Integrative Monitoring and Control Framework Based on Software Distributed Shared Memory Non-Locking Model

Mohamed Abdalla Khalil

A thesis submitted in partial fulfilment of the requirements of The Nottingham Trent University for a degree of Doctor of Philosophy

July 2004

Abstract

Distributed shared memory (DSM) paradigm provides an illusion of one physical shared memory in network of workstations where in reality shared data physically reside on different machines and different address spaces. DSM algorithms facilitate accessing the shared memory and exchanging data via normal read and write operations, concealing the inter-process communication and remote memory accesses. Such algorithms, and the environments they belong to, are used as processing platforms for distributed and parallel applications. This thesis investigates new approaches and algorithms for improving the performance of distributed shared memory systems. It assumes that the reduction of data retrieval time from the point of view of distributed applications is a major factor for measuring the performance of DSM systems.

The investigation introduces a framework that uses a non-locking approach and a purposely designed memory consistency model in order to achieve the above mentioned goal. This approach allows an application in a distributed environment to access the shared memory in a nearby location in a relatively short time thus saving valuable time for performing its native tasks. The framework is presented as a computing environment for building hierarchical traffic telematics distributed systems. It develops further the successful features of DIME DSM system (developed and designed at DOCM, NTU) and at the same time avoids its shortcomings. The main feature of the architectural design of the new framework is its flexibility, which allows the reconfiguration of the communication paths or routes of the system at run-time, thus improving the overall performance of the whole system.

To maintain consistent view of the distributed shared memory in the framework, a variant definition of sequential consistency (SC) model has been developed. This model has been designed specifically to support certain features in the urban traffic control (UTC) system. It also incorporates the flavour of SC definition that is intuitively favoured by distributed applications programmers. Also, to reliably manage the dissemination of messages and data across the distributed system, the thesis presents a novel proprietary communication protocol for the framework. This protocol sends

messages only to the applications that are involved in the operation rather than broadcasting the messages to every application in the distributed system. The algorithm of the protocol reduces the number of messages exchanged in the system, and therefore saves the resources of the network. Furthermore, a new novel heuristic algorithm is presented in this thesis, which allows system re-configurability at run-time and optimization of the performance of DSM systems.

The research work discusses some important issues for designing and building distributed systems. It also discusses new techniques that have been introduced in recent research papers for improving the performance of DSM systems. The presented implementation of the proposed framework demonstrates the applicability of the non-locking approach and the consistency model for building DSM systems.

Dedication

I would like to dedicate this work to my beloved mother, Soad, for loving, raising, and guiding me in every single step of my life, without her I might not be here writing these words.

I also dedicate my work to the souls of my father Abdalla, my brother Hisham, and to the one who will live in my heart and memory forever, Mozamil.

Acknowledgement

I would like to thank my first supervisor Dr. E Peytchev for guiding and supporting me during my PhD project. Without his valuable advices and discussions I would not be able to see an end to this long journey. I am also very grateful to my second supervisor Prof. A. Bargiela for offering his guidance and constructive criticism and ensuring that what I did was worthwhile.

Special thanks go to all my friends who offered an unlimited support, and encouraging me to keep the posture throughout my study, thank you folks. When it comes to my family, it would be unexplainable how it is difficult to tell how much I appreciate their support and love that motivates me to graduate. Thank you my sisters, Manasik, Manahil, Maysoun and Alsarour. Thank you Samia, Ali, Osman, Fawzya, Siham, and of course, the youngsters, Omar, and Osman.

I would like to acknowledge the support and help that was given by my colleagues in the Intelligent Simulation and Modelling research group, and also by the technicians and the staff in the department.

And last, but not least, thank you Waleed and Ibrahim for being such good friends.

Table of Contents

List of Figures

List of Tables

Chapter One

INTRODUCTION

The last decade witnessed dramatic advancements in computer networking technologies in terms of continually introducing new and powerful workstations capable of providing high performance computing platforms. The relatively small cost of such platforms compared to supercomputer clusters has shifted researcher's interest to networked workstations as a computing environment for parallel frameworks. These platforms are used in various applications, such as distributed databases and distributed web servers.

Although super machines are dedicated platforms characterised by their high performance, the price and utilizations of networked platforms can bridge this performance gap. Moreover, using network of workstations (NOW) as computing platform has an advantage of building much more robust fault tolerant systems. In other words, in the occurrence of machine failure, networked parallel system can continue running with no disturbance and transfer the failed task to be executed on another machine. Furthermore, parallel systems can employ more than one server avoiding bottleneck.

The thesis introduces a distributed shared memory framework that uses a partially-replicated non-locking approach to reduce data retrieval time. This method allows an application in a distributed environment to perform read/write operations on the shared memory in its proximity in a relatively short time allowing it more time for performing its native tasks. The structure of the new framework is flexible in such a way that the system can reconfigure its communication channels at run-time in order to improve and optimize the performance of the system.

1.1 Distributed Shared Memory Systems- General Overview

Building parallel and distributed applications on network of workstation (NOW) requires middleware of software that can efficiently manage exchanging messages and data between different applications running on different machines. Traditionally there are two paradigms in building such middleware in distributed systems – the message passing (MP) paradigm and the distributed shared memory (DSM) paradigm. The former was predominantly used for building distributed systems, in which the programmers have to be conscious of where the data is and how the processes communicate with each other, making it hard to construct distributed systems using this paradigm.

Therefore, researchers decided to find more convenient approach for building distributed systems which led to the introduction of the distributed shared memory paradigm. DSM algorithm provides an illusion of one logical shared memory on NOW. Unlike the MP paradigm, DSM algorithm facilitates accessing the shared memory and exchanging data via normal read and write operations, making life easier for programmers of parallel and distributed applications.

Research efforts in DSM paradigm have resulted in presenting number of different algorithms applying the concept of DSM abstraction [Argile A. et. al 1997, Amza C. et. al 1996, Li K. 1998]. The presented systems are built at different levels of implementations: software, hardware, or hybrid of both. Many of distributed systems are built as software DSM systems due to the fact that exchanging complex data structures between different processes is supported [Protic J. et. al 1996]. One important conclusion of the research in DSM algorithm is that building distributed systems on network of workstations with DSM algorithm is a viable alternative to the traditional message-passing paradigm. This thesis focuses on the DIME DSM system [Argile A. et. al 1997] as a case study of the research and the new framework, presented in this thesis, is a revised version of it, and it is called DIME-II system. DIME system was designed and implemented in the Department of Computing and Mathematics- the Nottingham Trent University.

1.2 Distributed Shared Memory Systems – Optimization Strategy

The increasing demands of distributed applications require sufficiently high-performance DSM algorithms. Another direction of research has been dedicated to investigating new approaches for improving the performance of distributed shared memory systems in order to narrow the performance gap between message-passing oriented systems and DSM systems, besides, satisfying the needs of distributed applications. This trend is launched, alongside developing new DSM algorithms, to investigate new techniques in improving and enhancing the performance of DSM algorithms. Such techniques can be called complementary techniques as they are used in conjunction with DSM algorithms. This direction of research has presented a wide spectrum of techniques to optimize the performance of distributed and networked systems at different levels of optimization.

