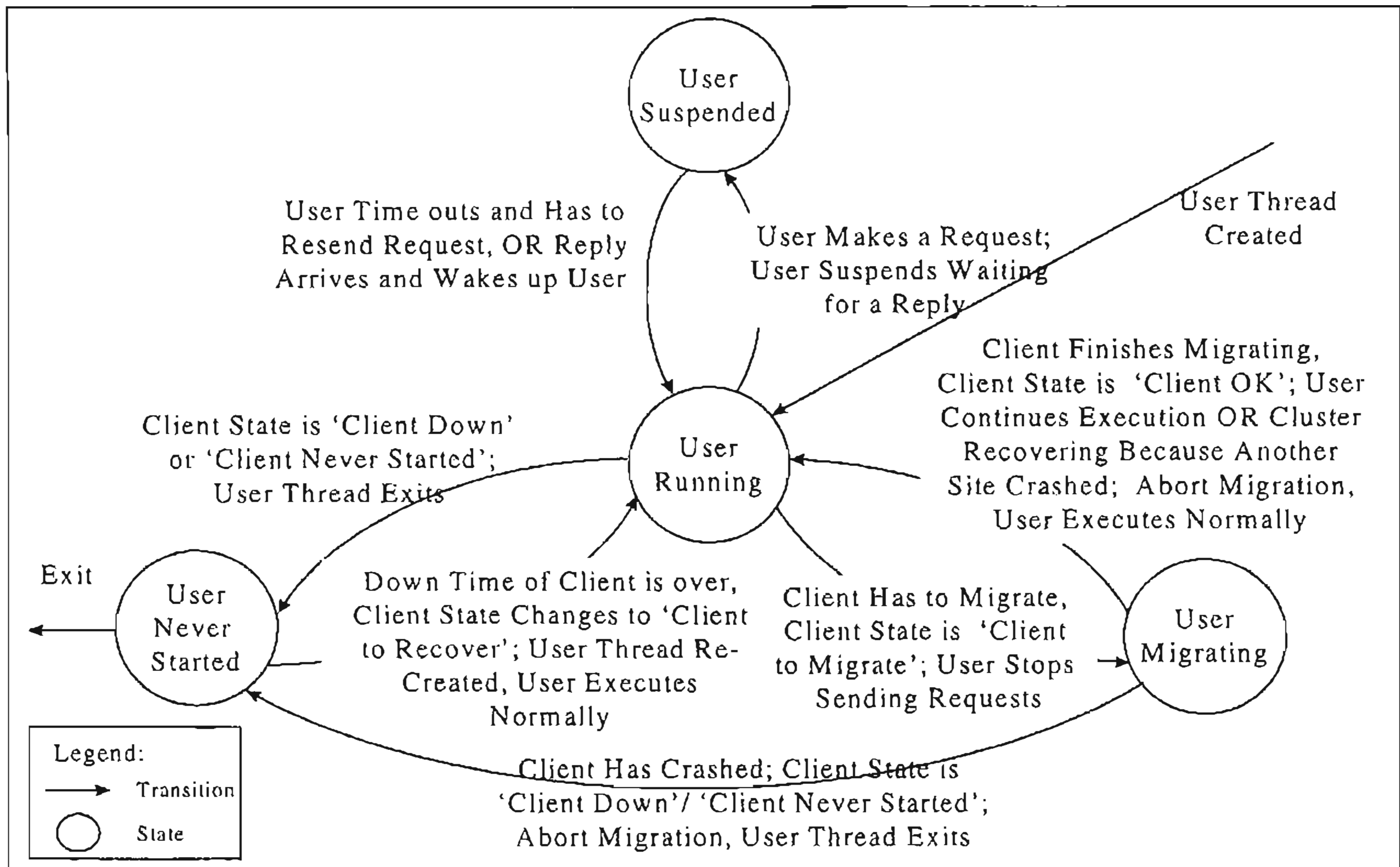

Figure 12. User State Diagram

OKLAHOMA STATE UNIVERSITY

When a site crashes, all threads on that processor must exit and the memory should be erased. When a client site crashes, the client state may be in the 'Client Down' state or the 'Client Never Started' state depending on what the client thread is doing at that point. When a user thread sees these client states, it moves from the 'User Running' state to the 'User Never Started' state. The user thread exits since this site is down. This state implies that there is no *active* user thread at this site. Once the down time of the client is over, the client moves to the 'Client to Recover' state. At this point, the user thread is re-created and the user moves to the 'User Running' state. The user thread continues normal execution and sends requests to its server. If the server is recovering, it will simply ignore all requests from users until recovery is completed.

When a client is to migrate to another cluster, its server tells it to migrate. The client moves to the 'Client to Migrate' state. When the user sees the 'Client to Migrate' state, it moves to the 'User Migrating' state. The user stops sending requests to its current server until migration is completed. Once the client hands over all its pages and the migration process is completed, the client changes its server and moves to the 'Client OK' state. When the user thread is in the 'User Migrating' state, and the client state changes to the 'Client OK' state, the user thread moves to the 'User Running' state, and continues execution in its new cluster.

When a client is migrating to another cluster, that particular client site can go down or another site in that cluster may go down. When the migrating client crashes, migration is aborted and the client site stays in the same cluster. Therefore, when a user is in the 'User Migrating' state and the client state changes to either 'Client Down' or 'Client Never Started', the user thread exits and moves to the 'User Never Started' state.

OKLAHOMA STATE UNIVERSITY

When a client site is migrating and another site in the cluster goes down, the migration process is aborted and the recovery process begins. When a user is in the 'User Migrating' state and the client state changes to either 'Client to Recover' or 'Client Recovering' states, the user changes to the 'User Running' state. The user thread continues normal execution. The server will ignore all requests from users until recovery is completed.

6.3 Synchronization

When multiple threads run on a single simulated processor node, some form of synchronization is essential. Multiple threads on the same node should not be allowed to enter critical sections of code (for instance, to modify a page table, data structures, or any other memory on the same processor) simultaneously. Proteus provides semaphore constructs that support spin locks [Brewer and Dellarocas 92]. Proteus also has the atomic region construct. An atomic region is where a thread cannot be preempted by another thread on the *same* processor. Instead of busy waiting on semaphores, atomic regions are used in the implementation to provide synchronization with passive waiting. Critical section code must be executed in an atomic region. This achieves the same objective as using semaphores and simplifies the programming. Of course, this means that a client thread in an atomic region will prevent a server thread on the same processor from accessing its data structures, even though the data structures are distinct. This is the price that is paid instead of just waiting on a spin lock. To offset this brute force hold on

OKLAHOMA STATE UNIVERSITY

the processor, every thread executes its critical sections in 'blocks', whenever possible, giving a chance to other threads on that processor to execute.

Caution must be exercised when using both semaphores and atomic regions to achieve synchronization. Deadlocks can occur. For example, say a thread A on a processor is holding a semaphore and another thread B on the same processor tries to access the semaphore *in an atomic region*. Thread B will be spinning on the semaphore wait call in an atomic region. Thread A will not be able to release the semaphore as it will not get scheduled when thread B is executing in an atomic region. This results in a circular wait that leads to deadlock.

6.4 Signal Handling

The Proteus library [Koppelman 97a] does not provide any constructs for implementing signal handling. Signal handling is required when a user thread sends a request and goes to sleep waiting for a reply. Two issues need to be considered when sending a request to a site. *One* of two 'signals' should wake up the user thread: the expiration of the time out period or a reply to the request. If a reply does not arrive in a certain time period, the user thread must wake up and resend the request. If a reply does arrive, the sleeping user thread should be 'signaled' to wake up. To complicate matters, the user thread might be awake and resending the request when the reply arrives. A synchronization mechanism is needed to make sure that the user thread is woken up only if it is asleep. This is where the user state variable can be used effectively to simulate signal handling.

A user thread keeps a count of the number of unique requests it has sent so far in the variable 'UniqueRequestNumber'. If a page that the user thread wants to access is not in memory, it calls the page fault handler (which conceptually executes at the kernel level). The page fault handler sends a request to the server. If a reply does not arrive in the time out period T, the user application must resend the request. Instead of putting the user thread to sleep, another thread called the 'watchdog thread' is created [Koppelman 97b]. This watchdog thread is put to sleep for time T. The user thread is suspended. The unique request number is sent as a part of the request to the server, and also passed to the watchdog thread.

Below is pseudo code for a user thread which executes the procedures UserThread and FaultHandler, which is the page fault handler:

```
Global Variables, visible to all threads on this processor:
      PageMode: Page in  Memory or Page Not in Memory
      UniqueRequestNumber:  The  number  of  unique(  not
                        duplicate 'resent') requests sent by
                        this user.
      UserState: The state of the user thread.
Accesses to these global variables, and accesses to any page
are to be made in atomic regions.  This is not shown in this
code segment.

procedure UserThread( )
{
      begin execution
      {
            While( 1 )
            {
                  If(Page in Memory)
                        Access the Page;
                  Else
                        FaultHandler( );
                        /* Page should be in memory. */
                        Access the page.
            } /* End of while. */
      }
      end  execution
}

procedure FaultHandler( )
{
      /* Increment global variable, Unique Request Number,
       * since this is a new request being sent. */
```

```
        UniqueRequestNumber = UniqueRequestNumber + 1;

        While( 1 )
        {
              /* Indicate that the page needed is not in
               * memory. */
              PageMode = PageNotInMemory;

              /* Send a Request with the following
               * information:
               *     Page Number,
               *     Access Mode: Read/Write,
               *     Unique Request Number. */

              SendRequestToServer(PageNumber, AccessMode,
                             UniqueRequestNumber);

              /* Change user state to Suspended. */
              UserState = UserSuspended ;

              /* Start a Watchdog Thread and put it to sleep
               * for time period, T. */
              CreateThread(WatchdogThread,
                     UniqueRequestNumber);
              Sleep(WatchdogThreadID, T);

              /* Suspend the user thread. */
              Suspend(UserThreadID);

              /* User thread wakes up here.  It is woken up by
               * the watchdog thread or by a reply. */

              If(PageMode == PageInMemory)/* Reply arrived. */
                    return;
              Else /* Watchdog woke up user. */
                    ; /* Go back to beginning of loop and
                       * resend request. */

        } /*end of while */
}
```

When a reply arrives, it is put in the client's request queue. The client thread takes the reply out of the request queue and calls procedure 'ReplyFromServer' which puts the page in memory and indicates that the page is in memory. The reply has the unique request number of this request as it was sent when the request was made. The user state variable is used to check if the user is still suspended. If it is, and if it is suspended on the same request number as that of this reply, then the user thread is woken

OKLAHOMA STATE UNIVERSITY

up (if the user state is 'User Running', then the watchdog wakes up the user first). The user state is changed to 'User Running'.

When the watchdog wakes at the end of time period T, it checks to see if the user is still suspended on the same request (the watchdog thread was given the request number). If the user is suspended on the same request, the watchdog thread wakes up the user (if the user state is 'User Running', then a reply already woke up the user). The user state is changed to 'User Running'. This mechanism ensures that even if a request is re-sent multiple number of times, and even if multiple replies arrive, and invalidation messages for each reply arrive, the system is in a consistent state.

Below is sample code for the watchdog thread, which executes procedure 'WatchdogThread'. The client thread executes procedure 'ReplyFromServer' on getting a reply for a request it's application process (user thread) made.

```
procedure ReplyFromServer(integer  page, integer
access_mode, integer unique_request_number)
{
      /* A reply will ALWAYS put a page in memory.  Even if
       * this reply does not wake up the user thread, the
       * user thread will find the page in memory before it
       * re-sends the request. */
      Put Page in Memory and Indicate Access Mode;
      PageMode = PageInMemory;

      /* If the user is still suspended and waiting on the
       * same request number, change the state of the user
       * to RUNNING, and wake up the user.
       */
      If (UserState == UserSuspended
            And
           UniqueRequestNumber == unique_request_number)
      {
            /* Change state of user to running and
             * wake up the user thread. */
            UserState = UserRunning ;
            Wakeup(UserThreadID) ;
      }
}

procedure WatchdogThread(integer unique_request_number)
```

```
{
        /* WatchdogThread wakes up when the time out period
         * expires.  If the user is suspended on the request
         * number for which this watchdog was created, wake
         * up the user. The user thread will resend the
         * request.*/

       If ( UserState == UserSuspended
             And
            UniqueRequestNumber == unique_request_number)
       {
             /* Change state of user to user running and
              * wake up the user thread. */
             UserState = UserRunning ;
             Wakeup(UserThreadID) ;
       }
}
```

OKLAHOMA STATE UNIVERSITY

CHAPTER VII

ASSUMPTIONS, INTERPRETATION AND OPTIMIZATIONS

It is impossible to simulate a real system with 100% accuracy. Some assumptions have to be made along the way. This chapter describes the assumptions made about the environment and how some events are simulated. The only information available about Reliable Mirage+ [DeMatteis 96] is that in the open literature. Some parts of Reliable Mirage+ have been modified slightly in the simulation, either to reduce the implementation complexity or to optimize the system. Justifications for the modifications are provided, whenever possible.

7.1 Assumptions About the Simulated Environment

This section lists assumptions made about the simulation environment for the single and multi-server cases. These are assumptions about the network, the type and number of failures allowed, the failure mechanism, and the memory.

1. The sites in the network have independent failure rates. The communication links may also fail independently of each other. The system is equipped to recover from a failstop, single-site failure. It is not equipped to recover from a communication link

failure, which may partition the network into subnets [Theel and Fleisch 96a] [DeMatteis 96].

2. There is enough main memory at every site to accommodate all the pages in the database. Page swapping algorithms are not implemented as adding page swapping may add extra time overhead (assuming there is enough secondary memory) and delay in the processing of requests, which may potentially shadow the true performance of the multi-server coherence protocol under investigation.
3. In Reliable Mirage+ [DeMatteis 96], once a site fails it goes out of the cluster. The system does not wait for that site to come back up after some down time. Failures are permanent failures. In case of a server failure, a new server is elected using standard election algorithms [DeMatteis 96]. In the simulated system, both single and multi-server, all failures are transient. A transient failure is one where a failed site comes back up after a randomly generated down time. Recovery of the system starts at the end of this down time. This simplifying assumption was made because of the implementation complexity of election algorithms, where the consensus problem might occur in the case of multiple cascading failures. Moreover, none of the literature on Mirage+ addresses the specific election algorithms used. If a specific election algorithm were chosen for this simulation, the overhead or the impact of the algorithm would have been difficult to examine.
4. All failures in the simulation and in Reliable Mirage+ are failstop failures. The simulated system is equipped to recover from a single site failure. The effect of cascading failures is not examined.

5. The original Reliable Mirage+ system runs on twelve IBM Ps/2s connected by a 10Mbps Ethernet. A failure is detected in a cluster using IBM's AIX/TCF, the Transparent Computing Facility, which allows a network of personal workstations to act as a transparent cluster of machines. Continuous topology monitoring is used to detect any changes to the cluster. Probe messages are multicast periodically to check if a site is running. If a site does not respond, a failure is assumed and the recovery process begun [DeMatteis 96].

   In the simulated system, continuous polling is not done to detect a failure in the cluster. A single 'failure thread' is created for the complete distributed system. The failure thread uses randomly generated data to determine when and for how long a specific site must go down. This failure thread broadcasts the failure of this site to all servers or to the single server when there is only one. When a server gets this message, it looks at its cluster to determine if the failed site belongs to its cluster. If so, it sends a message to the failed client site to go down for the specified down time. Once the down time is over, the server broadcasts recovery messages to all its clients to start the recovery process. Inter-processor interrupts are used by the failure thread to send messages to all servers, and by the servers to send messages to their clients.

   This processing overhead and message overhead *is* cycle counted. This will add to the execution time of the system, as it is included in the 'user code'. It is not straightforward to implement 'continuous topology monitoring' without sending messages across the network. All messages sent across the network are cycle counted, since it is under the control of the simulation engine. This extra overhead is

assumed to be minimal. However, no measurements are made to analyze this overhead accurately.

6. Every client site has only one user thread in the implementation.
7. Migration does not guarantee that the load on the servers is distributed equally. It is done to increase throughput by reducing the number of messages required to service a request. For instance, a request that goes out of the cluster to be serviced, will require more messages to be sent across the network than a request that is serviced within a cluster.
8. Even if a client and a server reside on the same node, they do not share memory or the page table for the pages they have access to. They have 'exclusive' memories. This is because the client at this site may migrate to a different cluster, and send its requests to a server which is not at this node.
9. In the simulated environment, it is assumed that messages are ordered and atomic broadcasts are possible. These assumptions are necessary to solve the consensus problem which occurs when a server has to hand-over a client to another server in an error-prone environment (refer to Section 4.5.4, page 44 for details).
10. Say a client C_1 is migrating from cluster 1 to cluster 2, which have servers S_1 and S_2, respectively. When S_1 hands over C_1 to S_2 (Figure 8, page 46), it broadcasts a message to C_1 telling it to join S_2, and to S_2 telling it to accept C_1. An assumption that client C_1 will not go down from the time that S_1 sent its broadcast message until S_2 accepts C_1 is made. This is because in the simulation, the event of a client going down is initiated by its server (see Item 5 above). During the time period when it is assumed that C_1 will not go down, C_1 does not belong to *any* server.

11. Say a client C_1 is in the 'process' of migrating from cluster 1 to cluster 2 (it might be handing over clock privileges) and recovery begins in cluster 1 before its server S_1 'hands over' client C_1 to server S_2. In this situation, migration of client C_1 is aborted. However, once S_1 hands over C_1 to cluster 2 (see Item 9 above), the migration is considered completed from S_1's point of view. After this hand-over, C_1 cannot participate in the recovery of cluster 1.

7.2 Differences in Interpretation

This section gives details about modifications made to the protocol design in Reliable Mirage+ [DeMatteis 96] and its simulation as reported in this thesis. These differences usually arise either because of lack of more specific information or because of implementation complexity. The effects of modifications made are not examined or analyzed. In one case (see Item 5 below), the simulated system is modified slightly to improve the availability of data.

1. Only eager recovery [DeMatteis 96] is implemented in the simulation of both the single and multi-server environments. Eager recovery is when the server ensures that every page has a backup before restarting the system. The system is stable and can survive another single site failure since a backup for every page exists.
2. In the scenario where a client blocks itself after sending a request, if the server goes down, the client would stay down forever because the server's request queue would be lost [Theel and Fleisch 96a]. Reliable Mirage+ solved this problem by making the clients time out and resend the request if a reply does not arrive within a certain time-

period. This solves the problem, but it could result in the server being flooded with duplicate requests. In Reliable Mirage+, the server uses sequence numbers to eliminate duplicate requests, if it has already serviced the request [DeMatteis 96]. In the simulation, sequence numbers are not used to eliminate duplicate requests. When a server gets a duplicate request, if its data structures indicate that the requesting site already has the requested page, it ignores the request. Otherwise, it forwards the request to the relevant clock site. The clock site ignores the request if its data structures indicate that the requesting client already has the page. Otherwise, it sends the page to the client. A slight overhead could be caused when the clock site has already serviced the request, invalidated that page later, and then gets the duplicate request. In such a case, the page would be sent twice to the requesting client.

3. Reliable Mirage+ has a time window mechanism at every clock site. This means that a clock site has a hold over a page during this time window. Even if a clock site gets a message from the server to invalidate a page, it does not invalidate the page [Fleisch and Popek 89]. This was done to reduce thrashing. This has not been implemented in either the single or the multi-server simulations because of implementation complexity.
4. In Reliable Mirage+, read requests for the same page are put in a batch and are serviced simultaneously by the server. Write requests are processed sequentially [Fleisch and Popek 89]. In the simulation, both read and write requests are processed sequentially. The impact of this modification on request latency is not examined.
5. In Reliable Mirage+, trailer pages are created when a clock gives up its clock privileges at the arrival of a write request. The clock site becomes a trailer

[DeMatteis 96]. In the simulation, trailer pages are created in two situations. As in Reliable Mirage+, they are created when a clock services a write request. They are also created when a clock downgrades from a writer to a reader because of a read request. Since the clock most probably just wrote to that page, it downgrades to a reader and sends a read page to the requesting client, say C_r. It also asks the client C_r to become the latest trailer page. Therefore, in this case, the trailer page is not the 'latest invalid page' but it is the last modified version. It is a valid page, which implies that C_r can access this page in the granted read mode. The changes made in the last write operation now have a backup at the new reader site. Data availability is increased in the sense that the latest version of this page will still be available even if the read page at C_r is invalidated and the clock site goes down. The latest available version will be the trailer page at C_r.

6. In the simulation, messages are ordered and requests are serviced atomically. This means that a client or a server cannot crash while servicing a request. They can crash either before or after a request is serviced. Every client and server maintains a count of the running threads related to its functions that can potentially change any data structures at that site. When a failure occurs at that site, such threads have to exit before the failure is deemed to have occurred. Also, a client or server cannot start recovery until such threads exit. This ensures that the data is in a consistent state and not partially updated, before recovery starts.

7. In Reliable Mirage+, it is not clear when a server changes a clock site for a page [DeMatteis 96]. Typically, the clock site for a page changes when a write request is serviced. The clock site may be changed to the requesting client when a write request

OKLAHOMA STATE UNIVERSITY

arrives. Or, the clock site may be changed to the requesting client *after* the page is installed at the requesting client. The latter is implemented in the simulation.

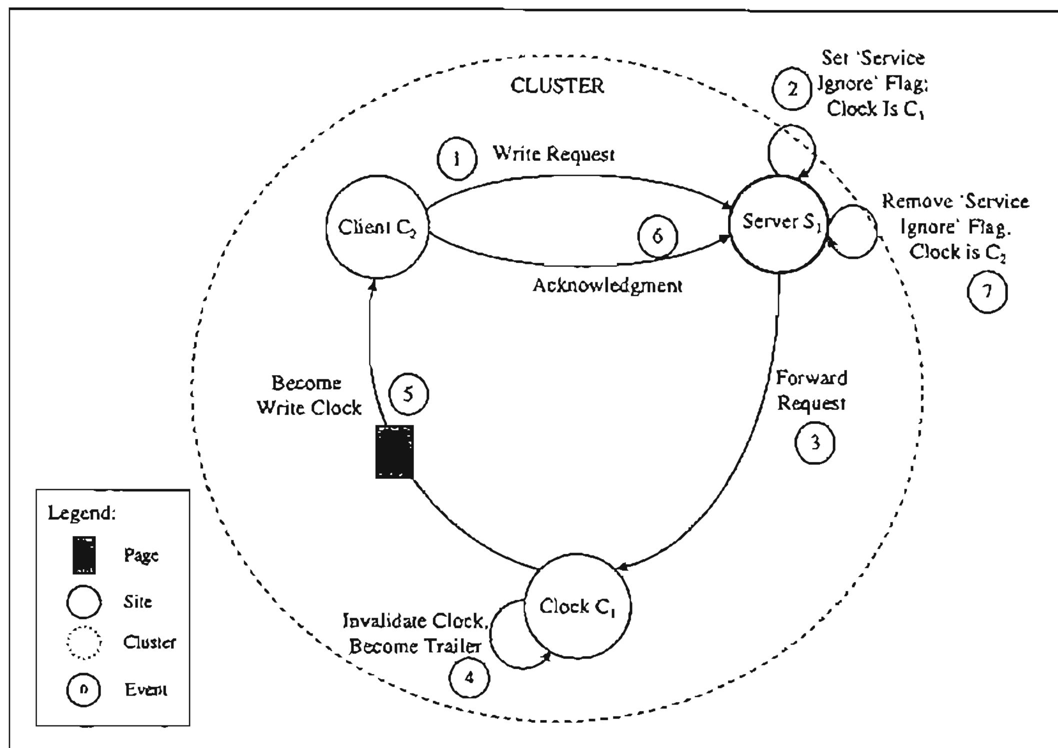

Figure 13. Change of Clock Site During Write Request

Let us consider a situation as depicted in Figure 13. Client C_2 makes a write request to its server S_1. When the write request arrives, the server forwards the request to the clock site C_1 and sets a 'service ignore' flag for that page. It does not change the clock site to the requesting client, C_2, at this point. The clock site becomes a trailer page and gives up clock privileges to the requesting client. When C_2 gets the message, it becomes the clock site and sends an acknowledgment to the server. On getting this acknowledgment, the server changes the clock site to the new clock site C_2, and removes the 'service ignore' flag. If any request(s) for this page arrived at the

server site when the service ignore flag was set (between events 2 and 7 in the figure), the server would ignore the request(s) because the clock was 'in transit'. Ignored requests will eventually time out and be resent to the server.

If the 'service ignore' method is not used, two race conditions can occur. All numbers in this and the following paragraph refer to the events in Figure 13. The clock site is changing from site C_1 to site C_2. The server still has site C_1 listed as the clock site for the page under consideration. Assume another request for this page arrives between events 2 and 7. If the server forwards this request to C_1, C_1 may have already sent event 5 by the time it got this request. The site C_1, which is now a trailer, can either ignore the request or forward it to the new clock site C_2 (it would then have to keep track of the new clock site). Or, alternatively, assume that the server changed the clock site to site C_2 at event 2 when it forwarded the write request to the current clock C_1. In this case, events 6 and 7 in the figure would not occur. Now, assume S_1 gets a request for this page after event 2 and before event 5 reaches C_2, and forwards the request to C_2. And, further assume that C_2 gets this request before it gets event 5. Such conditions make the system more complex. Therefore, the 'service ignore' method was implemented in the simulation.

7.3 Optimizations

A few optimizations were made in the simulation of the single and multi-server systems. The system was optimized to reduce the traffic on the network. This section describes the optimizations and gives some details about a few race conditions that can

occur. For example, an invalidation message for a write-update page can arrive at a client site even before the page itself arrives. If the invalidation message is ignored and the page arrives later, an inconsistent system will result.

1. In Figure 14, client C_{11} is a pseudo read clock for a page belonging to cluster 2. Client C_{11} can service any read requests, from its own cluster, for this page. C_{11} is called a pseudo read clock because it only maintains a list of readers and does not keep track of the latest trailer version for this page. C_{11} stays a clock for this page until a write request for this page is made. Client C_{12}, in cluster 2, is the read clock of this page. Client C_{13} in cluster 1, makes a write request for this page. When the request arrives at clock site C_2, it checks to see if the requesting server S_1 is already a reader. If it is, the page is not sent to that server. Instead, a message is sent telling S_1 to invalidate all its readers and give a write copy to the requesting client (2a in the figure). Message 2a is sent by C_2 to S_2 (not shown in the figure) and forwarded by S_2 to S_1. Upon getting message 2a, server S_1 sends a message to its pseudo read clock (which may have been the only reader in the absence of C_{12} being a reader). Site C_{11} invalidates all readers (4 in the figure) and sends a write-update copy to the requesting client C_{13} (5 in the figure). This reduces the size of the messages sent across the network because the page itself was not sent in messages 2a and 3.

2. In Item 1 above, a reply to a write-update request may be routed through a pseudo read clock. A race condition may occur if an invalidation message arrives for the write-update page, before the page itself reaches the site. In Figure 14, assume the clock site C_2 sends message 2a, giving write-update permission, and then sends message 2b, invalidating the write-update page. The reply to the write request made

OKLAHOMA STATE UNIVERSITY

by C_{13} is routed via $S_1 \rightarrow C_{11} \rightarrow C_{13}$, but the corresponding invalidation message is routed via $S_1 \rightarrow C_{13}$. If C_{11} has a heavy load, it is possible that the invalidation message (event 2b) reaches C_{13} before the write-update page itself reaches C_{13} (message 5).

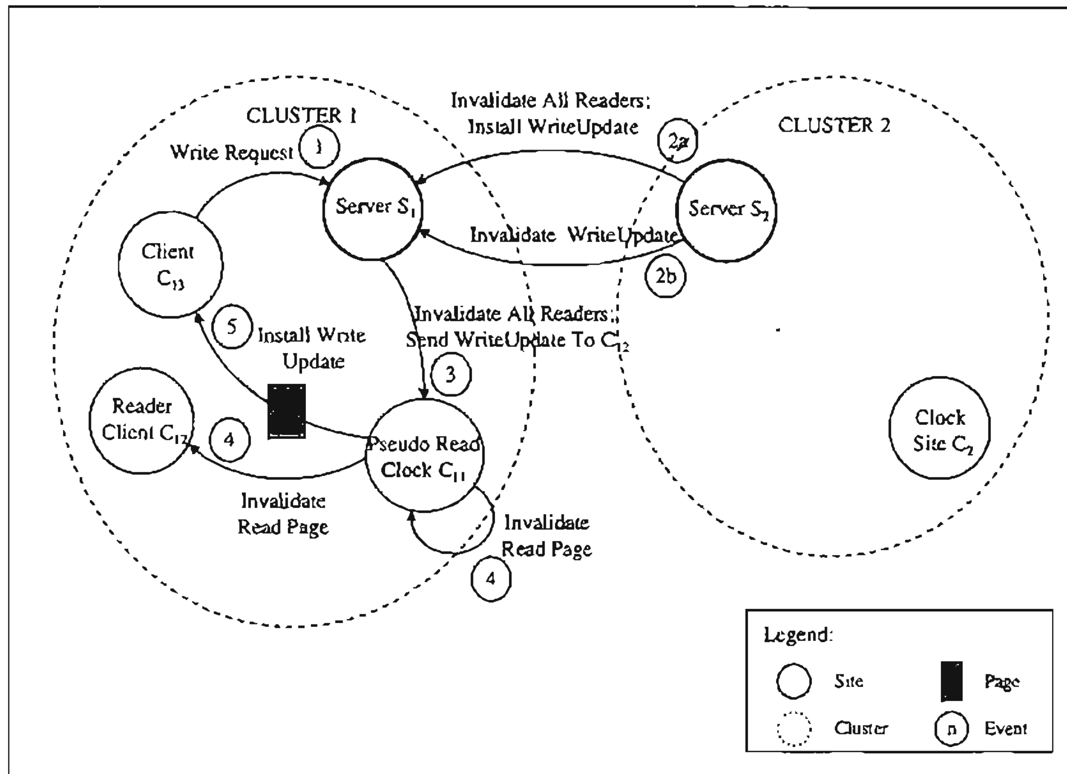

Figure 14. Out-Cluster-Reader Server Requests Write Page

C_{13} cannot just hold onto the invalidation message waiting for the page to arrive. There are two ways a write-update page can be invalidated. The first is when the time for which the page was loaned is over and the clock site C_2 sends an invalidation message. The second is when the user application at C_{13} finishes writing to the write-update page, invalidates the page from its memory, and returns the page to cluster 2. For instance, if C_{13} held on to the invalidation message 2b, since it did not have the

page in memory, C_{13} cannot be sure that the page has not arrived if it is not in memory. If the page did not arrive, 2b could be held at C_1 until 5 arrives. Alternatively, the page may have arrived, and the user at C_{13} may have returned it. This would mean that the invalidation message 2b would be held in C_{13}'s request queue indefinitely (this would cause an infinite loop if the client at C_{13} kept trying to service the message).

Since the invalidation message cannot be held in C_{13}'s request queue, it is held in S_1's request queue. When S_1 forwards message 3 to client C_{11} telling it to forward the write-update page to C_{13}, S_1 changes its page table to indicate that a write-update page is supposed to have been at C_{13}, but it has not reached the client site C_{13} as yet. When C_{13} gets the write-update page, it sends an acknowledgment to its server S_1, at which point S_1 sets a flag to indicate that the write-update page has reached C_{13}. Now, if the invalidation message arrives at S_1 and the page has not reached C_{13}, the invalidation message is modified to indicate that it is waiting for the write-update page to arrive at C_{13}. This invalidation message is then held in S_1's request queue. Once the acknowledgment arrives from C_{13}, this invalidation message can be forwarded to client C_{13}. In case C_{13} itself invalidates the page and returns it, S_1's page table will indicate that this write-update page does not exist in its cluster. Now, if the invalidation message (which is waiting on C_{13}) gets scheduled at S_1, it finds that the page is not in this cluster and it is ignored.

3. When a clock page was in transit during a write request, the server marked the page with the flag 'service ignore' (Figure 13, page 81). Here the time period for which the ignore flag is set is relatively small, so any client that makes a request for this page

during this time period will resend the request. Moreover, when a clock site has loaned a write-update page to another server or when a clock page is in transit because its clock site is migrating to another cluster, the server of the page does not service any requests for the page. However, the wait for the service to be turned back on can be very long. Instead of having the requesting client send many duplicate requests, which the server might anyway ignore, the server sets a 'service later reply' flag. That is, when a clock site is in transit because of its client migrating, or when a write-update page is loaned to another cluster, the server which owns this page marks the page with a 'service later' flag. Now if any request for a page arrives when this flag is set, the server sends a *reply* telling the requesting client to wait and resend the request later. This is similar to adding *flow control* to the system [Patterson and Hennessy 96]. Traditionally, flow control ensures that the sender does not overwhelm the receiver by sending data at a rate faster than the receiver can process the data. The idea is to use feedback to tell the sender when it is allowed to send the next packet. When a server tells a client to resend a request later, network congestion is reduced. The server will be able to do useful work instead of pulling out requests that it will have to ignore.

4. During migration, when a migrating clock is handing over its clock privileges, it must hand-over every clock in a single atomic region. This is done to ensure that another thread on the same processor does not attempt to make any changes to that page or to the data structures related to the page.

CHAPTER VIII

RESULTS

8.1 Basic Configuration

This thesis work simulated Reliable Mirage+ [DeMatteis 96] and the new multi-server protocol. The multi-server system is a dynamic system, where one or more clusters might be in the recovery phase, and one or more clients might be in the process of migrating. If the throughput of the 'complete' system is measured, it would be difficult to pin point the exact part of the protocol that might cause a change in throughput. Experiments are run with specific configurations, say without any failures, to measure the performance of the multi-server protocol.

A program driven simulator was developed which carried out a comparative study of WI, WB, and BR coherence protocols using simulation [Shah 97]. The simulation runs in this work used some standard applications and measured the number of messages that were sent across the network for each application, using variations of each coherence protocol.

In this thesis work, a synthetic application was used to evaluate the performance of the multi-server protocol versus the single server protocol in Reliable Mirage+

[DeMatteis 96]. Standard applications for parallel processors were not used, since the multi-server protocol assumes that the applications running on the system exhibit locality of reference towards a particular segment for a specific time period (see Sections 3.3 and 4.1). If a standard application were used, the results would be subject to the discrepancies caused by the method used to partition pages into segments and the creation of clusters. To avoid such discrepancies, a synthetic application was used. The input used was generated using a random number generator with specific distributions.

The performance of a multi-server architecture versus a single server architecture was measured. A user application, that runs on this system, accesses data from pages in the database, probably does some computation until it needs to access some more data, and so on. When a user application needs to access a page, it can directly read or write to the page if the page is in its local memory and it has the required permissions to access the page. If the page is not in its local memory, the underlying DSM system, at the kernel level, kicks in. The page fault handler at the kernel level will get the page from a remote site. The process of how a page is obtained from a remote site depends on the DSM coherence protocol used. The underlying DSM protocol is transparent to the user application running at a client site.

Request latency is the time when a client makes a request for a page to the time it gets the page in its local memory. Of course, if the page is already in its local memory, the request will be satisfied almost immediately. The request latency of a request for a page that is not in the local memory of the requesting client depends on the message latencies of all the messages that need to be sent to obtain the page, and the execution time of the augmented code of the coherence protocol. Message latency depends on the

network congestion of the system at that point, and is calculated by the underlying simulation engine of Proteus. The communication costs are analyzed by measuring the number of messages generated by the DSM protocol being simulated and the amount of traffic generated by it.

A synthetic application is used to run experiments on the single server and multi-server protocols. The inputs to this synthetic application consist of the following: the time when an application needs to access a page, which page from the database it needs to access, and whether it should read from or write to the page. All these values are randomly generated using specific distributions. For the single server case, measuring the average request latency at the end of a simulation run, when compared with the average request latency of another configuration, can give some indication about the performance of the system.

To be able to compare the performance of the single server protocol versus the multi-server protocol, the input to the multi-server protocol is further divided into categories. In the multi-server case, each user application is assumed to make some percentage of its total requests for pages belonging to its cluster, and the rest of its requests for pages belonging to all the other clusters. Each run of a multi-server system is configured so that all the clients make X% of requests for pages from their respective clusters, and (100- X)% of requests for pages not in their own clusters. If a client makes more than say N number of requests in time M (the migration interval) to a specific cluster which is not its own, it migrates to that cluster. The system is set up so that a client makes X% of requests to its own cluster, say cluster 1. That is, X% of all pages accessed belong to cluster 1. When this client migrates, say to cluster 2, it will then make

X% of requests for pages in its new cluster, cluster 2. This percentage of requests belonging to its cluster represents its locality of reference. In effect, it is assumed that its locality of reference for data within a cluster stays constant. That is, if it makes X% of its requests for pages in cluster 1 when in cluster 1, it does not make Y% (Y <> X) of its requests for pages in cluster 2 when it migrates to cluster 2.

The input is classified according to the percentage of its in-cluster requests, only in a multi-server case. As is intuitive, a client in a single server architecture will make 100% of all its requests from its own cluster, the single cluster, all the time. There are no migrations in a single server case. The performance of a single server protocol is compared with varying percentages of in-cluster requests in a multi-server protocol.

The different kinds of messages are divided into classes: normal messages, migration messages, and recovery messages. Normal messages are request messages, reply messages, and messages sent by the clients and the server(s) to ensure sequential consistency and to adhere to the particular coherence protocol. Migration messages are messages sent by the clients and the server(s) when a client migrates from one cluster to another. These messages are sent only in a multi-server environment. Recovery messages are messages sent when a cluster is recovering. These messages are sent in both single and multi-server environments. In a multi-server environment, normal messages are further divided into in-cluster and out-cluster messages. In-cluster messages are messages sent to various sites within a cluster when a client makes a request for a page belonging to its cluster. However, when a client makes a request for a page not belonging to its cluster, its server sends messages to the relevant server to obtain the page. The cluster which owns the page might have to send messages to various sites to satisfy

this request from a different cluster. Such messages are classified as out-cluster messages.

Each message class contains data messages and control messages. A data message is a message in which a page is actually sent across the network. A control message is a message in which a page is not sent across the network. The data collected for each kind of message (data or control) consists of the number of messages and the total bytes sent across the network.

8.2 Experiments

8.2.1 Performance Analysis of Single Server Architecture versus 2 Server Architecture

Experiments were run to compare the performance of a single server system versus a double server system, during normal operation. In the double server case, migrations of a client from one cluster to another were allowed. However, since the effect of failures was not being analyzed in this experiment, no failures were simulated.

The basic system configuration is given in Table I. There were no migrations in the single server case. In the 2 server case, the migration algorithm given in Table I was simulated. In the single server case, each client accessed 100% of the database. The 2 server system had 2 clusters, with each cluster owning 200 pages. Initially, each cluster had 16 clients. Depending on the migration of clients from one cluster to the other, the total number of clients in each cluster was dynamic throughout a run. In the 2 cluster system, various runs were executed varying the percentage of in-cluster requests made by

Simulation Parameter	Value
Number of Processors	32
Number of Clients	32
Mean Inter Request time	500 simulation cycles
Percent of Read Requests	80
Percent of Write Requests	20
Number of Pages	400
Page Size	1000 bytes
Total Simulation Time	1,073,741,824 simulation cycles
Migration Algorithm	Migrate if > 10 requests in 700,000 simulation cycles.

Table I. Basic System Configuration Parameters

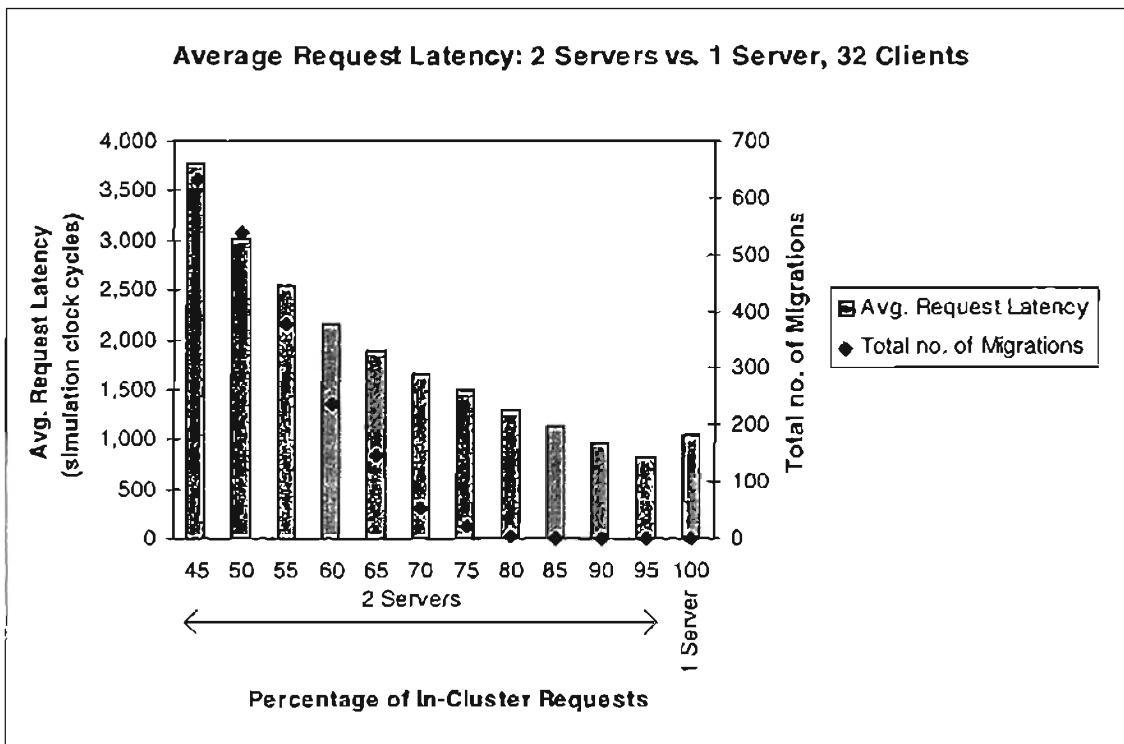

Figure 15. Effect of 2 Servers on Average Request Latency: As the locality of reference (percent of in-cluster requests) increases in a 2 server system, lesser migrations occur and lesser out-cluster requests are made. This reduces the average request latency. Single server latency falls between 85%-90% of in-cluster requests in a 2 server system.

clients. The percentage of in-cluster requests was varied from 45% to 95%, in increments of 5. Clients in the single server system made 100% of in-cluster requests (there being no other cluster).