Optimization levels range from client interface, through middleware and servers, to the communication infrastructure. In these techniques a variety of criteria are examined, including time, space and quality of service. Some research concentrated on developing mechanisms for maintaining consistent view of different replicas of the data throughout the networked system. This research reported that consistency models with more relaxed constraints may improve the performance [Tanenbaum et. al 2002]. Another research report [Amza C. et. al 1999] has introduced DSM algorithms that adopt more than one protocol that automatically adapt, at run-time, to the usage pattern of the shared data in the system. On the other hand, some distributed systems tend to use per-node multithreads to hide communication latencies [Mueller F. 1997]. Despite the inherent software and communication overhead, most of the techniques and strategies have exhibited significant success in scaling up and optimizing the performance of DSM systems.

1.3 Aims of the Project

Many different approaches and performance factors have been introduced and taken into account in past and recent research to measure the performance of DSM systems. For instance, Munin [Carter JB 1995] adopts multiple relaxed consistency

protocols in order to achieve good performance through reducing the number of messages exchanged in the network. On the other hand, TreadMarks [Amza C. et. al 1996] adopts the same means, but to speedup the distributed system as a whole. In our research, the major factor of measuring the performance is the reduction of data retrieval time from the perspective of user applications. The motive behind this factor is that the user application can have more time for performing its native tasks, which is often wasted in network communication.

The main objectives of this research are to:

- Investigate new means for building software DSM systems in which the time of data retrieval by user applications is reduced to the extreme, and
- To have a flexible design for the system, allowing start-up and run-time reconfiguration of the system connectivity when needed to optimize the performance.

With DSM algorithms, distributed applications often waste valuable time when retrieving data from the central shared memory, this time is spent by the middleware system during exchanging data and messages between different parts of the system to fetch the requested data. The framework presented in this thesis adopts a non-locking model to achieve the required enhanced performance. The algorithm adopted by the framework allows an application to retrieve the required data from a memory associated with that particular application. This intermediate memory contains copies of the data required by that application (not a whole replica of the main memory) and accessed only by that application.

The burden of making the intermediate memories consistent with the main memory is entirely left to the middleware system. Therefore, an application can retrieve the required data from its intermediate memory in a relatively short time. In other words, user applications will always find the required data without any significant delay, bearing in mind that the data is retrieved directly from the intermediate memory with no competition with other applications within the system. Providing an application with the requested data in a relatively short period of time is considered the one single most

important factor of measuring the performance of the system. The motive behind this assumption is that the user application can have more time for performing its native tasks, which time is very often wasted in network communication [Khalil M. et. al 2003b].

The utilization of this non-locking approach can allow distributed applications to perform read operations locally, resulting in the reduction of the number of exchanged messages in the system. This approach is expected to achieve the sought goal of improving the performance via reducing the time of data retrieval for user applications. This is based on the assumption that the speed of data processing is greater than the speed of exchanging data and messages over the network.

A relaxed model of sequential consistency (SC) has been designed to guarantee consistent view of the data in the framework. The reason behind choosing SC model among the others is that it is intuitively expected by programmers of distributed systems and the proposed for this research work requirements proved difficult or impossible to satisfy within other consistency models. At the same time, a lot of changes have to be defined in order to apply some improvement techniques such as multithreading and data replication approach, as SC model does not support such techniques. Moreover, as the DIME system is currently used as a computing platform for the Urban Traffic Control (UTC) distributed system, this new model supports data area and buffer structures which, naturally, exist in the UTC.

For reducing further the number of exchanged messages in the produced system, a communication protocol has been designed and implemented. This protocol is a middleware-level protocol which is used to exchange messages and commands within DIME-II DSM system, allowing user applications to perform read and write operations without being aware of the location of the data. This protocol is built on top of TCP/IP, and relies on the natural sequencing of the underlying network (Ethernet). The merit of this protocol is that it uses a multicast- based algorithm for propagating updates only to the applications that have replicas of the modified shared items. Thereby, the messages exchanged in the system are reduced and the network resources are saved.

On the other hand, optimizing the performance of DSM systems at run-time is also considered. The inherent software and communication overhead may cause poor performance in distributed shared memory system, in particular, when the demands of the distributed applications increase. This research investigates on developing algorithms that can scale up the performance of the system at run-time to adjust to the demands of the applications and the current state of the network. This thesis presents a strategy for optimizing the performance of DSM systems via the use of intermediate level of control. It also proposes a novel heuristic algorithm capable of reconfiguring DSM systems, at run-time, by adding up intermediate servers to support the main server supplying the service to the currently running applications.

More specifically, the algorithm optimizes the performance of DSM systems by improving the system connectivity via reconfiguring the communication paths of the system while preserving its backbone. This reconfiguration is based on the current state of the network which is evaluated by calcul+ating the round-trip times between different components of the system. Therefore, this algorithm considers the organization of the network in terms of its size, topology and the current workload, and accordingly changes the network connectivity of the DSM system to improve the performance and increase the data throughput. It has to be emphasised that, this algorithm aims, overall, at scaling up the overall performance of the system via maximizing the data retrieval rate at application level.

1.4 Overview of the Thesis

This thesis consists of six chapters as follows:

- Chapter one introduces the thesis by giving a brief introduction to the distributed shared memory paradigm and its rival, message-passing paradigm, and the trade-offs between them. Besides, it introduces the levels of implementation to the DSM paradigm in building distributed systems. It also sheds the light on one of the main challenges in prototyping DSM systems, which is performance optimization. This chapter identifies the main objectives of the research and the areas of research covered in this investigation.

➢ Chapter two presents a comprehensive literature survey about different issues in distributed shared memory systems. It starts by giving a general introduction to distributed systems, and then an extensive comparison between the traditional approaches of building middleware software for distributed systems, message-passing and distributed shared memory paradigms. It explores the design issues for building an efficient distributed system and some improvement techniques used to enhance the performance. Memory consistency design issue is thoroughly covered and a wide range of models is described. The implementation levels of the DSM paradigms in distributed systems are also elaborated in this chapter. The main contribution of this chapter is categorizing distributed shared memory systems into two big groups: locking and non-locking system according to the employed consistency model.

Chapter two elaborates comprehensively software distributed shared memory systems. It puts in plain words the reason of software DSM paradigms being a field of intensive research more than its competitors, hardware and hybrid levels of implementation. Some examples of DSM systems built at software level of implementation are elaborated in this chapter. The study describes the designing issues in each example, such as structure and granularity, consistency models, heterogeneity and scalability. The chapter wraps up the study by presenting some recent trends of the research in software DSM systems.

➢ Chapter three presents new framework that uses non-locking software DSM algorithm. It contains description for DIME system as a computing platform for the urban traffic control distributed system, which is called DIME-I in this thesis. A detailed description for DIME system as a user-level software DSM system is provided as well as identifying its limitations and justifies the need for improvement. Afterward, the new framework, DIME-II, is presented as a revised architecture of DIME-I architecture.

A model that ensures consistent view of the shared memory throughout the system is presented in the chapter. This model is designated to the new framework and it supports particular features in the urban traffic system. Also, it introduces a transmission protocol, called DIME-II Data Transfer Protocol (DDTP), for maintaining the underlying communication of the non-locking framework and its consistency model, and provides a complete description for the implementation of the produced framework. Finally, this chapter presents experiments on the new framework, discusses, and concludes the results.

➢ Chapter four takes us to the broad world of performance optimization by presenting some common strategies and techniques aim at scaling up the performance of DSM systems in the case of performance deterioration. These strategies are load balancing, load sharing and communication minimization. The main contribution of this chapter is the introduction of a novel heuristic algorithm for enhancing the performance of DSM systems. The strategy reconfigures the system at run-time by embedding intermediate servers ready to be initiated and activated to reduce the load on the main central server by redirecting some applications from the main server to take the service from the intermediate servers. Experimental results are also provided, discussed and concluded.

➢ Chapter five concludes the research and provides some directions to further research in distributed shared memory systems.

Chapter Two

DISTRIBUTED SHARED MEMORY SYSTEMS

2.1 Distributed Systems

Over the last decades Distributed System (DS) environments have attracted significant research interest. The aim was to investigate the applicability of such systems for building integrative frameworks. They enable computers to coordinate their activities and to share the resources of the system- hardware, software, and data. Users of a well-designed distributed system should perceive a single, integrated computing facility even though it may be implemented on many computers in different locations. In figure (2.1) distributed system software is represented as middleware service that links and coordinates the distributed activities running on different machines of different environments.

The development of distributed systems followed the emergence of high-speed local area networks at the beginning of the 1970s. More recently, the availability of high-performance personal computers, workstations and server computers has resulted in a major shift towards distributed systems and away from centralised and multi-user computers. Distributed system frameworks have been applied in a wide variety of applications, for commercial and academic purposes. Such system is characterised in [Tanenbaum A. et. al 2002] as we will see in the next section.

2.2 Distributed Systems Main Characteristics

There are several characteristics which determine the usefulness of the algorithm of a distributed system, and distinguish between the performances of number of

algorithms. Among these characteristics are: openness, concurrency, transparency, and resource sharing.

2.2.1 Openness

This characteristic means that distributed systems can be extended in various ways in terms of peripherals, memory, communication interface, operating system features, communication protocols and resource sharing services. Openness allows the addition and the removal of different kind of components easily without disturbing the whole running system.

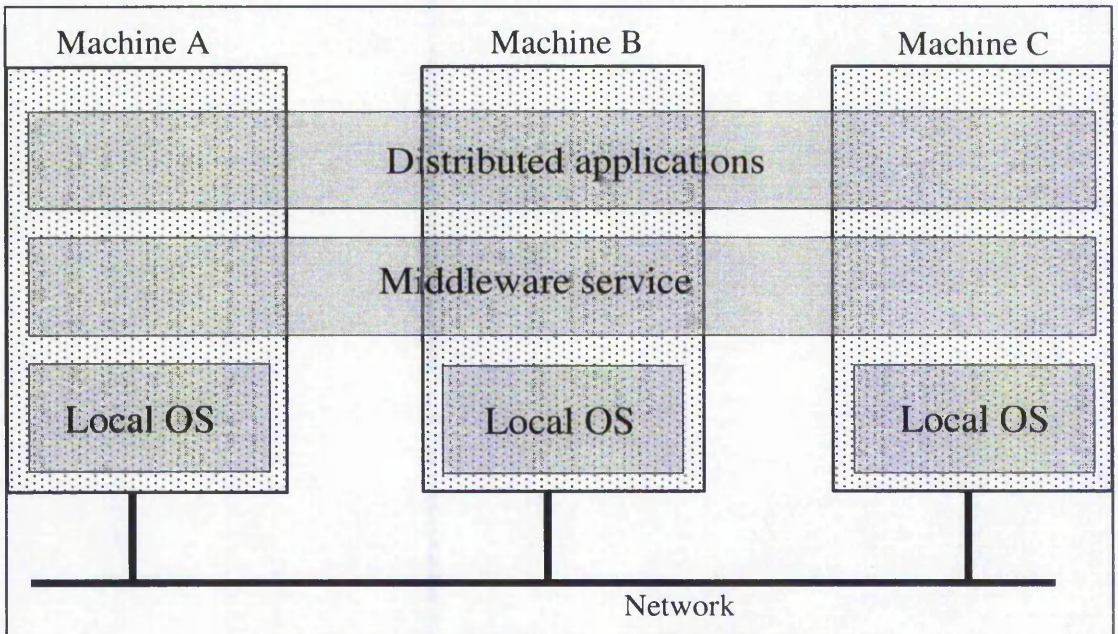


Figure 2.1: An Illustration of Distributed System

2.2.2 Concurrency

It is another influential feature of DS, as in such systems many processes can simultaneously run their tasks supporting parallel execution. This feature arises naturally in DS from the separate activities of applications, the independence of resources, and the locations of processes in separate machines, enabling these processes to run in parallel on different machines of different computing environments. Concurrent accesses and

updates in distributed system must be handled carefully in order to ensure that the benefits of concurrency are not lost. For example, concurrency means several updates to the same part of the shared memory can be issued at the same time, which means there should be some kind of control over the memory to determine which update to take place first. Therefore, the coherence of the system is maintained while having concurrent execution.

2.2.3 Transparency

Distributed systems are perceived as a whole rather than a set of independent components. Thus, system transparency addresses the needs of users and programmers to perceive a collection of networked computers as an integrated system, concealing the distributed nature of the resources used to perform the users' tasks. The separation of components is an inherent property of distributed systems. Its consequences include the need for communication and explicit system management and integration techniques. Separation of components allows the truly parallel execution of programs, and enables the containment of components faults and recovery from faults without disturbing the whole system.

2.2.4 Resource Sharing

Users of centralised and distributed computers are so accustomed to the benefits of resource sharing that they may easily overlook their significance. The term resource contains a wide range of hardware and software components such as printers, files, databases...etc. Sharing the hardware resources reduces the costs of buying new hardware and, on the other hand, data sharing is an essential requirement for many computer applications, such as commercial applications that enables users to share single data object in a single active database with no need to have the original database to be present on every machine where the manipulation may take place.

Sharing network resources can be obtained via message-passing or distributed shared memory models as we will see in the next section. In brief, it is the fundamental

characteristics in building distributed systems, and it strongly affects the software architecture of them.

2.3 Distributed Shared Memory Paradigm

In distributed system environment (often called loosely coupled multi-computers), processors in workstation clusters do not share global physical memory, and so all inter-process communication between processors must be performed by sending and receiving messages over the network. On the contrary, there is shared memory systems in which a common physical memory is accessible to all processors in the system. Such systems are called tightly coupled multiprocessors.

Formerly, Message-Passing paradigm was predominantly used for building distributed systems, in which primitives such as *Send* and *Receive* are used for maintaining communication between processes on different machines (figure 2.2).