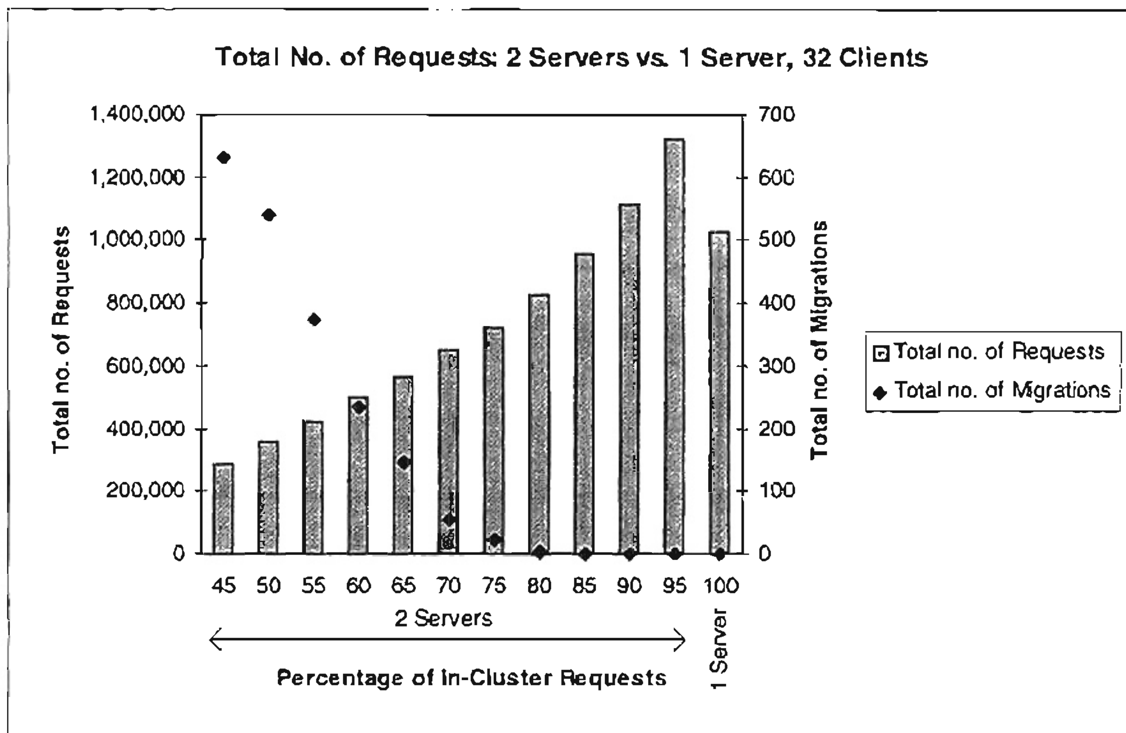

Figure 16. Effect of 2 Servers on Total Number of Requests: As the locality of reference (percent of in-cluster requests) increases in a 2 server system, lesser migrations occur and lesser out-cluster requests are made. This increases the total number of requests made. Single server performance falls between 85%-90% of in-cluster requests in a 2 server system.

Figures 15 and 16 illustrate the effect of varying the percentage of in-cluster requests, in the 2 server case, versus the single server case. When the percentage of in-cluster requests is at the lower end (45%), a client makes more out-cluster than in-cluster requests. This leads to the clients migrating more often, as is indicated in the total number of migrations in the 2 server case with 45% of in-cluster requests. The average

request latency is high in this case for two reasons: a majority of the requests are out-cluster requests, that have extra message and time overhead, and the larger number of migrations implies that the sites, both server and client, are spending more time migrating than servicing requests. The performance of the single server protocol falls between about 85% to 90% of in-cluster requests in the 2 server protocol.

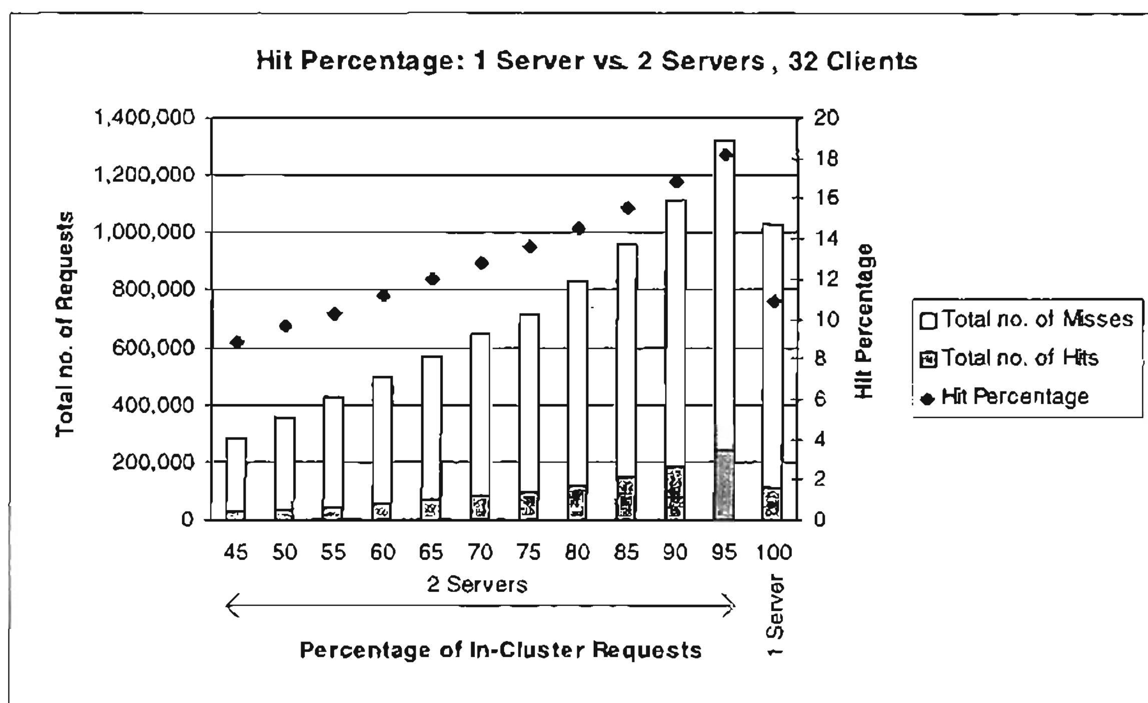

Figure 17. Effect of 2 Servers on Hit Percentage: Hit percent is the percentage of hits when a page is accessed. Though the hit percent of the single server falls between hit percent of 55%-60% of in-cluster requests in a 2 server system, the total number of requests made in a single server system is between 85%-90% in a 2 server system. This can be attributed to the overhead of a 2 server system.

The hit percent, which indicates if a page is in the local memory of a client when it needs to access the page, is shown in Figure 17. Comparing the total number of requests and the hit percent of the single server versus that of the 2 server system, even

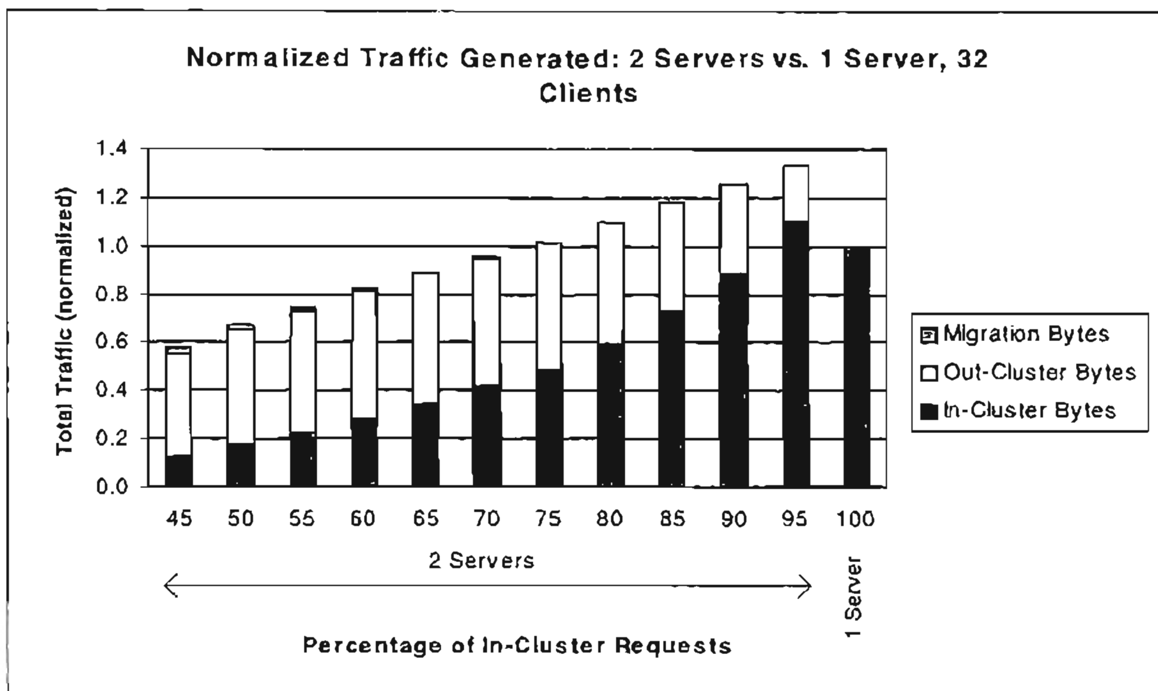

Figure 18. Total Traffic Generated, Normalized to Single Server: As the percentage of in-cluster requests increases, the out-cluster traffic and the migration traffic decreases. The higher the locality of reference (percent of in-cluster requests), the greater will be the number of requests serviced within a cluster.

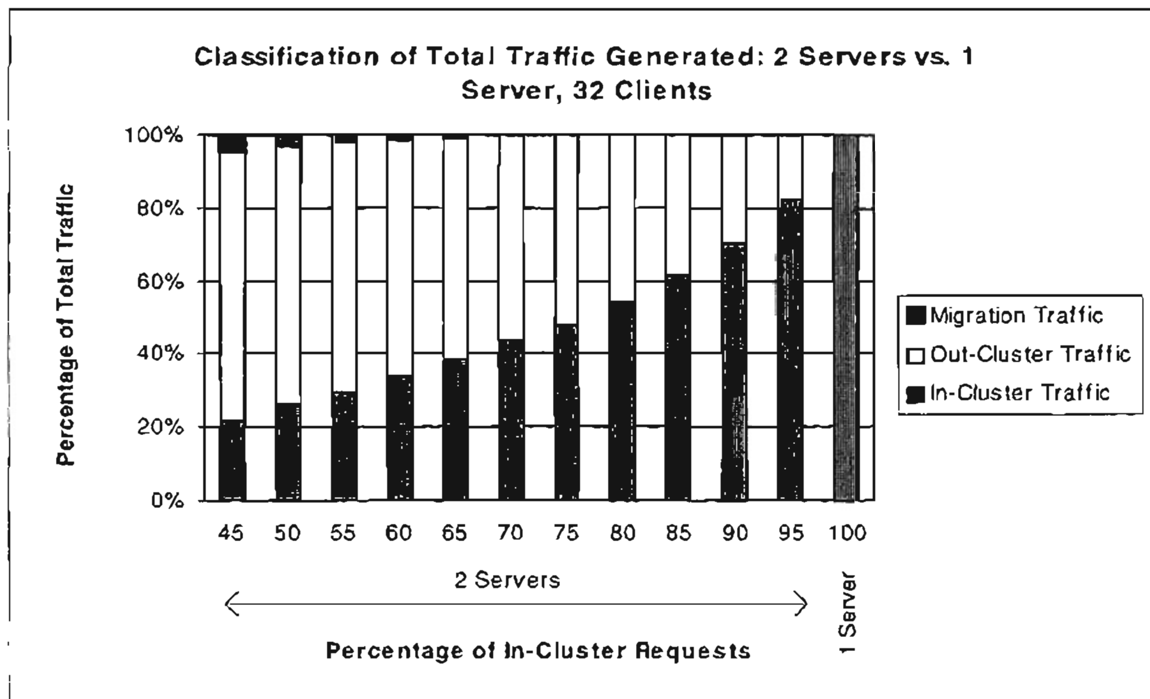

Figure 19. Classification of Total Traffic Generated as In-Cluster, Out-Cluster and Migration Traffic: A single server will have only in-cluster traffic. As the percent of in-cluster requests increases, the out-cluster percentage decreases. After 65% in 2 cluster system, the migration traffic is 0.

though the hit-percent of the single server is approximately equal to the hit-percent of the 60% in-cluster case, the total number of requests made in the single server run is more than double that of the 60% case. This is because the out-cluster traffic generated from 45% in-cluster to about 75% in-cluster is more than 50% of the total traffic generated, as shown in Figures 18 and 19. Therefore, it can be concluded that in the 2 server case, the overhead of servicing out of cluster requests nullifies the effect of parallel processing until the locality of reference is about 85% in-cluster.

8.2.2 Locality of Reference

Experiments were run to measure the effect of locality of reference on protocol performance. The basic system as configured in Table I, was with varying the number of servers. The migration algorithm for each multi-server case was the same. The initial cluster configuration for each configuration is shown in Table II.

Number of servers	1	2	3	4
Number of clients per cluster	32	16	11	8
Number of pages per cluster	400	200	133	100

Table II. Different System Configurations

Figure 20 illustrates the effect of the size of a cluster on the total number of requests, normalized to the single server case. The smaller the cluster the greater will be the total number of servers, the request latency for an in-cluster request will be lower, thus increasing the overall number of requests serviced in a run. Moreover, the effect of

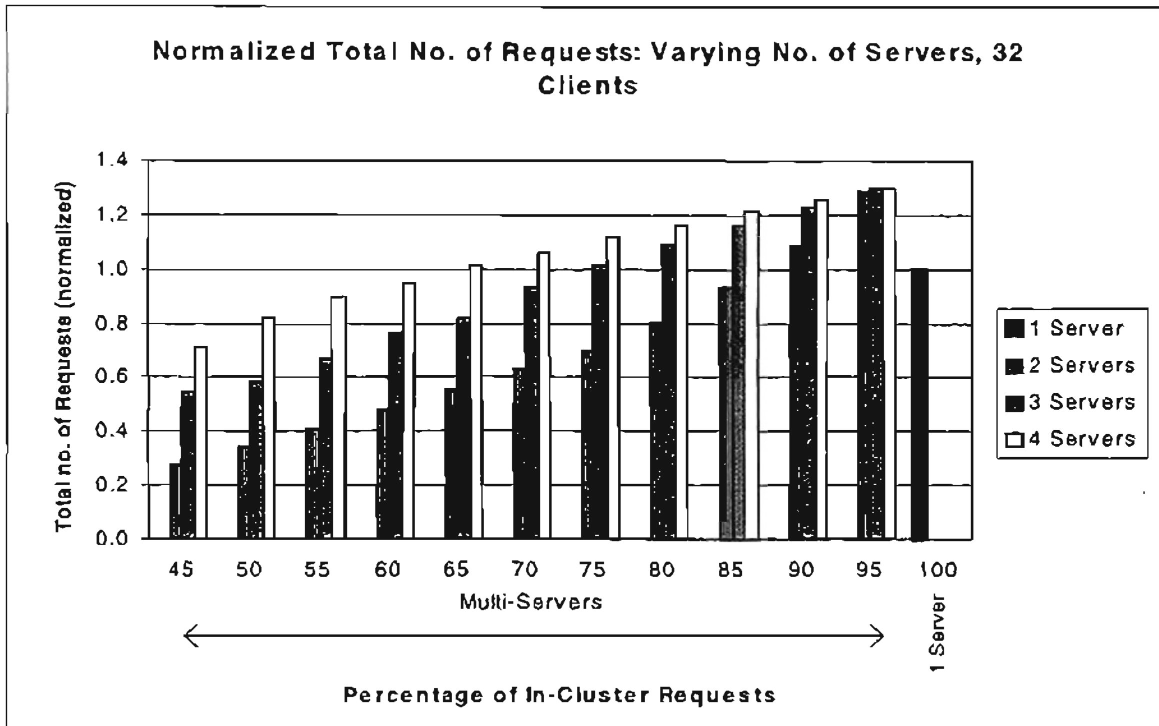

Figure 20. Effect of Locality of Reference on Total Number of Requests, Normalized to Single Server: As the size of a cluster decreases when the number of servers increases. The smaller a cluster, the better will be the overall performance of the system. Single server falls between: 85%-90% of 2 server, 70%-75% of 3 server, and 60%-65% of 4 server system.

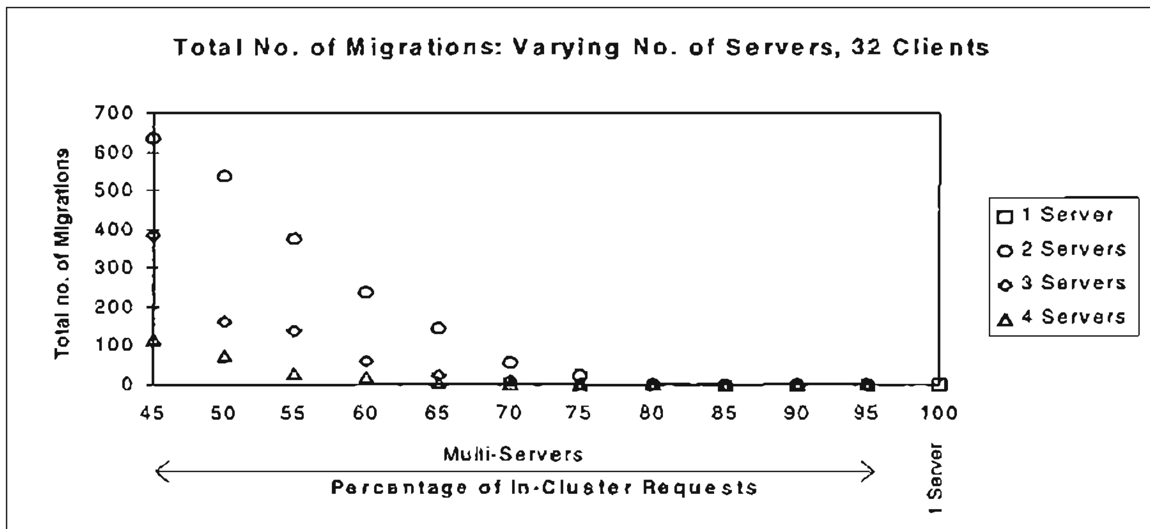

Figure 21. Effect of Locality of Reference on Total Number of Migrations: As the size of a cluster decreases, the total number of migrations decrease from 45%-75% of in-cluster requests. The number of migrations after about 75%, irrespective of the number of servers, is close to 0.

parallel performance by multiple servers is visible in this figure. Single server performance falls between: 85%-90% of 2 server, 70%-75% of 3 server, and 60%-65% of 4 server system. It should be noted that in each of these configurations, the locality of reference will depend on the number of pages owned by a client's server. Therefore, this figure does not reflect the performance of the *same* user application (having the same data access pattern) on systems with varying number of servers. As the size of a cluster differs across configurations, the data access pattern of a user application, which depends on its locality of reference *within* its own cluster, will differ. An applications locality of reference, which is a measure of the data it accesses frequently, is restricted to lesser pages as the size of the cluster reduces. Therefore the performance of the system improves.

Figure 21 shows the total number of migrations with varying configurations. The maximum migrations occur in a 2 server system. As the number of servers increases beyond 2, out cluster requests can be made to more than one server, thus reducing the maximum number of requests a user makes to one single outside cluster. This reduces the total number of migrations. Beyond 75% of in-cluster requests, irrespective of the cluster configuration, the number of migrations is 0. This is because a client will always request the maximum number of pages from its own cluster during the migration interval.

8.2.3 Multiple Servers

Section 8.2.2 analyzed the difference in performance as the locality of reference of an application changes. It was seen that as the number of servers is increased, the

performance of the system improves. The next instinctive question that arises is the performance of the same user application, with the same locality of reference, on systems with different number of servers.

Experiments were run with configurations shown in Table II. However, irrespective of the configuration, a user application on this system accessed the complete database, the pages belonging to all the servers, with equal probability. That is the data access pattern was the same as that in a single server system, where a user application

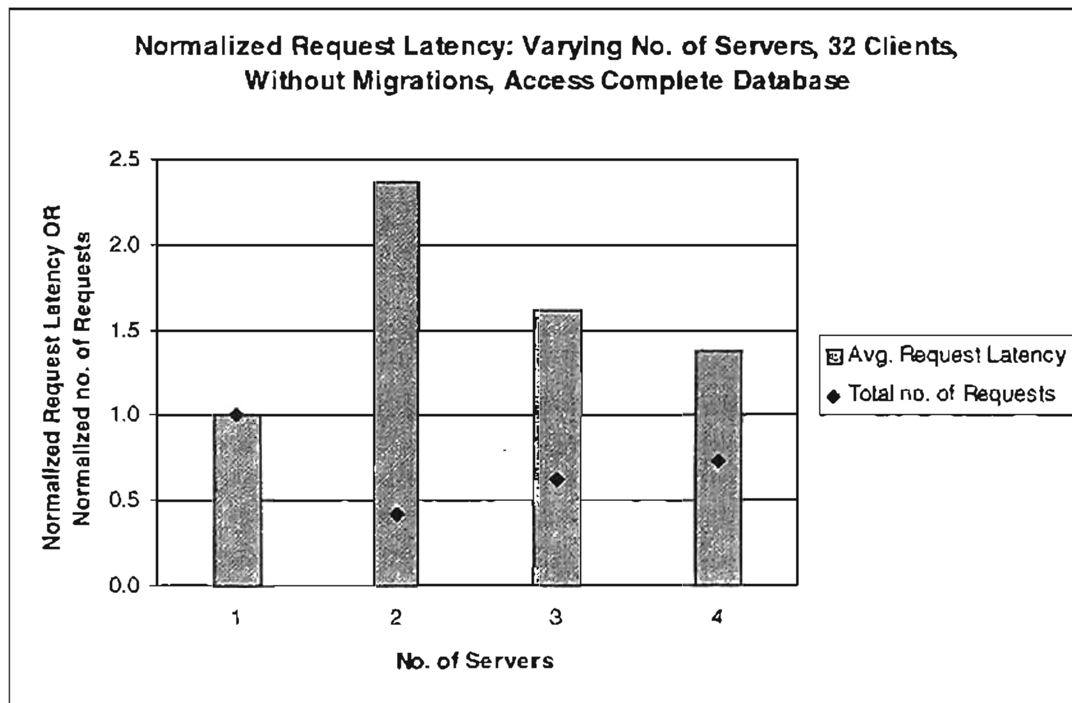

Figure 22. Effect of Multiple Servers on Request Latency, Normalized to Single Server: A user application running on each configuration has the same locality of reference. A user application accesses the complete database, and no migrations are allowed. Even if the number of servers is increased, no significant performance benefits are realized by parallel processing by multiple servers. The single server system outperforms the multi-server systems.

would make 100% in-cluster requests for the pages in the complete database, i.e., 400 pages. In the multi-server systems, no migrations were allowed.

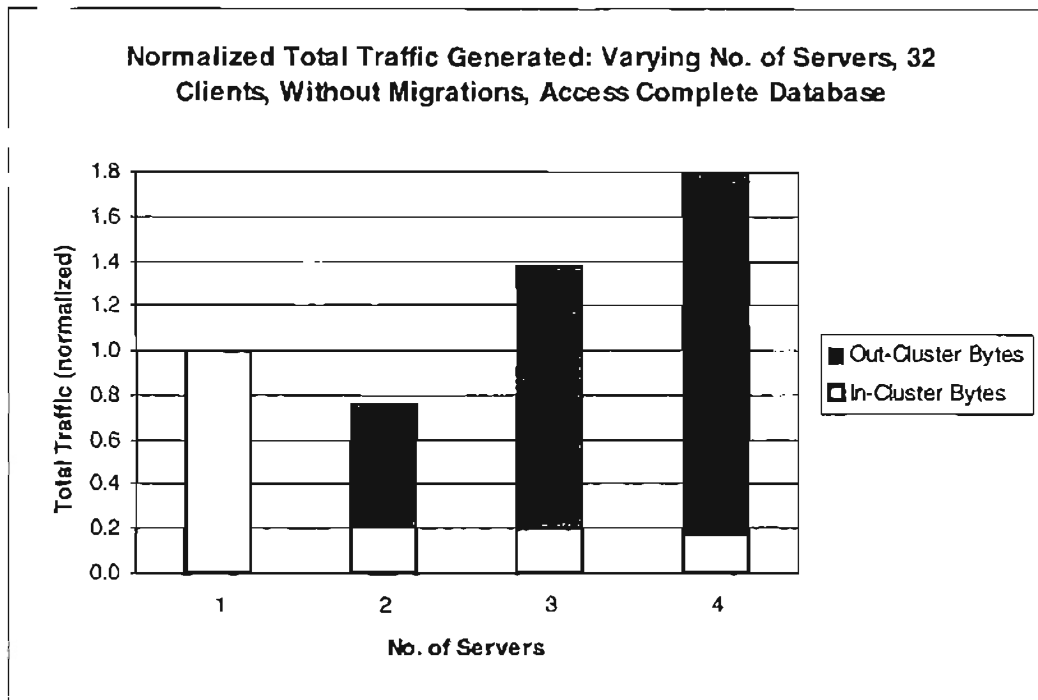

Figure 23. Effect of Multiple Servers on Total Traffic Generated, Normalized to Single Server: A user application running on each configuration has the same locality of reference. A user application accesses the complete database, and no migrations are allowed. When the number of servers is increased, the out cluster traffic increases, causing a high overhead because of cluster-to-cluster communication. The total percentage of out-cluster traffic is as follows: 72% for 2 server, 85% for 3 server, 90% for 4 server.

The variation in the request latency when the same user application is run on different configurations can be seen in Figure 22. A single server system outperforms all the multi-server systems, in terms of the average request latency, which effects the total number of requests that can be serviced in a run. Figure 23 gives insight into this performance analysis. Figure 23 shows the in-cluster traffic and the out-cluster traffic generated in each system. In a 2 server system about 72% of the total traffic is out cluster

traffic, in a 3 server system this percentage is about 85%, and it is 90% in a 4 server system.

An out-cluster will generate considerably more messages than an in-cluster request, adding to the overhead. Because of the high overhead caused by out-cluster traffic, it can be concluded that a single server system performs better when the complete database needs to be accessed equally. In other words, when a user application has no specific locality of reference, it is better to use a single server configuration.

8.2.4 Recovery

This section analyzes the recovery latency of systems having different configurations. A multi-server architecture reduces the size of the clusters. Autonomy is given to servers during recovery. A server can proceed with the recovery of its cluster while the other clusters function normally. A server has to recover all pages it owns by contacting all the clients in its cluster. The more the number of clusters in a system, the lesser are the number of pages owned by every server and the number of clients in every cluster. Therefore, the time a cluster takes to recover should be lesser for a smaller sized cluster.

If a recovery is simulated for one cluster in a multi-cluster system, the number of clients down and the number of pages lost will affect the recovery latency. A few specific scenarios are simulated. Instead of simulating these scenarios for configurations with varying number of servers, the simulations are run with varying number of clients, varying number of pages, but with only a single server. These single cluster

configurations are the same as the cluster configurations with 400 pages, 32 clients and varying number of servers. The multi-server configurations are shown in Table II.

In the following scenarios that were simulated, assume that client C_0 is at the server site. Clients C_1 and C_2 are two other clients in the single server system. The various scenarios when a site goes down are as follows:

- *Server Clock Failure:* C_1 writes to every page in the system. Then C_2 writes to every page in the system. This is followed by C_0 writing to every page in the system. When a client writes to a page, it becomes the new clock and the old clock becomes the trailer site. At this point, C_0 is a clock site for every page and C_2 is a trailer site for every page. Now, site 0 fails. That is, the server and client C_0 go down. The server would have lost its page table. It will have to recover all the directory information. Since the clock site C_0 for all the pages went down, the trailer site C_2 will be made the new clock site for all of the pages. Once this is done, backups for every page (since the trailer site became the clock site) will have to be installed.
- *Clock Failure:* C_0 writes to every page in the system. Then C_1 writes to every page in the system. This is followed by C_2 writing to every page in the system. When a client writes to a page, it becomes the new clock and the old clock becomes the trailer site. At this point, C_2 is a clock site for all the pages and C_1 is a trailer site for all the pages. Now, site 2 fails. That is the clock site for all the pages C_2 goes down. The server site does not go down. Since the clock site for all the pages went down, the server site will make the trailer site C_1 the new clock site for all of the pages. Once this is done, backups for every page (since the trailer site became the clock site) will have to be installed.

- *Server Failure:* C_0 writes to every page in the system. Then, C_1 writes to every page in the system. This is followed by C_2 writing to every page in the system. When a client writes to a page, it becomes the new clock and the old clock becomes the trailer site. At this point, C_2 is a clock site for all the pages and C_1 is a trailer site for all the pages. Now, site 0 fails. That is the clock site and the trailer site for all the pages do not fail. Only the server site fails. The server site, site 0, will have to recover all the directory information.

Figure 24 shows the recovery latency of a single site failure with the various failure scenarios for the different configurations. For a server clock failure, the recovery latency for a system having 200 pages and 16 clients is approximately 47% of the that of a system with 400 pages and 32 clients. The recovery latency of the 11 clients, 133 pages is about 27% that of the 400 pages, 32 clients system. The recovery latency of the 8 clients, 100 pages is about 20% that of the 400 pages, 32 clients system.

Recovery of one cluster in a multi-cluster system with the various scenarios was not simulated because of implementation complexity. The recovery time of a cluster in a multi-server system may be compared to the recovery time of a single server system with the same cluster size. The advantages of a multi-server system in terms of recovery latency are significant. Especially when time critical applications are run on a DSM system, the recovery latency will be a decisive factor in the underlying DSM coherence protocol used.

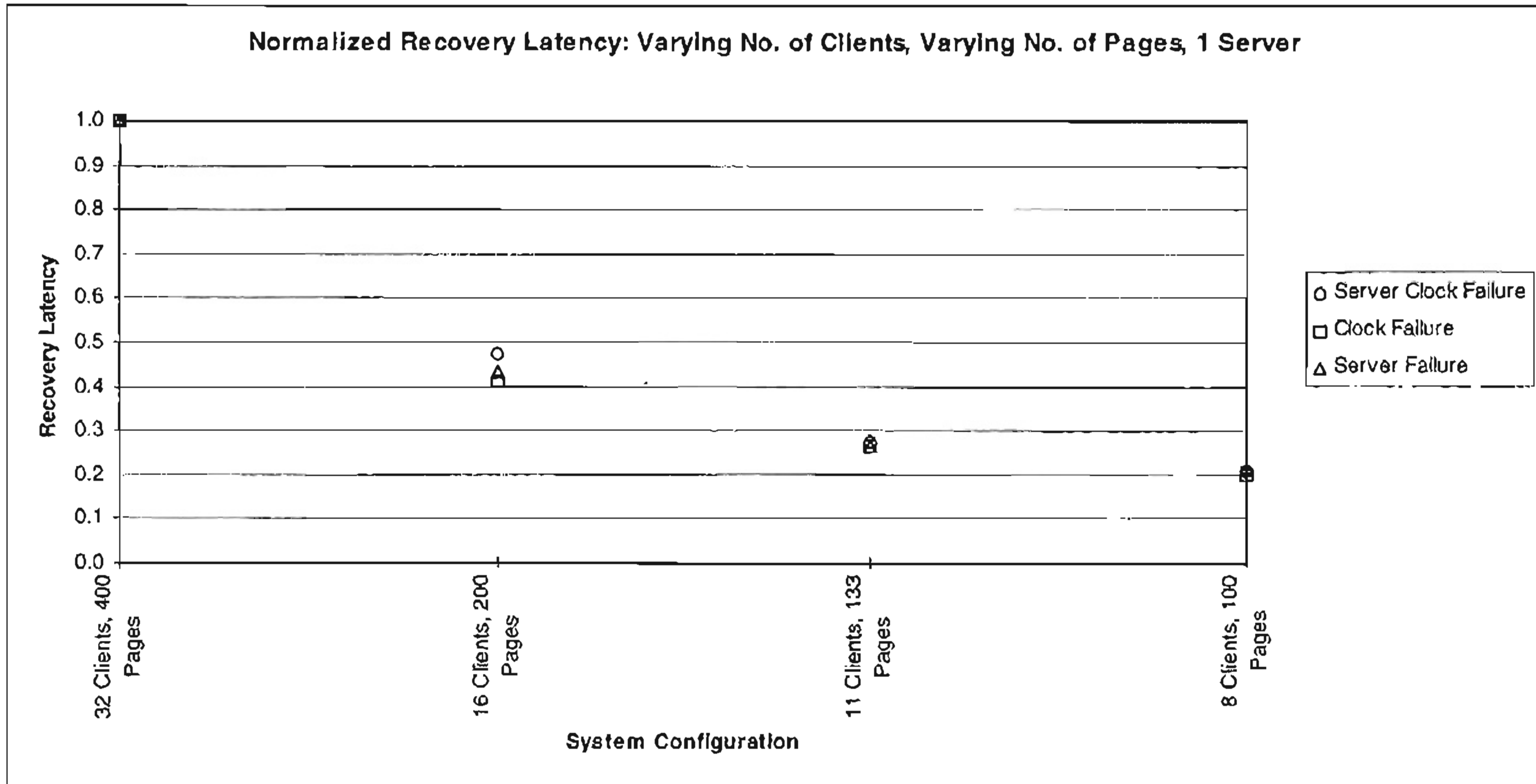

Figure 24. Effect of Recovery Latency on Single Cluster of Different Configurations: Server Clock Failure is when a server site, which is also a clock site goes down. Clock failure is when a client site, which is not a server site, goes down. Server failure is when a server site, which is neither a clock site or a trailer site, goes down. As the size of the cluster reduces from 400 pages to 200 pages, the recovery latency is approximately halved.

8.3 Limitations

As can be expected in any ongoing work of this sizable nature, during the development of this thesis work, a few problems were encountered. Initially Proteus version L3.10 was used for this research. A lot of the bugs that were encountered were fixed [Koppelman 97b]. The upgraded version of Proteus, version L3.12, was then installed as it had new features and was more robust. One vital bug was encountered when threads on some processors were not being scheduled, stalling the simulation by creating a bottleneck at say a server site. Eventually this bug was identified and fixed when it was found that timer interrupts would only occur on idle processors. This could severely distort the timing of any program which used timer interrupts while a thread was active or which has more than one ready thread per processor. This fix and other minor bug fixes were released in Proteus version L3.13. The experiments on which the results are based were run on Proteus version L3.13. As larger systems are simulated using Proteus, the heap space required during run-time and the secondary memory required to store the large data files generated are practical drawbacks.

The multi-server protocol is extremely complex. Because of the dynamism inherent to the protocol (out-of cluster requests and migrating clients) various race conditions occurred. These bugs occurred when a specific sequence of messages arrived at a client site causing an invalid system state. Eventually, memory consistency checks at every stage were incorporated to track such bugs. At the end of this research work, there are no known bugs in the system.

CHAPTER IX

CONCLUSION AND FUTURE WORK

9.1 Conclusion

Extending a single server system to a multi-server system is promising depending on the data access patterns of the application running on it. Previous attempts at multi-server systems took the primary-backup approach and found it to be unfeasible [DeMatteis 96]. The multi-server protocol that was investigated in this research exploited the inherent locality of reference exhibited by programs.

To investigate the feasibility, design, and to evaluate the performance of the multi-server protocol, a prototype was built. This prototype was a simulation of this protocol using randomly generated input data. The Proteus simulator for MIMD multiprocessors was used for building this prototype [Koppelman 97a]. All the multi-server results mentioned in this section used the migration algorithm shown in Table I (page 92). It may be possible that different results would have been obtained if a different migration algorithm were used.

The single server protocol in Reliable Mirage+ [DeMatteis 96], and the multi-server protocol under investigation (refer to Chapter IV for details) were simulated and compared. It was found that the performance of a single server system falls between

85%-90% of in-cluster requests in a 2 server protocol, where the percentage of in-cluster requests represents an applications locality of reference within its cluster. When the in-cluster percentage in a 2 server system is below 70%, the out-of-cluster overhead is more than 50% of the total traffic generated.

The out-cluster overhead is generally much greater than the migration overhead in multi-server systems. Tuning the migration algorithms to allow more migrations may reduce the out-cluster overhead. The percentage of in-cluster requests made by applications will play an important role in the amount of performance improvement over a single server system.

Experiments were conducted to observe the effect of locality of reference on protocol performance. Locality of reference is measured in terms of the total set of pages that are reused. A cluster having lesser pages will imply that the locality of reference of a user application in that cluster is restricted to this set of pages. It was found that as the locality of reference reduces, implying that the set of reused data reduces, the performance of the system improves. Reducing the locality of reference amounts to increasing the number of clusters, thus reducing the size of each cluster. It was found that the single server falls between 85%-90% of in-cluster requests in the 2 server case, 70%-75% of in-cluster requests in the 3 server case, and between 60%-65% of in-cluster requests in the 4 server case. From this it can be inferred that reduced locality of reference coupled with parallel processing by multiple servers gives better performance.

It was also concluded that a single server outperforms multi-server configurations when user applications exhibit no specific locality of reference. All the above experiments were conducted on dynamic systems with randomly generated input.

It was inferred that the benefits of faster recovery in multi-server systems are significant. For instance, the recovery latency for a system having 200 pages and 16 clients is approximately 47% of the that of a system with 400 pages and 32 clients. The recovery latency of the 11 clients, 133 pages is about 27% that of the 400 pages, 32 clients system. The recovery latency of the 8 clients, 100 pages is about 20% that of the 400 pages, 32 clients system. These results will hold special significance especially when time-critical applications run on a DSM system and minimal down time is a necessity.

Finally, it can be concluded that the multi-server protocol proposed in this research will have its benefits depending on the data access pattern of the user application running on the system. The migration algorithm may be changed depending on the user application to give better performance. The server was not a single point of failure and recovery was significantly faster in multi-server systems. Substantial gains in throughput over a single server system were obtained when the number of servers was increased, reducing the size of each cluster.

9.2 Advantages and Disadvantages of the Multi-Server Protocol

9.2.1 Advantages of the Multi-Server Protocol

The perceivable advantages of the multi-server system include the following items:

1. A multi-server system will generally out perform a single server system. However, the smaller the locality of reference of user applications, the greater will be the performance improvement.
2. Parallel processing by multiple servers increases the throughput of the system if the number of clusters is larger and the size of each cluster is smaller.
3. Even though there are multiple servers, location transparency is provided to the clients. A client need not be aware of the location of every page and/or server. Its server ensures that a client gets the requested page.
4. The server is no longer a single point of failure. If a server goes down in a multi-server system, only its cluster will participate in the recovery process. The rest of the system can still function, though not at full potential.
5. The benefits of faster recovery are significant. Recovery from a single site (and by extension a single cluster) failure in 2 server system may be generally approximately 40%-47% faster than recovery from a single site failure in a single server system having the same number of clients and pages as the 2 server system.
6. A 'window of vulnerability' is from the time a failure occurs until every page has a backup installed at an alternate site. The greater the number of servers, the lesser will be the number of pages per cluster. Subsequently, the window of vulnerability of a multi-server system will be smaller.
7. It is anticipated that even if all clusters in a multi-server system fail, the whole multi-server system will recover faster than the corresponding single server system. This is because recovery is limited to a cluster. Therefore in a multi-server system, each cluster (assuming the cluster configurations are the same) will take the same amount

of time to recover. But since the size of each cluster is smaller that the single cluster in a single server system, total system recovery in a multi-server environment is *expected* to be faster.

8. For applications that do not exhibit locality of reference, a single server environment is better than a multi-server environment. However, if the tradeoff is worthwhile, the benefits of faster recovery may offset the high out-cluster overhead if a multi-server protocol was used for such applications.

9.2.2 Disadvantages of the Multi-Server Protocol

The perceivable disadvantages of the multi-server system include the following items:

1. All the experiments conducted in this thesis partitioned the set of pages and clients equally to create clusters. The set of pages in each cluster represented the data that was reusable. In other words, it was a measure of the locality of reference of an application running in that cluster.

   In reality, this task of partitioning the network initially, based on client requirements, is not straightforward. Prior information about client requirements is difficult to obtain. The way a distributed database is partitioned into segments and the behavior of clients would depend on the particular application being run.

2. Irrespective of the way the database is divided into segments, it may turn out that all the clients in the system exhibit locality of reference for the same set of pages - all migrating to the same cluster at the same time. In such a situation, one server will be

overloaded, while the others are idle. If this happens often, the purpose of adding parallelism to the system by increasing the number of servers might be defeated.

9.3 Future Work

A few ideas about further analysis and possible enhancements to the multi-server protocol are listed below:

1. Scalability of the multi-server protocol has not been examined in this thesis. This was because as the number of processors is increased in the simulated system, heap space requirements during run-time and the secondary memory storage required for the considerably large data files generated by Proteus were potentially serious limitations. Investigating the scalability of the multi-server protocol, without changing the locality of reference of an application, may give more information about the potential of the DSM coherence protocol.
2. The input used for all the experiments in this thesis was randomly generated. Running standard applications on the multi-server system will give crucial insight into how the multi-server system may be used.
3. The locality of reference of a user application was constant in all the experiments that were conducted. That is, if a client made X% of in-cluster requests when in one cluster, it made the same percentage of in-cluster requests even after it migrated to another cluster. Perhaps a real application would have a more dynamic locality of reference and investigating it would provide valuable insight into the kind of applications that might benefit from this system.

4. Some of the parameters set in the test runs stayed constant throughout a run. Every segment was assumed to be of the same size (all clusters owned the same *number* of pages). The migration algorithm for all clients in all the clusters was the same for each run. The locality of reference of all clients, within a single cluster, and across clusters was the same. All of the above parameters are tunable. By allowing dynamic changes in one or more of them might give better performance.
5. Measuring the effect on throughput when one or more clusters go down would give valuable input on how a real-time system might get effected if a failure should occur.
6. The system architecture in this thesis assumed an indirect network. The distance between any pair of processors was constant. This was assumed so that the communication overhead between any client-server pair would be the same if the network load were the same. In such a system, a client would not add to the communication overhead if it migrated and had to contact its new server. Another alternative would be the following: a direct network may be used where the distance between each pair of processors is not the same. The processors that form a cluster would be closer to each other than to processors in other clusters. That is, at a hardware level, communication overhead within a cluster would be lesser than the communication overhead across clusters. Migration on such a system would not imply that a client contacted a new server. Instead, when a client migrated to a new cluster, the corresponding process would be executed on one of the processors in the new cluster. This is similar to the traditional concept of migration where a process moves from one processor to another. Such a system might work if the

communication locality within a cluster was minimal, say if a cluster was a small scale multiprocessor.

Finally, from this research it can be concluded that depending on the behavior of an application and its locality of reference, a specific multi-server configuration can be chosen. The migration algorithm can also be modified to give superior performance. Depending on the necessity for providing uninterrupted service, and weighing the tradeoff between the overhead of a multi-server system and the accelerated recovery, an appropriate architecture coupled with the multi-server protocol can be used effectively.