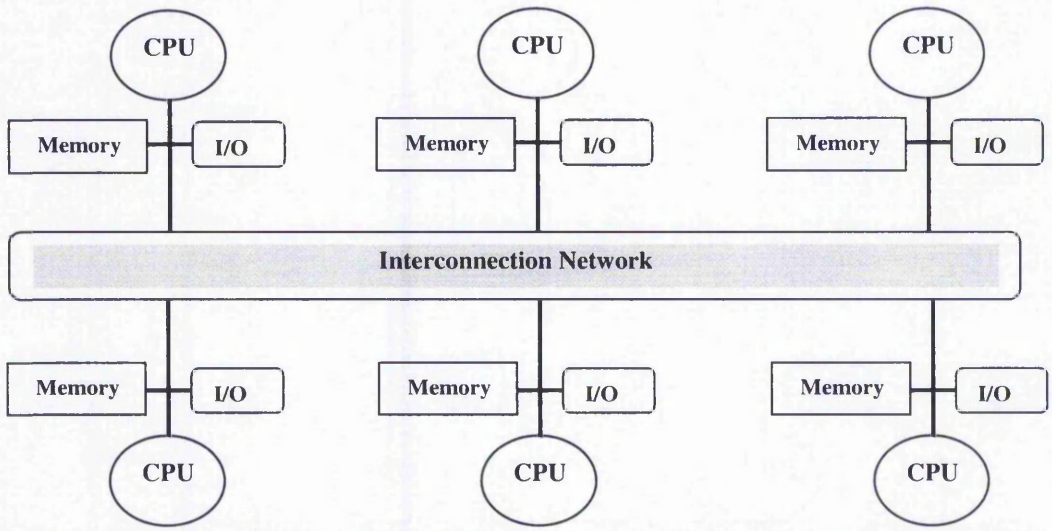


Figure 2.2: Distributed Memory Multiprocessors - The local memories are shared by explicitly passing messages over the network.

In such system, the inter-process communication is entirely programmers' responsibility, and therefore one must have a complete knowledge of the data usage pattern in the system which makes it hard to program using this model. Moreover, sending complex data structure using message-passing model involves considerable complexity in programming and substantial overhead in both space and time [Li. K 1988], which is the main drawback of this paradigm.

On the contrary, Distributed Shared Memory (DSM) paradigm logically implements the shared memory model in a physically distributed memory system (Figure 2.3). Thereby, applications can be written as if they were executing on a shared memory multiprocessor, accessing shared memory with ordinary read and write operations.

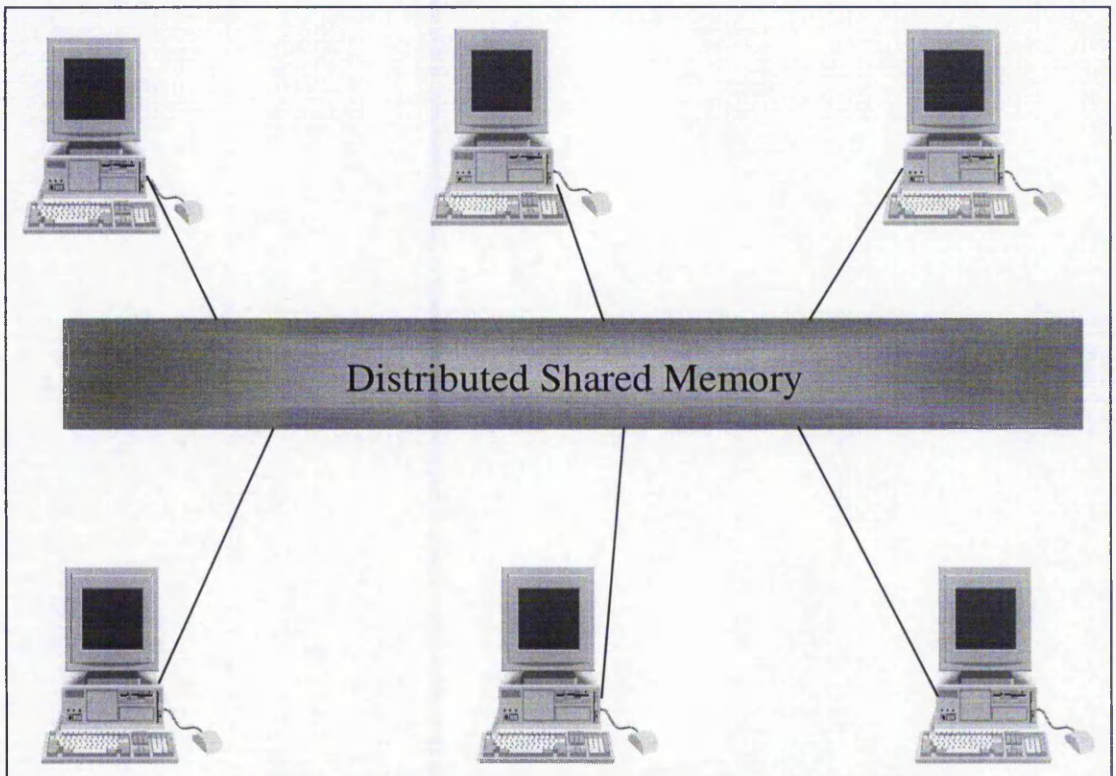


Figure 2.3: Distributed Shared Memory Abstraction (accessed via normal *Read* and *Write* operations)

Although DSM systems tend to generate more communication and then tend to be less efficient than message-passing systems, research has shown that DSM systems

are comparable with their analogous systems [Lu H. et. al 1995]; moreover, they can sometimes outperform message-passing models [Auld P. 2001].

In [Lu H. et. al 1995] experiments assessing the differences in programmability and performance between TreadMarks DSM [Amza C. et. al 1996] and PVM message-passing systems have shown that more messages and more data are sent in TreadMarks. This extra communication is due to the separation of synchronization and data transfer, besides, extra messages are sent to request updates for data in TreadMarks, therefore are responsible for lower performance of all the TreadMarks programs. However, in terms of programmability, it has been concluded that it is easier to program using TreadMarks DSM system. The ease of programmability was the major aim of such algorithms in the first place.

Another experiment presented in [Auld P. 2001] (concerning the operation of DSM systems and not the overall performance of the test programs) made a comparison between Broadcast DSM system and MPI message-passing system using straightforward parallel algorithms. The result has shown that for some applications the DSM paradigm of Broadcast DSM system outperforms the MPI message passing system.

Hence, to recall, due to the explicit use of *send/receive* primitives in message-passing paradigm, we find that programmers tend to use distributed shared memory paradigm, as it hides the remote communication mechanism from the application writer, so the ease of programming and the portability of the system are preserved [Protic J. et. al 1996]. The ability to provide a transparent interface and a convenient programming environment for distributed and parallel applications has made the DSM model the focus of numerous research efforts. One of the main objectives of the current research efforts in DSM systems is the development of new algorithms that reduces the overall delay of the shared data retrieval, while maintaining memory consistency in the whole system.

2.4 Issues in Designing DSM systems.

Many of Distributed shared memory algorithms have been successfully implemented in a wide range of experimental and commercial purposes. Building an

efficient, successful software DSM system depends enormously on the nature of applications that implement the algorithm. However, there are number of requirements or design issues that influence the performance and the efficiency of the system, as enumerated in [Nitzberg B. et. al 1991]. The degree of satisfying these issues varies from one application to another; therefore, considering the nature of an application in the designing stage can effectively increase the performance of that application.

2.4.1 Structure and Granularity

The structure and granularity of DSM system are closely related. Structure refers to the underlying representation of the shared data in the memory. This representation in most cases is a linear array of words, while other systems represent shared data items as object, language type, or even an associative memory (as in database systems). Granularity refers to the size of the unit of sharing (byte, word, page or complex data structure).

Some of DSM systems are implemented using the virtual memory hardware of the underlying architecture. In Ivy [Li. K. 1988] the memory is structured as 1-Kbyte pages. Mirage [Fleisch BD et. Al 1989] extended Ivy's single shared-memory space to support a paged segmentation mechanism. In this system, users share arbitrary-size region of memory (segments) while the system maintains the shared space in pages. In these implementation-determined systems, it is convenient to choose a multiple of the hardware page size provided in the Memory Management Unit (MMU) as the unit of sharing, in order to use its protection mechanism to detect inherent memory references and trap them to the appropriate fault handlers where coherence strategy is used to keep the memory coherent all the time.

DSM algorithms tend to provide each user application in the system with the required data in its local memory space in order to reduce contention in the system leading to better performance. With this locality of references, a process is likely to access large region of its shared address space in small amount of time. Theoretically, larger page sizes reduce paging overhead, but may ironically increase the likelihood of contention in the system. In other words, building systems with large-sized pages

increases the possibility of having more than one process competing in accessing the page. Sometimes two unrelated variables each used by different process are placed in one page. This situation is called false sharing. False sharing may cause a page to be sent back-and-forth between numbers of processes leading to network overhead. Using smaller sizes of page reduces the possibility of false sharing. Another factor affecting the choice of page size is the need to keep directory information about the pages in the system, as the smaller the size the larger the directory. Therefore, special care has to be taken into account when choosing the size of a page in order to reduce the likelihood of contention and false sharing while preserving the resources of the network by sending the minimal amount of messages.

On the other hand, the granularity of other DSM systems is determined by the application itself. One method of structuring the shared memory is by data type. With this method, shared memory is structured as objects in distributed object-oriented systems, as in CLOUDS system; or it is structured as variables in the source language, as in Munin [Carter JB 1995]. In such systems, granularity can vary to match the requirements of the applications. However, if different parts of objects are accessed by distinct processes, these systems can yet be liable to false sharing.

Another method is to structure the shared memory as database. In Linda [Rzeczkowski W et. al 1980], the shared memory is accessed as an associative memory called a tuple space. Although this model allows the separation of the location of data from its value, it requires the use of special access functions to interact with the shared memory. In most other systems, access to shared data is transparent.

2.4.2 Scalability

Theoretically, DSM systems scale better than tightly coupled shared memory multiprocessors, since it can be extended horizontally to contain new hardware and application software. However, the limits of scalability are reduced greatly by two factors: central bottlenecks (e.g. the bus of a tightly coupled shared memory multiprocessors) and a global common knowledge operations and storage (e.g. broadcast messages or full directories whose sizes are proportional to the number of nodes).

Most of DSM systems are implemented on top of Ethernet, which in itself is centralised bottleneck and can support only 100 nodes at a time. However, this limitation is most likely a result of these systems being research tools rather than an indication of any real design flaw.

2.4.3 Heterogeneity

Sharing memory between different environments can give a system more portability, although it seems impossible. In fact, different machines may use different underlying representations for basic data types (e.g. integer, floating-point numbers and so on). Structuring shared memory as variables or data objects in the source language makes the life much easier. Then a DSM compiler can add conversion routines to all accesses to shared memory. In Mermaid [Zhou S. et. al 1990], the memory is shared in pages, each contains only one type of data, and when a page is moved between two architecturally different systems, a conversion procedure is used to convert the page data to the appropriate format.

Although heterogeneous DSM systems allow more machines to participate in a computation, the overhead of conversion can outweigh the benefits.

2.4.4 Memory Consistency

Many DSM algorithms adopt data replication approach to enhance the reliability and improve the overall performance of distributed systems [Tanenbaum A. et. al 2002]. Enhancing the reliability can be obtained via the provision of several replicas of data. Usually, when one replica crashes, the system can simply handle this problem by switching to one of the other replicas. On the other hand, data replication for performance is crucial when the distributed system needs to scale in numbers and geographical area. In this case the performance can be improved by replicating the server and subsequently dividing the work over several servers.

Having multiple replicas may lead to consistency problem. Informally, whenever a replica is updated, that replica becomes different from the rest, and as a consequence, the update must be carried out on all replicas to ensure consistent view of every replica

in the entire system. Therefore, special attention has been paid to ensure consistency in such systems.

In general, a consistency model is essentially a contract between processes and data store, in other words, it says that if processes agree to obey certain rules, the store promises to work correctly. Normally, a process expects the last written value in return of a read operation; however, in the absence of a global clock, it is difficult to define precisely which write operation is the last one. Consequently, the need of alternative definitions for the meaning of last or most recently written value arose, leading to number of consistency models tackling this problem. Each consistency model provides number of rules or constraints to be imposed to maintain consistency. These constraints vary from a model to another in order to satisfy certain requirements, usually emerge from the nature of the application. The next sections navigate through number of definitions for consistency, viewing the gradual process of the development from most strict to more relaxed models in order to maintain the ease of programming, which was the major aim of DSM paradigm in the first place.

2.4.4.1 Strict Consistency

It is the most stringent consistency model, and can be defined by the following condition [Tanenbaum A. et. al 2002]:

> *Any read on a data item returns a value corresponding to the result of the most recent write on x.*

This definition is natural and intuitive, although it implicitly assumes the present of global timing that justifies the term *most recent.* This definition has already been applied in uniprocessor systems; therefore programmers of such systems are familiar with this definition. But matters are more complicated in a system where data are spread across multiple machines, and can be accessed by multiple processes at the same time. Apparently, strict definition relies on an absolute global time, but in essence, it is impossible in a distributed system to stamp each operation with a unique time

corresponds to actual global time. Hence, later research efforts have been dedicated for investigation on the applicability of models with less strict constraints.

2.4.4.2 Sequential and Linearizability Consistency

Sequential consistency model [Lamport L. 1979] is slightly weaker than the strict model, and its definition as presented by Lamport is:

> *The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

This definition allows interleaving of read and write operations of concurrent processes on different machines, but all processes see the same operations in the same order. For example, if a process P_1 changes the value of a shared item x to a, and after an arbitrary amount of time another process P_2 changes the value of x to b, however if the update b appears to take place before a for another process P_3, all other processes in the system must see the updates in the same sequence in order to be sequentially consistent.

On the other hand, Linearizability model [Herlihy M. et. al 1991] is stronger than the sequential counterpart, as it assumes to receive a timestamp using a globally available clock. This clock can be implemented in a distributed system by assuming processes use loosely synchronised clocks. Denoting $ts_{OP}(x)$ as the timestamp assigned to operation OP on shared item x, implementing linearizable consistent memory must obey the condition:

> *The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. In addition, if tsOP1(x) < tsOP2(y), then operation OP1(x) should precede OP2(y) in this sequence.*

According to this definition, Linearizability is essentially sequential model with an additional synchronised clock to stamp access operations to the shared space. This

feature makes implementing linearizable consistency much more expensive than sequential model [Attiya H. et. al 1994].