REFERENCES

[Agarwal 91] A. Agarwal, "Limits on Interconnection Network Performance", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 398-412, October 1991.

[Avizienis and Laprie 86] A. Avizienis and J. -C. Laprie, "Dependable Computing: From Concepts to Design Diversity", *Proceedings of IEEE*, Vol. 74, No. 5, pp. 629-638, May 1986.

[Barborak et al. 93] M. Barborak, M. Malek, and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing", *ACM Computing Surveys*, Vol. 25, No. 2, pp. 171-220, June 1993.

[Bennett et al. 90] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Seattle, WA, pp. 168-176, March 1990.

[Brewer 92] E. A. Brewer, "Aspects of a Parallel-Architecture Simulator", Technical Report MIT/LCS/TR-527, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, January 1992.

[Brewer and Dellarocas 92] E. A. Brewer and C. N. Dellarocas, "Proteus User Documentation Version 0.5", available in the distribution of Proteus at the URL 'http://www.ee.lsu.edu/koppel/proteus.html', December 1992.

[Brewer et al. 91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl, "Proteus: A High-Performance Parallel-Architecture Simulator", Technical Report MIT/LCS/TR-516, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1991.

[Coulouris et al. 94] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, Second Edition, Addison-Wesley Publishing Company, Inc., 1994.

[Cristian 91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of the ACM*, Vol. 34, No. 2, pp. 56-78, February 1991.

[Denning 72] P. J. Denning, "On Modeling Program Behavior", *Proceedings of the Spring Joint Computer Conference*, Atlantic City, N.J., pp. 937-944, May 1972.

[DeMatteis 96] C. K. DeMatteis, "A Fault Tolerant Distributed Shared Memory System: Reliable Mirage+", Masters Thesis, Computer Science Department, University of California, Riverside, CA, March 1996.

[Eggers and Katz 88] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation", *Proceedings of the Fifteenth Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, also published in the *ACM Computer Architecture News*, Vol. 16, No. 2, pp. 373-382, May 1988.

[Erlichson et al. 96] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy, "SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory", *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, also published in *Computer Architecture News, Special Issue*, Vol. 24, pp. 210-220, October 1996.

[Feeley et al. 94] M. J. Feeley, J. S. Chase, V. R. Narasayya, and H. M. Levy, "Integrating Coherency and Recoverability in Distributed Systems", *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, pp. 215-227, November 1994.

[Fischer et al. 85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, April 1985.

[Fleisch et al. 94] B. D. Fleisch, R. L. Hyde, and N. C. Juul, "Mirage+: A Kernel Implementation of Distributed Shared Memory on a Network of Personal Computers", *Software - Practice and Experience*, Vol. 24, No. 10, pp. 887-909, October 1994.

[Fleisch and Popek 89] B. D. Fleisch and G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design", *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, also published in *Operating Systems Review, Special Issue*, Vol. 23, No. 5, pp. 211-223, December 1989.

[Gray and Siewiorek 91] J. Gray and D. P. Siewiorek, "High-Availability Computer Systems", *IEEE Computer*, Vol. 24, No. 9, pp. 39-48, September 1991.

[Hyde and Fleisch 96] R. L. Hyde and B. D. Fleisch, "An Analysis of Degenerate Sharing and False Coherence", *Journal of Parallel and Distributed Systems*. Vol. 34, No. 2, pp. 183-195, June 1996.

[Juul and Fleisch 95] N. C. Juul and B. D. Fleisch, "A Memory Approach to Consistent, Reliable Distributed Shared Memory", *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, IEEE Computer Society Press, pp. 108-112, May 1995.

[Kermarrec et al. 95] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut, "A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability", *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing, Digest of Papers*, Pasadena, CA, pp. 289-298, June 1995.

[Koo and Toueg 87] R. Koo and S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 23-31, January 1987.

[Koppelman 97a] D. M. Koppelman, "Proteus Version L3.13", available at 'http://www.ee.lsu.edu/koppel/proteus.html', Louisiana State University, Baton Rouge, LA, 1997.

[Koppelman 97b] D. M. Koppelman, private communication, Department of Electrical & Computer Engineering, Louisiana State University, Baton Rouge, LA, 1997.

[Lamport 79] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690-691, September 1979.

[Lo 94] V. Lo, "Operating Systems Enhancements for Distributed Shared Memory", *Advances in Computers*, Vol. 39, pp. 191-237, 1994.

[Mohindra and Ramachandran 91] A. Mohindra and U. Ramachandran, "A Survey of Distributed Shared Memory in Loosely-Coupled Systems", Technical Report GIT-CC-91/01, Georgia Institute of Technology, Atlanta, GA, January 1991.

[Mullender 93] S. J. Mullender, "Kernel Support for Distributed Systems", *Distributed Systems*, Second Edition, ACM Press, New York, NY, pp. 385-409, 1993.

[Nitzberg and Lo 91] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms", *IEEE Computer*, Vol. 24, No. 8, pp. 52-60, August 1991.

[Patterson and Hennessy 96] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[Pong and Dubois 95] F. Pong and M. Dubois, "A New Approach for the Verification of Cache Coherence Protocols", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8, pp. 773 - 787, August 1995.

[Protic et al. 96] J. Protic, M. Tomasevic, and V. Milutinovic, "Distributed Shared Memory: Concepts and Systems", *IEEE Parallel and Distributed Technology*, Vol. 4, No. 2, pp. 63-79, Summer 1996.

[Satyanarayanan et al. 94] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler, "Lightweight Recoverable Virtual Memory", *ACM Transactions on Computer Systems*, Vol. 12, No 1, pp. 33-57, February 1994.

[Schneider 90] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", *ACM Computing Surveys*, Vol. 22, No 4, pp. 299-319, December 1990.

[Shah 97] S. K. Shah, "Fault Tolerance and Scalability in DSM Coherence Protocols - A Simulation Approach", Masters Thesis, Computer Science Department, University of California, Riverside, CA, June 1997.

[Silberschatz and Galvin 94] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fourth Edition, Addison-Wesley Publishing Company, Inc., 1994.

[Singhal and Shivaratri 94] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems: Distributed, Database and Multiprocessor Operating Systems*, McGraw-Hill, Inc., New York, NY, 1994.

[Tanenbaum 95] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[Theel and Fleisch 95] O. E. Theel and B. D. Fleisch, "Design and Analysis of Highly Available and Scalable Coherence Protocols for Distributed Shared Memory Systems", Technical Report UCR-CS-95-1, Department of Computer Science, University of California, Riverside, CA, April 1995.

[Theel and Fleisch 96a] O. E. Theel and B. D. Fleisch, "A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 9, pp. 915-930, September 1996.

[Theel and Fleisch 96b] O. E. Theel and B. D. Fleisch, "The Boundary-Restricted Coherence Protocol for Scalable and Highly Available Distributed Shared Memory Systems", *The Computer Journal*, Vol. 39, No. 6, pp. 496-510, Oxford Press, London, U. K., 1996.

[Turek and Shasha 92] J. Turek and D. Shasha, "The Many Faces of Consensus in Distributed Systems", *IEEE Computer*, Vol. 25, No. 6, pp. 8-17, June 1992.

[Wu and Fuchs 90] K.-L. Wu and W. K. Fuchs, "Recoverable Distributed Shared Virtual Memory", *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 460-469, April 1990.

APPENDICES

APPENDIX A

GLOSSARY

Availability	A measure of delivery of proper service with respect to the alternation of delivery of proper and improper service.
AIX/TCF	AIX is IBM's UNIX product consisting of a monolithic kernel, based on UNIX System V. The Transparent Computing Facility (TCF) allows a network of personal workstations to act as a transparent cluster of machines. Through the use of TCF, AIX becomes a transparent distributed operating system, a collection of computers are treated as a single resource.
BR Coherence Protocol	Boundary-Restricted Coherence Protocol; a write invalidate protocol defined by two parameters: w the minimum number of copies of a page cached at client sites, and n the maximum number of client sites in the network. It provides data availability at low costs.
Byzantine Failure	A failure where the faulty component can exhibit arbitrary and malicious behavior, with or without collusion with other faulty components.
Clock Site	The site which has the most recent copy of a page.
Coherence Semantics	The definition of the notion of correctness and *what* is guaranteed by a DSM system.
Coherence Unit	An abstract memory object that is guaranteed to be consistent.
DBR Coherence Protocol	Dynamic Boundary-Restricted Coherence Protocol; an extension to the BR coherence protocol class. The value of the parameter w, the minimum number of copies of a page

	cached at client sites, is monitored and modified dynamically depending on the workload of the application. It provides higher data availability at lower costs than BR coherence protocol.
Distributed Shared Memory	An abstraction used for sharing data among computers that do not share physical memory.
DSM	Distributed Shared Memory.
Error	That which may lead to a failure; it is the manifestation of a fault in a system.
Failstop Failure	One where the faulty component changes to a state that permits other components to detect the failure, and then stops.
Failure	Effect of an error on service; it happens when the delivered service deviates from the specified service.
Fault	That which causes an error.
Fault Avoidance	A method used to prevent the occurrence of a fault.
Fault Tolerance	A method of providing services complying with the specification in spite of faults.
Locality of Reference	A program property where programs tend to reuse data and instructions that have been used recently. An implication of locality of reference is that one can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past.
Loosely Coupled System	A distributed system where the processors do not share memory or a clock. Each processor has its own local memory. The processors communicate with one another via an interconnection network that connects all the processors.
Memory Coherence	A memory is *coherent* if the value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

MIMD	Multiple instruction streams, multiple data streams; A multiprocessor system where each processor fetches its own instructions and operates on its own data.
Recovery	Restoring a system to its normal operational state.
Reliability	A measure of the continuous delivery of proper service from a reference initial instant.
Sequential Consistency	A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.
Service	That which is delivered by a system; it is a system's behavior as perceived by its user(s).
System	A set of hardware and software components designed to provide a specified service.
Temporal Locality	When recently accessed words are likely to be accessed in the near future.
Thrashing	A process is said to be thrashing if it spends more time paging than executing.
Throughput	Amount of work done by a system in unit time.
Tightly Coupled System	A multiprocessor system where the processors share memory, a bus, and a clock. Processors exchange data and synchronize through a global address space that is accessible by all processors.
Trailer Site	The site that has the most recent invalid copy of a page.
WB Coherence Protocol	Write Broadcast Coherence Protocol; a write to a shared data object causes all copies to be updated. Once a site has cached a copy of a shared object, that copy is never deleted. Thus, an arbitrary copy can be used for reading but all copies must be modified as a part of a write operation.
WI Coherence Protocol	Write Invalidate Coherence Protocol; a write to a shared data object causes the invalidation of all copies except one

before the write can proceed. Once invalidated, copies are no longer accessible.

APPENDIX B

TRADEMARK INFORMATION

AIX	Trademark of International Business Machines.
IBM	Registered trademark of International Business Machines.
PS/2	Registered trademark of International Business Machines.
Solaris	Trademark of Sun Microsystems, Inc.
SPARC	Trademark of SPARC International, Inc. SPARCstation® is licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.
SunOS	Trademark of Sun Microsystems, Inc.
UNIX	Registered trademark of The Open Group.

APPENDIX C

PROGRAM LISTING

The program files are presented in this appendix. The following files contain code written in a simple superset of the C programming language and a set of supported simulator calls (see Section 5.2) [Brewer and Dellarocas 92]. These files simulate the single-server protocol of Reliable Mirage+ [DeMatteis 96] and the multi-server protocol which has been presented in this thesis.

The files with extension .h are the header files. The files with extension .ca are the augmented C files, that is C code that is cycle counted. The order of the files in the following pages is given below:

```
ft.h
requestQ.h
requestQ.ca
linked.h
linked.ca
linkedmulti.h
linkedmulti.ca
multiserver.h
ft_random.h
```

ftmain.ca

failure.ca

initSC.ca

messages.ca

serverfn.ca

clientfn.ca

requests.ca

servermulti.ca

clientmulti.ca

requestsmulti.ca

servermig.ca

clientmig.ca

serverrec.ca

clientrec.ca

The following sample code includes the following files:

ft.h

serverfn.ca

clientfn.ca

```
/***********************************************************************
 * FILE     :   ft.h                                                   *
 * CONTENTS:    This file contains all the declarations and            *
 *              data structures used by the single server and          *
 *              multi server system.                                   *
 ***********************************************************************/

#ifndef FAULT_TOL_H
#define FAULT_TOL_H
#include <string.h>
#include <malloc.h>
#include <stdlib.h>
#include <assert.h>
#include "misc.h"
#include "user.h"
#include "paramUser.h"
#include "event.h"
#include "linked.h"
#include "requestQ.h"

#include "multiserver.h"
#include "linkedmulti.h"

#define ARCHITECTURE                                        32
        /* 32 bit architecture. This is used to pack page information
         * into bitmaps during recovery.
         */

/* MAX_SIZE_ARRAY defined in paramUser.h, and can be modified in
 * prot.par*/
/* MAX_SIZE_ARRAY should be at least
        * (NO_OF_PROCESSORS *2 ) + 8 : clientMigrateSendAllInClusterReadClocks
        *                            : migrateHandoverReadClockAtServer
        * (NO_OF_PAGES *2) + 5: clientMigrateSendAllInAndOutClusterReadPages
        * 5 + ARCHITECTURE - 1: sendAllInClusterRecoveryInfo
        * 6 + NO_OF_PROCESSORS: checkAndInstallClocks
        * (NO_OF_PAGES + 10) :serverInvalidatingReadPagesForWhichMigClient
        *                     IsReader
        */

/* SEND_ARRAY is the name of an array in every function which
 * sends a message across the network. All values in SEND_ARRAY
 * are initialized to this value. Makes it easy to recognize the
 * value. This value is probably not a valid value which will
 * be sent. However, the value should not exceed the max value
 * of 'Word'.
 */
#define INIT_SEND_ARRAY                                     7777777

/* State of user threads:
 * USER_SUSPENDED and USER_RUNNING are used by the watchdog and the
 * reply thread to check the state of the user, and to see who woke up
 * the user.
 * When memory is allocated for perClientDS, user_state is initialized
 * to USER_NEVER_STARTED, so that once the server begins accepting
 * requests, the user is started for the first time (after checking the
 * value for USER_NEVER_STARTED).
 * This value is never needed again (unless the user crashed).
 * The following values are in octal.
 * These values are directly  assigned, not bitwise or'd.
 * A check is done by bitwise and'ing.
 * Only one of the following values is possible at a time.
 */
#define USER_SUSPENDED                                      01
#define USER_RUNNING                                        02
#define USER_NEVER_STARTED                                  04
#define USER_MIGRATING                                      010 /*octal 10, decimal  8*/

/* If the client is recovering, the user thread must be killed. So, the
 * user thread checks the variable client_state before making a request.
 * If it is CLIENT_RECOVERING, the user exits out of the infinite loop
 * and ends.
 * If the client is CLIENT_NEVER_STARTED, after installation a client
 * thread is started (to handle requests coming to clock sites, or
 * invalidations coming to client).
 * A check is done by bitwise and'ing.
```

```
 * Assignment is done by bitwise or'ing. Do not directly assign
 * The following values of client states are in octal.
 */
/* A client can only be in one of the following states.
 * A combination is not possible.
 */
#define CLIENT_OK                       01      /* octal 1, decimal 1*/
        /* The client thread is running, and processing requests.
         */

#define CLIENT_NEVER_STARTED            02      /* octal 2, decimal 2*/
        /* The client thread is not running, it must be created.
         */

#define CLIENT_DOWN                     04      /* octal 4, decimal 4 */
        /* The client site crashed, so the client thread must exit.
         * The client thread exits when it sees this state and goes
         * to CLIENT_NEVER_STARTED state.
         * The client moves to this state from either one of:
         * CLIENT_OK, CLIENT_TO_MIGRATE,CLIENT_MIGRATING.
         */

#define CLIENT_TO_RECOVER               010     /* octal 10, decimal 8 */
        /* When the server tells the client to start the recovery process,
         * the client goes into this state.The client could be in any of
         * the following states when it is told to recover:
         * CLIENT_OK, CLIENT_TO_MIGRATE,CLIENT_MIGRATING.
         * It has not yet started the recovery process in this state.
         * It moves to CLIENT_RECOVERING state from here.
         * Since only one site can go down at a time, it cannot
         * move to CLIENT_DOWN from this state.
         */

#define CLIENT_RECOVERING               020     /* octal 20, decimal 16 */
        /* The client moves from CLIENT_TO_RECOVER to CLIENT_RECOVERING
         * state. In this state, the client thread has stopped processing
         * requests. From this state, it moves to CLIENT_OK.
         */

#define CLIENT_TO_MIGRATE               040     /*octal 40, decimal 32*/
        /* When the client is in state CLIENT_OK, it changes to
         * CLIENT_TO_MIGRATE. The clientMigrationThread is started by
         * this time, but the client is waiting for the user state
         * to go to USER_MIGRATING, and the request queue to be
         * empty before the client thread goes into the
         * CLIENT_MIGRATING state.
         * Since it is still in s1's cluster, it can move to
         * CLIENT_TO_RECOVER or CLIENT_DOWN from this state
         */


#define CLIENT_MIGRATING                0100    /*octal 100, decimal 64*/
        /* When the client is CLIENT_TO_MIGRATE, the user thread
         * is USER_MIGRATING and the request queue is empty, the
         * client moves to  CLIENT_MIGRATING. It is still in the
         * old cluster. At this point, the clientMigrationThread
         * will signal a client ready to migrate to s1.
         * Since it is still in s1's cluster, it can move to
         * CLIENT_TO_RECOVER or CLIENT_DOWN from this state.
         */


#define CLIENT_MIGRATED_POLLING         0200    /*octal 200, decimal 128*/
        /* The client moves to this state from the CLIENT_MIGRATING
         * state. It is waiting to be accepted by its new server
         * s2. It cannot start recovery or go down at this point.
         * It first has to move to CLIENT_OK when s2 ack's the
         * polling messages.
         */

/* state of server (and its cluster)
 * After installation is done at the server, if the value
 * is SERVER_NEVER_STARTED, a server thread is started to handle
 * incoming requests.
 * A check is done by bitwise and'ing.
 * Assignment is done by bitwise or'ing. Do not directly assign
```

```
 * Assigned only when a value is sent by a server to a client.
 */
#define SERVER_OK               01      /* octal 1, decimal 1*/
                                        /* The serverThread is running, and processing
                                         * requests.
                                         */

#define SERVER_NEVER_STARTED    02      /* octal 2, decimal 2*/
                                        /* From SERVER_DOWN, it moves to this state.
                                         * The serverThread exits and sets the server state
                                         * to SERVER_NEVER_STARTED.
                                         */

#define SERVER_DOWN             04      /* octal 4, decimal 4*/
                                        /* The server site crashed, so the serverThread
                                         * must exit. It exits and moves to
                                         * SERVER_NEVER_STARTED.
                                         * Moves to this state only from SERVER_OK.
                                         */

#define SERVER_TO_RECOVER       010     /* octal 10, decimal 8*/
                                        /* The server site has to start recovery.
                                         * The server starts recovery either when
                                         * it comes from from failure, or if a down
                                         * client sends it a message saying its come
                                         * up from failure. When this state is set,
                                         * the serverThread sees this and moves to
                                         * SERVER_RECOVERING.
                                         * If the server was down, it moves to this
                                         * state from SERVER_NEVER_STARTED. When the
                                         * transition to this state is made, the
                                         * serverThread is re-started and the server
                                         * state changed.
                                         * If only a client crashed, and not the server,
                                         * it moves to this state from SERVER_OK.
                                         */

#define SERVER_RECOVERING       020     /* octal 20, decimal 16*/
                                        /* The server moves from SERVER_TO_RECOVER to
                                         * SERVER_RECOVERING. This state change is made
                                         * by serverThread. It means that the server
                                         * thread is no longer processing requests.
                                         * In this state the serverRecoveringThread is
                                         * running. It moves to SERVER_OK from here
                                         * and the serverRecoveringThread exits.
                                         */


/* State of failure thread. This is used to destroy the failure
 * thread at the end of the simulation.
 * Valid states are FAILURE_NEVER_STARTED. FAILURE_SUSPENDED,
 * FAILURE_RUNNING. This state is changed in the files
 * ftmain.ca and failure.ca
 */
#define FAILURE_SUSPENDED               01
#define FAILURE_RUNNING                 02
#define FAILURE_NEVER_STARTED           04


/* Used to wakeup the failureThread after installation is
 * complete.
 */
#define INSTALLATION_NOT_COMPLETE       400
#define INSTALLATION_COMPLETE           401

/* Should the server reply to duplicate requests or not
 * DO NOT swap these values. The way the values are
 * tested depends on the following values.
 * Exclusive -or is used to test the value.
 */
#define SERVE_DUPLICATE                 01      /* octal 1 = binary 01 = decimal 1*/
#define NOT_SERVE_DUPLICATE             00      /* octal 0 */

/* Should a request for that page be serviced now or later.
 * SERVICE_NOW and SERVICE_LATER are used by the client aux pte. On a
 * SERVICE_NOW, the client replies to any fwded requests.
```

```
 * On a SERVICE_LATER, the client ignores any fwded requests.
 * The server aux pte has values SERVICE_NOW, SERVICE_LATER_IGNORE,
 * SERVICE_LATER_REPLY. The server does the foll:
 * SERVICE_NOW: Forward the msg to the clock site
 * SERVICE_LATER_IGNORE: Do not forward the msg to clock  Ignore the
 *                        request and the client will re-send.
 * SERVICE_LATER_REPLY : D not fwd msg to clock. Reply to the client to
 *                       resend (the code for the reply thread will be LATER).
 *                       The client will not resend immediately but will wait
 *                       for some time before resending.
 *                       The server may have a SERVICE_LATER_REPLY flag if:
 *                       The clock site is migrating:
 *                       That page is in write mode in a different cluster;
 *                       In cases where some major updates are going on in
 *                       the clock site (which may require more than 1 msg),
 *                       and hence the requestor having a normal timeout
 *                       period before resending will not ensure that it
 *                       gets a reply.
 * SERVICE_OUT_OF_CLUSTER, a page which does not belong to this cluster and
 *                       is not in the memory of any site in this cluster.
 */
/* At client, SERVICE_NOW, SERVICE_LATER are the valid values.
 * At server, SERVICE_NOW,SERVICE_LATER_IGNORE,SERVICE_LATER_REPLY are
 *           valid values.
 * Always assign instead of bitwise or'ing.
 */

#define SERVICE_NOW                    01
#define SERVICE_LATER                  02
#define SERVICE_IGNORE                 04
#define SERVICE_REPLY                  010     /* octal 10, decimal 8*/
#define SERVICE_LATER_IGNORE           06
#define SERVICE_LATER_REPLY            012
#define SERVICE_OUT_OF_CLUSTER         020     /*octal 20, decimal 16*/

/* The modes a page can be in ( page_status in auxClient). All valid values
 * are listed here. Others are invalid.
 * These values are used in auxClient.page_status, auxServer.clock_mode
 * READ means page is in memory in READ mode.
 * WRITE_CLOCK means page is in memory in WRITE mode (therefore is also CLOCK).
 * READ_TRAILER means page is valid and in memory.
 * WRITE_TRAILER is an invalid option.
 * TRAILER means page is invalid, only a TRAILER copy.
 * PAGE_NOT_IN_MEMORY means page is not in memory.
 * Turning the READ, WRITE,CLOCK and TRAILER bits off will give
 * PAGE_NOT_IN_MEMORY.
 * To turn bit X ( X defined in octal) off in integer n. do
 * n = n & ~(X); This will turn bit X off in n.
 * A check is done by bitwise and'ing.
 * Assignment is done by bitwise or'ing. Do not directly assign.
 */
/* A zero preceding a number implies it is in octal format.*/

#define PAGE_NOT_IN_MEMORY      0100  /* octal 100, decimal 64*/
#define READ                    01    /* octal 1, decimal 1*/
#define WRITE                   02    /* octal 2, decimal 2*/
#define CLOCK                   04    /* octal 4, decimal 4*/
#define TRAILER                 010   /* octal 10, decimal 8*/
#define EXTRA                   020   /* octal 20, decimal 16*/
#define READ_CLOCK              05    /* octal 5, decimal 5*/
#define WRITE_CLOCK             06    /* octal 6, decimal 6*/
#define READ_TRAILER            011   /* octal 11, decimal 9 */
#define WRITE_TRAILER           012   /* octal 12, decimal 10 */
#define UPDATE                  040   /* octal 40 , decimal 32*/
#define WRITE_UPDATE            042   /* octal 42, decimal 34*/
#define WRITE_CLOCK_UPDATE      046   /*octal 46, decimal 38*/


/* These are codes used by a reply thread and a watchdog to update
 * the state of the page in the auxpte at the client site
 * (request_mode) in ClientDS.
 * All valid values are listed here. Rest are invalid.
 * Do not assign unless u want to overwrite previous value. Use bitwise
 * or'ing.
 */
```

```
#define NOT_IN_MEM              00
            /* Page mode is PAGE_NOT_IN_MEMORY. */
#define IN_MEM                  01
                        /* Page mode has READ or WRITE in it.*/
#define UPDATE_PAGE             02
#define LATER                   04
#define WATCHDOG                010 /* octal 10,decimal 8*/
#define IN_MEM_UPDATE_PAGE      03
            /* Check if this does not get overwritten by
             * watchdog, esp update flag */
#define IN_MEM_LATER            05
#define IN_MEM_UPDATE_PAGE_LATER   07
                        /* Reply to write update woke up user. Consequently a
             * LATER message for this page also arrived. This later
             * message was sent in response to a duplicate request.
             */


/* Kind of page access. LOCAL_ACCESS means the private memory of a client.
 * SHARED_ACCESS means memory that is shared with other clients in the
 * system. A SHARED_ACCESS page which is in the local memory of the client
 * is considered a shared operation.
 */
#define LOCAL_ACCESS    1
#define SHARED_ACCESS   2

/* The value returned by a  memory operation to a user thread.
 * Also the value returned by sendRequest to memory operation.
 * MEM_OK     : The memory operation was successful. Send next request.
 * QUIT       : Irrespective of success/failure of mem operation, user
 *              thread quits (Recovery in progress).
 * MEM_RESEND: When a user tries to read a page in memory,  and the user
 *              site is a clock site, and the clock has SERVICE_LATER,
 *              the clock could be transfering clock site. Then
 *              userThread retries the same operation again without
 *              calling the random no. generator.
 */
#define MEM_OK          1
#define MEM_RESEND      3
#define QUIT            2

/* These values are used while invalidating readers. If an existing
 * reader wants write access, it s read copy is not invalidated.
 * These values are used by invalidateAllReaders in clientfn.ca.
 */
#define ALREADY_A_READER        1
#define NOT_A_READER            2

#define NOT_REACHED             -1      /* The value of extra when a server
                                         * is indirectly forwarding a
                                         * WRITE_UPDATE out of cluster page.
                                         */

/* Message codes, used for the same ipi  Will give details about the
 * exact type of message.
 * The *_*_MSG are codes for MSG_FROM_SERVER_IPI.
 * ACK's are acknowledgement types. Put together, the ACK and MSG values
 * are unique to simplify debugging.
 * The *_*_ACK are codes for ACK_FROM_CLIENT_IPI.
 */
/* IMPORTANT: If any new messages are added, any REPLY or
 * invalidate messages (which come at a client site) should be added to
 * the switch statement in clientThread (inside the CLIENT_MIGRATING
 * while loop).
 */
#define INSTALL_CLOCK_MSG                                                    1
        /* Install the page*/
#define INSTALL_READER_TRAILER_MSG                                           2
        /* List of readers for a page are being sent.
         * A part of the readers for that page could
         * have been sent earlier. This is NOT to be
         * used to send only a set of readers without
         * prior communication(in a previous msg
         * (INSTALL_MSG) about the total number of
         * readers.
         */
```

```
#define INSTALL_DONE_AT_CLIENT_ACK                                              3
#define SET_SERVER_STATE_MSG                                                    4
        /* Uses MSG_FROM_SERVER_IPI*/
#define SET_CLIENT_STATE_MSG                                                    5
        /* Uses MSG_FROM_SERVER_IPI*/

/* Used when the server is forwarding a request to a clock, and
 * when a clock is sending a reply to a user.
 */
#define SEND_ACK_CLOCK_CHANGE_MSG                                               6
#define DO_NOT_SEND_ACK_MSG                                                     7
#define SEND_ACK_WRITE_UPDATE_REACHED_MSG                                       8
#define SEND_ACK_MIG_READ_CLOCK_INSTALLED                                       9
#define SEND_ACK_MIG_WRITE_CLOCK_INSTALLED                                      10
#define SEND_ACK_MIG_TEMP_READ_CLOCK_INSTALLED                                  11
#define SEND_ACK_REC_READ_CLOCK_INSTALLED                                       12
#define SEND_ACK_REC_TRAILER_INSTALLED                                          13


/* When a clock is sending a reply to a user. */
#define REPLY_MAKE_URSLF_READER_MSG                                             20
#define REPLY_MAKE_URSLF_READER_TRAILER_MSG                                     21
#define REPLY_MAKE_URSLF_WRITER_CLOCK_MSG                                       22
#define REPLY_MAKE_URSLF_READER_OUT_CLUSTER_PAGE_MSG                            23
                        /* Sent by its server, by a temp read clock*/
#define REPLY_MAKE_URSLF_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG                     24
                        /* Sent by its server, by a reader in the cluster*/
#define REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG                                 25
                        /* s1 sends this to c1 when s1 either has a service
                         * later reply or s1 gets a SS_REQUEST_FOR_PAGE_LATER
                         * from s2.
                         */
#define REPLY_UPGRADE_URSLF_FROM_READER_TO_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG   26
                        /* Sent by its server, by a temp read clock*/
                        /* When a server is directly sending a reply to a user without
                         * forwarding to a clock.
                         */

#define REPLY_WAKE_UP_USER_MSG                                                  27
                        /* A write clock gets a read request from the user at its
                         * own site. It downgrades to a reader, adds itself as a
                         * reader in the list of readers and send this message to
                         * itself to wake up its user. It does not send the page
                         * with this message as the page is already in its memory
                         * in the required mode.
                         */

#define S_FROM_USER_REQUEST                                                     100
                        /* If this value is changed, change the corresponding value
                         * in requestQ.h
                         */
        /* S: To server. Request from user.*/
#define C_FWD_REQUEST                                                           101
                        /* C: To client. Server fwding request. */
#define C_INVALIDATE_READER                                                     102
                        /* C: To client. Telling it to invalidate
                         * a read page.
                         * Sent by a client to a client.
                         */

#define S_ACK_CLOCK_CHANGE                                                      103
                        /* S: User makes itself clock and sends
                         * ack to server. */

#define S_INFORM_CLOCK_MODE_CHANGE                                              104
                        /* S: The user at a site changed the clock mode at
                         *    that site for a page. It informs the server
                         *    of this change */
#define S_MSG_NOT_DELIVERED                                                     105
                        /* S: The clock site could not deliver a message
                         *    to the requestor. The server has changed
                         *    the service to service_later_ignore and is
                         *    waiting for an ack. This msg will tell the
                         *    server to change it back to service_ok.
                         */
```

```
#define S_INFORM_PAGE_ACCESS                                          106
                        /* When a client accesses a shared page locally,
                         * it informs the server. The server updates the
                         * migration data. This information is needed to
                         * keep track of the history of requests made by
                         * a client. Used for migration. Uses REQUEST_FOR_SERVER_
                         * _IPI.
                         * This message is sent ONLY in the multi server case.
                         */
/* SS prefix : message from server to server, is request
 * related if sent via REQUEST_FOR_SERVER_IPI, not request
 * related if sent via SERVER_TO_SERVER_IPI
 */
#define SS_INFORM_SERVER_INFO                                         107
                        /* SERVER_TO_SERVER_IPI */
                        /* SS : A server sends its state and other
                         *      info to another server
                         *         - after initialization.
                         */

#define SS_OUT_CLUSTER_REQUEST                                        108
                        /* REQUEST_FOR_SERVER_IPI, request related.
                         * server 1 is sending a request for a page belonging
                         * to server 2 to server 2.
                         */

#define C_FWD_OUT_OF_CLUSTER_REQUEST                                  109
                        /* REQUEST_FOR_CLIENT_IPI, A server forwards a
                         * out of cluster request to the clock of that
                         * page(server 2 fwd'ing to c2).
                         */

#define S_READ_REPLY_OUT_CLUSTER_FROM_MY_CLIENT                       110
                        /* C2 sends a reply to S2; REQUEST_FOR_SERVER_IPI.*/

#define SS_READ_REPLY_OUT_CLUSTER_REQUEST                             111
                        /* S2 sends a reply to s1 to a read request made by s1.
                         * uses REQUEST_FOR_SERVER_IPI.*/

#define C_FWD_MAKE_URSLF_TEMP_CLOCK_HANDLE_READ_REQUEST               112
                        /* c1 in s1 has a read copy of page p which belongs
                         * to s2. Now, c1-2 in s1 requests a read copy of page
                         * p. s1 does not fwd request to s2. s1 sends a msg
                         * to c1 telling it to become a temp read clock for p.
                         * uses REQUEST_FOR_CLIENT_IPI.
                         */

#define S_INVALIDATE_ALL_READERS_AT_OTHER_SERVERS                     113
                        /* c2 in s2 has s1 as a reader. c2 is a read clock.
                         * When it gets a write request, it invalidates
                         * all readers. If a server is a reader ( a read
                         * page is given to any other server), c2 sends
                         * a this msg code to s2 using REQUEST_FOR_SERVER_IPI.
                         * s2 will check for that read page with all servers
                         * and send a invalidation msg to every server(say s1)
                         * that has this read page. s1, in turn, will tell
                         * its temp clock to invalidate all its readers.
                         */

#define S_INVALIDATE_ALL_BUT_ONE_READERS_AT_OTHER_SERVERS             114
                        /* c2 in s2 has s1 as a reader. c2 is a read clock.
                         * When it gets a write request, it invalidates
                         * all readers. If the write request was made by
                         * the server which is a reader, this message is
                         * sent to s2 using  REQUEST_FOR_SERVER_IPI.
                         * s2 will check for that read page with all servers
                         * and send a invalidation msg to every server(say s1)
                         * that has this read page except the server that
                         * requested the write. s1, in turn, will tell
                         * its temp read clock to invalidate all its
                         * readers.
                         */

#define SS_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE                 115
                        /* s1 has a read copy of page belonging to s2.
                         * s2 sends s1 this message telling it to
```

```
                        * invalidate all its readers of specified
                        * page. Uses REQUEST_FOR_SERVER_IPI.
                        */

#define C_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE                    116
                       /* c1 in s1 is a temp read clock of page p belonging
                        * to s2. s1 sends a message to c1 telling to it
                        * invalidate all its readers. c1 does not have to send
                        * an ack back to s1 and s1 does not have to send an
                        * ack back to s2(the owner of the page).
                        * Uses REQUEST_FOR_CLIENT_IPI.
                        * Sent by a server to a client.
                        */
#define C_INVALIDATE_OUT_CLUSTER_READER                                 117
                       /* A temporary read clock telling a reader to invalidate
                        * its read page. Sligtly differnt from
                        * C_INVALIDATE_READER.
                        * Send by a client to another client.
                        */
#define S_WRITE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT                        118
                       /* When a write request is made by a server s1 to server s2,
                        * s2 fwd's to client c2. c2 uses this message code to give
                        * the reply to this write request to its server
                        * Uses REQUEST_FOR_SERVER_IPI.
                        */

#define S_WRITE_UPGRADE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT                119
                       /* When a write request is made by a server s1 to server s2,
                        * s2 fwd's to client c2. c2 uses this message code to give
                        * the reply to this write request to its server. This
                        * message code is used when the requesting server s1, is
                        * already a reader of this page.
                        * Uses REQUEST_FOR_SERVER_IPI.
                        */


#define SS_WRITE_REPLY_OUT_CLUSTER_REQUEST                              120
                       /* S2 sends a reply to s1 to a write request made by s1.
                        * uses REQUEST_FOR_SERVER_IPI.*/

#define SS_WRITE_UPGRADE_REPLY_OUT_CLUSTER_REQUEST                      121
                       /* s1 is a reader for an out of cluster page. It made
                        * a write request for that page. It gets this reply
                        * from s2.
                        */

#define S_RETURNING_OUT_CLUSTER_UPDATE_PAGE                             122
                       /* c1 returns an update write (out of cluster) page to s1 after
                        * writing to it. Uses REQUEST_FOR_SERVER_IPI*/

#define SS_RETURNING_OUT_CLUSTER_UPDATE_PAGE                            123
                       /* s1 returns an update write page to s2 (the owner) after
                        * writing to it. */

#define C_RETURNING_UPDATE_PAGE                                         124
                       /*s2 sends the updated write page to c2. Uses
                        *REQUEST_FOR_CLIENT_IPI.
                        */

#define S_INVALIDATE_WRITE_UPDATE_PAGE                                  125
                       /* c2 has given a write update page to s2 to be given to another
                        * server. c2's watchdog wakes up and it finds that it hasnt
                        * yet got the update page. c2 sends this msg to s2 telling it
                        * to invalidate the write update page.*/
#define SS_INVALIDATE_WRITE_UPDATE_PAGE                                 126
                       /* s2 has given s1 a write update page. This message is sent by s2
                        * to s1 telling it to invalidate that write update page.
                        */

#define SS_INVALIDATE_WRITE_UPDATE_PAGE_WAITING                         127
                       /* s2 has given a write update page to s1. It sends
                        * an invalidation message to s1. If s1 sees that
                        * the initial write update page has not yet
                        * reached the client, it re-inserts the invalidation
                        * message as this message.
                        */
```

```
#define C_INVALIDATE_WRITE_UPDATE_OUT_CLUSTER_PAGE                              128
                        /* c1 has a write update copy of page. s1 sends it an invalidate
                         * message since c1 has kept it for more than update time.
                         * Sent by a server to a client.
                         */
#define SS_RESEND_REQUEST_FOR_PAGE_LATER                                         129
                        /* s2 gets a request for a page from s1. s2 has service later reply
                         * for this page. It sends a message back to s1.
                         */

#define C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE                    130
                        /* c1 in s1 is a temp read clock of page p belonging to s2.
                         * A site in s1,c1-1, asks for write permission for page p.
                         * On getting a reply from s2, s1 sends a message to c1
                         * telling it to invalidate all the readers and forward
                         * the reply to c1-1.
                         * Sent by a server to a client.
                         */

#define  S_ACK_RECEIVED_WRITE_UPDATE_PAGE                                        131
                        /* A client c1 is a reader of an out cluster page.
                         * It made a write request. When forwarding the reply,
                         * the server set extra to NOT_REACHED. On, recv'ing
                         * the reply REPLY_UPGRADE_URSLF_FROM_READER_TO_UPDATE
                         * WRITER_OUT_CLUSTER_PAGE_MSG, it acks the msg.
                         */

#define  S_ACK_NOT_SENDING_WRITE_UPDATE_PAGE                                     132
                        /* s1 is a reader of page belonging to s2. A client in
                         * s1, c1-1, requested a write. s2 sent a reply.. s1
                         * changed extra to NOT_REACHED, and forwarded the message
                         * 'C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE'
                         * with ack code 'SEND_ACK_WRITE_UPDATE_REACHED_MSG' to
                         * the temp read clock, c1. But c1 was migrating. And
                         * since the clientThread and clientMigratingThread at c1
                         * are running in parallel, c1 is no longer the reader
                         * when it gets C_INVALIDATE_ALL... So c1 sends this
                         * message back to s1. (B8-2)
                         * s1 will remove c1-1 from memory when it gets this.
                         */

/******* MESSAGES CONNECTED TO MIGRATION ******/

#define SS_CLIENT_TO_MIGRATE_IN                                                  140
                        /* A client from c1 is to migrate to s2 from
                         * s1. s1 informs s2 that c1 will be  migrating
                         * in. Uses SERVER_TO_SERVER_IPI.
                         */

#define S_MIG_HANDOVER_READ_CLOCK                                                141
                        /* A client c1 is migrating. It sends this message
                         * for every in-cluster page for which it is a
                         * a read clock. This message is sent to its server
                         * using REQUEST_FOR_SERVER_IPI.
                         */

#define C_MIG_HANDOVER_INSTALL_READ_CLOCK                                        142
                        /* c1 is a read clock that is migrating. It hands
                         * over the read clock to server s1, which then
                         * sends this message to the new clock c2.
                         * Uses REQUEST_FOR_CLIENT_IPI.
                         */

#define S_ACK_MIG_READ_CLOCK_INSTALLED                                           143
                        /* c1 was a read clock. It is migrating to s2. A new
                         * read clock is installed at c2. This message sends
                         * an ack to s1 telling it that the new read clock is
                         * installed successfully. Uses REQUEST_FOR_SERVER_IPI.
                         */

#define S_MIG_HANDOVER_WRITE_CLOCK                                               144
                        /* A client c1 is migrating. It sends this message
                         * for every in-cluster page for which it is a
                         * a write clock. This message is sent to its server
                         * using REQUEST_FOR_SERVER_IPI.
```

```
                         */

#define C_MIG_HANDOVER_INSTALL_WRITE_CLOCK                              145
                        /* c1 is a write clock that is migrating. It hands
                         * over the write clock to server s1, which then
                         * sends this message to the new clock c2.
                         * Uses REQUEST_FOR_CLIENT_IPI.
                         */

#define S_ACK_MIG_WRITE_CLOCK_INSTALLED                                 146
                        /* c1 was a write  clock. It is migrating to s2. A new
                         * write clock is installed at c2. This message sends
                         * an ack to s1 telling it that the new write clock is
                         * installed successfully. Uses REQUEST_FOR_SERVER_IPI.
                         */
#define S_MIG_HANDOVER_INVALIDATE_READ_PAGES                            147
                        /* c1 is a client that is migrating. It sends this
                         * message to s1 giving it the list of pages for
                         * which it is a reader. Uses REQUEST_FOR_SERVER_IPI.
                         */

#define C_MIG_REMOVE_CLIENT_AS_READER_FOR_PAGES                         148
                        /* c1 is a client that is migrating. s1 sends the following
                         * message to a client which is a read clock for a page
                         * for which c1 is a reader. s1 does not send the pages for
                         * which c1 was a reader. This client scans its read clocks,
                         * and removes c1 as a reader. REQUEST_FOR_CLIENT_IPI.
                         */

#define S_CLIENT_READY_TO_MIGRATE                                       149
                        /* A client c1 has sent all its handover messages to
                         * s1 during migration. It's user and client are
                         * waiting to migrate. It sends this message to s1
                         * so s1 can check if all the handover's have been
                         * ack'd, and allow c1 to migrate.
                         */

#define SS_HANDING_OVER_NEW_CLIENT                                      150
                        /* A client from c1 is to migrate to s2 from
                         * s1. s1 is handing over c1 to s2.
                         * Uses SERVER_TO_SERVER_IPI.
                         */

#define C_SERVER_GIVING_PERMISSION_TO_MIGRATE                           151
                        /* A client c1 is to migrate from s1 to s2.
                         * c1 has handed over its pages and is waiting
                         * for permission from s1 to migrate. This
                         * is s1 giving permission to c1.
                         * Uses MSG_FROM_SERVER_IPI.
                         */

#define S_NEW_CLIENT_ASKING_FOR_ACCEPTANCE                              152
                        /* A client c1 has migrated from s1 to s2 (It
                         * got permission from s1). s2 could have
                         * crashed without getting the handover message
                         * from s1. So, this client acts intelligently
                         * and keeps polling s2 asking for an ack.
                         * REQUEST_FOR_SERVER_IPI.
                         */

#define C_SERVER_ACCEPTING_NEW_CLIENT                                   153
                        /* A client c1 has migrated from s1 to s2.
                         * c1 polls s2, and this is the reply from
                         * s2 to c1. Uses MSG_FROM_SERVER_IPI.
                         */

#define SS_REMOVE_MY_SERVER_AS_READER_FOR_PAGES                         154

#define S_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK 155
#define SS_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK                   156

/************ MESSAGES CONNECTED TO RECOVERY **************/
#define S_REC_DOWN_TIME_FOR_CLIENT_OVER                                 160
                        /* Uses ACK_FROM_CLIENT_IPI.
                         * A client c1 that went down, informs its
                         * server when it comes back up.
```

```
                    */

#define S_REC_SENDING_CLOCK_READER_BITMAPS                                  161
                    /* Uses RECOVERY_TO_SERVER_IPI. A client sends this
                     * message to the server when told to recover.
                     * It sends information its clock and reader pages.
                     */

#define S_REC_SENDING_TRAILER_BITMAP                                        162
                    /* Uses RECOVERY_TO_SERVER_IPI. A client sends this
                     * message to the server when told to recover.
                     * It sends information about its trailer pages and
                     * their trailer version.
                     */

#define S_REC_CLIENT_RECOVERED                                              163
                    /* Uses RECOVERY_TO_SERVER_IPI. A client tells its
                     * server that it has finished sending all its
                     * clock/reader/trailer page information.
                     */

#define C_REC_INSTALL_READ_CLOCK                                            164
                    /* Uses RECOVERY_TO_CLIENT_IPI. A server sends this
                     * message to a client during recovery. Tells it
                     * to install a read clock (for a page whose clock
                     * is lost).
                     */

#define S_REC_ACK_READ_CLOCK_INSTALLED                                      165
                    /* A client sends this msg acknowledging installation
                     * of a read clock. This is in reply to C_REC_INSTALL
                     * _READ_CLOCK.
                     * Uses RECOVERY_TO_SERVER_IPI.
                     */

#define C_REC_INSTALL_TRAILER_TO_CLOCK                                      166
                    /* A clock is told to install another trailer.
                     * The designated trailer site is sent to the
                     * clock. If the clock has multiple readers,
                     * it makes one of the readers a trailer site.
                     * The new trailer is also a reader.
                     * Uses RECOVERY_TO_CLIENT_IPI
                     */

#define C_REC_INSTALL_TRAILER                                               167
                    /* A clock tells a site to become a reader trailer
                     * of a page. This site has to send an ack
                     * to the server.
                     * Uses RECOVERY_TO_CLIENT_IPI.
                     */

#define S_REC_ACK_TRAILER_INSTALLED                                         168
                    /* A client sends this msg acknowledging installation
                     * of a trailer. This is in reply to C_REC_INSTALL
                     * _TRAILER.
                     * Uses RECOVERY_TO_SERVER_IPI.
                     */

#define S_REC_RECOVERY_COMPLETE_SET_SERVER_OK                               169
                    /* The serverSendingRecoveryMessagesThread sends this
                     * message when it knows that recovery is complete.
                     * It sends this message to its own server. The
                     * serverRecoveringThread makes some changes and then
                     * exits. This message is sent as an IPI so that the
                     * serverRecoveringThread can exit.
                     */

#define C_REC_RECOVERY_COMPLETE_SET_CLIENT_SERVER_OK                        170
                    /* The serverRecoveringThread sends this message to
                     * all its clients when it gets the S_REC_RECOVERY
                     * _COMPLETE_SET_SERVER_OK message.
                     * Uses RECOVERY_TO_CLIENT. The clientRecoveringThread
                     * changes server/client state and exits.
                     */

#define SS_SET_SERVER_STATE                                                 171
```

```
                    /* Uses REQUEST_FOR_SERVER_IPI.
                     * When a server goes down or starts recovery (without
                     * having gone down), it informs all other servers
                     * about its status.
                     * A server which gets this message, changes it
                     * page table to indicate that none of its pages
                     * are in the down/recovering cluster.
                     * Also, it reclaims any write update pages it
                     * loaned to that cluster.
                     * Read pages are removed from read pages loaned to
                     * that server, messages are not sent to the clock
                     * sites. When a read invalidation message arrives
                     * later, it is ignored.
                     */

#define C_REC_REMOVE_SERVER_AS_READER_FOR_A_PAGE                    172
                    /* Used in migration and recovery.*/

#define BUSY_WAIT(delay)        \
        { if( delay < TIMER_PERIOD *3) \
          { \
                AddTime(delay);\
                thread_yield_to_scheduler();\
          }\
          else\
                thread_sleep(MY_TID,delay/TIMER_PERIOD); \
        }


int     STATE;
int     REQUEST;
int     CHECK;
int     TEMP_RANDOM_GLOBAL;


int     PRIORITY;
int     LARGE_STACKSIZE;
int     SMALL_STACKSIZE;
int     INIT_SERVER_IPI;
                            /* To initialize data structures in each
                             * server.
                             */
int     INIT_CLIENT_IPI;
                            /* To initialize data structures in each
                             * client.
                             */

int     ACK_FROM_CLIENT_IPI;
                            /* Sent by the client when all pages
                             * have been installed.
                             */


int     MSG_FROM_SERVER_IPI;
int     REQUEST_FOR_SERVER_IPI;       /* Used by client to send a request
                                       * to a server. */
int     REQUEST_FOR_CLIENT_IPI;       /* A request sent to a client by
                                       * a server */

#ifdef MULTI_SERVER
int     SERVER_TO_SERVER_IPI;
#endif

int     SERVER_DETECT_FAILURE_IPI;
int     WAKEUP_FAILURE_THREAD_IPI;
int     DESTROY_FAILURE_THREAD_IPI;   /* Used to destroy the failure thread
                                       * at the end of the simulation.
                                       */

int     RECOVERY_TO_SERVER_IPI;
int     RECOVERY_TO_CLIENT_IPI;

Sem     mutex; /* Output semaphore; used for printf. */

Sem     file_mutex; /* Semaphore used to write to file.*/
FILE    *opFilePtr;/*Pointer to output file */
```

```
/* One auxpte for a page. An array of auxpte constitute the state of
 * memory. One array per client and one per server needs to be
 * allocated.*/

/*Aux at client site per page.*/
struct  auxC
{
        intList                 *list_of_readers;
                                /* Call initIntList when
                                 * memory is allocated for auxC.
                                 * no_readers is here.
                                 */
                                /* If the mode is WRITE_CLOCK, then
                                 * the list of readers is NULL. The
                                 * clock site is the writer.
                                 */
        unsigned        int     page_status;
                                /* READ,WRITE,CLOCK,TRAILER,
                                 * EXTRA,PAGE_NOT_IN_MEMORY
                                 * (WRITE is an invalid
                                 * mode, a WRITE site must have
                                 * the mode WRITE_CLOCK)
                                 * READ_TRAILER, WRITE_TRAILER,
                                 * READ_CLOCK, WRITE_CLOCK are the
                                 * only valid combinations.
                                 * WRITE_CLOCK_UPDATE is a valid value when
                                 * the page is loaned to another server for
                                 * a write.
                                 */
        int                     page_value;
                                /* Good tohave this; easier to
                                 * debug.
                                 * During initialization, all
                                 * page values are -1.
                                 * When a server installs a page
                                 * at a client site, it sends
                                 * a value of 1.
                                 * So, a value <= 0 is not possible (presuming
                                 * a write is always positive).
                                 */
        int                     clock_site;
                                /* Though the page_status tells if
                                 * the current site is a clock or
                                 * not, this maintains the processor
                                 * number of the clock site. This may
                                 * be required if a clock site hands
                                 * over clock data to another site, but
                                 * may still need to fwd requests for
                                 * clock of that page, until the server
                                 * is notified of the clock site
                                 * change. This value is incorporated
                                 * so the design change is possible.
                                 * But if the clock_site value
                                 * is the present client, page_status
                                 * MUST be CLOCK (READ or WRITE).
                                 * Initial value is -1.
                                 */
        int                     trailer_version; /* Initialized to 0.*/
                                /* This is the latest trailer value
                                 * for that page in the cluster.
                                 */
        unsigned        int     service ;
                                /* SERVICE_NOW, SERVICE_LATER
                                 * used by the clock site If the clock site
                                 * has a SERVICE_NOW, it will take care of
                                 * any fwd'd requests from the server.
                                 * If it gets a fwd'd request while it is
                                 * SERVICE_LATER, it simply ignores it
                                 * init to SERVICE_LATER.
                                 * The SERVICE_LATER value is set when
                                 * . the clock is making some changes
                                 *     - adding a reader
                                 *     - invalidating readers
                                 *     - changing mode
                                 *
```

```
                                 *     - clock transfering to another site
                                 *     - site is migrating
                                 *     - site is recovering.
                                 * When the above operation is complete,
                                 * the service is changed to SERVICE_NOW
                                 * It is used only by a clock site. So, a
                                 * READ or TRAILER page will have a
                                 * SERVICE_LATER value.
                                 */
        int                     extra;
                                /* Used for number of readers to
                                 * be installed for that page.
                                 * Will be compared to list_of_readers->
                                 * no_elements to know if the required
                                 * no of readers have been installed.
                                 * This is because the number of readers
                                 * may be sent in more than 1 message.
                                 */

};

typedef struct auxC     *perSitePageTable, **allPageTables;
typedef struct auxC     auxClient;


/* Auxilary page table maintained at each server site. */
struct auxS
{
        int                     clock; /*Site of clock, initialized to -1.*/
        unsigned        int     clock_mode;
                                /* READ,WRITE, READ_CLOCK,
                                 * WRITE_CLOCK,PAGE_NOT_IN_MEMORY
                                 * are valid values.
                                 * The page_mode of the clock.
                                 * It is needed to know if a clock site will
                                 * be  changed while fwding a request to
                                 * the clock. If so, either a SERVICE_LATER
                                 * is sent as a reply to another request
                                 * that comes in, or the request is ignored:
                                 * and will be resent by that client when
                                 * it times out.
                                 */
        int                     trailer;        /* Site of trailer. initialized to -1.*/
        int                     t_version;      /* Version number of trailer;
                                     * might not be uptodate if SERVER_OK,
                                     * init to -1
                                     */
        unsigned        int     service ;
                                /* SERVICE_NOW,SERVICE_LATER_IGNORE,SERVICE_LATER_REPLY
                                 * The server will see this before forwarding a request
                                 * to a clock: will ignore the request if it is
                                 * SERVICE_LATER_IGNORE. Will send a 'send request
                                 * later' msg back to the requester(client) if it is
                                 * SERVICE_LATER_REPLY.
                                 * On receiving the SERVICE_LATER_REPLY, the client will
                                 * will not resend immediately after being woken up
                                 * It will wait a  little longer.
                                 * If the client does not receive a reply. it will
                                 * time out and resend.
                                 * init to SERVICE_LATER_IGNORE in single server case
                                 * In multi server case, pages within cluster are
                                 * intialized to SERVICE_NOW once installed at clients.
                                 * pages out of cluster are initialized  to
                                 * SERVICE_OUT_OF_CLUSTER.
                                 */
        int                     extra;  /* Used by the server when sending installation
                                 * messages. If 2 pages have to be installed,
                                 * (clock and trailer), extra is set to 2;
                                 * Upon recving ack from the 2 clients (clock
                                 * and trailer, this extra is decremented.
                                 * When extra reaches 0, service is changed
                                 * to SERVICE_NOW.
                                 */
};

typedef struct auxS     auxServer;
```

```
/* PerClientDS
 * To be maintained by each client.
 * IMP:   Any changes to this structure should be taken care of in
   initClientDSHandler and in the corresponding print functions.*/
struct ClientDS
{
        unsigned int            request_mode;
                                /* NOT_IN_MEM, IN_MEM, UPDATE, LATER, WATCHDOG,
                                 * IN_MEM_UPDATE are valid values. Rest are invalid.
                                 * This field must only be used by :
                                 *   the user thread
                                 *   the watchdog
                                 *   the reply thread
                                 */

        int                     client_proc;
        int                     current_page_requested;
        int                     current_page_mode_requested;
        unsigned int            time_page_requested;
        unsigned int            time_last_migration;
        unsigned int            time_start_migration;

        int                     cluster_index;

        int                     user_state ;
                                /* USER_SUSPENDED, USER_RUNNING, USER_NEVER_STARTED*/
        int                     server_site;
        unsigned int            server_state ;
                                /* SERVER_OK, server thread is running.
                                 * SERVER_DOWN, server thread is down, SERVER_RECOVERING
                                 *              bit should also be set.
                                 * SERVER_NEVER_STARTED/SERVER_TO_RECOVER are not told
                                 *                              to the client.
                                 * SERVER_RECOVERING, 1 or more sites are down (if
                                 *                    server is down, SERVER_DOWN is
                                 *                    also set).
                                 */
        unsigned int            client_state;
                                /* CLIENT_OK, CLIENT_RECOVERING, CLIENT_NEVER_STARTED
                                 * If the client is recovering, the user thread must
                                 * be killed. So, the user thread checks the field
                                 * client_state before making a request. If it is
                                 * CLIENT_RECOVERING, the user exits out of the
                                 * infinite loop and ends.
                                 * If the value is CLIENT_NEVER_STARTED, a client
                                 * to handle incoming requests is started after
                                 * installation of all pages is complete.
                                 */
        int                     lowest_page,highest_page;
                                /* Needed here to map a random request X% within cluster*/
        int                     no_pending_client_threads;
                                /* This should be 0 before a client can migrate.
                                 * Not all threads created depend on the client
                                 * functioning. Before a thread is created, the
                                 * state of the client and the state of the server
                                 * are checked. no_pending_client_threads is
                                 * incremented. Before that thread ends, this
                                 * field is decremented.
                                 * no_pending_threads should be 0 before a
                                 * client can start recovery or be migrated to
                                 * another cluster.
                                 * If a client thread is killed arbitrarily,
                                 * sequential consistency is not guaranteed.
                                 */
        auxClient               *client_ptable;
                                /* Each client has to maintain
                                 * a page table; each pte
                                 * is of the type auxClient */
        int                     extra;
                                /* extra field added. might be needed for
                                 * something. May be used for different
                                 * purposes.
                                 * Eg:
                                 * 1.  Count of number of page_nos which
```

```
                                 *      which have installed sucessfully.
                                 */
        int                      uniq_requests_sent;
                                 /*unique requests sent so far; not
                                  * duplicates of requests.
                                  */
        reqQueue                 *client_request_queue;
                                 /* This is the list of incoming requests to
                                  * a client which is a clock site for a page, or
                                  * any invalidation messages.
                                  * This queue is not used for sending replies to
                                  * a user. A reply to a user is sent as a direct ipi
                                  * to the user site, which starts a reply thread
                                  * without putting the information into the request
                                  * queue of the client at that site.
                                  * It is not for recovery messages. (This is
                                  * because if the client is down, and recovery is
                                  * going on, the client thread might not be running).
                                  * Also, since this request queue is deleted if the
                                  * client site crashes, any possible recovery messages
                                  * might be lost.
                                  * It is initialized by calling initRequestQ while
                                  * allocating memory for perClientDS.
                                  */
        reqQueue                 *client_recovery_queue;
                                 /* This is a queue of incoming messages sent to a client
                                  * during recovery. An IPI to RECOVERY_TO_CLIENT_IPI puts
                                  * the recovery message in this queue.
                                  */

};

typedef struct ClientDS          perClientDS;

perClientDS     **globalClientDS;
perClientDS     **ptrToGlobalClientDS;

/* To be maintained by each server.
 * IMP:   Any changes to this structure should be taken care of in
   initServerDSHandler and in the corresponding print functions.*/
/* If MULTI_SERVER is defined (as a compile flag), some more data
   structures related to multi-server environment will be included
   in perServerDS.*/
struct  ServerDS
{
        intList                  *list_of_clients;
                                 /* Number of clients is here
                                  * call initIntList when memory is
                                  * allocated for ServerDS.
                                  */
        int                      lowest_page;
        int                      highest_page;
        int                      proc_no;        /* The processor number of this server.
                                 * listOfServers[s_index] is the proc number
                                 */
        int                      s_index;
                                 /* listOfServers[s_index] will be this
                                  * server's processor number. This field
                                  * is convenient to have instead of iterating
                                  * through listOfServers. Used for statistics
                                  * collection.
                                  */
        unsigned        int      server_state ;
                                 /* State of the server's cluster valid values are
                                  * SERVER_OK,SERVER_RECOVERING, SERVER_DOWN,
                                  * SERVER_NEVER_STARTED, valid combinations are
                                  * defined in the beginning of this file.
                                  */
        unsigned        int      dupReq : 1;
                                 /* Should duplicate requests be serviced.
                                  * Values are SERVE_DUPLICATE, NOT_SERVE_DUPLICATE. */
        auxServer                *server_ptable;
                                 /* Each server has to maintain
                                  * a page table; each pte
                                  * is of the type auxServer. */
        int                      no_pending_server_threads;
```

```
                    /* This should be 0 before a server can start
                     * the recovery process. Not all threads
                     * created depend on the server's functioning.
                     * Before a thread is created, the state of
                     * the server is checked.
                     * no_pending_client_threads is
                     * incremented. Before that thread ends, this
                     * field is decremented.
                     */
int                 extra;
                    /* extra field added. might be needed for
                     * something. May be used for different
                     * purposes.
                     * Examples:
                     * 1. Count of number of clients
                     *    which have installed pages sucessfully.
                     * 2. Count  of number of clients which have
                     *    sent in all their reader/trailer/clock page
                     *    information during recovery.
                     */
reqQueue            *server_request_queue;
                    /* This is the list of incoming requests to
                     * a server. This queue is only for read/write
                     * requests or related messages. It is not for
                     * recovery messages. (This is because if the
                     * server is down, and recovery is going on, the
                     * server thread might not be running). Also, since
                     * this request queue is deleted if the server site
                     * crashes, any possible recovery messages might
                     * be lost.
                     * It is initialized by calling initRequestQ while
                     * allocating memory for perServerDS.
                     */
Multi4List          *list_of_down_clients;
                    /* Multi4Node has:
                     * val1 : client site which is down
                     * val2 : the state of that client site:
                     *        takes values , CLIENT_DOWN,
                     *        CLIENT_RECOVERING (set when
                     *        client sends a message saying its
                     *        up from recovery
                     */
                    /* This is the list of clients that are down. Only
                     * one site can be down at a time. So, this should
                     * actually be single value. But is a list here, in
                     * case multiple failures are allowed.
                     * Used in both single/multi server environments.
                     */

reqQueue            *server_recovery_queue;
                    /* This is a queue of incoming messages sent to a server
                     * during recovery. An IPI to RECOVERY_TO_SERVER_IPI puts
                     * the recovery message in this queue.
                     */
intList             **ReadersForPages;
                    /* This is an array of intLists, of size NO_OF_PAGES.
                     * It is used ONLY DURING RECOVERY.
                     * The readers for each page are added to this array
                     * as and when the recovery messages come in. Since
                     * the server normally does not maintain list of readers
                     * for each page, this is added here and not in
                     * the page table itself.
                     */

int                 total_clients_in_cluster;
                    /* Total number of clients in cluster. A server knows this
                     * value while waiting for recovered messages from its
                     * clients. So, it is needed even in a single server
                     * case.
                     * This value is updated during installation, when a
                     * client migrates (is finally handed over to new server).
                     */

dataPerServer       *other_servers_info;
                    /* This is an array of dataPerServer
                     * of size NO_OF_SERVERS. other_servers_info[i] is info
```

```
                                 * this server maintains about the server located
                                 * at processor number listOfServers[i].
                                 * other_servers_info[i] is not used when
                                 * listOfServers[i] is CURR_PROCESSOR
                                 * i is the index for the server whose info
                                 * is in other_servers_info[i].
                                 */
        int                     **migData;
                                /* A 2D array of integers is created of size,
                                 * (Total no. clients in whole system) x (total no.
                                 * servers).
                                 * #rows= total no. clients in the whole system
                                 * #columns= total no. of servers
                                 * migData[i] is a pointer. Memory of size no. servers
                                 * is allocated for each i. migData[j], where j is
                                 * a client in this server's cluster is used.
                                 * migData[k], where k is not a client in this servers
                                 * cluster is not used.
                                 * When client j makes an out of cluster request for
                                 * a page owned by server s1, then migData[j][s1]
                                 * is incremented.
                                 * So, migData[cj][sr] is the number of requests
                                 * client j made for a page owned by server r.
                                 * If this server is s-me, then migData[cp][s-me] is
                                 * incremented when a client makes an in-cluster request.
                                 */
        Multi4List              *migratingInClients, *migratingOutClients;
                                /* migratingInClients is the list of client coming into
                                 *    this server's cluster.
                                 * migratingOutClients is the list of clients going out
                                 *    of this server's cluster.
                                 */
        int                     *lastRequestByClient;
                                /* This is used to eliminate duplicate request numbers.
                                 * This is an array of size 'number of clients'. When a
                                 * server gets a request from its client, it updates
                                 * the last unique request made by that client. If a
                                 * duplicate request arrives from this client with an
                                 * earlier unique request number, it is ignored. This is
                                 * used only in the message S_FROM_USER_REQUEST. Therefore,
                                 * a server only eliminates duplicate requests made by its
                                 * own clients, and not duplicate requests made by other
                                 * servers.
                                 */

};

typedef struct ServerDS  perServerDS ; /* per server data structure. */

perServerDS     **globalServerDS;
perServerDS     **ptrToGlobalServerDS;

/* This is a an integer array of size TOTAL_NO_SERVERS.
 * listOfServers[i] is the processor number of the i th server.
 * This is because the servers may not be located at consecutive
 * processors.
 * This array is written into only in usermain. After that it is
 * used as a read-only array. Even in case of any site crashing,
 * this array is not effected. It does not belong to any site or
 * any cluster.
 */
int             *listOfServers;

/* When a server finds that there is a failure in its
 * cluster (a client/the server/or both), it adds the
 * down site to teh globalListOfDownSites.
 * Two clusters may go down when a single site fails (the
 * client is in one cluster, the server in another.
 * In such a case, both clusters add the down site to
 * this global list.
 * When a server recovers, it removes the down site from
 * the global list.. If the global list if empty when the
 * server removes the down site, the server has to wake
 * up the failure thread.
 */
intList         *globalListOfDownSites;
```

```
/* When a server finishes installation, it adds its server index
 * to this list. The last server to finish installation will
 * wake up the failure thread.
 * The semaphore access_failure_data_mutex is used to access
 * globalListOfInitServers.
 */
intList         *globalListOfInitServers;
int             globalStateInstall;
int             tid_failure_thread;    /* Tid of failure thread; this value
                                        * is updated only once when the
                                        * failure thread is first created.
                                        */
int             proc_failure_thread;   /* The processor on which the failure
                                        * thread runs. An IPI will have to
                                        * be sent to that processor to
                                        * wake up the failure thread.
                                        */

int             state_failure_thread;  /* Needed to destroy the thread at
                                        * the end of the simulation.
                                        * Valid values are
                                        * FAILURE_RUNNING,
                                        * FAILURE_SUSPENDED,
                                        * FAILURE_NEVER_STARTED.
                                        */

Sem             access_failure_data_mutex;
                        /* Semaphore used to wakeup the failure thread;
                         * and to add/delete from the globalListOfDownSites
                         * DO NOT ACCESS THIS SEMAPHORE IN ATOMIC REGION.
                         */

unsigned long   TOTAL_SIM_RUN_TIME_CYCLES;

/* Data structures used for data collection. Memory is not
 * allocated using memory routines in Proteus. Arrays of size
 * NO_OF_CLIENTS, and NO_OF_SERVERS are dynamically created
 * in usermain.
 */
struct message_level
{
        unsigned        long    data_msg_count;
                                /* Number of data messages (messages where
                                 * the page itself is sent).
                                 */
        unsigned        long    data_msg_bytes;
                                /* Number of bytes sent in data messages
                                 */
        unsigned        long    control_msg_count;
                                /* Number of control messages (messages
                                 * sent where a page is not sent). Such
                                 * messages are sent as replies, requests,
                                 * to ensure consistency, for recovery and
                                 * for migration,
                                 */
        unsigned        long    control_msg_bytes;
                                /* Number of bytes sent in control
                                 * messages.
                                 */
};
typedef struct message_level MsgLevel;

struct ClientStatsDS
{
        int             read_requests; /* Total number of read requests made.*/
        int             write_requests;/* Total number of write requests made.*/
        MsgLevel        normal;        /* Data collected during normal operation
                                        * of the client.*/
        MsgLevel        migration;     /* Data collected when this client is
                                        * migrating. */
        MsgLevel        recovery;      /* Data collected when this client is
                                        * recovering. */

};

struct ClientStatsDS  *globalClientStats;
```

```
        /* Array of size NO_OF_CLIENTS. Memory allocated in usermain.
         */

struct ServerStatsDS
{
        MsgLevel        normal;         /* Data collected during normal operation
                                         * of the server. */
        MsgLevel        migration;      /* Data collected for this server's
                                         * migrating clients. */
        MsgLevel        recovery;       /* Data collected when this server is
                                         * recovering.
                                         * The data_msg of this is not used.
                                         * Only control messages are sent to
                                         * the server during recovery.
                                         */
};

struct ServerStatsDS  *globalServerStats;
        /* Array of size NO_OF_SERVERS. Memory allocated in usermain.
         * NO_OF_SERVERS is defined in paramUser.h.
         */


/* The total simulation run time is divided into time periods.
 * The information below is maintained on a 'per cluster' basis,
 * for each time period. This information is used to evaluate
 * the load on a cluster before, during and after a migration.
 */
struct ClusterIntervalData
{
        /* Control messages received by a client or a server in
         * a particular cluster. These are normal messages received
         * related to any requests from within the cluster (not a request
         * from another cluster).
         */
        unsigned long  *in_cluster_control_byte;
        unsigned long  *in_cluster_control_msg;

        /* Data messages received by a client or a server in
         * a particular cluster. These are normal messages received
         * related to any requests from within the cluster (not a request
         * from another cluster).
         */
        unsigned long *in_cluster_data_byte;
        unsigned long *in_cluster_data_msg;

        /* Control messages received by a client or a server in
         * a particular cluster. These are normal messages received
         * related to any requests from outside the cluster (not a request
         * from the same cluster).
         */
        unsigned long *out_cluster_control_byte;
        unsigned long *out_cluster_control_msg;

        unsigned long *out_cluster_data_byte;
        unsigned long *out_cluster_data_msg;

        /* Control messages received by a client or a server in
         * a particular cluster when a client is migrating. This is whether
         * a client is migrating in or migrating out of this cluster.
         */
        unsigned long *cluster_mig_control_byte;
        unsigned long *cluster_mig_control_msg;

        unsigned long *cluster_mig_data_byte;
        unsigned long *cluster_mig_data_msg;
};

struct ClusterIntervalData    *globalIntervalData;
        /* Array of size NO_OF_SERVERS (which is number of clusters).
         * Relevant messages received by a client or a server are
         * added to the relevant metric for that time interval.
         * Memory allocated in usermain.
         * NO_OF_SERVERS is defined in paramUser.h.
         */
```

```
FILE    *migFilePtr;/*Pointer to migration interval  output file.*/

int     global_total_hit_metric;
int     global_total_miss_metric;

/* Average and standard deviation of the time taken
 * for a request for a page not in the local memory of
 * a client. Measured in sendRequest by userThread.
 */
int     global_request_latency_avg_metric;
int     global_request_latency_stddev_metric;

/* Average and standard deviation of the total number of requests
 * (local/non-local, in/out cluster) made by a client in the
 * CHECK_FOR_MIGRATION_INTERVAL. This is used to tweak interval
 * if necessary.
 * The system should be set up so that every client makes about
 * 100 (?) requests on the average in this interval. Based on
 * these 100 requests, a decision is made as to when a client
 * should migrate to another cluster.
 * Measured by checkMigrationThread in findNewServer.
 */

int     global_tot_requests_mig_interval_by_client_avg_metric;
int     global_tot_requests_mig_interval_by_client_stddev_metric;


/* This max is greater than the stipulated minimum number of
 * requests a client must make to migrate.
 */
int     global_max_incluster_requests_by_client_avg_metric;
int     global_max_incluster_requests_by_client_stddev_metric;

/* This max is greater than the stipulated minimum number of
 * requests a client must make to migrate.
 */
int     global_max_outcluster_requests_by_client_avg_metric;
int     global_max_outcluster_requests_by_client_stddev_metric;


/* This max is the maximum number of requests made by a client in
 * the CHECK_FOR_MIGRATION_INTERVAL. However, this max is less than the
 * stipulated minimum number of requests a client must make to migrate.
 */
int     global_max_not_enough_requests_by_client_avg_metric;
int     global_max_not_enough_requests_by_client_stddev_metric;

/* The percentage of in cluster requests (out of total requests
 * made in interval) made by a client in the migration interval.
 * Will give indication of how realistic the  CHECK_FOR_MIGRATION_INTERVAL
 * is. This interval should record sufficient number of
 * requests to make a decision based on the 'cluster' data access
 * pattern of a client.
 */
int     global_percent_incluster_requests_by_client_avg_metric;
int     global_percent_incluster_requests_by_client_stddev_metric;

/* When the total number of requests is 0, how do you calculate
 * the percent of in cluster requests? (Not counted in the percent)
 * For now, keep a count of the total number of readings for
 * the global_percent .metrics. And calculate what percentage
 * of the requests had 0 as the total
 */
int     global_mig_interval_positive_requests;
int     global_mig_interval_zero_requests;

/* Average and standard deviation of the total time taken
 * by a client to migrate. Migration involves the old server,
 * the client and the new server. This metric measures
 * the time between the instant a client is told to migrate
 * until it is accepted by its new server.
 */
int     global_client_mig_duration_avg_metric;
int     global_client_mig_duration_stddev_metric;

int     global_mean_inter_request_time_avg_metric;
```

```
FILE    *migFilePtr;/*Pointer to migration interval  output file.*/

int     global_total_hit_metric;
int     global_total_miss_metric;

/* Average and standard deviation of the time taken
 * for a request for a page not in the local memory of
 * a client. Measured in sendRequest by userThread.
 */
int     global_request_latency_avg_metric;
int     global_request_latency_stddev_metric;

/* Average and standard deviation of the total number of requests
 * (local/non-local, in/out cluster) made by a client in the
 * CHECK_FOR_MIGRATION_INTERVAL. This is used to tweak interval
 * if necessary.
 * The system should be set up so that every client makes about
 * 100 (?) requests on the average in this interval. Based on
 * these 100 requests, a decision is made as to when a client
 * should migrate to another cluster.
 * Measured by checkMigrationThread in findNewServer.
 */

int     global_tot_requests_mig_interval_by_client_avg_metric;
int     global_tot_requests_mig_interval_by_client_stddev_metric;


/* This max is greater than the stipulated minimum number of
 * requests a client must make to migrate.
 */
int     global_max_incluster_requests_by_client_avg_metric;
int     global_max_incluster_requests_by_client_stddev_metric;

/* This max is greater than the stipulated minimum number of
 * requests a client must make to migrate.
 */
int     global_max_outcluster_requests_by_client_avg_metric;
int     global_max_outcluster_requests_by_client_stddev_metric;


/* This max is the maximum number of requests made by a client in
 * the CHECK_FOR_MIGRATION_INTERVAL. However, this max is less than the
 * stipulated minimum number of requests a client must make to migrate.
 */
int     global_max_not_enough_requests_by_client_avg_metric;
int     global_max_not_enough_requests_by_client_stddev_metric;

/* The percentage of in cluster requests (out of total requests
 * made in interval) made by a client in the migration interval.
 * Will give indication of how realistic the  CHECK_FOR_MIGRATION_INTERVAL
 * is. This interval should record sufficient number of
 * requests to make a decision based on the 'cluster' data access
 * pattern of a client.
 */
int     global_percent_incluster_requests_by_client_avg_metric;
int     global_percent_incluster_requests_by_client_stddev_metric;

/* When the total number of requests is 0, how do you calculate
 * the percent of in cluster requests? (Not counted in the percent)
 * For now, keep a count of the total number of readings for
 * the global_percent..metrics. And calculate what percentage
 * of the requests had 0 as the total
 */
int     global_mig_interval_positive_requests;
int     global_mig_interval_zero_requests;

/* Average and standard deviation of the total time taken
 * by a client to migrate. Migration involves the old server,
 * the client and the new server. This metric measures
 * the time between the instant a client is told to migrate
 * until it is accepted by its new server.
 */
int     global_client_mig_duration_avg_metric;
int     global_client_mig_duration_stddev_metric;

int     global_mean_inter_request_time_avg_metric,
```

```
int     global_mean_inter_request_time_stddev_metric;

struct BottleneckServerMetric
{
        int     avg_metric;
        int     stddev_metric;
};

struct BottleneckServerMetric *globalBottleneckMetric ;

/* No of intervals, or the size of the array, for globalIntervalMetrics
 */
int     MAX_MIG_INTERVALS;

int     NO_OF_CLIENTS;

int     TOTAL_NO_MIGRATIONS;
                        /* The total number of successful migrations.
                         * When a server accepts a new client it increments
                         * this value. Does not use semaphore.
                         */

int     UPDATE_TIME;    /* Time for which a client can keep an out of cluster page.
                         * A client can keep an out of cluster page for
                         * UPDATE_TIME * TIMER_PEROID cycles.
                         */
int     TIME_TO_REACH_OUT_CLUSTER_SERVER;
                        /* Time taken for an update page to reach another server.
                         * This time is TIME_TO_REACH_OUT_CLUSTER_SERVER * TIMER_PERIOD
                         * cycles.*/
int     TIME_LATER;     /* Extra time for which a user must wait if it gets a RESEND
                         * REQUEST LATER for a request it sent.
                         * This time is TIME_LATER * TIMER_PERIOD cycles.
                         */
int     WATCHDOG_SLEEP_TIME;
int     RECOVERY_SLEEP_TIME;
                        /* The duration of the server/client sleep
                         * time when the system is recovering.
                         * The server/client thread sleeps for
                         * RECOVERY_SLEEP_TIME * TIMER_PERIOD cycles
                         */
int     WAIT_RECOVERY_CONDITION_SLEEP_TIME;
                        /* Used by:
                         * serverCheckIfFailureInClusterThread,
                         * serverSendingRecoveryMessagesThread
                         * clientDownThread, clientUpFromFailureThread,
                         * clientSendingRecoveryMessagesThread.
                         */

int     MIGRATING_SLEEP_TIME;
                        /* This is the duration for which threads which
                         * are waiting for a migration to complete sleep.
                         * Used by clientThread,clientMigratingThread.
                         * clientPollingNewServerThread,userThread
                         * serverGivingPermissionToMigrateThread
                         * The thread sleeps for
                         * MIGRATING_SLEEP_TIME * TIMER_PERIOD cycles.
                         */

int     CHECK_FOR_MIGRATION_INTERVAL;
                        /* Interval at which the check to see if any
                         * clients should migrate is made. Used by
                         * checkMigrationThread. The thread sleeps for
                         * CHECK_FOR_MIGRATION_INTERVAL * TIMER_PERIOD cycles
                         */

int     MIGRATED_WAITING_FOR_ACCEPTANCE_INTERVAL;
                        /* Interval for which a client (that has been given
                         * permission to migrate) waits when polling its
                         * new server asking for acceptance.
                         */
int     CHECK_SERVER_BOTTLENECK_INTERVAL;
                        /* The bottleneck of the server when system is
                         * scaled up will give an idea about whether
                         * adding more servers is increasing throughput
                         * by parallel processing.
```

```
                        * A thread is created at every server. At fixed
                        * intervals of time, it checks the number of requests
                        * in the servers request queue. This bottleneck
                        * is measured ONLY when no failures are allowed
                        * in the system.
                        */

int     EMPTY_QUEUE_WAIT_TIME; /* When threads pull out requests from
                                * a request queue, and find the queue
                                * empty, they wait for EMPTY_QUEUE_WAIT_TIME
                                * * TIMER_PERIOD cycles.
                                * Used by:
                                * clientThread, serverThread,
                                * serverRecoveringThread,
                                * clientRecoveringThread.
                                */
int     SYSTEM_INIT_TIME;      /* When the userThread is first created
                                * (in messages.ca), it is put to sleep for
                                * SYSTEM_INIT_TIME * TIMER_PERIOD cycles,
                                * so that other servers and the rest of
                                * the system is initialised before the
                                * user starts sending requests.
                                */
Sem request_mutex;
                       /* Semaphore used to increment TOTAL_NO_REQUESTS.
                        * A semaphore is required, as user threads on
                        * various processors will increment this value.
                        */

int     global_to_test_quantum;

#ifdef  RECOVERY_HARD_CODE

Sem     recovery_user1, recovery_user2;
Sem recovery_mutex[NO_OF_PROCESSORS]; /* Output semaphore; used for printf. */

int     RECOVERY_HARD_CODE_TIME;
int     RECOVERY_STARTED;


#endif

/* Functions in initSC.ca. */
void    initServerDSHandler(int argc, Word *argv);
void    initClientDSHandler(int argc, Word *argv);
void    installPagesAtClientsThread();
void    distributePagesAmongClients(int ***cList, int ***tList);
int     getSecondIndex(int no_clients);
int     getSecondFromFirst(int first, int diff, int no_clients);
void    checkForInstallationEndThread();


/* Functions in messages.ca */
void    msgFromServerHandler(int argc, Word *argv);
void    installClockMsgFromServerThread(int no_args,int msg_code,
                int page_no, int page_mode, int trailer_version,
                int page_service, int page_value, int no_readers,
                int reader1, int reader2);
void    installReaderTrailerMsgFromServerThread(int no_args,int msg_code,
                 int page_no ,int page_mode, int trailer_version,
                 int page_service,int page_value, int a, int b, int c);
void    ackFromClientHandler(int argc, Word *argv);
void    broadcastServerStateThread();
void    setServerStateThread(int no_args, int msg_code,int server_state,
                int a,int b, int c, int d, int e, int f, int g);
void    setClientStateThread(int msg_code, int client_state, int dummy);


/* Functions in requests.ca; related to sending requests, client site. */
void    userThread(void);
void    getRequestFromRandom(int request_no,int *page_no,
                int *access_mode, int *where, int *when);
int     memoryOperation(int page_no, int access_mode, int kind_of_access);
int     sendRequest(int page_no, int access_mode);
void    watchdogThread(int tid_of_user,int uniq_req);
```

```
void    replyMakeReader(reqNode *request);
void    replyMakeReaderTrailer(reqNode *request);
void    replyMakeWriterClock(reqNode *request);
void    replyResendRequestLater(reqNode *request);
void    replyWakeUpUser(reqNode *request);

/* Functions in serverfn.ca ; server sending replies, server side.*/
void    requestForServerHandler(int argc, Word *argv);
void    serverThread();
void    measureServerBottleneckThread();
void    processRequestAtServer(reqNode *request);

void    readRequestClockWriterServer(reqNode *request);
void    writeRequestClockReaderOrWriterServer(reqNode *request);

void    ackOfClockChangeServer(reqNode *request);

void    informClockModeChangeServer( reqNode *request);
void    informMsgNotDeliveredServer( reqNode *request);

void    getServerCodeFromServerProc(int server_proc,int *server_code);
void    getServerIndexFromServerCode(int server_code,int *server_index);
void    getServerIndexFromServerProc(int server_proc,int *server_index);
int     isItServerCode(int server_code);


/* Functions in clientfn.ca; client handling clock operations,
 * service requests,send msgs, invalidate, etc.*/

void    requestForClientHandler(int argc, Word *argv);
void    clientThread();
void    processRequestAtClient(reqNode *request);

void    readerClockHandlingReadRequest(reqNode *request);
void    writerClockHandlingReadRequest(reqNode *request);
void    readerClockHandlingWriteRequest(reqNode *request);
void    writerClockHandlingWriteRequest(reqNode *request);
int     invalidateAllReaders(int page_no, int check_reader);
void    invalidateReaderOfPage(reqNode *request);

/* functions in servermulti.ca: server functions related to
 * multiserver case.*/
void    serverToServerHandler(int argc, Word *argv);
void    ssInformServerThread(int server_proc, int server_index,
             int lowest_page, int highest_page,int server_state);

void    serverUpdatingClientDataAccessPattern(reqNode *request);
void    requestForOutClusterPageAtServer(reqNode *request);
void    requestFromOutOfClusterPageAtServer(reqNode *request);
void    serverHandlingReadReplyOutClusterFromItsClient(reqNode *reply);
void    serverHandlingReadReplyToOutClusterRequest(reqNode *reply);

void    serverInvalidatingOtherServerReaders(reqNode *request);
void    serverInvalidatingAllButOneOtherServerReaders(reqNode *request);
void    serverInvalidatingItsOutClusterReadPage(reqNode *request);

void    serverHandlingWriteReplyOutClusterFromItsClient(reqNode *reply);
void    serverHandlingWriteUpgradeReplyOutClusterFromItsClient(reqNode *reply);

void    serverHandlingWriteReplyToOutClusterRequest(reqNode *reply);
void    serverHandlingWriteUpgradeReplyToOutClusterRequest(reqNode *reply);

void    serverReturningWriteUpdatePageToOwnerServer(reqNode *request);
void    serverGettingItsWriteUpdatedPage(reqNode *request);
void    serverInvalidatingWriteUpdatePageItGaveToServer(reqNode *request);
void    serverInvalidatingWriteUpdateOutClusterPageTimeout(reqNode *request);
void    serverInvalidatingWriteUpdateOutClusterPageTimeoutWaiting(reqNode *request);

void    serverAskedToResendRequestOutClusterPageLater(reqNode *request);
void    ackWriteUpdateReachedServer(reqNode *request);
void    ackWriteUpdateNotSentServer(reqNode *request);

/* functions in clientmulti.ca; client functions related to multiserver
 * case*/
void    clockHandlingOutOfClusterRequest(reqNode *request);
void    readerClockHandlingOutOfClusterReadRequest(reqNode *request);
```

```
void    writerClockHandlingOutOfClusterReadRequest(reqNode *request);
void    readerOrWriterClockHandlingOutOfClusterWriteRequest(reqNode *request);

void    makeUrslfTempClockHandleOutClusterReadRequest(reqNode *request);
void    invalidateAllOutClusterReadersOfPage(reqNode *request);
void    invalidateOutClusterReaderOfPage(reqNode *request);
void    watchdogUpdateThread(int page_no, int update_coordination_code);
void    clockGettingItsWriteUpdatePage(reqNode *request);
void    invalidateOutClusterUpdateWriterPage(reqNode *request);
void    invalidateAllOutClusterReadersFwdWriteUpdate(reqNode *request);

/* functions in requestsmulti.ca; request functions related to multiserver
 * case, replies to out of cluster requests.*/
void    replyMakeReaderOutClusterPage(reqNode *request);
void    replyMakeUpdateWriterOutClusterPage(reqNode *request);
void    replyUpgradeFromReaderToUpdateWriterOutClusterPage(reqNode *request);


/* Functions in servermig.ca. */
void    checkMigrationThread();
void    iterateMigData();
int     findNewServer(int client_proc);

void    ssClientToMigrateIn(reqNode *request);
void    serverInvalidatingOutClusterReadPagesForWhichMigClientIsReader
            ( int mig_client);
void    ssRemoveServerAsReaderForPages(reqNode *request);

void    migrateHandoverReadClockAtServer(reqNode *request);
void    ackOfMigrateReadClockInstalled(reqNode *request);

void    migrateHandoverWriteClockAtServer(reqNode *request);
void    ackOfMigrateWriteClockInstalled(reqNode *request);

void    migrateHandoverAllInvalidateReadPagesAtServer(reqNode *request);

void    clientReadyToMigrate(reqNode *request);
void    serverGivingPermissionToMigrateThread(int client_mig);
void    ssHandingOverNewClient(reqNode *request);
void    newClientAskingForAcceptance(reqNode *request);

void    serverHandlingMigNotInstallingOldClusterWriteClockFromItsClient
            (reqNode *request);
void    serverHandlingMigNotInstallingOldClusterWriteClockFromAnotherServer(reqNode
*request);


/* Functions in clientmig.ca. */
void    clientMigratingThread();
void    clientMigrateSendAllInClusterReadClocks(int page_no);
void    clientMigrateHandoverInstallNewReadClock(reqNode *request);

void    clientMigrateSendAllInClusterWriteClocks(int page_no);
void    clientMigrateHandoverInstallNewWriteClock(reqNode *request);

void    clientInvalidateAllOutClusterReadClocksBeforeMigrating(int page_no),
void    clientMigrateInvalidateOneOutClusterReadClock(int page_no),

void    clientMigrateSendAllInAndOutClusterReadPages();
void    clientMigrateHandoverRemoveClientAsReader(reqNode *request);

void    clientMigrateRecheckIfAllPagesSent();
void    clientPollingNewServerThread(int code, int new_server);

/* Functions in failure.ca.*/
void    failureThread();
void    wakeupFailureThreadHandler(int argc, Word *argv);
void    destroyFailureThreadHandler(int argc, Word *argv);

/* Functions in serverrec.ca. */
void    serverDetectFailureHandler(int argc, Word *argv);
void    serverCheckIfFailureInClusterThread(int site, int down_time);
void    serverSendingRecoveryMessagesThread(int down_site);
void    checkAndInstallClocks();
void    checkAndInstallTrailers();
void    changeInAnotherCluster(reqNode *request);
```

```
void    recoveryToServerHandler(int argc, Word *argv);
void    serverRecoveringThread();
int     processRecoveryMessageAtServer(reqNode *request);
void    serverRecoveryUnpackingClockReaderBitmaps(reqNode *request);
void    serverRecoveryUnpackingTrailerBitmap(reqNode *request);
void    serverRecoveryClientRecovered(reqNode *request);
void    serverRecoveryAckReadClockInstalled(reqNode *request);
void    serverRecoveryAckTrailerInstalled(reqNode *request);
void    serverRecoveryCompleteSetServerOK();


/* Functions in clientrec.ca.*/
void    clientDownThread(int down_time);
void    clientUpFromFailureThread();
void    clientSendingRecoveryMessagesThread(int down_site);
void    sendAllInClusterRecoveryInfo();
void    setBitInBitmap(Word *bitmap, int arch_size, int pos);
int     isBitSet(int bitmap, int size, int pos);
int     getNextPage(int bitmap, int size, int pos);
void    recoveryToClientHandler(int argc, Word *argv);
void    clientRecoveringThread();
int     processRecoveryMessageAtClient(reqNode *request);
void    clientRecoveryInstallReadClock(reqNode *request);
void    clientRecoveryInstallTrailerToClock(reqNode *request);
void    clientRecoveryInstallReaderTrailer(reqNode *request);
void    clientRecoveryComplete();
void    clientRecRemoveServerAsReaderForAPage(reqNode *request);

void usermain(int argc, char **argv);

#endif FAULT_TOL_H
/*ifndef FAULT_TOL_H */
```

```
/*****************************************************************************
 * FILE     :     serverfn.ca                                                *
 * CONTENTS:      This file contains all the functions which deal            *
 *                with an in-cluster message at a server site (in both       *
 *                single and multi-server systems).                          *
 *****************************************************************************/
#include 'ft.h'


/*****************************************************************************
  FUNCTION: requestForServerHandler
  PURPOSE : This handler is called when a request is sent to it.
            This handler puts the request into the server requests
            queue. The server thread pulls out requests in a FIFO
            manner.
            Uses REQUEST_FOR_SERVER_IPI, runs at server

  INPUTS  :

        GENERAL MESSAGE FORMAT
        ----------------------

        argv[0]                                        request_code
        argv[1]                                        no of args (from argv[2])
        argv[2]-argv[ no of args -2]                   the arguments to be put
                                                       into the array in a request.

        REQUEST CODES USED
        ------------------

        **  0. request code                            S_FROM_USER_REQUEST,request
                                                       sent by a user in this
                                                       cluster.
            1. no args                                 6
            2. page_no                                 the page required
            3. access_mode                             READ/WRITE
            4. duplicate request number
            5. tid of user
            6. processor of user
            7. unique requests sent so far

    **  0. request_code                                S_ACK_CLOCK_CHANGE, ack of
                                                       clock change from user.
            1. no args                                 4
            2. page_no
            3. clock_site                              the processor of the site
                               sending this ack. This is the new clock site.
            4.  clock_mode                             READ,WRITE,

                                                       READ_CLOCK,WRITE_CLOCK
            5. t_version                               Since the clock has changed,it
                               will have a new trailer version number. This need not
                               be sent to the clock (is needed only during recovery).
                               but no harm updating the server now in the same msg.
                               (The site of the trailer is old)
    **  0. request_code                                S_INFORM_CLOCK_MODE_CHANGE, if
                               user at clock site changes the mode of the
                               clock, it informs the server
            1. no args                                 3
            2. page_no
            3. clock_mode                              READ,WRITE,
                                                       READ_CLOCK,WRITE_CLOCK
            4. clock_site                              the processor of the site
                               sending this msg. This is needed to check if the clock
                               site is still the same.

        **  0. request_code                            S_MSG_NOT_DELIVERED,
                               the server has set service_later_ignore or
                               service_later_reply for this page. Since the clock site
                               has not been able to deliver this msg, it is informing
                               the server to change the service back to service_now
                               for this page.
            1. no_args                                 1
            2. page_no

        **  0. request_code                            S_INFORM_PAGE_ACCESS
```

```
                    When a client access a shared page locally, it
                    sends this message to it server. The server can
                    maintain the history of accesses of every client
                    to determine when a client should migrate, and to
                    what cluster. This message is sent only in a
                    multi-server environment.
   1. no args                                  2
   2. client_proc
   3. page no                                  the page that was accessed.


** 0. request_code                             SS_OUT_CLUSTER_REQUEST,
                    another server is sending this server a request for
                    a page belonging to this server.
   1. no args                                  5
   2. from_server
   3. key_at_from_server
   4. page_no
   5. reqd_access                              READ,WRITE
   6. dup_req_no


** 0. request_code
                    S_READ_REPLY_OUT_CLUSTER_FROM_MY_CLIENT.
                    C2 sends a reply to out of cluster read request to s2.
   1. no args                                  5
   2. this_server_key                      key 2 at s2.
   3. requesting_server_code         used by s2 to locate
                    incoming_request node for s1.
   4. page no
   5. page_mode                                READ
   6. page_value


** 0. request_code
                    SS_READ_REPLY_OUT_CLUSTER_REQUEST
                    s2 sends a reply to a read request made by s1.
   1. no args                                  5
   2. server_replying                          s2 (s1 looks up other_servers_
                    info[s2's index]
   3. your_server_key                          key1 at s1
   4. page no
   5. page_mode                                READ
   6. page_value

** 0. request_code
                    S_INVALIDATE_ALL_READERS_AT_OTHER_SERVERS
                    This server gets a message from its clock of
                    page_no to invalidate all readers outside this
                    cluster. (This clock sends this msg when it is
                    invalidating its readers and sees that a server
                    is a reader).
   1. no args                                  1
   2. page_no

** 0. request_code
                    S_INVALIDATE_ALL_BUT_ONE_READERS_AT_OTHER_SERVERS
                    This server gets a message from its clock of
                    page_no to invalidate all readers outside this
                    cluster. (This clock sends this msg when it is
                    invalidating its readers and sees that a server
                    is a reader). This message is sent when a server
                    that is already a reader, requests a write. The
                    server sends invalidation messages to all servers
                    that are readers, except the server that made the
                    write request.
   1. no args                                  2
   2. page_no
   3. server_code                              the code of the server reader
                    which should not be invalidated.


** 0. request_code
                    SS_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE
                    s1 has a read copy of page belonging to s2.
                    s2 sends s1 this message telling it to invalidate
```

```
                    all its readers of specified page.
   1. no_args                                    1
   2. page_no

** 0. request code
                    S_WRITE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT,
                c2 sends a reply to out of cluster write request to
                s2, when the requesting server is not already a reader.
   1. no_args                                    7 (excluding msg code)
   2. this_server_key                            key2 at s2
   3. requesting_server_code                     used to locate incoming
                    request node for s1.
   4. page_no
   5. page_mode                                  WRITE_UPDATE
   6. page value
   7. update time                                the time for which s1 can keep
                    the write update page
   8. update coordination code            (stored in auxClient(page_no).
                    extra at c2) This is used for coordination between the
                    updated page returned by s1 and c2.

** 0. request code
                    S_WRITE_UPGRADE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT,
                    (B4-1)
                c2 sends a reply to out of cluster write request to
                s2, when the requesting server is already a reader.
                (does not send the page value)
   1. no_args                                    7 (excluding msg code)
   2. this_server_key                            key2 at s2
   3. requesting_server_code                     used to locate incoming
                    request node for s1.
   4. page_no
   5. page_mode                                  WRITE_UPDATE
   6. page value                                 This value is sent as the
                    requesting server might have invalidated the page if
                    its reader client was migrating.
   7. update time                                the time for which s1 can keep
                    the write update page
   8. update coordination code                   This is used for coordination
                    between the updated page returned by s1 and c2.
                    (Stored in     auxClient(page_no).extra at c2.)

** 0. request_code
                    SS_WRITE_REPLY_OUT_CLUSTER_REQUEST
                    S2 sends a reply to a write request made by s1.
1. no args                                       6
   2. server_replying                            s2 (s1 looks up other_servers_
                    info[s2's index]
3. your_server_key                               key1 at s1
4. page no
5. page_mode WRITE_UPDATE
6. page_value
   7. update time

** 0. request_code
                    SS_WRITE_UPGRADE_REPLY_OUT_CLUSTER_REQUEST
                    s1 is a reader for an out of cluster page. It made a
                    write request for that page. It gets this reply from
                    s2 (B5-1)
   1. no args                                    6
   2. server_replying                            s2 (s1 looks up other_servers_
                    info[s2' s index]
   3. my_server_key                              key1 at s1
   4. page no
   5. page_mode                                  WRITE_UPDATE
   6. page value
7. update time

** 0. request code
                    S_RETURNING_OUT_CLUSTER_UPDATE_PAGE
                    c1 returns an update write (out of cluster) page to s1
                    after writing to it. s1 will return it to s2.
   1. no args                                    3
   2. page_no
   3. page value                                 (new updated value)
   4. client site                                the site returning the page.
```

** 0. request code
SS_RETURNING_OUT_CLUSTER_UPDATE_PAGE
s1 returns an update write page to s2 (the owner) after writing to it
1. no args 3
2. page no
3 page value
4. server_returning_page, s1's processor

** 0. request code S_INVALIDATE_WRITE_UPDATE_PAGE
c2 has given a write update page to s2 to be given to another server. c2's watchdog wakes up and it finds that it hasnt yet got the update page. c2 sends this msg to s2 telling it to invalidate the write update page
1. no args 2
2. page no
3. update code

** 0. request code
SS_INVALIDATE_WRITE_UPDATE_PAGE (B11)
s2 has given s1 a write update page. This message is sent by s2 to s1 telling it to invalidate that write update page.
1. no args 1
2. page no

** 0. request code
SS_INVALIDATE_WRITE_UPDATE_PAGE_WAITING(B13)
s2 gave a write update page to s1. s2 sends SS_INVALIDATE_WRITE_UPDATE_PAGE to s1. Before the write update page reaches the client(c1) in s1 which requests it, s1 got the invalidation message. Instead of sending it to c1, s1 re-inserted it as this message.
1. no args 2
2. page no
3. client site c1's processor number.

** 0. request code
SS_RESEND_REQUEST_FOR_PAGE_LATER (D1)
s2 gets a request for a page from s1. s2 has service later reply for this page. Sends this message back to s1.
1. no args, 2
2 page no
3. key at from server, key1

** 0. request code
S_ACK_RECEIVED_WRITE_UPDATE_PAGE (B8-1)
s1 is a reader of page belonging to s2. A client in s1,c1-1, requested a write. s2 sent a reply . s1 changed extra to NOT_REACHED, and forwarded the message to the temp read clock. The temp read clock forwarded the reply to c1-1.. on getting the reply c1-1 sends this message to s1.
1. no args 2
2. page no
3. c1-1's processor number