Although sequential consistency is a programmer-friendly model, it has a serious performance problem. Experimental result in [Lipton R. et. al 1988] proved that any attempt, in a sequentially consistent distributed shared memory system, to change the protocol improving read performance makes write performance worse, and vice versa. As a result, researchers have investigated more relaxing or weaker models.

2.4.4.3 Causal Consistency

Causal consistency is a weak version of sequential model [Hutto P. et. al 1996]. However, this model distinguishes between events that are potentially causally related and those that are not. In short, if event B is caused or influenced by an earlier event A, causality requires other processes see the result of the operations performed by the event A followed by the results of the operations of the event B. Formally, causality requires the following condition:

> *Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.*

For instance, if process P_1 writes a variable x, then P_2 reads x and writes y. In this case reading x and writing y are causally related, because the computation of y may have depended on the value of x, which read by P_2 after written by P_1. On the other hand, if two processes spontaneously and simultaneously write two different variables, these are not causally related, and said to be concurrent. Thus, this model permits that two processes may see concurrent operations in different orders, which might be seen as an inconsistent behavior and unacceptable by certain applications.

2.4.4.4 FIFO Consistency

In contrast to Causal model, FIFO consistency allows causally-related operations to appear in different orders by processes in distributed environment, giving more relaxation [Lipton R. et. al 1988]. Considering such relaxation, we have the condition:

> *Writes done by a single process are seen by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.*

FIFO model guarantees that write operations issued from single process are viewed by other processes within the system in the same order they were generated, while no guarantee that writes from different processes are viewed throughout the system in specific order. In contrast to the definition of causal model, if an event B is causally-related to a previous event A, the results of the event A are not necessarily seen everywhere before the results of the event B. Put in other words, with the definition of FIFO model, a process P_1 can see the results of A before the results of the event B, while another process P_2 may see them in the reverse order, which in not permitted in causal model. On the other hand, if a variable x is updated by a process P_1 with a value a, and subsequently another process P_2 updates the same variable with a value b. Unlike sequential model, the FIFO model allows a situation where a process P_3 may see the update a followed by b, while they may be visible to another process P_4 as b and then a. The key difference between FIFO and SC model is that with the SC, although the order of operation execution is nondeterministic, at least all processes agree what it is. This model is also called Pipelined RAM (PRAM), because writes by a single process can be pipelined as the process does not have to halt waiting for each write to complete before starting the next one.

2.4.4.5 Weak Consistency

Although many relaxed models have been presented, still there are a lot of restrictive features that are not required in some applications. These restrictions can be represented by the fact that the presented models require that all writes in a single

process be seen everywhere in some order. However, many applications do not need to see all writes, let alone seeing them in order. Specifically, when a process inside a critical region updating shared items, other processes are not allowed to even touch the items until that process leaves the region. In this case the processes within the system only see the last update made by that process, without being conscious of the likelihood of other updates that might have happened before the available last one in the course of critical section execution. However, before the introduction of synchronization mechanisms, the system has no way of perceiving that there is a process in a critical region, therefore number of updates are unnecessarily propagated within the system, wasting valuable processor time and network communication in nothing.

Synchronization mechanisms provide alternative solutions through their synchronization variables. A synchronization variable S can be associated with the shared data. This variable can only be held by one process at a time. An exclusive access for that variable guarantees that no other process can perform any kind of operations in the course of the execution. Therefore, the system can only propagate the update after the process leaves the critical region, reducing numerous overhead in the system in terms of processor time, and communications. Weak model [Dubois M. et. al 1988] uses such mechanism to maintain consistency in DSM environments. This model has the following properties:

- *Accesses to synchronization variables associated with a data store are sequentially consistent.*
- *No operation on a synchronization variable is allowed to be performed until all previous writes have completed everywhere.*
- *No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.*

The first property says that all processes in the system observe all operations on synchronization variables in the same order. Informally, if a process P_1 requests synchronised access to certain shared area and at the same time another process P_2

makes the same request, all processes in the system see the request of P_1 before of P_2, or vice versa.

The second one guarantees that no read or write operation can be performed until all previous writes have completed everywhere, and the updates are reflected in all local replicas throughout the system.

The last point says that having an exclusive access over shared item means that all previous write operations have been successfully performed and propagated. In other words, an operation is performed inside a critical region only when all the results of previous operations of the same region have been seen throughout the system. That is, write or read operations always performed on the most recent value.

Since this model enforces consistency on a group of operations rather on individuals reads and writes; unlike the previous models, it is most useful in applications where isolated accesses to shared data are rare. Furthermore, this model could be seen as sequential model where consistency is enforced between groups of operations instead of between individual operations.

On the other hand, Weak consistency has a disadvantage of that the system can not distinguish between entering a critical region and leaving it when an access to the synchronization variable is requested. Upon requesting the variable the system performs actions required in both cases. It ensures that all previous writes have been completed, as well as gathering all updates from other processes to propagate them, putting an additional burden on the system. Such deficiency has been overcome in later model called Release consistency.

2.4.4.6 Release Consistency

Release model [Gharachorloo K. et. al 1990] makes distinction between entering critical region and leaving it providing two kinds of variables: *acquire* and *release*. Those two operations do not have to be applied to all shared data in the system; rather they may guard only specific shared items, in which case only those items are kept

inconsistent [Tanenbaum A. et. al 2002]. In general, the following conditions apply this model:

- *Before a read or write operation on shared data is performed, all previous acquires done by the process must have completed successfully.*
- *Before a release is allowed to be performed, all previous reads and writes done by the process must have been completed.*
- *Accesses to synchronization variables are FIFO consistent (sequential consistency is not required).*

Enforcing these conditions guarantees that when a process does an *acquire*, all local copies of the protected data are brought up to date. A process can not perform *acquire* operation on a shared item unless all replicas of that item have been updated. On the other hand, upon *release* operation, all protected data have been updated are propagated out to other local copies within the system. However, there is no guarantee that other non protected data are consistent. In the light of these facts, release model does not affect non protected data.

It is also possible to use *barriers* instead of critical regions with release model. A barrier is a synchronization mechanism that prevents any process from starting phase *n+1* of a program until all processes have accomplished phase *n*. Therefore, when a process reaches a barrier it must wait until all other processes arrive at the same point, at that time all shared data are synchronised and all processes are resumed. In other words, departure from barrier is done on an *acquire*, whereas performing a *release* operation indicates arrival of all processes at barrier. That is, all updates are propagated to all processes that have replicas of the updated item only on an arrival event.

2.4.4.7 Lazy Release Consistency

With release consistency model, all updates are carried out to all processes that have local replicas of the protected shared data upon release operation, even though not all of the processes might need these updates. Therefore, to overcome such inefficiency Lazy variant of release model pushes out the updates only when an acquire operation is

performed [Keleher P. et al 1995]. In other words, when an acquire operation is executed by a process, that process gets the most recent values of the data.

The difference in performance can appear if a critical region is located inside a loop. With release model (sometime called eager release consistency to be distinguished from lazy variant [Tanenbaum A. et. al 2002]), in every iteration a release is done, and all the updates have to be carried out to all processes, resulting in wasting bandwidth and needless delay. On the contrary, with lazy model, performing release operation does not end up with propagating out any update done. Whereas, when acquire is performed, the updates are brought only to the process that performs the acquisition. Consequently, there is no network traffic is generated unless there is an execution of acquire operation.