** 0. request code
S_ACK_NOT_SENDING_WRITE_UPDATE_PAGE (B8-2)
s1 is a reader of page belonging to s2. A client in s1, c1-1, requested a write. s2 sent a reply.. s1 changed extra to NOT_REACHED, and forwarded the message 'C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE' with ack code 'SEND_ACK_WRITE_UPDATE_REACHED_MSG' to the temp read clock, c1. But c1 was migrating. And since the clientThread and clientMigratingThread at c1 are running in parallel, c1 is no longer the reader when it gets C_INVALIDATE_ALL... So c1 sends this message back to s1.
s1 will remove c1-1 from memory when it gets this.
1. no args 1
2. page no

```
REQUESTS RELATED TO MIGRATION
-----------------------------

**  0. request code
                        SS_CLIENT_TO_MIGRATE_IN (G2)
                        c1 from s1 will be migrating to s2. s1 informs s2 about
                        this. This msg is recd by s2.
    1. no args                                  2
    2. c1's processor number
    3. s1's processor number

**  0. request code
                        SS_REMOVE_MY_SERVER_AS_READER_FOR_PAGES
                        The server which sends this message is invalidating
                        some read pages it borrowed (because its temp read
                        clock was migrating). This server removes those read
                        pages from list of read pages given to that server.
    1. no args                                  2 + no_pages
    2. sending server index
    3. no pages
                        (CHECKED CORRECT PAGE RETRIEVAL)
    4. page1
    5. page2
    6. page3,. and so on

**  0. request code
                        S_MIG_HANDOVER_READ_CLOCK (H1)
                        A client c1 is migrating. It sends this
                        message for every in-cluster page for which it is
                        a read clock.
    1. no args                                  6 + no of readers
    2. page number
    3. page status                              READ_CLOCK
    4. page value
    5. client processor                         client which is handing over
                        the clock, c1, which is migrating
    6. trailer version
    7. number of readers
                        (CHECKED CORRECT READER RETRIEVAL)
    8. reader 1
    9  reader 2 . and so on

**  0. request code
                        S_ACK_MIG_READ_CLOCK_INSTALLED
                        c1 was a read clock. It is migrating to s2. A new
                        read clock is installed at c2. This message sends
                        an ack to s1 telling it that the new read clock is
                        installed successfully. (H3)
    1. no args                                  2
    2. page no
    3. new clock's processor


**  0. request code
                        S_MIG_HANDOVER_WRITE_CLOCK (K1)
                        Client c1 is migrating. It sends this
                        message for every in-cluster page for which it is
                        a write clock.
    1. no args                                  5
    2. page number
    3. page status                              WRITE_CLOCK
    4. page value
    5. client processor                         client which is handing over
                        the clock, c1, which is migrating
    6. trailer version

**  0. request code
                        S_ACK_MIG_WRITE_CLOCK_INSTALLED(K3)
                        c1 was a write clock. It is migrating to s2. A new
                        write clock is installed at c2. This message sends
                        an ack to s1 telling it that the new write clock is
                        installed successfully.
    1. no args                                  2
    2. page no
    3. new clock's processor
```

```
** 0. request code
                    S_ACK_MIG_READ_CLOCK_INSTALLED (I3)
                    c1 was a temp read clock for an out cluster page.
                    It is migrating to s2. A new read clock is installed at
                    c2. This message sends an ack to s1 telling it that the
                    new temp read clock is installed successfully
   1. no args                                       2
   2. page no
   3. new clock's processor

** 0. request code
                    S_MIG_HANDOVER_INVALIDATE_READ_PAGES (J1)
                    c1 is a client which is migrating to a different
                    cluster. It sends this message to s1 giving it
                    a list of pages for which it is a reader (not read
                    clock). This list includes both in and out cluster
                    pages.It sends this message to s1 even if it
                    doesnt have any 'just read' pages.
   1. no args                                       2 + no_pages
   2. c1's processor number
   3. no_pages being sent
                    (CHECKED CORRECT PAGE RETRIEVAL)
   4. page 1
   5. page 2.. and so on

** 0. request code
                    S_CLIENT_READY_TO_MIGRATE(L1)
                    A client c1 has sent all its handover messages to
                    s1 during migration. It's user and client are
                    waiting to migrate. It sends this message to s1
                    so s1 can check if all the handover's have been
                    ack'd, and allow c1 to migrate
   1. no args                                       1
   2. c1's processor number

** 0. request code
                    SS_HANDING_OVER_NEW_CLIENT(L3)
                    Client c1 is migrating to s2 from s1. This is the final
                    message s1 sends to s2 telling it that c1 will migrate
                    to s2.
   1. no args                                       2
   2. c1, client migrating in
   3. from_server_proc

** 0. request code
                    S_NEW_CLIENT_ASKING_FOR_ACCEPTANCE (N1)
                    A client c1 has migrated from s1 to s2 (It
                    got permission from s1). s2 could have
                    crashed without getting the handover message
                    from s1. So, this client acts intelligently
                    and keeps polling s2 asking for an ack.
                    REQUEST_FOR_SERVER_IPI
   1. no args                                       1
   2. new_client_proc

** 0. request_code
                    S_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK
                    A client c1 was migrating from cluster 1 to cluster 2.
                    When in cluster 1 it requested a write page in cluster 1
                    It got a reply 'after' it migrated to cluster 2. Now
                    that page is an out of cluster page. So, c1 cannot install
                    the page as a write clock in replyMakeWriteClock. It sends
                    the page back to its new server, the server in cluster 2.
                    s2 will have to forward the page to s1. s1 will have to
                    install a new write clock. s1 was waiting for an
                    acknowledgment from c1 if c1 was in cluster 1, and hence
                    it has to be told that this write clock has not been
                    installed.
                    This server is in cluster 2.
  1. no args                                        5
  2. client proc                                    the client sending this
message
  3. page no
  4. access_mode                                    WRITE_CLOCK
```

```
            5. page_value
            6. trailer_version                                the latest t version in
                                                              cluster 1
        ** 0. request_code
                              SS_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK
                              A client c1 was migrating from cluster 1 to cluster 2.
                              When in cluster 1 it requested a write page in cluster 1.
                              It got a reply 'after' it migrated to cluster 2. Now
                              that page is an out of cluster page. So, c1 cannot install
                              the page as a write clock in replyMakeWriteClock. It sends
                              the page back to its new server, the server in cluster 2.
                              s2 will have to forward the page to s1. s1 will have to
                              install a new write clock. s1 was waiting for an
                              acknowledgment from c1 if c1 was in cluster 1, and hence
                              it has to be told that this write clock has not been
                              installed.
                              This server is in cluster 1 which got the message from
                              server 2.
            1. no args                                        4
            2. page no
            3. access_mode                                    WRITE_CLOCK
            4. page_value
            5. trailer_version                                the latest t version in
                                                              cluster 1
        REQUEST RELATED TO OTHER SERVER(S) STATE (RECOVERY)
        -------------------------------------------------

        **  0. request code                                   SS_SET_SERVER_STATE
                              When a server goes down or starts the recovery process,
                              it informs all other servers about its state.
                              A server that gets this message has to make sure
                              its cluster is consistent.
                              This message does not use the SERVER_TO_SERVER_IPI. as
                              a lot of processing to make data consistent has to be
                              done. It was considered more reasonalbe to do this
                              via the request queue, than give it a special priority
                              via the SERVER_TO_SERVER_IPI (which does not go
                              through queue)
             1. no args                                       3
             2. server_processor
             3. server_index
             4. server_state                                  SERVER_DOWN/SERVER_RECOVERING/
                                                                             SERVER_OK

OUTPUTS : none
 *****************************************************************************/
void    requestForServerHandler(int argc, Word *argv)
{
        perServerDS     *thisServer;
        int             ret;

        thisServer = (perServerDS *) (*(ptrToGlobalServerDS));
        /* argv[1] is number of values in the request. It should not
         * be the value to which SEND_ARRAY is initialized.
         */
        assert(argv[1] != INIT_SEND_ARRAY);

        if(globalStateInstall != INSTALLATION_COMPLETE)
                return;

        /* If SERVER_OK, put the request in the request queue.*/
        /* If the server is SERVER_NEVER_STARTED,SERVER_DOWN,
         * SERVER_TO_RECOVER,SERVER_RECOVERING, do not put in request q.*/
        if ( (thisServer->server_state & SERVER_OK)  == SERVER_OK)
        {
                ret = insertIntoRequestQ(thisServer-
>server_request_queue,argv[1],argv[0],&argv[2]);
                if ( ret == ERROR_Q)
                {
                        fatal(" ERROR: server failed to insert request into server request
queue\n");
                }
        }
}

/*****************************************************************************
```

```
  FUNCTION: serverThread
  PURPOSE : This thread runs in an infinite while loop. It pulls out
            requests from the server request queue. If the queue is
            empty it sits in an idle loop and checks again. It checks
            for the server and system state. If the server has gone
            down, it exits. Otherwise, it checks for some conditions
            and calls appropriate functions to handle requests.
            This thread is not counted in the no pending server threads.
  INPUTS  : none
  OUTPUTS : none
*****************************************************************************/
void    serverThread()
{
        perServerDS     *thisServer;
        reqNode         *next_request;
        int             ret_code; /* return value of getting a request
                                   * from request queue
                                   */
        long            initsleep,endsleep;
        int             cycles_slept;
        int             len,new_tid;

        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));

        /* BOTTLENECK is defined in paramUser.h and can be
         * modified in the corresponding prot.par file.
         * A thread is created which checks the number of requests
         * in a server's queue periodically. For now, this is done
         * only when there are no failures in the system. (The coding
         * doesnt take care of this). When no failures are simulated,
         * (specified in prot.par), only then shd BOTTLENECK be true.
         */
        if ( BOTTLENECK)
        {
                new_tid =
thread_create((FuncPtr)measureServerBottleneckThread,SMALL_STACKSIZE,PRIORITY,0);
                thread_wakeup(new_tid);
        }

        while( CURR_TIME < TOTAL_SIM_RUN_TIME_CYCLES)
        {
                /* If the SERVER_DOWN bit is set, the server thread
                 * must exit.
                 * set the SERVER_NEVER_STARTED bit and exit from the server
                 * thread. This thread will be re-started when
                 * the system recovers.
                 * The SERVER_TO_RECOVER bit is set by serverSendingRecovery
                 * MessagesThread.
                 * If the server is recovering, the server thread does not have
                 * to exit. It can just sit in a loop.
                 * The server_never_started and server_ok cannot both
                 * be set at the same time.
                 * The server_down and server_ok should not be set at
                 * the same time.
                 */
                begin_atomic();
                if(  (thisServer->server_state & SERVER_DOWN) == SERVER_DOWN)
                {
                        /* Since the server Thread is exiting, set the
                         * SERVER_NEVER_STARTED bit on.
                         * This is used to restart the server thread after
                         * recovery is complete.
                         */
                        emptyRequestQ(thisServer->server_request_queue);
                        thisServer->server_state &= 0;
                        thisServer->server_state |= SERVER_NEVER_STARTED;
                        end_atomic();
                        return;
                }
                end_atomic();

                /* If the server is recovering, no requests should be
                 * serviced. The request queue is only to manage
                 * read /write requests or related messages. It is
                 * not for recovery messages.
                 * But if the system is recovering, the server thread
```

```
             * will sleep in a loop waiting for it to recover, before
             * it can start servicing requests.
             */

            begin_atomic();
            if((thisServer->server_state & SERVER_TO_RECOVER) ==
                         SERVER_TO_RECOVER)
            {
                    /* Empty request queue*/
                    emptyRequestQ(thisServer->server_request_queue);
                    thisServer->server_state &= 0;
                    thisServer->server_state |= SERVER_RECOVERING;
            }

            while(  (thisServer->server_state & SERVER_RECOVERING) ==
SERVER_RECOVERING)
            {
                    if( CURR_TIME > TOTAL_SIM_RUN_TIME_CYCLES)
                    {
                            thisServer->server_state &= 0;
                            thisServer->server_state |= SERVER_NEVER_STARTED;
                            end_atomic();
                            return;
                    }
                    end_atomic();
                    BUSY_WAIT(RECOVERY_SLEEP_TIME_CYCLES);

                    begin_atomic();
                    /* Do not need a thread_sleep_end_atomic for
                     * this.*/
                    /* SINCE ONLY ONE SITE CAN GO DOWN AT A TIME, DO NOT
                     * NEED TO CHECK FOR SERVER_DOWN HERE. */
            }
            end_atomic();


            /* Pull out a request from the request queue.*/

            begin_atomic();
            next_request = getNextRequest(thisServer->server_request_queue, &ret_code);
            end_atomic();

            if ( ret_code == EMPTY_Q)
            { /* Wait in a loop for sometime, or put the server
               * thread to sleep.
               */

               /* Add 50 cycles to server thread*/
                    initsleep = CURR_TIME;
                    ASSERT_NOT_ATOMIC();
                    BUSY_WAIT(EMPTY_QUEUE_WAIT_TIME_CYCLES);

                    endsleep = CURR_TIME;
                    cycles_slept = (endsleep - initsleep)/TIMER_PERIOD;
            }
            else
            {
                    /* Process the request.*/
                    /* Must be in non atomic region here */
                    processRequestAtServer(next_request);
                    /* Will return here. Free the memory for
                     * the request here instead of doing it in
                     * the function where the request has been
                     * handled.
                     */
                    freeRequestNode(&next_request);
                    ASSERT_NOT_ATOMIC();
            }

    } /* end of while */
    /* End of simulation. Destroy the failure thread if it is
     * suspended/running. All servers will execute this code
     * at the end of the simulation, but only one call will
     * destroy the failure thread. This is done by using the
     * state of the failure thread. If this server is down
     * at the end of the simulation, alternate ways to destroy
```

```
         * the failure thread are needed.
         */
        begin_atomic();
        thisServer->server_state &= 0;
        thisServer->server_state |= SERVER_NEVER_STARTED;
        end_atomic();

        len =1;
        send_ipi(proc_failure_thread, PRIORITY, DESTROY_FAILURE_THREAD_IPI,len,0);
}

void    measureServerBottleneckThread()
{
        perServerDS     *thisServer;
        int             index, no_requests;

        begin_atomic();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
        index = thisServer->s_index;
        end_atomic();

        while( CURR_TIME < TOTAL_SIM_RUN_TIME_CYCLES)
        {
                begin_atomic();
                no_requests = thisServer->server_request_queue->no_of_requests;
                end_atomic();

                CYCLE_COUNTING_OFF;
                AVG_METRIC(globalBottleneckMetric[index].avg_metric, no_requests);
                STDDEV_METRIC(globalBottleneckMetric[index].stddev_metric,
globalBottleneckMetric[index].avg_metric,no_requests);
                CYCLE_COUNTING_ON;

                BUSY_WAIT(CHECK_SERVER_BOTTLENECK_CYCLES);
        }
}


/************************************************************************
  FUNCTION: processRequestAtServer
  PURPOSE : This function processes a request sent to the server
            with the REQUEST_FOR_SERVER_IPI. These are requests
            sent during the normal operation of the system ( and
            not during recovery). It handles requests from its
            clients and from other servers.

            This function takes care of its atomic regions, so the
            calling function should be in a non atomic region when
            this function is called.
  INPUTS  : reqNode *request: the request pulled out of the
                              server_request_queueu. It is deleted
                              by the calling function.
  OUTPUTS : none
************************************************************************/
void    processRequestAtServer( reqNode *request)
{
        int                             page_no,access_mode,dup_req_no;
        int                             tid_of_origin, proc_origin;
        int                             uniq_request_no;
        perServerDS                     *thisServer;
        auxServer                       *serverAuxTable;
        Word                            SEND_ARRAY[20];
        int                             clock_site;
        unsigned int                    clock_mode;
        int                             len, i, size_msg;
        int                             array_index;

        #ifdef MULTI_SERVER
        int                             key_at_from_server, from_server;
        int                             this_server_index;
        #endif

        ASSERT_NOT_ATOMIC();

        begin_atomic();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
```

```
        serverAuxTable = thisServer->server_ptable;
        end_atomic();

        for(i=0; i< 20; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        /* Not in atomic region here.*/
        switch ( request->msg_code)
        {
                case S_FROM_USER_REQUEST
                {       /* request->array:
                         * 0. page_no........the page required
                         * 1. access_mode....READ/WRITE
                         * 2. duplicate request number
                         * 3. tid of user
                         * 4. processor of user
                         * 5. unique requests sent so far
                         */
                        page_no = request->array[0];
                        access_mode = request->array[1];
                        dup_req_no = request->array[2];
                        tid_of_origin = request->array[3];
                        proc_origin = request->array[4];
                        uniq_request_no = request->array[5];

                        /* Update statistics at this server site.
                         * This is a control message during normal operation.
                         */
                        CYCLE_COUNTING_OFF;
                        begin_atomic();
                                globalServerStats[thisServer-
>s_index].normal.control_msg_count++;
                                size_msg = 8 * sizeof(Word);
                                globalServerStats[thisServer-
>s_index].normal.control_msg_bytes += size_msg;
                        end_atomic();

                        /* Update statistics for the time interval into which
                         * this message falls.
                         */
                        array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

                        if(MIG_INTERVAL)
                        {
                        assert(thisServer->s_index >=0 &&
                                thisServer->s_index < NO_OF_SERVERS);
                        assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);

                        globalIntervalData[thisServer-
>s_index].in_cluster_control_byte[array_index] += size_msg;
                        globalIntervalData[thisServer-
>s_index].in_cluster_control_msg[array_index] += 1;
                        }

                        CYCLE_COUNTING_ON;

                        begin_atomic();
                        /* First check if this is a duplicate request
                         * that can be ignored.
                         */
                        if ( ELIMINATE_DUPLICATE_REQUESTS == 1 )
                        {
                                if( thisServer->lastRequestByClient[proc_origin]
                                        > uniq_request_no)
                                {
                                        if (dup_req_no > 1)
                                                return;
                                }
                                if (thisServer->lastRequestByClient[proc_origin]
                                        < (uniq_request_no -1) )
                                {
                                        /* Don't know if this current request
                                         * will be serviced successfully. So,
                                         * set the latest request serviced to
                                         * 1 less than the current request no.
                                         */
```

```
                                        thisServer->lastRequestByClient[proc_origin] =
uniq_request_no -1;
                                }
                        }

                        /* If the page has SERVICE_LATER_IGNORE,
                         * ignore the request
                         * Can check with == directly instead of & as
                         * not all values are possible.
                         */
                        if ( serverAuxTable[page_no].service  == SERVICE_LATER_IGNORE)
                        {
                                /* Ignore the request.*/
                                end_atomic();
                                return;
                        }
                        if ( serverAuxTable[page_no].service == SERVICE_LATER_REPLY)
                        {
                                end_atomic();
                                /* s1 has loaned this page to another server or the clock
                                 * site of this page is migrating. It tells its
                                 * client to resend this request
                                 * s1 sends a REQUEST_FOR_CLIENT_IPI to c1 (D2)
                                 * 0. REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG
                                 * 1. no args, 3
                                 * 2. page no
                                 * 3. tid of user
                                 * 4. unique request number
                                 */
                                SEND_ARRAY[0] = REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG;
                                SEND_ARRAY[1] = 3;
                                SEND_ARRAY[2] = page_no;
                                SEND_ARRAY[3] = tid_of_origin;
                                SEND_ARRAY[4] = uniq_request_no;
                                /* MESSAGE_LENGTH*/
                                len = 5 * sizeof(Word);

        send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,5,SEND_ARRAY);
                                assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);
                                return;

                        }
                        /* If the request is for a page which is out
                         * of the server's cluster, call a method for
                         * it.
                         */
                        if( (page_no < thisServer->lowest_page) ||
                            ( page_no > thisServer->highest_page))
                        {
                                /* Make sure it is a valid page no.*/
                                end_atomic();

                                #ifndef MULTI_SERVER
                                        fatal(" ERROR: request for out of cluster request in
single server environment\n");
                                        return;
                                #else
                                        requestForOutClusterPageAtServer(request);
                                        ASSERT_NOT_ATOMIC();
                                        return;
                                #endif
                        }
                        /* If the server is not servicing duplicate
                         * requests, do not service any request with
                         * duplicate request value > 1.
                         */
                        if( thisServer->dupReq ^ SERVE_DUPLICATE)
                        {
                        /* Do not service duplicate requests. Do not
                         * even fwd them to the clock site.
                         */
                                if( dup_req_no > 1)
                                {
                                        end_atomic();
                                        return;
```

```
			}
		}
		end_atomic();

		/* The request cannot be ignored now! */
		begin_atomic();
		clock_site = serverAuxTable[page_no].clock;
		clock_mode = serverAuxTable[page_no].clock_mode;
		end_atomic();

		/* Not in atomic region here. */
		switch(access_mode)
		{
		case	READ: /* Read request.*/
			/* There are 2 cases:
			 * 1.  Clock is reader, read request;
			 *      - forward the request to the
			 *        clock site, no changes
			 *        are made to the server's data
			 *        structures.
			 * 2.  Clock is writer, read request.
			 *      - changes are made to the
			 *        server's data structures.
			 */
			switch(clock_mode)
			{
			case	READ:	/* Clock is reader.*/
			case	READ_CLOCK:
			/* Clock is reader, read request. */
			/* Forward the request to the clock site.
			 * Indicate that the user does not need
			 * to send an ack to the server
			 * Arguments:
			 * 0. request_code :
			 *		C_FWD_REQUEST
			 * 1. no_args
			 * 2. ack_code : DO_NOT_SEND_ACK_MSG
			 * 3. page_no
			 * 4. access_mode
			 * 5. tid_of_origin
			 * 6. processor of origin
			 * 7. uniq_request_no
			 * The clock will know its a reader and
			 * will do the processing
			 * The server makes no changes to its
			 * data structures, and is not waiting
			 * for any acknowledgement.
			 */
				SEND_ARRAY[0] = C_FWD_REQUEST;
				SEND_ARRAY[1] = 6;
				SEND_ARRAY[2] = DO_NOT_SEND_ACK_MSG;
				SEND_ARRAY[3] = page_no;
				SEND_ARRAY[4] = access_mode;
				SEND_ARRAY[5] = tid_of_origin;
				SEND_ARRAY[6] = proc_origin;
				SEND_ARRAY[7] = uniq_request_no;

				/* MESSAGE_LENGTH*/
				len = 8 * sizeof(Word);

send_ipiV(clock_site,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,8, SEND_ARRAY);
				assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

				/* The request is no longer
				 * needed. Memory for it will
				 * be freed in serverThread
				 * on return
				 */
				break; /* Clock as reader.*/
			case	WRITE: /* Clock is writer.*/
			case	WRITE_CLOCK:
			/* Clock is writer, read request. */
			/* Server's data structures need to be
			 * changed. Call a function that
			 * will do the processing.
			 */
```

```
                                        readRequestClockWriterServer(request);
                                        ASSERT_NOT_ATOMIC();
                                        break:/* Clock as writer.*/
                                default:
                                        fatal("ERROR:  Invalid clock_mode %d for page %d at
server P%d\n",clock_mode,page_no,CURR_PROCESSOR);
                                        break;
                                } /* End of switch, clock_mode.*/
                                break; /* End of read request.*/
                        case    WRITE: /* Write request. */
                                /* Not in atomic region here.*/
                                /* There are 2 cases:
                                 * 1.  Clock is reader, write request:
                                 *     - forward the request to the
                                 *       clock site, no changes
                                 *       are made to the server's data
                                 *       structures.
                                 * 2.  Clock is writer, write request:
                                 *     - changes are made to the
                                 *       server's data structures
                                 *       Create a thread.
                                 */
                                switch(clock_mode)
                                {
                                case    READ:   /* Clock is reader.*/
                                case    READ_CLOCK:
                                /* Clock is reader, write request. */
                                /* Server's data structures need to be
                                 * changed. Call a function.
                                 */
                                /* Not in atomic region here.*/
                                writeRequestClockReaderOrWriterServer(request);
                                ASSERT_NOT_ATOMIC();

                                break; /* Clock as reader.*/

                                case    WRITE:  /* Clock is writer.*/
                                case    WRITE_CLOCK:
                                /* Clock is writer, write request. */
                                /* Server's data structures need to be
                                 * changed. Call a function.
                                 */
                                writeRequestClockReaderOrWriterServer(request);
                                ASSERT_NOT_ATOMIC();
                                break:/* Clock as writer.*/
                                default:
                                fatal(" Invalid clock_mode %d for page %d at server
P%d\n",clock_mode,page_no,CURR_PROCESSOR);
                                break;
                                }/* End of switch clock_mode.*/
                                break; /* End of write request.*/
                        default:
                                fatal(" Invalid access mode %d sent in
REQUEST_FROM_CLIENT_IPI\n", access_mode);
                                break;

                        }/* Switch access_mode.*/

                        #ifdef MULTI_SERVER
                        /* If a multi-server environment, the in cluster requests
                         * made by a client are also considered to decide if it
                         * should migrate to another cluster.
                         * Since a client can send multiple duplicate requests,
                         * increment migData only for the first request
                         */
                        begin_atomic();
                        this_server_index = -1;
                        for ( i=0; i < NO_OF_SERVERS; i++)
                        {
                                if( listOfServers[i] == CURR_PROCESSOR)
                                {
                                        this_server_index = i;
                                        break;
                                }
                        }
                        if ( dup_req_no == 1)
```

```
                        {
                                (thisServer->migData)[proc_origin][this_server_index] =
(thisServer->migData)[proc_origin][this_server_index] + 1;

                        }
                        end_atomic();
                        #endif /* MULTI_SERVER*/

                        break; /* End of case S_FROM_USER_REQUEST*/
                }
                case S_ACK_CLOCK_CHANGE:
                {
                        ackOfClockChangeServer(request);
                        ASSERT_NOT_ATOMIC();
                        break; /* End of case S_ACK_CLOCK_CHANGE*/
                }
                case S_INFORM_CLOCK_MODE_CHANGE:
                {
                        informClockModeChangeServer(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                } /* End of case S_INFORM_CLOCK_MODE_CHANGE*/
                case S_MSG_NOT_DELIVERED:
                {
                        informMsgNotDeliveredServer(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                } /* End of case S_MSG_NOT_DELIVERED*/

/*!!!!!!!!!!!!!!!!!!!!!!!!! BEGIN MULTI_SERVER!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
                #ifdef MULTI_SERVER

                case S_INFORM_PAGE_ACCESS:
                {
                        /* When a client access a shared page locally, it
                         * sends this message to its server. The server can
                         * maintain the history of accesses of every client
                         * to determine when a client should migrate, and to
                         * what cluster. This message is sent only in a
                         * multi-server environment.
                         */
                         serverUpdatingClientDataAccessPattern(request);
                         ASSERT_NOT_ATOMIC();
                         break;
                } /* End of case S_INFORM_PAGE_ACCESS*/
                case SS_OUT_CLUSTER_REQUEST:
                { /* Another server is sending this server a request
                   * for a page belonging to this server.
                   * s1 sends s2 (this server) a request.
                   * request->request_code = SS_OUT_CLUSTER_REQUEST
                   * request->no_values = 5
                   * request->array:
                   * 0. from_server
                   * 1. key_at_from_server
                   * 2. page_no
                   * 3. reqd_access .. R/W
                   * 4. dup_req_no
                   */

                        from_server = request->array[0]; /* s1's processor*/
                        key_at_from_server= request->array[1];
                        page_no = request->array[2];
                        access_mode = request->array[3];
                        dup_req_no = request->array[4];

                        /* Update statistics at this server site.
                         * This is a control message during normal operation.
                         */
                        CYCLE_COUNTING_OFF;
                        begin_atomic();
                                globalServerStats[thisServer-
>s_index].normal.control_msg_count++;
                                size_msg = 7 * sizeof(Word);
                                globalServerStats[thisServer-
>s_index].normal.control_msg_bytes += size_msg;
                        end_atomic();
```

```
                    /* Update statistics for the time interval into which
                     * this message falls.
                     */
                    array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

                    if(MIG_INTERVAL)
                    {
                    assert(thisServer->s_index >=0 &&
                            thisServer->s_index < NO_OF_SERVERS);
                    assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);

                    globalIntervalData[thisServer-
>s_index].out_cluster_control_byte[array_index] += size_msg;
                    globalIntervalData[thisServer-
>s_index].out_cluster_control_msg[array_index] += 1;
                    }

                    CYCLE_COUNTING_ON;


                    begin_atomic();
                    if( (page_no < thisServer->lowest_page) ||
                         ( page_no > thisServer->highest_page))
                    {
                            fatal(" SS_OUT_CLUSTER_REQUEST came to server P%d for page
%d, this page does not belong to this server\n",CURR_PROCESSOR,page_no);
                            end_atomic();
                            return;
                    }
                    /* If the page has SERVICE_LATER_IGNORE,
                     * ignore the request.
                     * Can check with == directly instead of & as
                     * not all values are possible.
                     */
                    if ( serverAuxTable[page_no].service  == SERVICE_LATER_IGNORE)
                    {
                            /* Ignore the request.*/
                            end_atomic();
                            return;
                    }
                    if ( serverAuxTable[page_no].service == SERVICE_LATER_REPLY)
                    {
                            end_atomic();
                            /* Send a later reply to the server s1,
                             * and return.
                             */
                            /* s2 sends a REQUEST_FOR_SERVER_IPI to s1 (D1)
                             * 0. request_code: SS_RESEND_REQUEST_FOR_PAGE_LATER
                             * 1. no args =2
                             * 2. page no
                             * 3. key at from server, key1
                             */
                            SEND_ARRAY[0] = SS_RESEND_REQUEST_FOR_PAGE_LATER;
                            SEND_ARRAY[1] = 2;
                            SEND_ARRAY[2] = page_no;
                            SEND_ARRAY[3] = key_at_from_server;
                            /* MESSAGE_LENGTH*/
                            len = 4 * sizeof(Word);

        send_ipiV(from_server,PRIORITY,REQUEST_FOR_SERVER_IPI,len,4,SEND_ARRAY);
                            assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

                            return;

                    }
                    /* If the server is not servicing duplicate
                     * requests, do not service any request with
                     * duplicate request value > 1.
                     */
                    if( thisServer->dupReq ^ SERVE_DUPLICATE)
                    {
                    /* Do not service duplicate requests. Do not
                     * even fwd them to the clock site.
                     */
                           if( dup_req_no > 1)
```

```
                {
                        end_atomic();
                        return;
                }
        }
        end_atomic();

        /* The request cannot be ignored now! */
        requestFromOutOfClusterPageAtServer(request);
        ASSERT_NOT_ATOMIC();
        break;
} /*end of case SS_OUT_CLUSTER_REQUEST*/
case  S_READ_REPLY_OUT_CLUSTER_FROM_MY_CLIENT:
{
        /* An out of cluster read request from s1 was
         * fwd'd by s2 to its clock c2. c2 is sending
         * a reply to this read request.*/
        serverHandlingReadReplyOutClusterFromItsClient(request);
        ASSERT_NOT_ATOMIC();
        break;
}/* End of case S_READ_REPLY_OUT_CLUSTER_FROM_MY_CLIENT*/
case SS_READ_REPLY_OUT_CLUSTER_REQUEST:
{
        /* s1 gets a reply from s2 to a read request it made
         * earlier.
         */
        serverHandlingReadReplyToOutClusterRequest(request);
        ASSERT_NOT_ATOMIC();
        break;
} /* End of case SS_READ_REPLY_OUT_CLUSTER_REQUEST*/
case S_INVALIDATE_ALL_READERS_AT_OTHER_SERVERS:
{
        /* This server gets a message from its clock of
         * page_no to invalidate all readers outside
         * this cluster. (This clock sends this msg when
         * it is invalidating its readers and sees that
         * a server is a reader).
         */
        serverInvalidatingOtherServerReaders(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case S_INVALIDATE_ALL_BUT_ONE_READERS_AT_OTHER_SERVERS:
{
        /* This server gets a message from its clock of
         * page_no to invalidate all readers outside this
         * cluster. (This clock sends this msg when it is
         * invalidating its readers and sees that a server
         * is a reader). This message is sent when a server
         * that is already a reader, requests a write. The
         * server sends invalidation messages to all servers
         * that are readers, except the server that made the
         * write request.
         */
        serverInvalidatingAllButOneOtherServerReaders(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case SS_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE:
{
        /*s1 has a read copy of page belonging to s2.
         *s2 sends s1 this message telling it to invalidate
         *all its readers of specified page.
         */
        serverInvalidatingItsOutClusterReadPage(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case S_WRITE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT:
{
        /* An out of cluster write request from s1 was
         * fwd'd by s2 to its clock c2. c2 is sending
         * a reply to this write request*/
        serverHandlingWriteReplyOutClusterFromItsClient(request);
        ASSERT_NOT_ATOMIC();
        break;
```

```
}
case S_WRITE_UPGRADE_REPLY_OUT_CLUSTER_FROM_MY_CLIENT:
{
        /* An out of cluster write request from s1 was
         * fwd'd by s2 to its clock c2. c2 sees that s1
         * is already a reader of this page is sending
         * a reply to this write request B4-1.
         */
        serverHandlingWriteUpgradeReplyOutClusterFromItsClient(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case SS_WRITE_REPLY_OUT_CLUSTER_REQUEST:
{
        /* s1 gets a reply from s2 to a write request it made
         * earlier.
         */
        serverHandlingWriteReplyToOutClusterRequest(request);
        ASSERT_NOT_ATOMIC();
        break;
} /* End of case SS_WRITE_REPLY_OUT_CLUSTER_REQUEST*/
case SS_WRITE_UPGRADE_REPLY_OUT_CLUSTER_REQUEST:
{
        /* s1 is a reader for an out of cluster page. It made a
         * write request for that page. It gets this reply from
         * s2.
         */
         serverHandlingWriteUpgradeReplyToOutClusterRequest(request);
         ASSERT_NOT_ATOMIC();
         break;
}
case S_RETURNING_OUT_CLUSTER_UPDATE_PAGE:
{
        /* c1 returns an update write (out of cluster) page to s1 after
         * writing to it. s1 will return it to s2.
         */
        serverReturningWriteUpdatePageToOwnerServer(request);
        ASSERT_NOT_ATOMIC();
        break;
} /* End of case S_RETURNING_OUT_CLUSTER_PAGE */
case    SS_RETURNING_OUT_CLUSTER_UPDATE_PAGE:
{
        /* s1 returns an update write page to s2 (the owner) after
         * writing to it. */
        serverGettingItsWriteUpdatedPage(request);
        ASSERT_NOT_ATOMIC();
        break;
}/* Eend of case SS_RETURNING_OUT_CLUSTER_PAGE*/
case S_INVALIDATE_WRITE_UPDATE_PAGE:
{
        /* c2 has given a write update page to s2 to be given to another
         * server. c2's watchdog wakes up and it finds that it hasnt
         * yet got the update page. c2 sends this msg to s2 telling it
         * to invalidate the write update page.
         */
        serverInvalidatingWriteUpdatePageItGaveToServer(request):
        ASSERT_NOT_ATOMIC();
        break;
}
case SS_INVALIDATE_WRITE_UPDATE_PAGE:
{
        /* s2 has given s1 a write update page. This message is
         * sent by s2 to s1 telling it to invalidate that write
         * update page.
         */
        serverInvalidatingWriteUpdateOutClusterPageTimeout(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case SS_INVALIDATE_WRITE_UPDATE_PAGE_WAITING:
{
        /* s2 gave a write update page to s1. s2 sends
         * SS_INVALIDATE_WRITE_UPDATE_PAGE to s1. Before
         * the write update page reaches the client(c1) in s1
         * which requests it, s1 got the invalidation message.
         * Instead of sending it to c1, s1 re-inserted it
```

```
                 * as this message. B13
                 */
serverInvalidatingWriteUpdateOutClusterPageTimeoutWaiting(request);
                 ASSERT_NOT_ATOMIC();
                 break;
        }
        case SS_RESEND_REQUEST_FOR_PAGE_LATER:
        {
                /* s1 requested a page from s2. s2 sends a reply asking
                 * s1 to resend the request later. (s2 has loaned that
                 * page to another server).
                 */
                serverAskedToResendRequestOutClusterPageLater( request);
                ASSERT_NOT_ATOMIC();
                break;
        }
        case S_ACK_RECEIVED_WRITE_UPDATE_PAGE:
        {
                /* s1 is a reader of page belonging to s2. A client in
                 * s1,c1-1, requested a write. s2 sent a reply.. s1 changed
                 * extra to NOT_REACHED, and forwarded the message to
                 * the temp read clock. The temp read clock forwarded the
                 * reply to c1-1.. on getting the reply c1-1 sends this
                 * message to s1. B8-1
                 */
                 ackWriteUpdateReachedServer(request);
                 ASSERT_NOT_ATOMIC();
                 break;
        }
        case S_ACK_NOT_SENDING_WRITE_UPDATE_PAGE:
        {
                /* s1 is a reader of page belonging to s2. A client in
                 * s2, c1-1, requested a write. s2 sent a reply.. s1
                 * changed extra to NOT_REACHED, and forwarded the message
                 * 'C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE'
                 * with ack code 'SEND_ACK_WRITE_UPDATE_REACHED_MSG' to
                 * the temp read clock, c1. But c1 was migrating. And
                 * since the clientThread and clientMigratingThread at c1
                 * are running in parallel, c1 is no longer the reader
                 * when it gets C_INVALIDATE_ALL... So c1 sends this
                 * message back to s1. (B8-2)
                 * s1 will remove c1-1 from memory when it gets this.
                 */
                 ackWriteUpdateNotSentServer(request);
                 ASSERT_NOT_ATOMIC();
                 break;
        }
        case SS_CLIENT_TO_MIGRATE_IN:
        {
                /* c1 from s1 will be migrating to s2. s1 informs s2 about
                 * this. This msg is recd by s2. (G2)
                 */
                ssClientToMigrateIn(request);
                ASSERT_NOT_ATOMIC();
                break;
        }
        case SS_REMOVE_MY_SERVER_AS_READER_FOR_PAGES:
        {
                /* The server which sends this message is invalidating some
                 * read pages it borrowed (because its temp read clock was
                 * migrating). This server removes those read pages from
                 * list of read pages given to that server.
                 */
                ssRemoveServerAsReaderForPages(request);
                ASSERT_NOT_ATOMIC();
                break;
        }
        case S_MIG_HANDOVER_READ_CLOCK:
        {
                /* C1 is a client that is migrating. It sends this
                 * message for every in-cluster page for which it is
                 * a read clock. H1
                 */
                 migrateHandoverReadClockAtServer(request);
                 ASSERT_NOT_ATOMIC();
```

```
        break;
}
case S_ACK_MIG_READ_CLOCK_INSTALLED:
{
        /* c1 was a read clock. It is migrating to s2. A new
         * read clock is installed at c2. This message sends
         * an ack to s1 telling it that the new read clock is
         * installed successfully. (H3)
         */
        ackOfMigrateReadClockInstalled(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case  S_MIG_HANDOVER_WRITE_CLOCK:
{
        /* Client c1 is migrating. It sends this
         * message for every in-cluster page for which it is
         * a write clock.(K1)
         */
         migrateHandoverWriteClockAtServer(request);
         ASSERT_NOT_ATOMIC():
         break;
}
case S_ACK_MIG_WRITE_CLOCK_INSTALLED:
{
        /* c1 was a write clock. It is migrating to s2. A new
         * write clock is installed at c2. This message sends
         * an ack to s1 telling it that the new write clock is
         * installed successfully. K3
         */
        ackOfMigrateWriteClockInstalled(request);
        ASSERT_NOT_ATOMIC();
        break;
}
case S_MIG_HANDOVER_INVALIDATE_READ_PAGES:
{
        /* c1 is a client which is migrating to a different
         * cluster. It sends this message to s1 giving it
         * a list of pages for which it is a reader (not read
         * clock). This list includes both in and out cluster
         * pages.It sends this message to s1 even if it
         * doesnt have any 'just read' pages.  (J1)
         */
        migrateHandoverAllInvalidateReadPagesAtServer(request);
        ASSERT_NOT_ATOMIC();
        break:
}
case S_CLIENT_READY_TO_MIGRATE:
{
        /* A client c1 has sent all its handover messages to
         * s1 during migration. It's user and client are
         * waiting to migrate. It sends this message to s1
         * so s1 can check if all the handover's have been
         * ack'd, and allow c1 to migrate. (L1)
         */
        clientReadyToMigrate(request):
        ASSERT_NOT_ATOMIC():
        break;
}
case SS_HANDING_OVER_NEW_CLIENT:
{
        /* Client c1 is migrating to s2 from s1. This is the final
         * message s1 sends to s2 telling it that c1 will migrate
         * to s2.  (L3)
         */
         ssHandingOverNewClient(request);
         ASSERT_NOT_ATOMIC();
         break;
}
case S_NEW_CLIENT_ASKING_FOR_ACCEPTANCE:
{
        /* A client c1 has migrated from s1 to s2 (It
         * got permission from s1). s2 could have
         * crashed without getting the handover message
         * from s1. So, this client acts intelligently
         * and keeps polling s2 asking for an ack (N1)
```

```
                        */
                        newClientAskingForAcceptance(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case S_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK:
                {
                        /* A client c1 was migrating from cluster 1 to cluster 2.
                         * When in cluster 1 it requested a write page in cluster 1.
                         * It got a reply 'after' it migrated to cluster 2. Now
                         * that page is an out of cluster page. So, c1 cannot
                         * install the page as a write clock in replyMakeWriteClock.
                         * It sends the page back to its new server, the server in
                         * cluster 2. This is the server in cluster 2.
                         */
serverHandlingMigNotInstallingOldClusterWriteClockFromItsClient(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case SS_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK:
                {
                        /* See comment for S_MIG_NOT_INSTALLING_OLD_CLUSTER_WRITE_CLOCK
                         * This is server 1 getting back its write clock from server 2
                         */
serverHandlingMigNotInstallingOldClusterWriteClockFromAnotherServer(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case SS_SET_SERVER_STATE:
                {
                        /* When a server goes down or starts the recovery process,
                         * it informs all other servers about its state.
                         * A server that gets this message has to make sure
                         * its cluster is consistent.
                         * This message does not use the SERVER_TO_SERVER_IPI, as
                         * a lot of processing to make data consistent has to be
                         * done. It was considered more reasonalbe to do this
                         * via the request queue, than give it a special priority
                         * via the SERVER_TO_SERVER_IPI (which does not go
                         * through queue)
                         * This function is in serverrec.ca
                         */
                        changeInAnotherCluster(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }

                #endif /*of MULTI_SERVER*/
/*!!!!!!!!!!!!!!!! END of MULTI_SERVER !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

                default:
                {
                        fatal( " ERROR: Invalid msg code %d in processRequestAtServer\n",
request->msg_code);
                        break;
                }

        }

}


/*****************************************************************************
  FUNCTION: readRequestClockWriterServer
  PURPOSE : This function is called by processRequestAtServer when
            the clock is a writer and there is a read request.
            When the clock is a writer, it has the latest copy.
            It downgrades itself to a reader, and gives a copy
            to the user requesting a read. Now, since a writer
            clock has presumably just written to that page, the
            latest trailer of this page in the system does not
            have this change. So, this clock, when it becomes
            a reader, will give the user a read copy and also
            make it a trailer. The clock will inform the server
```

```
          of a clock mode change. The server makes no change
          to this page's DS now.

          The server sends a DO_NOT_ACK_CLOCK_CHANGE_MSG
          to the clock which will be forwarded to the user.

          This function manages its atomic regions. Should be called
          in non atomic region. Should return in non atomic region.
  INPUTS  : reqNode *request
          request codes used:
          request->msg_code                          S_FROM_USER_REQUEST,
                                                     request sent by a user.
          request->no_values                         6
          request->array:
          0. page_no                                 the page required
          1. access_mode                             READ/WRITE
          2. duplicate request number
          3. tid of user
          4. processor of user
          5. unique request no
  OUTPUTS : none
***********************************************************************/
void    readRequestClockWriterServer(reqNode *request)
{
        perServerDS     *thisServer;
        auxServer       *serverAuxTable;
        int             page_no, access_mode,tid_of_origin;
        int             proc_origin, uniq_request_no;
        int             len,i;
        Word            SEND_ARRAY[8];
        int             clock_site;

        ASSERT_NOT_ATOMIC();

        for(i=0; i< 8; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        page_no = request->array[0];
        /* The access_mode is the access reqd by the user. It
         * is read, since this is a readRequest function.
         */
        access_mode = request->array[1];

        /* dup_req_no is not needed here. Was needed only by the
         * serverThread when it checked if it was to service
         * duplicate requests.
         * dup_req_no = request->array[2];
         */
        tid_of_origin = request->array[3];
        proc_origin = request->array[4];
        uniq_request_no = request->array[5];

        begin_atomic();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
        serverAuxTable = thisServer->server_ptable;

        clock_site = serverAuxTable[page_no].clock;
        /* The server knows the clock will downgrade
         * from writer to reader, but it will not change
         * the clock mode now. This is because by the time
         * this request reaches the clock, the clock could
         * have changed (unlikely cos a clock change is
         * first known by the server before the clock knows it),
         * or the clock state could have been changed by
         * a user at that site. If a user changes the clock mode,
         * it will inform the server. By the time this request
         * reaches the clock, the clock mode could have changed
         * from what the server thinks it is at this point.
         * It is better for the clock to inform the server
         * of a clock mode change when it makes the change, rather
         * than the server assuming it will change and make
         * that change now.
         * Therefore, the server does not change the mode of the
         * clock. The clock will inform the server when it changes
         * from a READ clock to a WRITE clock.
         */
```

```
        end_atomic();

        /* Forward the request to the clock site.
         * REQUEST_FOR_CLIENT_IPI
         * Indicate that the user does not need
         * to send an ack to the server
         * Arguments:
         * 0. request_code :
         *           C_FWD_REQUEST
         * 1. no_args
         * 2. ack_code : DO_NOT_SEND_ACK_MSG
         * 3. page_no
         * 4. access_mode
         * 5. tid_of_origin
         * 6. processor of origin
         * 7. uniq_request_no
         * The clock will make itself a reader and
         * will do the processing
         * The server is not waiting for an acknowledgement from
         * the user
         */
        SEND_ARRAY[0] = C_FWD_REQUEST;
        SEND_ARRAY[1] = 6;
        SEND_ARRAY[2] = DO_NOT_SEND_ACK_MSG;
        SEND_ARRAY[3] = page_no;
        SEND_ARRAY[4] = access_mode;
        SEND_ARRAY[5] = tid_of_origin;
        SEND_ARRAY[6] = proc_origin;
        SEND_ARRAY[7] = uniq_request_no;

        /* MESSAGE_LENGTH*/
        len = 8 * sizeof(Word);
        send_ipiV(clock_site,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,8, SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

        /* The request is no longer needed. Memory for it will
         * be freed in serverThread on return.
         */
}

/**********************************************************************
  FUNCTION: writeRequestClockReaderOrWriterServer
  PURPOSE : This function is called by processRequestAtServer when
            the clock is a reader and there is a write request.
            Since a write request has to be sent to the user, and
            this is a write invalidate protocol, the reader clock
            will invalidate itself, and give a write page to the
            user. The user becomes the new clock.

            Since there is going to be a change in clock site, the
            server will send a  SEND_ACK_CLOCK_CHANGE_MSG to the
            clock. Meanwhile, the server marks this page as
            SERVICE_LATER_IGNORE. Though the server, at this point,
            knows what the new clock site will be(the user site making
            the write request), it will not forward subsequent requests
            for this page to the new clock site until the new clock site
            actually becomes a clock site for this page.
            Any requests that arrive at the server when the SERVICE_
             LATER_IGNORE flag is set will timeout and resend.
            This function manages its atomic regions. Should be called
            in non atomic region. Should return in non atomic region.
  INPUTS  : reqNode *request
            request codes used
            request->msg_code                S_FROM_USER_REQUEST,
                                             request sent by a user.
            request->no_values               6
            request->array:
            0. page_no                       the page required
            1. access_mode                   READ/WRITE
            2. duplicate request number
            3. tid of user
            4. processor of user
            5. unique request no
  OUTPUTS : none
**********************************************************************/
void    writeRequestClockReaderOrWriterServer(reqNode *request)
```

```
{
        perServerDS     *thisServer;
        auxServer       *serverAuxTable;
        int             page_no,tid_of_origin,proc_origin;
        int             access_mode,uniq_request_no;
        Word            SEND_ARRAY[8];
        int             len,i;
        int             clock_site;

        ASSERT_NOT_ATOMIC();
        thisServer = (perServerDS *) (*(ptrToGlobalServerDS));
        serverAuxTable = thisServer->server_ptable;

        for(i=0 ; i<8; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        page_no = request->array[0];
        /* The access_mode is the access reqd by the user. It
         * is write, since this is a writeRequest function.
         * (whether the clock is a reader or writer).
         */
        access_mode = request->array[1];

        /* dup_req_no is not needed here. Was needed only by the
         * serverThread when it checked if it was to service
         * duplicate requests.

         * dup_req_no = request->array[2];
         */
        tid_of_origin = request->array[3];
        proc_origin = request->array[4];
        uniq_request_no = request->array[5];

        /* The server knows the clock will change. It sets
         * the page service to SERVICE_LATER_IGNORE.
         */
        begin_atomic();

        /* Assign new service, do not bitwise or it. */
        serverAuxTable[page_no].service  = SERVICE_LATER_IGNORE;

        /* The mode of the new clock will be sent with the ack. But do
         * not invalidate the clock site or the mode now. This is
         * because, before the server gets an ack for this msg,
         * it can get a clock mode change. The new clock will be sent
         * in the message S_ACK_CLOCK_CHANGE.
         */
        /* Get the clock site to fwd the request to, before
         * invalidating it!*/
        clock_site = serverAuxTable[page_no].clock;
        end_atomic();

        /* Forward the request to the clock site.
         * REQUEST_FOR_CLIENT_IPI
         * Indicate that the user does need
         * to send an ack to the server.
         * Arguments:
         * 0. request_code : C_FWD_REQUEST
         * 1. no_args
         * 2. ack_code : SEND_ACK_CLOCK_CHANGE_MSG
         * 3. page_no
         * 4. access_mode
         * 5. tid_of_origin
         * 6. processor of origin
         * 7. uniq_request_no
         * The clock will know its a reader and
         * will do the processing
         * The server makes the SERVICE_LATER_IGNORE change to
         * that page, and is waiting for an acknowledgement.
         */
        SEND_ARRAY[0] = C_FWD_REQUEST;
        SEND_ARRAY[1] = 6;
        SEND_ARRAY[2] = SEND_ACK_CLOCK_CHANGE_MSG;
        SEND_ARRAY[3] = page_no;
        SEND_ARRAY[4] = access_mode;
        SEND_ARRAY[5] = tid_of_origin;
```

```
        SEND_ARRAY[6] = proc_origin;
        SEND_ARRAY[7] = uniq_request_no;

        /* MESSAGE_LENGTH*/
        len = 8 * sizeof(Word);
        send_ipiV(clock_site,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,8, SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

        /* The request is no longer needed. Memory for it will
         * be freed in serverThread on return.
         */
}

/***********************************************************************