To sum up, since it has been proved difficult to design an efficient distributed system with sequential model, many research efforts have been dedicated to introduce alternative models to maintain consistent view of the shared memory in accordance with the nature of the distributed and parallel applications, as well as preserving the ease of use and implementation. All these models differ in many aspects, such as degree of restriction, ease of implementation and programming, and performance. However, before the use of synchronization mechanisms, the introduced models were accompanied with some consistency behaviors that are theoretically not acceptable to the majority of distributed applications. Such behavior is represented by the fact that there is no guarantee about when the updates made on the shared memory will be visible to all other processes in the system. Therefore, further research papers have introduced consistency models with an explicit use of synchronization mechanisms as in the last three models, in which a process has to have an exclusive access to the shared memory before performing any read or write operation. Although these models vary in when the updates are propagated throughout the system, they allow programmers to pretend that a shared memory is sequentially consistent, when in fact; it is not.

2.5 Locking and Non-locking DSM Algorithms

From the previous coverage of memory consistency models it can easily be pointed out that some of the models explicitly use synchronization mechanisms to

maintain an exclusive access over the shared memory, while such mechanisms are not employed in the rest. In the DSM algorithms that use the former models, a process performs a special function to obtain an exclusive access to the shared memory prior to any read or write operation, while another function is used to relinquish the memory after accomplishing the current operation.

Therefore, and according to the employed consistency model, DSM algorithms can be divided into two big categories. The first one is locking DSM algorithms that explicitly use synchronization mechanisms such as Locks and Barriers, as in Weak, Lazy Release, and Eager Release. On the other hand, models such as Sequential consistency, Linearizability, and Causal consistency, do not use synchronization mechanisms to control the shared memory; therefore, can be categorised as non-locking DSM algorithm.

2.6 Implementation Levels of DSM Algorithms

Since the introduction of the DSM mechanism, several DSM algorithms have been developed and implemented at different levels: software, hardware and hybrid levels of implementation. The choice of implementation level usually depends on price/performance trade-offs. Although typically superior in performance, hardware implementations require additional complexity, allowable only in high performance large-scale machines. On the other hand, low-end systems, such as networks of workstations, yet do not allow cost of additional hardware for DSM, and are limited to software implementation. For the class of mid-range systems, such as clusters of workstations, low-cost additional hardware, typically used in hybrid solutions, seems to be suitable [Protic J. et. al 1996]. This section concentrates on the hardware and hybrid levels of implementation as the software level implementation is covered more thoroughly later in this chapter.

2.6.1 Software-oriented DSM Systems

It started with the idea of hiding the explicit passing of messages within the system and providing an abstraction of one single non-physical shared memory on

networked machines. Some of the software-oriented mechanisms rely on specific runtime libraries that are to be linked to the applications that use the shared memory. Other systems implement DSM abstraction on the level of programming language, since the compiler can detect accesses to the shared memory and insert calls to the synchronization and coherence routines into the executable code. On the other hand, the DSM mechanism can be incorporated into the distributed operating system, inside or outside the kernel. Operating system and run time library-oriented approaches often integrate the DSM mechanism with an existing virtual memory management system. However, some of the DSM systems usually combine elements of different software-oriented approaches. Software-oriented mechanisms are used in building number of DSM systems such as DIME [Argile A. et. al 1996], TreadMarks [Amza C. et. al 1996] and Brazos [Speight E. et. al 1997]].

2.6.2 Hardware-oriented DSM Systems

Such systems utilize dedicated hardware devices responsible for locating, copying shared data items, and keeping their coherence. The main objective is to implement the abstraction of DSM and its concepts in hardware device, which can be added to the system to provide distributed environment. They can be classified into three groups according to their memory architecture:

- Cache Coherent Non-Uniform Memory Access (CC-NUMA) systems. The address space is statically distributed across the local memory of clusters, and can be accessed by the local processor and processors from other clusters in the system with different access latencies. Memnet [Delp G., 1988] and Dash [Lenoski D. et. al 1992] use this mechanism.

- Cache-Only Memory Architecture (COMA) systems. The data is dynamically partitioned in the form of distributed memories, organised as large second-level caches. In other words, local memories are used as huge caches for data blocks from virtual memory address space, and data can be replicated and migrated across local memories on demand. KSR1 [Frank S. et. al 1993] is an example of such systems.

- Reflective Memory Systems (RMS). A hardware-implemented mechanism is employed to propagate updates as they are performed to all sharing sites using broadcast or multicast algorithm. Merlin [Wittie L. et. al 1989] is one example of these systems.

The prescribed mechanisms ensure automatic and transparent replication of shared data in local memories and processor caches, and they efficiently support small, fine-grain physical unit of replication. Further techniques have been introduced and applied in order to increase the performance of hardware-oriented DSM systems, in terms of reducing network latency, maintaining coherent cache and so on.

Maintaining finer granularity in these mechanisms minimizes negative effects of false sharing and thrashing, which results in a significant reduction in the communication overhead in DSM systems that use such mechanisms. Furthermore, searching and directory functions implemented at hardware level are faster compared to their software-oriented counter-parts. However, advanced techniques used for maintaining memory coherence and reducing communication latency in hardware-oriented DSM systems usually make the design complex and difficult to verify. Therefore, we find that hardware DSM mechanisms are often used in high-end machines where performance is more important than cost.

2.6.3 Hybrid Level DSM Systems

Since the evolution of the DSM abstract, numerous implementations at hardware and software levels have been presented. However, even entirely hardware DSM approaches use software-controlled features explicitly visible to the programmer for the purpose of memory reference optimization. On the other hand, many purely software solutions require some hardware support, such as virtual memory management hardware and error correction code.

Therefore, since either approach has its disadvantages, some approaches have been introduced as an attempt to combine software and hardware features in order to balance the cost/performance trade-offs. A solution implemented in some industrial

computers provided a DSM-like paradigm in software executed by a microcontroller located on a separate communication board [Protic J. et. al 1993].

Some solutions are typically to achieve replication and migration of data from a shared virtual address space across the clusters in software, while their coherence is managed at the hardware level, as in PLUS [Bisiani R. et. al 1990].

2.7 Software DSM Systems

As mentioned earlier, the DSM abstraction can be applied at different level of implementations. This research chooses to use the software level of implementation in building the integrative framework for number of reasons and based on previous research reports.

In [Cox A. et. al 1994] experiments were conducted comparing the performance of software implementation of DSM paradigm; TreadMarks, on a general-purpose network of uniprocessor nodes to hardware-oriented DSM; SGI 4D/480, on a dedicated interconnect multiprocessors. An important feature of this comparison was the similarity between the two platforms in processor, cache, compiler, and parallel programming interface, with only one difference of the level of DSM implementation. The results showed that the two implementations perform comparably for the applications with moderate synchronization and communication demands. When these demands increase, the communication latency and the software overhead of TreadMarks causes it to fall off in performance. On the other hand, since it provides a processor with a private path to memory, networked workstations of software DSM implementation scales well with applications of high memory bandwidth requirements.