  FUNCTION: ackOfClockChangeServer
  PURPOSE : This function is called by processRequestAtServer when
            the clock has changed and the server has marked that
            page SERVICE_LATER_IGNORE until it gets confirmation
            of the clock change.
            The clock changes in case of :
                a write request (clock could have been R or W)
            This is the response the server gets for its
            SEND_ACK_CLOCK_CHANGE_MSG it sent in a reply to a
            write request.
            This message tells the server the new clock site, trailer
            version for that page. The server makes these changes and
            then changes the service of this page to SERVICE_NOW.

            This function manages its atomic regions. Should be called
            in non atomic region. Should return in non atomic region.
  INPUTS  : reqNode *request

            request codes used:
            request->msg_code                S_ACK_CLOCK_CHANGE, ack of clock
                                             change from user
            request->no_values               4
            request->array:
             0. page_no
            1. clock_site                    the new clock site
            2. clock_mode                    READ/WRITE/READ_CLOCK/WRITE_CLOCK
            3. t_version                     the latest trailer version
                                 Since the clock has changed, it will have a new
                                 trailer version number. This need not be sent
                                 to the clock (is needed only during recovery),
                                 but no harm updating the server now in the same msg
                                 (The site of the trailer is old)
  OUTPUTS : none
***********************************************************************/
void    ackOfClockChangeServer(reqNode *request)
{
        perServerDS     *thisServer;
        auxServer       *serverAuxTable;
        int             page_no,clock_site,clock_mode,t_version;
        int             size_msg,array_index;

        #ifdef MULTI_SERVER
        int             ret;
        Multi4Node      ret_out_mig_node;
        #endif


        ASSERT_NOT_ATOMIC();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
        serverAuxTable = thisServer->server_ptable;

        page_no = request->array[0];
        clock_site = request->array[1];
        clock_mode = request->array[2];
        t_version = request->array[3];

        /* Update statistics at this server site.
         * This is a control message during normal operation.
         */
        CYCLE_COUNTING_OFF;
        begin_atomic();
                globalServerStats[thisServer->s_index].normal.control_msg_count++;
```

```
                size_msg = 6 * sizeof(Word);
                globalServerStats[thisServer->s_index].normal.control_msg_bytes +=
size_msg;
        end_atomic();

        /* Update statistics for the time interval into which
         * this message falls.
         */
        array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

        if (MIG_INTERVAL)
        {
        assert(thisServer->s_index >=0 &&
                thisServer->s_index < NO_OF_SERVERS);
        assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);
        globalIntervalData[thisServer->s_index].in_cluster_control_byte[array_index] +=
size_msg;
        globalIntervalData[thisServer->s_index].in_cluster_control_msg[array_index] += 1.
        }

        CYCLE_COUNTING_ON;

        begin_atomic();

        /* The server changed the service of the page to SERVICE_LATER_IGNORE
         * on a write request in writeRequestClockReaderOrWriterServer.
         * This is the ack of the clock change on a write.
         *
         * All the following are done irrespective of
         * whether the user at the clock site changed the
         * clock mode or not. */
        serverAuxTable[page_no].clock = clock_site;
        /* Assign the new clock mode. do not bitwise or it.*/
        serverAuxTable[page_no].clock_mode = clock_mode;
        serverAuxTable[page_no].t_version = t_version;

        /* See migration notes and the comments below:
         * *6 c1 made a write request to in cluster page. say c2 was
         *    the clock. When s1 forwarded c1's request to c2, it
         *    asked c1 to SEND_ACK_CLOCK_CHANGE_MSG. c1 replies with
         *    a S_ACK_CLOCK_CHANGE and s1 changes the clock to c1.
         *    Suppose c1's user is waiting for that write reply and c1
         *    is to migrate. s1 should not forward any subsequent requests
         *    to c1 (once it becomes the write clock), as c1 is migrating.
         *    So, in S_ACL_CLOCK_CHANGE, if c1 is migrating, s1 changes
         *    the service from SERVICE_LATER_IGNORE to SERVICE_LATER_REPLY
         *    instead of SERVICE_NOW. This change should be made only
         *    for the multi-server case.
         */
        #ifdef MULTI_SERVER
                /* Check if clock site is migrating. If it is, change the
                 * service to SERVICE_LATER_REPLY
                 */
                ret = findInMulti4List(thisServer-
>migratingOutClients,clock_site,&ret_out_mig_node);
                if ( ret == FOUND)
                        serverAuxTable[page_no].service = SERVICE_LATER_REPLY;
                else
                        serverAuxTable[page_no].service = SERVICE_NOW:
        #else
                serverAuxTable[page_no].service = SERVICE_NOW;
        #endif

        end_atomic();

        /* The request is no longer needed. Memory for it will
         * be freed in serverThread on return.
         */

        /* The request is no longer needed. Memory for it will
         * be freed in serverThread on return.
         */

}

/*****************************************************************************
```

```
  FUNCTION: informClockModeChangeServer
  PURPOSE : This function is called by processRequestAtServer when
            the clock at a client site has changed its clock
            mode (changeMyClockMode) and is informing the server
            of this change.
            The clock mode changes in case of :
             -  a user at a clock site, wants to read when the
                       clock mode is write.
             -  a user at a clock site, wants to write when the
                       clock mode is read.
            The server changes only the clock mode. Nothing else.

            This function manages its atomic regions. Should be called
            in non atomic region. Should return in non atomic region.
  INPUTS  : reqNode *request

            request codes used
            request->msg_code                      S_INFORM_CLOCK_MODE_CHANGE
            request->no_values                     3
            request->array:
             0. page_no
             1. new_clock_mode                     READ,WRITE,READ_CLOCK,WRITE_CLOCK
             2. clock_site                         the ocessor of the site sending
                                  this msg. This is needed to check if the clock site
                                  is still the same.
  OUTPUTS : none
*************************************************************************/
void    informClockModeChangeServer(reqNode *request)
{
        perServerDS     *thisServer;
        auxServer       *serverAuxTable;
        int             page_no, new_clock_mode,clock_site,
        int             size_msg, array_index;

        ASSERT_NOT_ATOMIC();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
        serverAuxTable = thisServer->server_ptable;

        page_no = request->array[0];
        new_clock_mode = request->array[1];
        clock_site = request->array[2];

        /* Update statistics at this server site.
         * This is a control message during normal operation.
         */
        CYCLE_COUNTING_OFF;
        begin_atomic();
                globalServerStats[thisServer->s_index].normal.control_msg_count++;
                size_msg = 5 * sizeof(Word);
                globalServerStats[thisServer->s_index].normal.control_msg_bytes +=
size_msg;
        end_atomic();

        /* Update statistics for the time interval into which
         * this message falls.
         */
        array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

        if (MIG_INTERVAL)
        {
        assert(thisServer->s_index >=0 &&
                thisServer->s_index < NO_OF_SERVERS);
        assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);

        globalIntervalData[thisServer->s_index].in_cluster_control_byte[array_index] +=
size_msg;
        globalIntervalData[thisServer->s_index].in_cluster_control_msg[array_index] += 1.
        }

        CYCLE_COUNTING_ON;

        begin_atomic();

        /* Assign the new clock mode. Do not bitwise or it
         * with the previous value.
         */
```

```
        /* Check that there is a clock site. Even if it has
         * a SERVICE_LATER_IGNORE due to a write request
         * forwarded to the clock site, the clock mode can
         * be safely changed. The SERVICE_LATER_IGNORE flag
         * is set for all write requests and does not depend
         * on the clock mode.
         * Make sure there is a clock site.
         */
        if( serverAuxTable[page_no].clock !=  clock_site)
        {
                fatal(" Invalid clock site in informClockModeChangeServer client clock %d,
servers clock %d.. for page %d\n",clock_site, serverAuxTable[page_no].clock, page_no);
                end_atomic();
                return;

        }
        else /* correct clock site, change the mode*/
        { /* This is done even if this page has service_later_ignore
           * or service_later_reply set.
           */
                serverAuxTable[page_no].clock_mode &= 0;
                serverAuxTable[page_no].clock_mode |= new_clock_mode;
        }
        end_atomic();

        /* The request is no longer needed. Memory for it will
         * be freed in serverThread on return.
         */
}

/*****************************************************************
  FUNCTION: informMsgNotDeliveredServer
  PURPOSE : This function is called by processRequestAtServer when the
            server forwarded a request to the clock site and set the
            SERVICE_LATER_IGNORE bit. The server is now waiting for
            an acknowledgement. But the clock could not service the
            request. So the clock sends this msg to the server telling it
            to change its service back to SERVICE_NOW for this page.

            This function manages its atomic regions. Should be called
            in non atomic region. Should return in non atomic region.
  INPUTS  : reqNode *request

            request codes used:
            request->msg_code                            S_MSG_NOT_DELIVERED
            request->no_values                           1
            request->array:
             0. page_no
  OUTPUTS . none
*****************************************************************/
void    informMsgNotDeliveredServer(reqNode *request)
{
        perServerDS     *thisServer;
        auxServer       *serverAuxTable;
        int             page_no;
        int             size_msg, array_index;

        #ifdef MULTI_SERVER
        Multi4Node      ret_node;
        int             ret;
        #endif

        ASSERT_NOT_ATOMIC();
        thisServer = (perServerDS*) (*(ptrToGlobalServerDS));
        serverAuxTable = thisServer->server_ptable;

        page_no = request->array[0];

        /* Update statistics at this server site.
         * This is a control message during normal operation.
         */
        CYCLE_COUNTING_OFF;
        begin_atomic();
                globalServerStats[thisServer->s_index].normal.control_msg_count++;
                size_msg = 3 * sizeof(Word);
```

```
		globalServerStats[thisServer->s_index].normal.control_msg_bytes +=
size_msg;
	end_atomic();

	/* Update statistics for the time interval into which
	 * this message falls.
	 */
	array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

	if (MIG_INTERVAL)
	{
	assert(thisServer->s_index >=0 &&
		thisServer->s_index < NO_OF_SERVERS);
	assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);

	globalIntervalData[thisServer->s_index].in_cluster_control_byte[array_index] +=
size_msg;
	globalIntervalData[thisServer->s_index].in_cluster_control_msg[array_index] += 1;
	}

	CYCLE_COUNTING_ON;

	begin_atomic();

	/* This message has been sent when:
	 * - the clock site loaned a write update page to another
	 *   cluster. By then a write request from a client in
	 *   its cluster was forwarded to the clock.
	 * - An out of cluster write request arrives.
	 *   Server forwards to clock, does not change service.
	 * - An in cluster write request arrives.
	 *   Server forwards to clock, changes service to
	 *   SERVICE_LATER_IGNORE.
	 * - Clock services out of cluster write request. Becomes
	 *   write update, changes service to SERVICE_LATER.
	 * - The write update page is forwarded to another server by
	 *   this cluster's server. Server changes service from
	 *   SERVICE_LATER_IGNORE to SERVICE_LATER_REPLY.
	 * - Clock gets the in-cluster write request. Cannot service
	 *   request because it has WRITE_UPDATE,SERVICE_LATER.
	 * - Clock sends a S_MSG_NOT_DELIVERED to server.
	 * - Server sees  the service is SERVICE_LATER_REPLY. Makes
	 *   no changes and quits from this function.
	 */
	/* It is possible that the clock ignored the request because
	 * it was migrating. So, do not change the service back
	 * to SERVICE_NOW, if the clock is migrating. Currently
	 * it is SERVICE_LATER_IGNORE. Change it to SERVICE_LATER_REPLY.
	 */

	if(serverAuxTable[page_no].service == SERVICE_LATER_IGNORE)
	{
	#ifndef MULTI_SERVER
		/* Change the service back to SERVICE_NOW.do not bitwise or it */
		serverAuxTable[page_no].service  = SERVICE_NOW;
	#else
		ret = findInMulti4List(thisServer-
>migratingOutClients,serverAuxTable[page_no].clock, &ret_node);
		if( ret == FOUND)
			/* Change from SERVICE_LATER_IGNORE to SERVICE_LATER_REPLY*/
			serverAuxTable[page_no].service = SERVICE_LATER_REPLY;
		else
			serverAuxTable[page_no].service = SERVICE_NOW;
	#endif
	}

	end_atomic();

	/* Do not change clock site, clock mode or anything else
	 * Now the server will service further requests for this
	 * page.
	 */

	/* The request is no longer needed. Memory for it will
	 * be freed in serverThread on return.
	 */
```

```
}

/*********************************************************************
  FUNCTION: getServerCodeFromServerProc
  PURPOSE : This function returns an integer which is a code
            for a server making a request. It is not the
            processor number of the server, but a number which
            is different from all the processor numbers in the
            system
            IMP: this function should be coded considering the
            code in getServerIndexFromServerCode
  INPUTS  :
            int server_proc           server processor number
            int *server_code          return value of the server code
  OUTPUTS : none
*********************************************************************/
void    getServerCodeFromServerProc(int server_proc, int *server_code)
{
        int     server_index;  /* The index of this server used in
                                * listOfServers.*/
        int     i;

        for( i=0; i< NO_OF_SERVERS; i++)
        {
                if( listOfServers[i] ==  server_proc)
                {
                        server_index = i;
                        break; /* From for loop.*/
                }
        }
        /* Server index will vary from 0 to NO_OF_SERVERS-1
         * All the processors in the system vary from 0 to
         * NO_OF_CLIENTS.
         * so, server code for:
         * server_index 0 can be NO_OF_CLIENTS
         * server index 1 can be NO_OF_CLIENTS+1
         * server index (NO_OF_SERVERS-1) can be NO_OF_CLINETS +
         *                       NO_OF_SERVERS -1
         */

        *server_code = NO_OF_CLIENTS + server_index;
}

/*********************************************************************
  FUNCTION: getServerIndexFromServerCode
  PURPOSE : This function returns the server index given the
            server code. listOfServers[server index] will be
            the processor number of this server. The server code
            is used by a client which has a server as a reader.

            IMP: this function should be coded considering the
            code in getServerCodeFromServerProc.
  INPUTS  :
            int server_code           server  code
            int *server_index         the index used in listOfServer
                                      which will give that server's processor
  OUTPUTS : none
*********************************************************************/
void    getServerIndexFromServerCode(int server_code, int *server_index)
{
        *server_index = server_code - NO_OF_CLIENTS;

        /* The processor number will be listOfServers[server_index].*/
}

/*********************************************************************
  FUNCTION: getServerIndexFromServerProc
  PURPOSE : This function returns i where listOfServers[i] is
            the input server_proc.
  INPUTS  :
            int server_proc           server processor number
            int *server_index         return value of the server index
  OUTPUTS : none
*********************************************************************/
void    getServerIndexFromServerProc(int server_proc, int *server_index)
{
```

```
        int     i;

        for( i=0; i< NO_OF_SERVERS; i++)
        {
                if( listOfServers[i] ==  server_proc)
                {
                        *server_index = i;
                        return;
                }
        }
}
/***********************************************************************
  FUNCTION: isItServerCode
  PURPOSE : This function returns TRUE if the input value is
            a server code. Server codes range from NO_OF_CLIENTS
            to NO_OF_CLIENTS + NO_OF_SERVERS-1
            IMP: this function should be coded considering the
            code in getServerCodeFromServerProc
  INPUTS  : int server_code
  OUTPUTS : TRUE/FALSE
 ***********************************************************************/
int     isItServerCode(int server_code)
{
        if(server_code >= NO_OF_CLIENTS)
                return TRUE;
        else
                return FALSE;
}
```

```
/*********************************************************************
 * FILE     :   clientfn.ca                                          *
 * CONTENTS:    This file contains all the functions which deal      *
 *              with an in-cluster message at a client site (in both *
 *              single and multi-server systems).                    *
 *********************************************************************/

#include "ft.h"
#include "consistent.h"


/*********************************************************************
  FUNCTION: requestForClientHandler
  PURPOSE : This handler is called when a request is sent to it from
           a server. This handler puts the request into the client
           requests queue. The client thread pulls out requests in a
           FIFO manner.
              REQUEST_FOR_CLIENT_IPI, runs at client.
  INPUTS   :
        GENERAL MESSAGE FORMAT
        ----------------------

        argv[0]                              request_code
        argv[1]                              no of args (from argv[2])
        argv[2]-argv[no of args - 2]         the arguments to be put into
                                             the array in a request.

        REQUEST CODES USED
        ------------------

           To forward a request to a client.
         **0. request_code             C_FWD_REQUEST.
           1. no args                  6.
           2. ack_code                 DO_NOT_SEND_ACK_MSG.
                                       SEND_ACK_CLOCK_CHANGE_MSG.
           3. page_no
           4. access_mode              READ/WRITE.
           5. tid of origin
           6. processor of origin
           7. unique request no

           To tell a client to invalidate a read page.
         **0. request_code             C_INVALIDATE_READER.
           1. no_args                  1
           2. page_no



          This clients server is forwarding a request from out
          of this cluster.
         **0. request_code             C_FWD_OUT_OF_CLUSTER_REQUEST.
           1. no args                  4.
           2. from_server_code         The server which sent the request.
                                       The servers are given a different code
                                       to identify them. It is not the processor
                                       number of the server, because this client
                                       cannot directly send a message to  another
                                       server.
           3. this_server_key          The key which this cluster's server
                                       uses (incoming requests) to identify the
                                       details of the server which sent the request
                                       (key2).
           4. page_no
           5. reqd_access

        This client has a read copy of out of cluster page. Another
        read request for the same page is made by another client in
        this cluster. The server s1 does not fwd the request to the
        server owning the page. Instead, it sends the request to
        this client having a read copy, making it a temporary clock
        of an out of cluster page.
        ** 0. request_code:
              C_FWD_MAKE_URSLF_TEMP_CLOCK_HANDLE_READ_REQUEST
              c1 in s1 has a read copy of page p which belongs
              to s2. now, c1-2 in s1 requests a read copy of page
              p. s1 does not fwd request to s2, s1 sends a msg
```

```
        to c1 telling it to become a temp read clock for p.
   1. no args                6.
2. ack_code                  DO_NOT_SEND_ACK_MSG.
3. page_no
4. access_mode               READ.
5. tid of origin
6. processor of origin
7. unique request no

c1 in s1 is a temp clock of page p belonging to s2.
s1 sends a message to c1 telling it to invalidate all
the readers. c1 does not have to send an ack back to
s1 and s1 does not have to send an ack back to s2.
** 0. request_code           C_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE
   1. no args                1
   2. page_no

A temporary read clock (for out of cluster page) tells its
reader to invalidate its read page. It is different from
C_INVALIDATE_READER. In C_INVALIDATE_READER, the page mode
is changed, but the page stays in memory(it could be  a trailer).
Since this is a multi server case, the page is removed from
memory.
**0. request_code            C_INVALIDATE_OUT_CLUSTER_READER
  1. no_args                 1
  2. page_no

s2 is returning a write update page to c2. c2 uses
update code to coordinate the update with the watchdog
update thread.
** 0. request code           C_RETURNING_UPDATE_PAGE,
   1. no args                3
   2. page no
   3. page value             updated page value
   4. update code

c1 has a write update copy of page. s1 sends it an
invalidate message since c1 has kept it for more than
update time.
** 0. request code           C_INVALIDATE_WRITE_UPDATE_OUT_CLUSTER_PAGE
   1. no args                1
   2. page no

c1 in s1 is a temp read clock of page p belonging to s2.
A site in s1,c1-1, asks for write permission for page p. On
getting a reply from s2, s1 sends a message to c1 telling it to
invalidate all the readers and forward the reply to c1-1. While
forwarding the message to c1, s1 set extra to NOT_REACHED. So,
s1 is waiting for an ack from c1-1 (that it received the page)
(56-1).
** 0. request_code
C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE
   1. no args                7
   2. ack_code               SEND_ACK_WRITE_UPDATE_REACHED_MSG
   3. page no
   4. requesting client      client requesing write
   5. access mode            WRITE_UPDATE
   6. tid of origin
   7. unique request no
   8. update time



REPLIES TO REQUESTS:
--------------------
** 0. msg_code               REPLY_MAKE_URSLF_READER_MSG , telling the
                                       user to make itself a reader of
                                       this page.
   1. no args                6
   2. ack_code               DO_NOT_SEND_ACK_MSG. This is the code the
                                       server sent the clock, since the
                                       server is not waiting for an ack
                                       since the clock is not changing.
   3. page_no                the page no of the requested page
   4. access_mode            READ
   5. page_value             the value at the clock site
```

```
   6. tid_of_origin          needed by user to synchronize the
                                        user/reply/watchdog
   7. unique request no       "

** 0. msg_code               REPLY_MAKE_URSLF_READER_TRAILER_MSG ,
                                        telling the user to make itself a
                                        reader and trailer of this page.
   1. no args                7
   2. ack_code               DO_NOT_SEND_ACK_MSG This is the code the
                                        server sent the clock, since the
                                        server is not waiting for an
                                        acknowledgment.
   3. page_no                the page no of the requested page
   4. access_mode            READ_TRAILER
   5. page_value             the value at the clock site
   6. tid_of_origin          needed by user to synchronize the
                                        user/reply/watchdog
   7. unique request no               "
   8. trailer_version  the latest t version in the system


** 0. msg_code               REPLY_MAKE_URSLF_WRITER_CLOCK_MSG,
                                        telling the user to make itself a
                                        clock and a writer of this page.
   1. no args,7
   2. ack_code               SEND_ACK_CLOCK_CHANGE_MSG
                                        This is the code the server sent the
                                        clock, since the server is waiting
                                        for an ack for completion of clock
                                        change.
   3. page_no                the page no of the requested page.
   4. access_mode            WRITE_CLOCK.
   5. page_value             the value at the clock site.
   6. tid_of_origin          needed by user to synchronize the
                                        user/reply/watchdog.
   7. unique request no               ".
   8. trailer_version  the latest t version in the system.

** 0. msg_code               REPLY_MAKE_URSLF_READER_OUT_CLUSTER_PAGE_MSG .
                                        telling the user to make itself a
                                        reader of this page.
   1. no args                6
   2. ack_code               DO_NOT_SEND_ACK_MSG
                             server is not waiting for an ack.
   3. page_no                the page no of the requested page.
   4. access_mode            this should be READ.
   5. page_value
   6. tid_of_origin          needed by user to synchronize the
                             user/reply/watchdog.
   7. unique request no               ".

s1 has loaned this page to another server or
the clock site of this page is migrating. It tells its
client to resend this request.
s1 sends a REQUEST_FOR_CLIENT_IPI to c1 (D2).
** 0. msg_code               REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG
   1. no args                3
   2. page no
   3. tid of user
   4. unique request number

A write clock gets a read request from the user at its own
site. It downgrades to a reader, adds itself as a reader in
the list of readers and send this message to itself to
wake up its user. It does not send the page with this
message as the page is already in its memory in the
required mode.
** 0. msg_code               REPLY_WAKE_UP_USER_MSG
   1. no args                3
   2. ack_code               DO_NOT_SEND_ACK_MSG
   3. tid of user
   4. unique request number

s1 sends a reply to the user which made the request for out of
cluster write page.
** 0. msg_code               REPLY_MAKE_URSLF_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG
```

```
                                                  telling the user to make itself a
                                                  update writer of this page.
   1. no args                7.
   2. ack_code               DO_NOT_SEND_ACK_MSG
                                                  Server is not waiting for an ack
                                                  or SEND_ACK_WRITE_UPDATE_REACHED_MSG
                                                  IMP: If ack code is
                                                  SEND_ACK_WRITE_UPDATE_REACHED_MSG, it
                                                  means that this write update was
                                                  forwarded to a reader which
                                                  forwarded the reply to this site. The
                                                  server set the value of extra to
                                                  NOT_REACHED. So this client will have
                                                  to send it an ack message
                                                  S_ACK_RECEIVED_WRITE_UPDATE_PAGE
                                                  (B8-1)
   3. page_no                the page no of the requested page
   4. access_mode            this should be WRITE_UPDATE
   5. page_value
   6. tid_of_origin          needed by user to synchronize the
                             user/reply/watchdog
   7. unique request no
   8. update time


c1 is a reader of a page belonging to s2. c1-1 requested write
permission to this page. c1, c1-1 are in s1. s1 got a 'upgrade
yourslf from reader to write update' reply from s2.
s1 forwards the reply to c1, which in turn will forward the
reply to c1-1. This reply if forwarded from c1 to c1-1 when c1
was already a reader and it requested a write.(B7-1)
** 0. msg_code               REPLY_UPGRADE_URSLF_FROM_READER_TO_UPDATE_
                                                  WRITER_OUT_CLUSTER_PAGE_MSG,
                                                  telling the user to make itself a
                                                  update writer of this page,
   1. no args                6
   2. ack_code               SEND_ACK_WRITE_UPDATE_REACHED_MSG
                                                  The server has set the value of extra
                                                  to NOT_REACHED. So. it is waiting for
                                                  the ack msg
                             S_ACK_RECEIVED_WRITE_UPDATE_PAGE
                                                  (B8-1)
   3. page_no                the page no of the requested page
   4. access_mode            this should be WRITE_UPDATE
   5. tid_of_origin          needed by user to synchronize the
                             user/reply/watchdog
   6. unique request no
   7. update time


REQUESTS RELATED TO MIGRATION:
-----------------------------

c1 is a read clock that is migrating. It hands over
the read clock to the server s1, which then sends
this message to this client c2, telling it to become
the new read clock. c2 could already be a reader.
** 0. msg_code               C_MIG_HANDOVER_INSTALL_READ_CLOCK (H2)
   1. no args                6 + no readers
   2. ack_code               SEND_ACK_MIG_READ_CLOCK_INSTALLED
   3. page number
   4. page_status            READ_CLOCK
   5. page value
   6. trailer version
   7. no of readers (excluding this site) CHECKED CORRECT PAGE RETRIEVAL
   8. reader 1
   9. reader 2, etc


c1 is a write clock that is migrating. It hands over
the write clock to the server s1, which then sends
this message to this client c2, telling it to become
the new write clock. c2 could already be a writer.
** 0. msg_code               C_MIG_HANDOVER_INSTALL_WRITE_CLOCK (K2)
   1. no args                5
```

```
        2. ack_code               SEND_ACK_MIG_WRITE_CLOCK_INSTALLED
        3. page number
        4. page_status            WRITE_CLOCK
        5. page value
        6. trailer version


     c1 is a client that is migrating. s1 sends the following
     message to a client which is a read clock for a page
     for which c1 is a reader. s1 does not send the pages for
     which c1 was a reader. This client scans its read clocks,
     and removes c1 as a reader.
     ** 0. msg_code               C_MIG_REMOVE_CLIENT_AS_READER_FOR_PAGES (J2)
        1. no args                1
        2. client_to_remove


     This message is used for both migration and
     recovery. Hence it is put in file, clientrec.ca so
     it can be used for recovery in single server case.
     ** 0. msg_code               C_REC_REMOVE_SERVER_AS_READER_FOR_A_PAGE
                                           A clock site is told to remove a
                                           server as a reader for a specific
                                           page.
        1. no args                2
        2. server code,           of the server to be removed. This is
                                           NOT the server index. It is the read
                                           site number that the client will have
                                           in its list of readers.
       3. page no                 the page number to check.


  OUTPUTS : none
 ***********************************************************************/
void    requestForClientHandler(int argc, Word *argv)
{
        perClientDS     *thisClient;
        int             ret;

        thisClient = (perClientDS*) (*ptrToGlobalClientDS);

        assert(argv[1] != INIT_SEND_ARRAY);

        /*If the server is ok bit is set, the server thread could
         *be running, but the server could be recovering. So check
         *for the down or recovering before before putting a request
         *into the request q.
         */

        /* If the client or server are down or recovering, the
         * server waiting for an ack. to a request will not make
         * any difference. The server's data structures will be
         * validated during recovery.
         */
        if ( !((thisClient->server_state & SERVER_DOWN) == SERVER_DOWN)
              &&
             !((thisClient->server_state & SERVER_RECOVERING) ==
                SERVER_RECOVERING)
           )
        {
                /* If the client is not down and not recovering , put
                 * the request in the request queue.
                 * Even if the client is migrating, the request will be
                 * put in the request queue. It may be a reply for which
                 * the user thread is waiting, or it may be an invalidate
                 * message.
                 * If the client is not CLIENT_DOWN and not CLIENT_RECOVERING,
                 * put the request in the request queue.
                 * If the client is : CLIENT_NEVER_STARTED, it should
                 * not be able to accept requests.
                 * If CLIENT_TO_RECOVER,CLIENT_TO_MIGRATE, CLIENT_MIGRATING
                 * it can still accept requests.
                 * If CLIENT_MIGRATED_POLLING, its not in either cluster,
                 * so it cant accept requests  The only requests it
                 * could get at this point are invalidation messages
                 * from its old cluster (directly from clients in the old
```

```
 * cluster). But since its already erased
 * everything in its page table while migrating,
 * these messages can be ignored.
 *
 *      PARTICULAR TIMING UNUSUAL CASE:
 * - Client c1 is migrating from cluster 1 to cluster 2
 * - Server s1 hands over c1 to s2
 * - Server s2 accepts c1 and adds it to its list of of
 *   clients
 * - Another client c2 in cluster 2 is migrating out, so
 *   s2 is handing over c2's pages to other clients in
 *   cluster 2.
 * - Client C1 has sent a message S_NEW_CLIENT_ASKING_FOR
 *   ACCEPTANCE and is waiting for a reply. So, c1 has
 *   state CLIENT_MIGRATED_POLLING.
 * - But since s2 has c1 in its client list, it 'hands over'
 *   a clock which was at c2 to c1 (and not another client
 *   in its cluster). C1's S_NEW_CLIENT_ASKING_FOR_ACCEPTANCE
 *   is in s2's queue, but it is at the end of the queue
 *   with a lot of handover clock messages before it.
 * - S2 sends a "clock handover" (C_MIG_HANDOVER_INSTALL
 *   READ_CLOCK/ C_MIG_HANDOVER_INSTALL_WRITE_CLOCK) to
 *   c1.
 * - C1's REQUEST_FOR_CLIENT_IPI doesn't put that handover
 *   message in the queue because c1 is in state
 *   CLIENT_MIGRATED_POLLING, so this handover message is
 *   lost forever.
 * - c2 will never migrate because s1 keeps checking to
 *   see if all clock pages at c2 have been installed
 *   elsewhere. Lead to an infinite loop and effects
 *   throughput.
 *
 *      SOLUTION:
 *      A client in CLIENT_MIGRATED_POLLING state must put
 *  messages into its message queue, even though clientThread
 *  will not service them until client state changes to
 *  CLIENT_OK.
 *  Possible messages that can arrive in CLIENT_MIGRATED_
 *  POLLING, via REQUEST_FOR_CLIENT_IPI:
 *  C_MIG_HANDOVER_INSTALL_READ_CLOCK
 *  C_MIG_HANDOVER_INSTALL_WRITE_CLOCK
 *
 *  via MSG_FROM_SERVER_IPI:
 *  SET_CLIENT_STATE_MSG : CLIENT_DOWN/CLIENT_TO_RECOVER
 *  C_SERVER_ACCEPTING_NEW_CLIENT
 *
 *  Replies for earlier duplicate requests it made can also
 *  arrive at this point. These could be replies to requests
 *  it made in its old cluster. If an in-cluster page is
 *  sent and this client sees the page is out-cluster or
 *  vice versa, it will have to return the page if
 *  necessary.
 *
 *      ANOTHER TIMING UNUSUAL CASE:
 *  This client c1 has CLIENT_MIGRATED_POLLING. s2 has
 *  c1 in its list of clients. s2 goes down. s2 tells
 *  c1 to either set CLIENT_DOWN or CLIENT_TO_RECOVER
 *  (via MSG_FROM_SERVER_IPI).
 *  This should break the clientThread out of the
 *  loop of CLIENT_MIGRATED_POLLING. This will also
 *  break the clientPollingNewServerThread out of
 *  its while loop. No bugs foreseen in this case.
 *  The recovery code will empty the client request queue.
 *
 */

if ( !((thisClient->client_state & CLIENT_DOWN) ==
       CLIENT_DOWN)
       &&
     !((thisClient->client_state & CLIENT_RECOVERING) ==
       CLIENT_RECOVERING)
       /********
       &&
     !((thisClient->client_state & CLIENT_MIGRATED_POLLING) ==
       CLIENT_MIGRATED_POLLING)******/
       &&
```

```
                    !((thisClient->client_state & CLIENT_NEVER_STARTED) ==
                      CLIENT_NEVER_STARTED)
                )
            {
                      ret = insertIntoRequestQ(thisClient-
>client_request_queue,argv[1],argv[0],&argv[2]);

                      if ( ret == ERROR_Q)
                      {
                              fatal(" ERROR: client failed to insert request into
queue\n");
                      }
            }
        }
}


/*****************************************************************************
  FUNCTION: clientThread
  PURPOSE : This thread runs in an infinite while loop. It pulls out
            requests from the client request queue. If the queue is
            empty it sits in an idle loop and checks again. It checks
            for the server,system and client state. If the client is
            down it exits. Otherwise, it checks for some conditions
            and calls appropriate functions to handle requests.
            This thread is not counted in the no pending client threads.
  INPUTS  : none
  OUTPUTS : none
*****************************************************************************/
void    clientThread()
{
        perClientDS     *thisClient;
        auxClient       *clientAuxTable;
        int             ret_code;
        reqNode         *next_request;
        long            initsleep;

        thisClient = (perClientDS *) (*(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        while( CURR_TIME < TOTAL_SIM_RUN_TIME_CYCLES)
        {
                /* If the CLIENT_DOWN bit is set, the client thread must
                 * exit.
                 * Set the CLIENT_NEVER_STARTED bit and exit from client
                 * thread. This thread will be restarted when the system
                 * recovers.
                 * The CLIENT_TO_RECOVER bit is set by the recovering
                 * thread. If the client is recovering, the
                 * client thread does not have to exit. It can just sit
                 * in a loop.
                 * The client_never_started and client_ok cannot both
                 * be set at the same time.
                 * The client_down and client_ok should not be set at
                 * the same time.
                 */

                /* If the client or server are down or recovering, the
                 * server waiting for an ack. to a request will not make
                 * any difference. The server's data structures will be
                 * validated during recovery.
                 */

                /* If the server of this cluster is in the recovery
                 * process, or if the server is down, or if the
                 * client is in the recovery process, ignore the request
                 * and go to sleep in a loop until the system recovers.
                 */
                begin_atomic();
                if((thisClient->client_state & CLIENT_DOWN) == CLIENT_DOWN)
                {
                        emptyRequestQ(thisClient->client_request_queue);
                        thisClient->client_state &= 0;
                        thisClient->client_state |= CLIENT_NEVER_STARTED;
                        end_atomic();
                        return;
```

```
        }
        end_atomic();

        begin_atomic();
        if((thisClient->client_state & CLIENT_TO_RECOVER) ==
                        CLIENT_TO_RECOVER)
        {
                /* Empty request queue*/
                emptyRequestQ(thisClient->client_request_queue);
                thisClient->client_state &= 0;
                thisClient->client_state |= CLIENT_RECOVERING;
        }

        while (
            ((thisClient->server_state & SERVER_DOWN) ==
                SERVER_DOWN )
             ||
            ((thisClient->server_state & SERVER_RECOVERING) ==
                SERVER_RECOVERING)
             ||
            ((thisClient->client_state & CLIENT_RECOVERING) ==
                CLIENT_RECOVERING)
           )
        {
                if( CURR_TIME > TOTAL_SIM_RUN_TIME_CYCLES)
                {
                        thisClient->client_state &= 0;
                        thisClient->client_state |= CLIENT_NEVER_STARTED;
                        end_atomic();
                        return;
                }
                end_atomic();
                BUSY_WAIT(RECOVERY_SLEEP_TIME_CYCLES);
                begin_atomic();
                /* Do not need a thread_sleep_end_atomic
                 * for this. */
                /* SINCE ONLY ONE SITE CAN GO DOWN AT A TIME, DO NOT
                 * NEED TO CHECK FOR CLIENT_DOWN HERE.
                 */
        }
        end_atomic();

        begin_atomic(),
        if(
            ((thisClient->client_state & CLIENT_TO_MIGRATE) ==
                        CLIENT_TO_MIGRATE )
             &&
            ((thisClient->user_state & USER_MIGRATING) ==
                        USER_MIGRATING)
           )
        {
                if ( thisClient->client_request_queue->no_of_requests == 0)
                {
                        thisClient->client_state &= 0;
                        thisClient->client_state |= CLIENT_MIGRATING;
}
        }
        end_atomic();


        begin_atomic();
        while ( (thisClient->client_state & CLIENT_MIGRATING) ==
                CLIENT_MIGRATING)
        {
                if( CURR_TIME > TOTAL_SIM_RUN_TIME_CYCLES)
                {
                        thisClient->client_state &= 0;
                        thisClient->client_state |= CLIENT_NEVER_STARTED;
                        end_atomic();
                        return;
                }
                end_atomic();

                BUSY_WAIT( MIGRATING_SLEEP_TIME_CYCLES);

                if((thisClient->client_state & CLIENT_MIGRATING)!=
```

```
                                        CLIENT_MIGRATING)
                        {
                                begin_atomic();
                                break;
                        }

                        /* Make sure the request queue is empty. Once the client
                         * reaches this state, it will not get out of this
                         * while loop until an external event changes the state
                         * of this client to CLIENT_MIGRATED_POLLING, CLIENT_DOWN,
                         * or CLIENT_TO_RECOVER.
                         * So, in this loop the client will not service requests.
                         * But replies to duplicate requests that were sent
                         * earlier may still arrive, and the server may never
                         * migrate this client. See comment in
                         * REPLY_MAKE_READER function at client side when
                         * clientMigrateRecheckIfAllPagesSent is called.
                         * This applies to all REPLY.. messages at a client
                         * site.
                         */
                        /* pull out a request from the request queue*/
                        /* Service all requests that are in the request queue.
                         * This is done so that these messages will not have to
                         * be processed after this client migrates, creating
                         * bugs.
                         */
                        while( 1)
                        {
                        begin_atomic();
                        next_request = getNextRequest(thisClient-
>client_request_queue,&ret_code);
                        end_atomic();

                        if ( ret_code != EMPTY_Q)
                        {
                                /* Process the request.
                                 * Must be in non atomic region here.
                                 * This should only be a REPLY.. request or an
                                 * invalidate message from any client or its
                                 * server.
                                 * The messages allowed to be processed here are those
                                 * that may be sent to this client site even if it is
                                 * migrating. Typically, a new request for a page, or
                                 * a handover of a clock site should not be sent to a
                                 * client that is migrating.
                                 * No other requests should be processed here.
                                 * All the invalidate messages might get processed
                                 * after this client is in its new cluster.
                                 */
                                switch(next_request->msg_code)
                                {
                                case REPLY_MAKE_URSLF_READER_MSG:
                                case REPLY_MAKE_URSLF_READER_TRAILER_MSG:
                                case REPLY_MAKE_URSLF_WRITER_CLOCK_MSG:
                                case REPLY_MAKE_URSLF_READER_OUT_CLUSTER_PAGE_MSG:
                                case REPLY_MAKE_URSLF_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG:
                                case
REPLY_UPGRADE_URSLF_FROM_READER_TO_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG:
                                case REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG:
                                case REPLY_WAKE_UP_USER_MSG:
                                case C_INVALIDATE_READER:
                                case C_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE:
                                case C_INVALIDATE_OUT_CLUSTER_READER:
                                case C_INVALIDATE_WRITE_UPDATE_OUT_CLUSTER_PAGE:
                                case C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE:
                                case C_RETURNING_UPDATE_PAGE:
                                case C_MIG_REMOVE_CLIENT_AS_READER_FOR_PAGES:
                                case C_REC_REMOVE_SERVER_AS_READER_FOR_A_PAGE:
                                        {
                                                /* OK to process this request. */
                                                break;
                                        }
                                default:
                                        {
```

```
                                        printf(" \n\n ClientP%d request:
\n",CURR_PROCESSOR);
                                        printRequestNode(next_request);
                                        fatal("\n\n Client P%d state is %d, processing a
request , request code is %d.. INVALID REQUEST\n", CURR_PROCESSOR,thisClient-
>client_state, next_request->msg_code);
                                        }
                                }/* End of switch statement.*/

                                processRequestAtClient(next_request);
                                /* Will return here. Free the memory for
                                 * the request here instead of doing it in
                                 * the function where the request has been
                                 * handled.
                                 */
                                freeRequestNode(&next_request);
                                /* Must be in non atomic region here.*/
                                ASSERT_NOT_ATOMIC();
                        }/* end of non empty queue*/
                        else
                                break;
                        }
                        begin_atomic();
                }
                end_atomic();
                begin_atomic();
                while ( (thisClient->client_state & CLIENT_MIGRATED_POLLING) ==
                        CLIENT_MIGRATED_POLLING)
                {
                        if( CURR_TIME > TOTAL_SIM_RUN_TIME_CYCLES)
                        {
                                thisClient->client_state &= 0;
                                thisClient->client_state |= CLIENT_NEVER_STARTED;
                                end_atomic();
                                return;
                        }
                        end_atomic();
                        BUSY_WAIT( MIGRATING_SLEEP_TIME_CYCLES);
                        begin_atomic();
                }
                end_atomic();
                /* pull out a request from the request queue*/
                begin_atomic();
                next_request = getNextRequest(thisClient->client_request_queue,&ret_code);
                end_atomic();

                if ( ret_code == EMPTY_Q)
                { /* wait in a loop for sometime, or put the client
                   * thread to sleep
                   */
                        initsleep = CURR_TIME;
                        ASSERT_NOT_ATOMIC();
                        BUSY_WAIT(EMPTY_QUEUE_WAIT_TIME_CYCLES);

                }
                else
                {
                        /* process the request*/
                        /* must be in non atomic region here*/
                        processRequestAtClient(next_request);
                        /* will return here. Free the memory for
                         * the request here instead of doing it in
                         * the function where the request has been
                         * handled
                         */
                        freeRequestNode(&next_request);
              /*must be in non atomic region here*/
              ASSERT_NOT_ATOMIC();
                }
        } /* end of while (1) */

        begin_atomic();
        thisClient->client_state &= 0;
        thisClient->client_state |= CLIENT_NEVER_STARTED;
        end_atomic();
}
```

```
/*****************************************************************************
  FUNCTION: processRequestAtClient
  PURPOSE : This function checks for some conditions like if this
           processor is the correct clock site, and if that page
           should be serviced now by this clock site.
               For each message, it calls a function. The definitions of
               these functions are distributed in various client*.ca
               files.
  INPUTS  : reqNode *request  the request pulled out of the request Q
  OUTPUTS : none
 *****************************************************************************/
void    processRequestAtClient(reqNode *request)
{
        perClientDS             *thisClient;
        auxClient               *clientAuxTable;
        unsigned int            this_clock_page_status;
        int                     page_no,access_mode;
        int                     server_site;
        int                     len,i, ack_code, size_msg;
        Word                    SEND_ARRAY[3];
        int                     array_index;

        for(i=0; i< 3; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        ASSERT_NOT_ATOMIC();

        begin_atomic();
        thisClient = (perClientDS *) (*(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;
        end_atomic();

        /*not in atomic region here*/
        switch(request->msg_code)
        {
                case C_FWD_REQUEST:
                        {

                        page_no = request->array[1];
                        access_mode = request->array[2];

                        /* Update statistics at this client site.
                         * This is a control message during normal operation.
                         */
                        CYCLE_COUNTING_OFF;
                        begin_atomic();
                        globalClientStats[CURR_PROCESSOR].normal.control_msg_count++;
                        size_msg = 8 * sizeof(Word);
                        globalClientStats[CURR_PROCESSOR].normal.control_msg_bytes +=
size_msg;
                        end_atomic();

                        /* Update statistics for the time interval into which
                         * this message falls.
                         */
                        array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

                        if(MIG_INTERVAL)
                        {

                        assert(thisClient->cluster_index >=0 &&
                                   thisClient->cluster_index < NO_OF_SERVERS);
                        assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);

                        globalIntervalData[thisClient-
>cluster_index].in_cluster_control_byte[array_index] += size_msg;
                        globalIntervalData[thisClient-
>cluster_index].in_cluster_control_msg[array_index] += 1;
                        }

                        CYCLE_COUNTING_ON;

                        /* If the client or server are down or
                         * recovering, the server waiting for an ack.
                         * to a request will not make any difference.
```

```
                 * The server's data structures will be
                 * validated during recovery.
                 */

                /* Check if this client is the correct clock
                 * site or if the clock page has a service
                 * later.*/
                /* If the page has SERVICE_LATER, ignore the
                 * request. Do not test for just the LATER
                 * bit. Only valid values for auxClient.service
                 * are SERVICE_LATER and SERVICE_NOW.
                 * If this client is migrating, in a multi-server
                 * environment, it doesnt service the request.
                 */
                begin_atomic();
                if ((clientAuxTable[page_no].clock_site !=
                        CURR_PROCESSOR) ||
                   (( clientAuxTable[page_no].page_status &
                        CLOCK) != CLOCK)
                                ||
                   ( clientAuxTable[page_no].service ==
                        SERVICE_LATER) )
                {
                        /* The clock could have changed by now.
                         * This has happened when:
                         * - The service is SERVICE_LATER because the write
                         *   clock has loaned the page to another cluster.
                         * - The clock is WRITE_CLOCK and the service is
                         *   SERVICE_LATER because this client is migrating.
                         */
                        if(( clientAuxTable[page_no].page_status &
WRITE_CLOCK_UPDATE)
                                != WRITE_CLOCK_UPDATE)
                        {
                        #ifdef PRINT_DEBUG
                        CYCLE_COUNTING_OFF;
                        sem_P(mutex);
                        printPerClientDS(thisClient);
                        sem_V(mutex);
                        CYCLE_COUNTING_ON;
                        printf("\n\n ERROR: MESSAGE SENT TO WRONG CLOCK SITE, client
P%d is not clock for page %d, reqd access = %d \n",CURR_PROCESSOR,page_no,access_mode);
                        snapshot();
                        #endif

                        }
                        /* Before ignoring the request:
                         * if the server is waiting for an
                         * ack and has set a service later
                         * ignore bit, it will never be
                         * changed to service_now because this
                         * request has died here. Any
                         * duplicate requests will not be
                         * serviced since the server has
                         * set the service later ignore!!
                         * Inform the server to undo the
                         * service later ignore. The clock site
                         * at the server will not change unless
                         * it gets a ack Change of clock msg.
                         * The clock site is the same.
                         */

                        ack_code = request->array[0];
                        if ( ack_code == SEND_ACK_CLOCK_CHANGE_MSG)
                        {
                        /* Send a msg not delivered, undo
                         * service later ignore, message to
                         * the server. This is a request
                         * related message, it is sent as a
                         * normal request to the server
                         * (REQUEST_FOR_SERVER_IPI).
                         *
                         * 0. S_MSG_NOT_DELIVERED
                         * 1. no_args..1
                         * 2. page_no
                         *  is clock site needed?
```

```
				 */
				/* Make sure this page is an in cluster page.
				 */
				if ( (page_no < thisClient->lowest_page) ||
					(page_no > thisClient->highest_page)
					)
				{
					CYCLE_COUNTING_OFF;
					sem_P(mutex);
					printf("\n\n in C_FWD_REQUEST, Client P%d not a
clock for page %d, this should be an in cluster page, but is an out cluster page. FATAL
ERROR\n",CURR_PROCESSOR,page_no);
					printPerClientDS(thisClient);
					fatal(" in C_FWD_REQUEST, Client P%d not a clock for
page %d, this should be an in cluster page, but is an out cluster page. FATAL
ERROR\n",CURR_PROCESSOR,page_no);
					sem_V(mutex);
					CYCLE_COUNTING_ON;
					end_atomic();
					return;
				}


				server_site = thisClient->server_site;

				SEND_ARRAY[0] = S_MSG_NOT_DELIVERED;
				SEND_ARRAY[1] = 1;
				SEND_ARRAY[2] = page_no;
				/* MESSAGE_LENGTH*/
				len = 3 * sizeof(Word);

	send_ipiV(server_site,PRIORITY,REQUEST_FOR_SERVER_IPI,len,3,SEND_ARRAY);
				assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);
				}

				end_atomic();
				return;
			}

			this_clock_page_status = clientAuxTable[page_no].page_status;

			/* The request cannot be ignored now.*/
			/* The service of this clock is changed to
			 * SERVICE_LATER first, so the user at the
			 * clock site cannot make changes to the
			 * clock until this request is serviced
			 * This setting to SERVICE_LATER should be done
			 * in the same atomic region as the testing of its
			 * value.
			 */
			clientAuxTable[page_no].service &= 0;
			clientAuxTable[page_no].service |= SERVICE_LATER;
			/* Only if the site remains a clock site
			 * will its service be changed to SERVICE_NOW.*/
			end_atomic();
			/* Switch not in atomic region.*/
			switch(access_mode)
			{
			case	READ: /* read request*/
				{
				/* There are 2 cases:
				 * 1.  Clock is reader, read request;
				 * 2.  Clock is writer, read request;
				 * In both cases, the clock makes some
				 * changes and sends a reply to the
				 * origin of the request.
				 */
				switch(this_clock_page_status)
				{
				case	READ:/* clock is reader*/
				case	READ_CLOCK:
					{
					/* Clock is reader, read request. */
					/* Call a function which will do
					 * the processing.
					 */
```

```
                                        readerClockHandlingReadRequest(request);
                                        ASSERT_NOT_ATOMIC();
                                        /* Change the service back to
                                         * SERVICE_NOW, since this is still
                                         * the clock site.*/
                                        begin_atomic();
                                        clientAuxTable[page_no].service &= 0;
                                        clientAuxTable[page_no].service
                                                |= SERVICE_NOW;
                                        end_atomic();
                                        break; /*End of read clock.*/
                                        }
                                case    WRITE: /*Clock is writer.*/
                                case    WRITE_CLOCK:
                                        {
                                        /* Clock is writer, read request. */
                                        /* Call a function which will do
                                         * the processing.
                                         */
                                        writerClockHandlingReadRequest(request);
                                        ASSERT_NOT_ATOMIC();
                                        /* Change the service back to
                                         * SERVICE_NOW, since this is still
                                         * the clock site.*/
                                        begin_atomic();
                                        clientAuxTable[page_no].service &= 0;
                                        clientAuxTable[page_no].service
                                                |= SERVICE_NOW;
                                        end_atomic();
                                        break; /* End of write clock.*/
                                        }
                                default:
                                        {
                                        fatal(" Invalid page mode %d for clock: at client
P%d for page %d\n",this_clock_page_status,page_no.CURR_PROCESSOR);
                                        break;
                                        }
                                } /* End of switch, this_clock_page_status.*/
                                break; /* End of read request.*/
                                }
                        case    WRITE: /*Write request.*/
                                {
                                switch(this_clock_page_status)
                                {
                                case    READ:   /* Clock is reader.*/
                                case    READ_CLOCK:
                                {
                                        /* Clock is reader,write request. */
                                        /* Call a function which will do
                                         * the processing.*/
                                        readerClockHandlingWriteRequest(request);
                                        ASSERT_NOT_ATOMIC();
                                        /* DO NOT change the service back to
                                         * SERVICE_NOW, since this was a
                                         * write request and the clock has
                                         * changed. Server is waiting for ack
                                         * S_ACK_CLOCK_CHANGE message from the new
                                         * clock.*/
                                        break; /* End of read clock.*/
                                }
                                case    WRITE:  /* Clock is writer.*/
                                case    WRITE_CLOCK:
                                {
                                        /* Clock is writer,write request. */
                                        /* Call a function which will do
                                         * the processing.*/
                                        writerClockHandlingWriteRequest(request);
                                        ASSERT_NOT_ATOMIC();
                                        /* DO NOT change the service back to
                                         * SERVICE_NOW, since this was a
                                         * write request and the clock has
                                         * changed. Server is waiting for ack
                                         * S_ACK_CLOCK_CHANGE message from the new
                                         * clock.*/
                                        break; /*End of write clock.*/
                                }
```

```
                                default:
                                {
                                        fatal(" Invalid page mode %d for clock; at client
P%d for page %d\n".this_clock_page_status,page_no,CURR_PROCESSOR):
                                        break;
                                }
                                } /* End of switch, this_clock_page_status.*/
                                break: /* End of write request.*/
                                }
                        default:
                                fatal(' Invalid access mode %d sent in
C_FWD_REQUEST\n".access_mode);
                                break;

                        } /* switch access_mode*/
                        /* Not in atomic region here.*/

                        break; /* End of case C_FWD_REQUEST. */
                        }
                case C_INVALIDATE_READER:
                        {
                        /* Call a function to invalidate the
                         * reader.*/
                        invalidateReaderOfPage(request);
                        ASSERT_NOT_ATOMIC();
                        break; /*End of case C_INVALIDATE_READER.*/
                        }

/*!!!!!!!!!!!!!!!!!!!!!!!!!! BEGIN MULTI SERVER!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
                #ifdef MULTI_SERVER

                case C_FWD_OUT_OF_CLUSTER_REQUEST:
                {
                        /* This clients server is forwarding a request
                         * from a  different server.
                         * There are 4 cases here:
                         * Clock R/W, and R/W request.
                         * Cant use the same functions as the
                         * C_FWD_REQUEST as those functions directly
                         * send a message to the user.
                         * Here, a reply has to be sent to the server.
                         */
                        clockHandlingOutOfClusterRequest(request):
                        ASSERT_NOT_ATOMIC();
                        break;
                }/*End of case C_FWD_OUT_OF_CLUSTER_REQUEST.*/
                case C_FWD_MAKE_URSLF_TEMP_CLOCK_HANDLE_READ_REQUEST:
                {
                        /* This client(c1) is a reader for an out of cluster
                         * page. Another client c11 in the same cluster
                         * cluster makes a read request for the same page.
                         * c1 becomes a temporary clock and gives a read
                         * copy to c11. In case of a invalidate all readers
                         * msg to this server, a msg is sent to c1, which
                         * in turn will invalidate c11.
                         */
                        /* The following function takes care of the service
                         * of this page.*/
                        makeUrslfTempClockHandleOutClusterReadRequest(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case C_INVALIDATE_ALL_OUT_CLUSTER_READERS_OF_PAGE:
                {
                        /* This client c1 is a reader(and maybe temp clock)
                         * for an out of cluster page. s1 tells c1 to invalidate
                         * all its readers. s1 has already invalidated the
                         * clock site for this page. c1 invalidates all its
                         * readers. c1 does not have to send an ack back to
                         * s1. The owner of this page is s2. s1 does not
                         * have to send an ack back to s2.
                         */
                        invalidateAllOutClusterReadersOfPage(request):
                        ASSERT_NOT_ATOMIC();
                        break;
                }
```

```
		case C_INVALIDATE_OUT_CLUSTER_READER:
		{
			/* A temporary read clock (for out of cluster page)
			 * tells its reader to invalidate its read page. It is
			 * different from C_INVALIDATE_READER. In
			 * C_INVALIDATE_READER, the page mode is changed,
			 * but the page stays in memory(it could be  a trailer).
			 * Since this is a multi server case, the page is
			 * removed from memory.
			 */
			invalidateOutClusterReaderOfPage(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case C_RETURNING_UPDATE_PAGE:
		{
			/* s2 is returning a write update page to c2. c2 uses update code
			 * to coordinate the update with the watchdog update thread.
			 */
			clockGettingItsWriteUpdatePage(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case C_INVALIDATE_WRITE_UPDATE_OUT_CLUSTER_PAGE:
		{
			/* c1 has a write update copy of page. s1 sends it an
			 * invalidate message since c1 has kept it for more than
			 * update time.
			 */
			invalidateOutClusterUpdateWriterPage(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case C_INVALIDATE_ALL_OUT_CLUSTER_READERS_FWD_WRITE_UPDATE:
		{
			/* c1 in s1 is a temp read clock of page p belonging to s2.
			 * A site in s1,c1-1, asks for write permission for page p.
			 * On getting a reply from s2, s1 sends a message to c1
			 * telling it to invalidate all the readers and forward the
			 * reply to c1-1. While forwarding the message to c1, s1
			 * set extra to NOT_REACHED. So, s1 is waiting for an ack
			 * from c1-1 (that it recd the page).
			 */
			invalidateAllOutClusterReadersFwdWriteUpdate(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		#endif /* of MULTI_SERVER*/
/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! END MULTI_SERVER!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/



		/* REPLIES TO REQUESTS*/
		case REPLY_MAKE_URSLF_READER_MSG:
		{
			/* A user is asked to become a reader
			 * of this page; it made a read request
			 * for a page in its cluster.
			 */
			replyMakeReader(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case REPLY_MAKE_URSLF_READER_TRAILER_MSG:
		{
			/* A user is asked to become a reader and
			 * trailer of this page; it made a read
			 * request for a page in its cluster.
			 * The clock was in write mode, so is
			 * making this site a trailer with the
			 * latest write.
			 */
			replyMakeReaderTrailer(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
```

```
		case REPLY_MAKE_URSLF_WRITER_CLOCK_MSG:
		{
			/* A user is asked to become a writer clock
			 * of this page; it made a write request
			 * for a page in its cluster. It may or may not
			 * already be a reader of this page.
			 */
			replyMakeWriterClock(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case REPLY_RESEND_REQUEST_FOR_PAGE_LATER_MSG:
		{
			/* A user has made a request for a page
			 * Its server tells it to resend this
			 * request. The server of this page has
			 * marked it SERVICE_LATER_REPLY.
			 */
		replyResendRequestLater(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case REPLY_WAKE_UP_USER_MSG:
		{
			/* A write clock gets a read request from the user at its
			 * own site. It downgrades to a reader, adds itself as a
			 * reader in the list of readers and send this message to
			 * itself to wake up its user. It does not send the page
			 * with this message as the page is already in its memory
			 * in the required mode.
			 */
			 replyWakeUpUser(request);
			 ASSERT_NOT_ATOMIC();
			 break;
		}
/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!! BEGIN MULTI SERVER!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
		#ifdef MULTI_SERVER

		case REPLY_MAKE_URSLF_READER_OUT_CLUSTER_PAGE_MSG:
		{
			/* A user is asked to become a reader
			 * for an out of cluster page. It made
			 * a read request for this page.
			 */
			replyMakeReaderOutClusterPage(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case REPLY_MAKE_URSLF_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG:
		{
			/* A user is asked to become an update writer
			 * for an out of cluster page. It made
			 * a write request for this page.
			 */
			replyMakeUpdateWriterOutClusterPage(request);
			ASSERT_NOT_ATOMIC();
			break;
		}
		case REPLY_UPGRADE_URSLF_FROM_READER_TO_UPDATE_WRITER_OUT_CLUSTER_PAGE_MSG:
		{
			/* c1 is a reader of a page. c1-1 requested write permission
			 * to this page. c1, c1-1 are in s1. s1 got a 'upgrade
			 * yourslf from reader to write update' reply from s2.
			 * s1 forwards the reply to c1, which in turn will forward
			 * the reply to c1-1. This reply is forwarded from c1 to
			 * c1-1 when c1 was already a reader and it requested a
			 * write (B7-1).
			 */
			 replyUpgradeFromReaderToUpdateWriterOutClusterPage(request);
			 ASSERT_NOT_ATOMIC();
			 break;
		}
		case C_MIG_HANDOVER_INSTALL_READ_CLOCK :
		{
			/* c1 is a read clock that is migrating. It hands over
```

```
                        * the read clock to the server s1, which then sends
                        * this message to this client c2, telling it to become
                        * the new read clock. c2 could already be a reader. (H2)
                        */
                        clientMigrateHandoverInstallNewReadClock(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case C_MIG_HANDOVER_INSTALL_WRITE_CLOCK:
                {
                        /* c1 is a write clock that is migrating. It hands over
                         * the write clock to the server s1, which then sends
                         * this message to this client c2, telling it to become
                         * the new write clock. c2 could already be a writer.(K2)
                         */
                        clientMigrateHandoverInstallNewWriteClock(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case C_MIG_REMOVE_CLIENT_AS_READER_FOR_PAGES:
                {
                        /* c1 is a client that is migrating. s1 sends the following
                         * message to a client which is a read clock for a page/pages
                         * for which c1 is a reader. s1 does not send the pages for
                         * which c1 was a reader. This client scans its read clocks,
                         * and removes c1 as a reader. (J2)
                         */
                        clientMigrateHandoverRemoveClientAsReader(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                case C_REC_REMOVE_SERVER_AS_READER_FOR_A_PAGE:
                {
                        /* This message is used for both migration and
                         * recovery. Hence it is put in file, clientrec.ca so
                         * it can be used for recovery in single server case.
                         * A clock site is told to remove a server as a reader
                         * for a specific page.
                         */
                        clientRecRemoveServerAsReaderForAPage(request);
                        ASSERT_NOT_ATOMIC();
                        break;
                }
                #endif /*of MULTI_SERVER*/
/*!!!!!!!!!!!!!!!!!!!!!!!!!!! END MULTI SERVER!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/

                default:
                {
                        fatal(" ERROR: Invalid msg code %d in
processRequestAtClient\n",request->msg_code);
                        break;
                }
        } /*End of switch request->msg_code.*/

}

/**********************************************************************
  FUNCTION: readerClockHandlingReadRequest
  PURPOSE : This function is called at the clock site when the
            clock is a reader for a page, and there is a read request
            for that page. Here the request is made by a client in the
            clock's cluster.
            In this case, no changes were made to the server and the
            acknowledgement code sent by the server is DO_NOT_SEND_ACK
            _MSG which is passed to the user which will not ack the
            receipt of the reply to the server.
            Since the clock is a reader, it checks to see if the
            requesting processor is already a reader for that page.
            If it is, it ignores the request.
            If not, it adds that processor as a reader, and sends
            a read copy of the page to the user at that processor.

            The service of this page is changed to SERVICE_LATER
            before this function call and is changed back to
            SERVICE_NOW on return from this function call.
            The reply sends a REQUEST_FOR_CLIENT_IPI to the site
```

```
          making the request.
          Not to be used in the recovery process.
  INPUTS  : reqNode *request
          request->msg_code  C_FWD_REQUEST
          request->no_values 6
          request->array:
          0. ack_code                         DO_NOT_SEND_ACK_MSG
                        This ack_code is sent by the server.The
                        clock passes it on to the user, which
                        sends an ack to the server, if reqd.
          1. page_no
          2. access_mode                      READ (reqd of page requested)
          3. tid of origin
          4. processor of origin
          5. unique request no
  OUTPUTS : none
**************************************************************************/
void readerClockHandlingReadRequest(reqNode *request)
{
        perClientDS     *thisClient;
        auxClient       *clientAuxTable;
        intNode         ret_node, *node;
        int             ret,len, i;
        int             page_value_at_clock;
        int             ack_code, page_no, access_mode;
        int             tid_of_origin,proc_origin,uniq_request_no;
        Word            SEND_ARRAY[8];

        ASSERT_NOT_ATOMIC();
        thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        for(i=0; i< 8; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        /* The state of the server, client and system have already
         * been checked when this request was removed from
         * the request queue.
         */

        /* The clock has to add proc_of_origin as a new reader of
         * page_no.
         * The clock remains a clock. No changes have to be sent back to
         * the server.
         */
        /* This could be a duplicate request, so the user site may
         * already be a reader.*/
         ack_code = request->array[0];
         page_no = request->array[1];
         access_mode = request->array[2];
         tid_of_origin = request->array[3];
         proc_origin = request->array[4];
         uniq_request_no = request->array[5];

        /* check for the ack code*/
        if ( ack_code != DO_NOT_SEND_ACK_MSG)
        {
                fatal( "ERROR: In readerClockHandlingReadRequest P%d, ack code is not
DO_NOT_SEND_ACK_MSG but %d\n",CURR_PROCESSOR,ack_code);
                return;
        }

        begin_atomic();
        ret =
findInIntList(clientAuxTable[page_no].list_of_readers,proc_origin,&ret_node);
        if( ret == FOUND)
        {
                /* Do not process the request.*/
                /* The  request is no longer needed. Memory for it will
                 * be freed in clientThread on return.
                 */

                checkSinglePage(clientAuxTable[page_no], page_no, CURR_PROCESSOR,
CHECK_TRAILER_VER," 1 readerClockHandlingReadRequest");

                end_atomic();
```

```
                return;
        }

        /* The requestor is not a reader. Add it to the list of
         * readers. Send an ipi to that processor.
         */

        node = createIntNode( proc_origin);
        if (node == NULL)
        {
                fatal(" ERROR: could not create int node to add reader to readers list\n");
                /* Not in atomic region.*/
                /* The  request is no longer needed. Memory for it will
                 * be freed in clientThread on return.
                 */
                return ;

        }

        ret = insertIntoIntList(clientAuxTable[page_no].list_of_readers,node);
        if (ret == ERROR_LIST)
        {
                fatal(" ERROR: could not insert node into readers list\n");
                end_atomic();
                /* The  request is no longer needed. Memory for it will
                 * be freed in clientThread on return.
                 */
                return ;

        }

        page_value_at_clock = clientAuxTable[page_no].page_value;
        checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 2 readerClockHandlingReadRequest");
        end_atomic();


        /* The clock sends a REQUEST_FOR_CLIENT_IPI to the user
         * who made the request. The args sent are:
         * 0. msg_code          REPLY_MAKE_URSLF_READER_MSG , telling the
         *                      user to make itself a reader of
         *                      this page.
         * 1. no args           6
         * 2. ack_code          DO_NOT_SEND_ACK_MSG. This is the code the
         *                      server sent the clock. since the server is not waiting
         *                      for an ack since the clock is not changing.
         * 3. page_no           the page no of the requested page
         * 4. access_mode       this should be READ
         * 5. page_value        the value at the clock site
         * 6. tid_of_origin     needed by user to synchronize the
         *                      user/reply/watchdog
         * 7. unique request no
         */
        SEND_ARRAY[0] = REPLY_MAKE_URSLF_READER_MSG;
        SEND_ARRAY[1] = 6;
        SEND_ARRAY[2] = ack_code;
        SEND_ARRAY[3] = page_no;
        SEND_ARRAY[4] = READ;
        SEND_ARRAY[5] = page_value_at_clock;
        SEND_ARRAY[6] = tid_of_origin;
        SEND_ARRAY[7] = uniq_request_no;
        /* MESSAGE_LENGTH*/
        len =( 7 + PAGE_SIZE) * sizeof(Word);
        send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,8,SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

        /* The  request is no longer needed. Memory for it will
         * be freed in clientThread on return.
         */
}

/***********************************************************************
  FUNCTION: writerClockHandlingReadRequest
  PURPOSE : Case 1:
            This function is called at the clock site when the
            clock is a writer for a page, and there is a read request
```

```
         for that page.

         In this case, the server made no changes though it
         knows that the clock will downgrade from a writer to
         a reader. The server is not waiting for an acknowledgement
         from the user as it knows the clock site will not
         change since this is a read request. It sends a
         DO_NOT_SEND_ACK_MSG.
         Once this clock changes its mode, it informs the
         server about this mode change by sending a
         S_INFORM_CLOCK_MODE_CHANGE msg to the server.

         Since the clock is a writer, it has the latest copy of
         the page. It downgrades itself to a reader, and gives
         a copy to the user. Now, since a writer clock has
         presumably just written to this page, the latest
         trailer of this page in the system does not have this
         change. So, this clock, when it downgrades to a reader,
         will give the user a read copy and also make it a trailer.

         Case 2:
         The user thread at the clock site itself could have
         requested a read when the clock is in write mode.
         In this case, the server made no changes though it
         knows that the clock will downgrade from a writer to
         a reader. The server is not waiting for an acknowledgement
         from the user as it knows the clock site will not
         change since this is a read request. It sends a
         DO_NOT_SEND_ACK_MSG.
         Once this clock changes its mode, it informs the
         server about this mode change by sending a
         S_INFORM_CLOCK_MODE_CHANGE msg to the server.

         Since the clock is a writer, it has the latest copy of
         the page. It downgrades itself to a reader, and actually
         does not need to give a copy to the user since the user
         is the same site. Since its the same site, there
         is no new trailer. So it just sends an IPI to its
         own processor, which telling the user to wake up. Since this
            site is already a reader now, the read page is NOT sent.

         That request is sent to the server. The server does
         not make any changes since it knows the clock will stay
         the same.

         The service of this page is changed to SERVICE_LATER
         before this function call and is changed back to
         SERVICE_NOW on return from this function call.

         In both cases, this request is made by a client in the
         clock's cluster.

         Not to be used in the recovery process.
 INPUTS  : reqNode *request
         request->msg_code          C_FWD_REQUEST
         request->no_values         6
         request->array:
         0. ack_code                DO_NOT_SEND_ACK_MSG
                         This ack_code is sent by the server.The
                         clock passes it on to the user, which
                         does not send an ack to the server
         1. page_no
         2. access_mode             READ (reqd of page requested)
         3. tid of origin
         4. processor of origin
         5. unique request no
 OUTPUTS : none
 *********************************************************************/
void writerClockHandlingReadRequest(reqNode *request)
{
      perClientDS      *thisClient;
      auxClient        *clientAuxTable;
      int              ack_code, page_no, access_mode;
      int              tid_of_origin,proc_origin,uniq_request_no;
      int              new_trailer_version;
      int              page_value_at_clock;
```

```
	int		i,len;
	Word		SEND_ARRAY[9];
	intNode		*node;
	int		ret,server_site;

	ASSERT_NOT_ATOMIC();
	thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
	clientAuxTable = thisClient->client_ptable;

	for(i=0; i< 9; i++)
		SEND_ARRAY[i] = INIT_SEND_ARRAY;

	 ack_code = request->array[0];
	 page_no = request->array[1];
	 access_mode = request->array[2];
	 tid_of_origin = request->array[3];
	 proc_origin = request->array[4];
	 uniq_reequest_no = request->array[5];

	/* Check for the ack code.*/
	if ( ack_code != DO_NOT_SEND_ACK_MSG)
	{
		fatal( "ERROR: In writerClockHandlingReadRequest P%d, ack code is not
DO_NOT_SEND_ACK_MSG but %d\n",CURR_PROCESSOR,ack_code);
		return;
	}

	/* The clock has to invalidate its write mode
	 * Do not do that here because this function
	 * may return with an error. Do it only after no
	 * further errors are possible.
	 */

	/* When a clock is a reader, the clock site is added to
	 * the list of readers.
	 */
	node = createIntNode( CURR_PROCESSOR);
	if (node == NULL)
	{
		fatal(" ERROR: could not create int node to add reader to readers list\n");
		/* Not in atomic region.*/
		/* the  request is no longer needed. Memory for it will
		 * be freed in clientThread on return.
		 */
		return;

	}

	begin_atomic();
	ret = insertIntoIntList(clientAuxTable[page_no].list_of_readers,node);
	if (ret == ERROR_LIST)
	{
		fatal(" ERROR: could not insert node into readers list\n");
		end_atomic();
		/* The  request is no longer needed. Memory for it will
		 * be freed in clientThread on return.
		 */
		return ;

	}

	/* Add the user as a reader,only if it is not this site
	 * itself. If the write clock itself requested a read,
	 * only its mode needs to be changed.
	 */
	if( proc_origin != CURR_PROCESSOR)
	{
		node = createIntNode( proc_origin);
		if (node == NULL)
		{
			fatal(" ERROR: could not create int node to add reader to readers
list\n");
			/* Not in atomic region.*/
			/* The  request is no longer needed. Memory for it will
			 * be freed in clientThread on return.
			 */
```

```
                        end_atomic();
                        return ;

                }

                ret = insertIntoIntList(clientAuxTable[page_no].list_of_readers,node);
                if (ret == ERROR_LIST)
                {
                        fatal(" ERROR: could not insert node into readers list\n");
                        end_atomic();
                        /* The  request is no longer needed. Memory for it will
                         * be freed in clientThread on return.
                         */
                        return ;

                }
        } /* End of if proc_origin != CURR_PROCESSOR.*/

        /* The clock has to first invalidate its write mode.
         */
        clientAuxTable[page_no].page_status &= ~(WRITE);

        /* The clock makes itself a reader.*/
        clientAuxTable[page_no].page_status |= READ;

        /* The current trailer version is the latest t_version
         * in the cluster. A read copy of this page is sent to
         * the user. This user becomes the next trailer site.
         * Increment trailer_version.
         */
        if ( proc_origin != CURR_PROCESSOR)
        {
                (clientAuxTable[page_no].trailer_version)++;
        }
        new_trailer_version = clientAuxTable[page_no].trailer_version;
        page_value_at_clock = clientAuxTable[page_no].page_value;
        server_site = thisClient->server_site;

        checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 1 writerClockHandlingReadRequest");

        end_atomic();


        /* The clock sends a REQUEST_FOR_CLIENT_IPI to the user
         * who made the request. The args sent are:
         * 0. msg_code          REPLY_MAKE_URSLF_READER_TRAILER_MSG.
         *                      telling the user to make itself a  reader and
         *                      trailer of this page.
         * 1. no args           7
         * 2. ack_code          DO_NOT_SEND_ACK_MSG. This is the code the
         *                      server sent the clock, since the server is not waiting
         *                      for an ack.
         * 3. page_no           the page no of the requested page
         * 4. access_mode       this should be READ_TRAILER
         * 5. page_value        the value at the clock site
         * 6. tid_of_origin     needed by user to synchronize the
         *                      user/reply/watchdog
         * 7. unique request no
         * 8. trailer_version  the latest t version in the system
         */
        if( proc_origin != CURR_PROCESSOR)
        {
                SEND_ARRAY[0] = REPLY_MAKE_URSLF_READER_TRAILER_MSG;
                SEND_ARRAY[1] = 7;
                SEND_ARRAY[2] = ack_code;
                SEND_ARRAY[3] = page_no;
                SEND_ARRAY[4] = READ_TRAILER;
                SEND_ARRAY[5] = page_value_at_clock;
                SEND_ARRAY[6] = tid_of_origin;
                SEND_ARRAY[7] = uniq_request_no;
                SEND_ARRAY[8] = new_trailer_version;
                /* MESSAGE_LENGTH*/
                len = ( 8 + PAGE_SIZE) * sizeof(Word);
                send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,9,SEND_ARRAY);
                assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);
```

```
}
else
{
        /* If the user at this write clock itself made the
         * read request, this write clock downgrades to a
         * reader, adding itself as a reader. Now this page is
         * already in a read mode at this site.
         *
         * PARTICULAR TIMING UNUSUAL CASE:
         * Do not send a REPLY_MAKE_URSLF_READER_MSG to itself.
         * Send a REPLY_WAKE_UP_USER which will not install the
         * page in memory, but will only wake up the user indicating
         * that the page is already in memory.
         * Consider the following:
         * 1. This client is c1. It is a write clock.
         * 2. c1 makes a read request for this page.
         * 3. Simulataneously with (2), c2(another client in this
         *    cluster) makes a write request.
         * 4. c1 gets the read request from itself. It downgrades
         *    to a read clock and sends a REPLY_MAKE_URSLF_READER_MSG
         *    to itself 'with the read page'. Already this clock
         *    has downgraded to a read clock with itself as a
         *    reader.
         * 5. Immediately after 4, it gets 3. It has to handover
         *    clock privileges to c2. It invalidates its read
         *    mode in invalidateAllReaders. This invalidation is
         *    done locally, without sending an invalidate message
         *    to itself. So it becomes a TRAILER page and hands
         *    over write clock to c2.
         * 6. The REPLY_MAKE_URSLF_READER_MSG that c1 sent to
         *    itself in 4 arrives. By this time this client has
         *    already become a trailer. That reply installs a
         *    read page, and this client becomes a READ_TRAILER.
         * 7. Parallel to 6, c2 gets 5 and becomes a WRITE_CLOCK.
         *    THE SYSTEM IS IN AN INCONSISTENT STATE because:
         *    - one client in the cluster is a write clock.
         *    - another client in the cluster is a read trailer (it
         *      should have been just a trailer).
         * SOLUTION:
         * Instead of sending a REPLY_MAKE_URSLF_READER_MSG in 4,
         * send a REPLY_WAKE_UP_USER. This message will not
         * re-install a read page. It just wakes up the user telling
         * it that the page is in memory. By the time the user
         * tries to read from this page, if the read clock has
         * already handed over to c2, the user will find that the
         * read page is not in memory and will have to re-send the
         * request to its server
         * Note: Instead of a REPLY_WAKE_UP_USER this clock can
         * wake up the user right here since the user is on the
         * same processor. But for elegance and to conform to
         * a reply arriving via REPLY_.. messages, this REPLY_WAKE_
         * UP_USER is sent anyway.
         *
         * This bug does not occur in readerClockHandlingWriteRequest
         * from itself, because the reader clock totally invalidates
         * itself and hands over write clock privileges to itself.
         */

        /* The clock sends a REQUEST_FOR_CLIENT_IPI to the user
         * at its own site. The args sent are:
         * 0. msg_code          REPLY_WAKE_UP_USER_MSG,
         *                      telling the user to wake up since the page is
         *                      already in memory.
         * 1. no args           3
         * 2. ack_code          DO_NOT_SEND_ACK_MSG. The user at this
         *                      site does not need to send any ack to the server.
         * 3. tid_of_origin     needed by user to synchronize the
         *                      user/reply/watchdog
         * 4. unique request no          "
         */
         SEND_ARRAY[0] = REPLY_WAKE_UP_USER_MSG;
         SEND_ARRAY[1] = 3;
         SEND_ARRAY[2] = DO_NOT_SEND_ACK_MSG;
         SEND_ARRAY[3] = tid_of_origin;
         SEND_ARRAY[4] = uniq_request_no;
```

```
            /*MESSAGE_LENGTH*/
            len = 5 * sizeof(Word);
            send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,8,SEND_ARRAY);
            assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

        }
        /* Now the write clock has become a read clock.
         * Inform the server of this change.
         */

        /* Inform the server of change in clock mode. This
         * is a request related message, it is sent as a
         * normal request to the server (REQUEST_FOR_SERVER_IPI).
         *
         * Send the request ipi to the server.
         * 4 arguments:
         * 0. request code              S_INFORM_CLOCK_MODE_CHANGE
         * 1. no_args                   3 (excluding msg code)
         * 2. page_no
         * 3. new_clock_mode   READ
         * 4. this site
         */
        SEND_ARRAY[0] = S_INFORM_CLOCK_MODE_CHANGE;
        SEND_ARRAY[1] = 3;
        SEND_ARRAY[2] = page_no;
        SEND_ARRAY[3] = READ;
        SEND_ARRAY[4] = CURR_PROCESSOR;
        /* MESSAGE_LENGTH*/
        len = 5 * sizeof(Word);
        send_ipiV(server_site, PRIORITY,REQUEST_FOR_SERVER_IPI,len,5,SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);
        /* 5 args sent in SEND_ARRAY*/
}

/*************************************************************************
  FUNCTION: readerClockHandlingWriteRequest
  PURPOSE : This function is called at the clock site when the
            clock is a reader for a page, and there is a write request
            for that page. Here, this request is made by a client in
            the clock's cluster.
            In this case, the server changed the service of that
            page to SERVICE_LATER_IGNORE. The server will wait
            for an acknowledgment from the user before servicing any
            further requests for that page.
            The server passes a SEND_ACK_CLOCK_CHANGE_MSG in the
            request fwd'd to the clock. The clock forwards this
            to the user.
            Since the clock is a reader, it has to invalidate itself
            and its other readers. It becomes a trailer and the user
            becomes the new clock.
            Even if the user that is requesting a write is already a
            reader, the read clock is completely invalidated and the
            reader is told to become a write clock. Therefore, the
            reply write clock message should not find the page in
            read mode.
            This clock forwards the reply to the user using a REQUEST_
            FOR_CLIENT_IPI.

            The service of this page is changed to SERVICE_LATER
            before this function call and is changed back to
            SERVICE_NOW on return from this function call.
            Not to be used in the recovery process.
  INPUTS  : reqNode *request
            request->msg_code           C_FWD_REQUEST
            request->no_values          6
            request->array:
            0. ack_code                 SEND_ACK_CLOCK_CHANGE_MSG
                          This ack_code is sent by the server.The
                          clock passes it on to the user, which
                          sends an ack to the server. The ack sent to
                          the server is S_ACK_CLOCK_CHANGE msg.
            1. page_no
            2. access_mode              WRITE (reqd of page requested)
            3. tid of origin
            4. processor of origin
```

```
            5. unique request no
  OUTPUTS : none
*****************************************************************************/
void readerClockHandlingWriteRequest(reqNode *request)
{
        perClientDS      *thisClient;
        auxClient        *clientAuxTable;
        int              ack_code, page_no, access_mode;
        int              tid_of_origin,proc_origin,uniq_request_no;
        int              new_trailer_version;
        int              page_value_at_clock;
        int              len,i;
        Word             SEND_ARRAY[9];

        ASSERT_NOT_ATOMIC();
        thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        for(i=0; i< 9; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

         ack_code = request->array[0];
         page_no = request->array[1];
         access_mode = request->array[2];
         tid_of_origin = request->array[3];
         proc_origin = request->array[4];
         uniq_request_no = request->array[5];

        /* Check for the ack code.*/
        if ( ack_code != SEND_ACK_CLOCK_CHANGE_MSG)
        {
                fatal( "ERROR: In readerClockHandlingWriteRequest P%d, ack code is not
SEND_ACK_CLOCK_CHANGE_MSG but %d\n",CURR_PROCESSOR,ack_code);
                return;
        }


        /* Invalidate all readers. Messages are  sent to all the
         * readers. The clock does not wait for an acknowledgement
         * for the invalidation. It just sends the messages and
         * assumes they will reach the destination.
         */
        /* Not in atomic region here.*/
        /* If the request was from the clock itself, the clock's
         * read page is also invalidated. There pass -1  as
         * the second argument.
         */
        begin_atomic();

        invalidateAllReaders(page_no, -1);
        ASSERT_ATOMIC();

        /* Now the read mode for the clock has been turned off. whether
         * or not the clock site itself made this write request.
         * Turn the clock mode off.
         * Increment the trailer version. if the request was not
         * from the clock itself.
         *
         * Give this clock a trailer mode if the request was not
         * from the clock.
         */
        clientAuxTable[page_no].page_status &= ~(CLOCK);
        /* The clock_site of this page was CURR_PROCESSOR.
         * Invalidate it by making it -1
         */
        clientAuxTable[page_no].clock_site = -1;
        /* Assign the page value before it is invalidated. */
        page_value_at_clock = clientAuxTable[page_no].page_value;

        if ( proc_origin != CURR_PROCESSOR)
        {
                /* Make it a trailer.*/

                clientAuxTable[page_no].page_status |= TRAILER;
                /* The current trailer version is the latest t_version
                 * in the cluster. This clock will be the next t_version.
```

```
                 * Increment trailer_version.
                 */
                (clientAuxTable[page_no].trailer_version)++;
                new_trailer_version = clientAuxTable[page_no].trailer_version;
                /* Do not invalidate the page value*/
        }
        else
        {
                clientAuxTable[page_no].page_status |= PAGE_NOT_IN_MEMORY;
                /* Even  if the same site requested the write, invalidate
                 * the trailer version so that a page consistency check
                 * can be made here. All the values for a write clock
                 * at the same site are sent in the reply message and
                 * can be used to restore clock status.
                 */
                new_trailer_version = clientAuxTable[page_no].trailer_version;
                clientAuxTable[page_no].trailer_version = 0;
                /* Since the clock is handed over, invalidate the page
                 * value
                 */
                clientAuxTable[page_no].page_value = -1;
        }
        /* The list of readers should be empty, done in
         * invalidateAllReaders.
         */

        /* Now, the reader clock has invalidated itself, and made
         * itself a trailer, if the write request was not from itself.
         * If the write request was from itself, the page has been
         * compeletely invalidated, and write clock privileges are
         * handed over to itself in a new message.
         * Send clock information to the origin of request. Give
         * it a WRITE_CLOCK mode.
         */

        checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 1 readerClockHandlingWriteRequest");

        end_atomic();

        /* The clock sends a REQUEST_FOR_CLIENT_IPI to the user
         * who made the request. The args sent are:
         * 0. msg_code          REPLY_MAKE_URSLF_WRITER_CLOCK_MSG,
         *                      telling the user to make itself a clock and a
         *                      writer of this page.
         *                      Even if the user is alreay a reader, the page value
         *                      is sent.
         * 1. no args           7
         * 2. ack_code          SEND_ACK_CLOCK_CHANGE_MSG This is the code
         *                      the server sent the clock, since the
         *                      server is waiting for an ack
         *                      for completion of clock change.
         * 3. page_no           the page no of the requested page
         * 4. access_mode       this should be WRITE_CLOCK
         * 5. page_value        the value at the clock site
         *                      Even if the requesting user is already a user,
         *                      the page value is sent. This is because even if the
         *                      requesting client is already a reader, it may invalidate
         *                      the read page if its migrating, yet be waiting for the
         *                      same write page to change to a migrating state. So, if
         *                      the requesting client does not have the read page in
         *                      memory when it gets the message,it uses the page value
         *                      sent in this message.
         * 6. tid_of_origin     needed by user to synchronize the
         *                      user/reply/watchdog
         * 7. unique request no
         * 8. trailer_version  the latest t version in the system
         */
        SEND_ARRAY[0] = REPLY_MAKE_URSLF_WRITER_CLOCK_MSG;
        SEND_ARRAY[1] = 7;
        SEND_ARRAY[2] = ack_code;
        SEND_ARRAY[3] = page_no;
        SEND_ARRAY[4] = WRITE_CLOCK;
        SEND_ARRAY[5] = page_value_at_clock;
        SEND_ARRAY[6] = tid_of_origin;
        SEND_ARRAY[7] = uniq_request_no;
```

```
        SEND_ARRAY[8] = new_trailer_version;
        /* MESSAGE_LENGTH*/
        len =( 8 + PAGE_SIZE) * sizeof(Word);
        send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,9,SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

        /* This function has been tested for the following cases:
         * readerClockHandlingWriteRequest:
         * - request from itself,      totally invalidated page, sent a message
         *                             to itself to install write clock.
         * - request from client that is not a reader
         *                             this site became trailer, handed over write clock.
         * - request from another client (not itself) that is a reader,
         *                             sent an invalidate message to itself
         *                             (in invalidateAllReaders), handed over write clock.
         */
}

/*************************************************************************
  FUNCTION: writerClockHandlingWriteRequest
  PURPOSE : This function is called at the clock site when the
            clock is a writer for a page, and there is a write request
            for that page. This request is made by a client in the clock's
            cluster.
            In this case, the server changed the service of that
            page to SERVICE_LATER_IGNORE. The server will wait
            for an acknowledgment from the user before servicing any
            further requests for that page.
            The server passes a SEND_ACK_CLOCK_CHANGE_MSG in the
            request fwd'd to the clock. The clock forwards this
            to the user.
            Since the clock is a writer, it invalidates itself.
            It becomes a trailer and the user becomes the new clock.
            This clock forwards the reply to the user. The reply sends
            a CLIENT_FOR_SERVER_IPI to the user

            The service of this page is changed to SERVICE_LATER
            before this function call and is changed back to
            SERVICE_NOW on return from this function call.
            Not to be used in the recovery process.
  INPUTS  : reqNode *request
            request->msg_code          C_FWD_REQUEST
            request->no_values         6
            request->array:
            0. ack_code                SEND_ACK_CLOCK_CHANGE_MSG
                           This ack_code is sent by the server.The
                           clock passes it on to the user, which
                           sends an ack to the server(S_ACK_CLOCK_CHANGE;
            1. page_no
            2. access_mode             WRITE (reqd of page requested)
            3. tid of origin
            4. processor of origin
            5. unique request no
  OUTPUTS : none
*************************************************************************/
void writerClockHandlingWriteRequest(reqNode *request)
{
        perClientDS     *thisClient;
        auxClient       *clientAuxTable;
        int             ack_code, page_no, access_mode;
        int             tid_of_origin,proc_origin,uniq_request_no;
        int             new_trailer_version;
        int             page_value_at_clock;
        int             len,i;
        Word            SEND_ARRAY[9];

        ASSERT_NOT_ATOMIC();
        thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        for(i=0; i< 9; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

         ack_code = request->array[0];
         page_no = request->array[1];
         access_mode = request->array[2];
```

```
        tid_of_origin = request->array[3];
        proc_origin = request->array[4];
        uniq_request_no = request->array[5];

        /* check for the ack code*/
        if ( ack_code != SEND_ACK_CLOCK_CHANGE_MSG)
        {
                fatal( "ERROR: In writerClockHandlingWriteRequest P%d, ack code is not
SEND_ACK_CLOCK_CHANGE_MSG but %d\n",CURR_PROCESSOR,ack_code);
                return;
        }


        /* Turn the write mode off.
         * Turn the clock mode off. Increment the trailer version,
         * and give this clock a trailer mode.
         */
        begin_atomic();
        clientAuxTable[page_no].page_status &= ~(WRITE);
        clientAuxTable[page_no].page_status &= ~(CLOCK);

        /* make it a trailer*/

        clientAuxTable[page_no].page_status |= TRAILER;
        /* The current trailer version is the latest t_version
         * in the cluster. This clock will be the next t_version.
         * Increment trailer_version.
         */
        (clientAuxTable[page_no].trailer_version)++;
        new_trailer_version = clientAuxTable[page_no].trailer_version;

        /* The clock_site of this page was CURR_PROCESSOR.
         * Invalidate it by making it -1.
         */
        clientAuxTable[page_no].clock_site = -1;
        /* The list of readers should be empty, was empty since
         * this was a write clock.
         */

        /* Send clock information to the origin of request. Give
         * it a WRITE_CLOCK mode
         */

        page_value_at_clock = clientAuxTable[page_no].page_value;
        checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 1 writerClockHandlingWriteRequest");
        end_atomic();

        /* The clock sends a REQUEST_FOR_CLIENT_IPI to the user
         * who made the request. The args sent are:
         * 0. msg_code                  REPLY_MAKE_URSLF_WRITER_CLOCK_MSG, telling
         *                      the user to make itself a clock and a
         *                      writer of this page.
         * 1. no args, 7
         * 2. ack_code                  SEND_ACK_CLOCK_CHANGE_MSG. This is the code
         *                      the server sent the clock, since the server is waiting
         *                      for an ack for completion of clock change.
         * 3. page_no                   the page no of the requested page
         * 4. access_mode               this should be WRITE_CLOCK
         * 5. page_value                the value at the clock site
         * 6. tid_of_origin             needed by user to synchronize the
         *                      user/reply/watchdog
         * 7. unique request no
         * 8. trailer_version  the latest t version in the system
         */
        SEND_ARRAY[0] = REPLY_MAKE_URSLF_WRITER_CLOCK_MSG;
        SEND_ARRAY[1] = 7;
        SEND_ARRAY[2] = ack_code;
        SEND_ARRAY[3] = page_no;
        SEND_ARRAY[4] = WRITE_CLOCK;
        SEND_ARRAY[5] = page_value_at_clock;
        SEND_ARRAY[6] = tid_of_origin;
        SEND_ARRAY[7] = uniq_request_no;
        SEND_ARRAY[8] = new_trailer_version;
        /* MESSAGE_LENGTH*/
        len = ( 8 + PAGE_SIZE) * sizeof(Word);
```

```
        send_ipiV(proc_origin,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,9,SEND_ARRAY);
        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);
}

/*****************************************************************************
  FUNCTION: invalidateAllReaders
  PURPOSE : It sends a REQUEST_FOR_CLIENT_IPI to each reader of
            the page, telling it to invalidate its reader.

            This function is called by a clock site invalidating
            its readers. The reader list is deleted in this function.
            The clock site does not invalidate itself here but sends a
            read invalidation message to itself. The clock mode is not
            touched in this function and should be changed by the calling
            function, if necessary.

            If check_reader is a reader, it is not sent an invalidation
            message. A value ALREADY_A_READER is returned. But that node
            is still removed from the list of readers.
            If it is a multi server environment, and if even a single
            server is a reader, it sends an invalidation message to its
            server telling it to invalidate all server readers of this page.
            If a server is a reader, and that is specified as the
            check_reader, a message is sent to its server to invalidate all
            readers except the check reader. If check_reader is not a reader,
            and there are server readers, it sends a message to its server
            telling it to invalidate ALL the server readers.

            This function manages its atomic regions. Should be
            called in non atomic mode. Returns in non atomic mode.
  INPUTS  :
            page_no                 the page who's readers are to be
                                            invalidated
            check_reader            if check_reader is a reader, do not invalidate
                                            it, return ALREADY_A_READER
                                            if check_reader is not a reader, return
                                            NOT_A_READER
  OUTPUTS : ALREADY_A_READER/NOT_A_READER
*****************************************************************************/
int     invalidateAllReaders(int page_no, int check_reader)
{
        perClientDS     *thisClient;
        auxClient       *clientAuxTable;
        intNode         *next_reader;
        int             ret_val;
        Word            SEND_ARRAY[3];/* used to send to clients*/
        Word            SEND_ARRAY_SERVER[4];/*used to send to server*/
        int             next_reader_site;
        int             len,i,ret_value,server_reader;

        #ifdef MULTI_SERVER
        int             server_site;
        #endif

        ASSERT_ATOMIC();
        thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        for(i=0; i< 3; i++)
                SEND_ARRAY[i] = INIT_SEND_ARRAY;

        for(i=0; i< 4; i++)
                SEND_ARRAY_SERVER[i] = INIT_SEND_ARRAY;


        /* See migration notes and the comments below:
         * *9 c1 is a reader for an in cluster page. c1 is migrating so
         *    it invalidates the read page. s1 is supposed to inform the
         *    clock to remove c1 as a reader, but suppose, parallely, the
         *    clock sends a C_INVALIDATE_READER to c1. c1 should invalidate
         *    the reader only if it is still a reader.
         *    Though this check should be made only in multi-server case,
         *    it is safe to do it always.
         *    Though this function is called after checking if the
         *    clock is a reader, still make this check. It is possible
         *    that parallel to this client thread, a clientMigrationThread
```

```
         *    invalidated this clock and sent it to the server.
         */
        if ((clientAuxTable[page_no].page_status & READ) != READ)
        {

                ASSERT_ATOMIC();
                return NOT_A_READER;
        }

        /* Pull out each reader from the list of readers, and
         * send an invalidation message to it.
         * This clock site will be one of the readers. There is
         * no need to send an IPI to itself.
         */
        ret_value = NOT_A_READER;
        server_reader = FALSE;

        next_reader =
getNextNodeFromIntList(clientAuxTable[page_no].list_of_readers,&ret_val);

        SEND_ARRAY[0] = C_INVALIDATE_READER;
        SEND_ARRAY[1] = 1;
        SEND_ARRAY[2] = page_no;
        /* MESSAGE_LENGTH*/
        len = 3 * sizeof(Word);

        while ( next_reader != NULL)
        {
                /* Not in atomic region.*/
                /* Send the request to the reader of the page
                 * REQUEST_FOR_CLIENT_IPI.
                 * Arguments:
                 * 0. request_code              C_INVALIDATE_READER
                 * 1. no args                   1
                 * 2. page_no                   the read page that has to
                 *                                      be invalidated
                 */
                next_reader_site = next_reader->val;

                if( next_reader_site == check_reader)
                {
                        ret_value = ALREADY_A_READER;

                        /* If check reader is a server, server state is not set
                         * to true. So if check_reader is the only server reader,
                         * an 'invalidate server readers' message will not be
                         * sent to the server.
                         */
                }
                else
                {
                        if ( next_reader_site == CURR_PROCESSOR)
                        {
                                /* The clock mode also has to be turned off by
                                 * the calling function.
                                 * If it was an earlier trailer version for
                                 * that same page, do not remove it.
                                 */

                                clientAuxTable[page_no].page_status &= ~(READ);

                                /* Even though the clock can invalidate
                                 * itself right here, it sends an invalidation
                                 * message to itself. This is because, in
                                 * case a reply to a request made by the user
                                 * at this site is still pending, this
                                 * invalidation would have been done  without
                                 * the reply having used the page
                                 * Example:
                                 * Single server, c1,c2 in s1.
                                 * C1 is a write clock.
                                 * M1: c1 wants to read.
                                 * M2: c2 wants to read
                                 *
                                 * Say, M1 reaches s1 before M2.
                                 * M3: s1 gets M1, makes no changes to ds
```

```
                         *    since read request, forwards to c1.
                         * M4: c1 becomes reader clock.
                         * M5: c1 gets M2 from s1.
                         * M6: c1 inserts a reply to M1 into its
                         *    request queue. M6 must be after M4.
                         * M7: c1 wants to invalidate itself because
                         *     of the write request M2.
                         * Now, if M7 is done directly here, that is
                         * this site directly invalidates the read mode.
                         *
                         */

                }
                else
                {
                        /* If multi-server, the reader site
                         * might be a server. In that case
                         * send a message to the server.
                         * If single-server, it is safe
                         * to send a msg directly to the
                         * next reader.
                         */
                        #ifndef MULTI_SERVER

        send_ipiV(next_reader_site,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,3,SEND_ARRAY);
                        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

                        #else /* If multi server case*/
                        /* If the next_reader_site is a server,
                         * send an ipi to this cluster's server.
                         * Else, send an ipi directly to the
                         * reader (which shd be a client in this
                         * cluster).
                         */
                        if( isItServerCode(next_reader_site) == TRUE)
                        {
                                server_reader = TRUE;
                        }
                        else
                        {
        send_ipiV(next_reader_site,PRIORITY,REQUEST_FOR_CLIENT_IPI,len,3,SEND_ARRAY);
                        assert(SEND_ARRAY[1] != INIT_SEND_ARRAY);

                        }

                        #endif /* MULTI_SERVER*/
                }
        }

        free(next_reader);
        next_reader =
getNextNodeFromIntList(clientAuxTable[page_no].list_of_readers,&ret_val);

        }/*end of while*/

        /* If at least another server is a reader, send a message
         * to this clusters server telling it to invalidate
         * all server readers.
         */
        #ifdef MULTI_SERVER
        if( server_reader == TRUE)
        {
                server_site = thisClient->server_site;

                if( ( isItServerCode(check_reader) == TRUE) &&
                        (ret_value == ALREADY_A_READER))
                {
                        /* If the check_reader is a server site,
                         * the server is sent a message to invalidate all
                         * server readers except this check_reader_server.
                         * So, this message only has to be sent if
                         * there are other server readers apart from the
                         * check reader.
                         */
                        SEND_ARRAY_SERVER[0] =
S_INVALIDATE_ALL_BUT_ONE_READERS_AT_OTHER_SERVERS;
```

```
                SEND_ARRAY_SERVER[1] = 2;
                SEND_ARRAY_SERVER[2] = page_no;
                SEND_ARRAY_SERVER[3] = check_reader;/*server code of that server*/
                /* MESSAGE_LENGTH*/
                len = 4 * sizeof(Word);

    send_ipiV(server_site,PRIORITY,REQUEST_FOR_SERVER_IPI,len,4,SEND_ARRAY_SERVER);
                assert(SEND_ARRAY_SERVER[1] != INIT_SEND_ARRAY);
            }
            else
            { /*If check reader is not a server or ret_value is NOT_A_READER.*/
                SEND_ARRAY_SERVER[0] = S_INVALIDATE_ALL_READERS_AT_OTHER_SERVERS;
                SEND_ARRAY_SERVER[1] = 1;
                SEND_ARRAY_SERVER[2] = page_no;
                /* MESSAGE_LENGTH*/
                len = 3 * sizeof(Word);

    send_ipiV(server_site,PRIORITY,REQUEST_FOR_SERVER_IPI,len,3,SEND_ARRAY_SERVER);
                assert(SEND_ARRAY_SERVER[1] != INIT_SEND_ARRAY);
            }
    }
    #endif /* MULTI_SERVER*/

    ASSERT_ATOMIC();
    return ret_value;

}


/*******************************************************************
  FUNCTION: invalidateReaderOfPage
  PURPOSE : This function is called to invalidate a read page at
            a site. This page could have been a trailer before.
            That remains unchanged.
            This function manages its atomic regions. Should be
            called in non atomic mode. Returns in non atomic
            mode.
  INPUTS  : reqNode *request
            request->msg_code          C_INVALIDATE_READER
            request->no_values         1
            request->array:
            0. page_no                 the read page to be invalidated.
  OUTPUTS : none
*******************************************************************/
void    invalidateReaderOfPage(reqNode *request)
{
        perClientDS     *thisClient;
        auxClient       *clientAuxTable;
        int             page_no, size_msg, array_index;

        ASSERT_NOT_ATOMIC();
        thisClient = (perClientDS*) ( *(ptrToGlobalClientDS));
        clientAuxTable = thisClient->client_ptable;

        page_no = request->array[0];

        /* Update statistics at this client site.
         * This is a control message during normal operation.
         */
        CYCLE_COUNTING_OFF;
        begin_atomic();
                globalClientStats[CURR_PROCESSOR].normal control_msg_count++;
                size_msg = 3 * sizeof(Word);
                globalClientStats[CURR_PROCESSOR].normal.control_msg_bytes += size_msg;
        end_atomic();

        /* Update statistics for the time interval into which
         * this message falls.
         */
        array_index = CURR_TIME/METRIC_INTERVAL_CYCLES;

        if( MIG_INTERVAL)
        {
        assert(thisClient->cluster_index >=0 &&
                        thisClient->cluster_index < NO_OF_SERVERS);
        assert(array_index >=0 && array_index < MAX_MIG_INTERVALS);
```

```
        globalIntervalData[thisClient->cluster_index].in_cluster_control_byte[array_index]
+= size_msg;
        globalIntervalData[thisClient->cluster_index].in_cluster_control_msg[array_index]
+= 1;
        }

        CYCLE_COUNTING_ON;

        /* COMMENT 1 :
         * This message is sent by a client to another client.
         *
         * This message can arrive for a client that has just migrated.
         * 1. c1 is a client in cluster 1, to migrate to cluster 2.
         *    c1 is a reader of this page.
         * 2. c1 tells s1 to invalidate c1 as a reader for this page.
         *    c1 migrated to s2.
         * 3. The clock in cluster 1 has still not removed c1 as
         *    a reader for this page, but sends an invalidateReaderOf
         *    Page message to c1. By the time c1 gets this message,
         *    it is in its new cluster. If c1 has already become a
         *    reader(or temp read clock) in its new cluster, the page
         *    consistency check will fail.
         * Therefore proceed only if this page is an in-cluster page.
         * But this C_INVALIDATE_READER message is sent directly by
         * a client, so its possible that it is sent by a clock
         * in the old cluster after this client has migrated to its
         * new cluster!
         */
        begin_atomic();
        if( (page_no < thisClient->lowest_page ) ||
                (page_no > thisClient->highest_page)
      )
        {
                end_atomic();
                return;
        }
        end_atomic();

        /* See migration notes and the comments below:
         * *9 c1 is a reader for an in cluster page. c1 is migrating so
         *    it invalidates the read page. s1 is supposed to inform the
         *    clock to remove c1 as a reader, but suppose, parallely, the
         *    clock sends a C_INVALIDATE_READER to c1. c1 should invalidate
         *    the reader only if it is still a reader.
         *    Though this check should be made only in multi-server case,
         *    it is safe to do it always.
         */
        begin_atomic();
        if ((clientAuxTable[page_no].page_status & READ) != READ)
        {
                checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 1 invalidateReaderOfPage");

                end_atomic();
                return;
        }
        /* set READ mode off.
         */

        clientAuxTable[page_no].page_status &= ~(READ);
        /* If the page was a trailer before, it stays a trailer.
         * But if it was not a trailer before, make it PAGE_NOT_IN_MEMORY.
         */
        if(( clientAuxTable[page_no].page_status & TRAILER) != TRAILER)
        {
                clientAuxTable[page_no].page_status &= 0;
                clientAuxTable[page_no].page_status |= PAGE_NOT_IN_MEMORY;
                /* Since this site is not a trailer site for this page,
                 * invalidate the page value to keep the page consistent.
                 */
                clientAuxTable[page_no].page_value = -1;
        }

        checkSinglePage(clientAuxTable[page_no], page_no,
CURR_PROCESSOR,CHECK_TRAILER_VER," 2 invalidateReaderOfPage");
```

```
        end_atomic();

        /* The  request is no longer needed. Memory for it will
         * be freed in clientThread on return.
         */
}
```

2

VITA

Pallavi K. Ramam

Candidate for the Degree of

Master of Science

Thesis: A FAULT-TOLERANT COHERENCE PROTOCOL FOR DISTRIBUTED SHARED MEMORY SYSTEMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Hyderabad, Andhra Pradesh, India on September 13, 1970, daughter of K. Vijai Ramam and Vijaya L. Ramam.

Education: Graduated from Nasr School, Hyderabad, India in April 1986; received Bachelor of Science degree in Mathematics, Physics, and Computer Science from St. Francis College for Women, Hyderabad, India in April 1991; received Master of Science degree in Mathematics with a specialization in Computer Science from the Indian Institute of Technology, Bombay, India in July 1993; completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in May 1998.

Experience: Employed by OMC Computers Ltd., Hyderabad, India as a Graduate Engineer Trainee from September 1993 to August 1994; employed by Siemens Communication Software Ltd., Bangalore, India as a Software Engineer from November 1994 to July 1995; employed by Teubner and Associates, Inc., Stillwater, OK as a Software Developer from May 1996 to July 1996; employed by Oklahoma State University, Computer Science Department as a Graduate Teaching Assistant from August 1995 to May 1997.