Therefore, although in most cases it can not compete with hardware mechanism in terms of performance, software mechanism has an advantage over hardware algorithm in many aspects. Software DSM algorithm gives a viable solution for building a computing environment for parallel and distributed applications on the commodity workstations; which still do not tolerate cost of additional hardware. Also, software support for DSM is generally more flexible and easy to program than hardware-oriented

counterparts, and enables better tailoring of the consistency mechanisms to the data usage patterns and the application behaviour.

2.7.1 Software DSM System Examples

Since research and experiments of the DSM approach can extensively rely on the available programming languages and operating systems on the networked workstations [Protic J. et. al 1996], number of implementations of software DSM systems has been introduced and some of them are discussed in the next subsections, featuring the development stages of the approach over the years and some techniques that have been developed and introduced to enhance the performance of the produced systems.

2.7.1.1 IVY

IVY [Li. K. 1988] is one of the first proposed software DSM solutions. It was implemented as a set of user-level modules built on top of Aegis operating system on the Apollo Domain workstations. However, some changes in the underlying operating system were imposed in order to support remote operation and memory mapping modules.

IVY system consists of 5 modules: remote operation, memory mapping, process management, memory allocation, and initialization. The top three modules in figure 2.4 comprise the IVY client interface, each module consists of a set of primitives that can be used by application programs.

The granularity of the system is 1-Kbyte page, which is consistent with the one provided by the Memory Manager Unit (MMU). This consistency is important in such a way that the protection mechanism of the MMU can be used to detect incoherent memory references and trap them to the appropriate fault handler that keeps the memory address space coherent all the time. To keep the shared memory address space consistent an invalidation approach is used. That is, all read-only copies of a page are invalidated before a processor writes to the page.

To maintain sequential consistent memory space in IVY system, three algorithms were used: the improved centralised manager; the fixed distributed manager, and the dynamic distributed manager. The centralised manager resides on a single processor and keeps information of all ownership of the pages. When having a page fault, the manager will be asked by the faulting processor for a copy of the page. The manager will then request a copy of the page from the owner of the page to be forwarded to the calling processor. To avoid the inherent bottleneck of the approach, a fixed distributed manager algorithm was used. Each processor has a manager responsible for the predetermined set of pages evenly given by the algorithm.

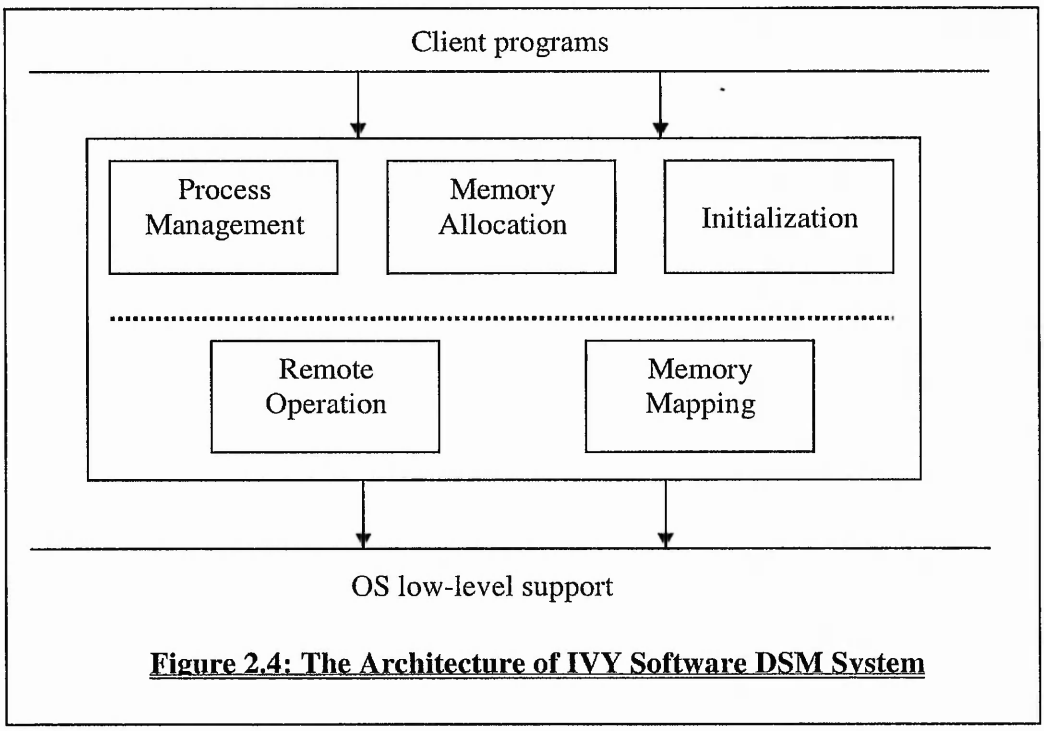


Figure 2.4: The Architecture of IVY Software DSM System

When a page fault occurs the faulting processor asks the true owner of the page for a copy, and after receiving the copy it proceeds in a centralised manner. The third approach was the dynamic distributed algorithm that keeps track of the ownership of all the pages in the system, using a field called *probOwner* (probable owner) in each page entry. The value of this field can either be the true owner or the probable owner. Initially, its value can be set to some default processor that can be considered the initial

owner of the page. As the system runs, each processor uses the *probOwner* field to keep track of the last change of the ownership of a page. This field is updated whenever a processor receives an invalidation request, relinquishes ownership of the page, or forwards a page fault request.

On the other hand, for memory allocation, a first fit algorithm with one-level centralised control was utilised. The processor that is contacted directly by the user is appointed to be the centralised memory manager. To reduce the contention on the memory, the memory allocator allocates each piece of memory to the boundary of a page. IVY made use of some synchronization mechanism (i.e. lock) for allocating memory page. At the beginning of each memory management primitive, a *test-and-set* operation is performed on the lock. A failed process will be sent back into a queue and will be awakened by an unlock operation on the lock which is done at every end of each primitive.

Experimental results showed that parallel programs using such a not well-tuned, user-mode system yields almost linear and occasionally super-linear speedups over a uniprocessor. Hence, the important contribution presented by this system was its proof of the viability of DSM models as an efficient alternative solution for parallel applications on distributed environment via overcoming the drawbacks of message-passing models.

2.7.1.2 TreadMarks

TreadMarks is a user-level software DSM system where messages and data traffic are reduced by relaxing consistency semantics of the shared memory [Amza C. et. al 1996]. TreadMarks implementation of the DSM abstraction relies on UNIX standard libraries in order to accomplish remote process communication, and memory management, therefore no need to make modifications in the operating system kernel. It runs on SPARC, DECStation, DECAlpha, IBM RS-6000, and on Ethernet and ATM networks. The intermachine communication is implemented using the Berkeley sockets interface. TreadMarks uses either UDP/IP or AAL3/4 as the message transport protocol

depending on the underlying network hardware (e.g. Ethernet or ATM). By default, the UDP/IP is used unless the machines are connected by an ATM LAN.

Since neither UDP/IP nor ALL3/4 guarantees reliable delivery, the system uses light-weight, operation-specific, user-level protocols to ensure message arrival, and every message sent is either a request message or a response message. Request messages are sent by TreadMarks as a result of an explicit call to a TreadMarks library routines or a page fault. Once a machine has sent a request, it blocks until a request message or the expected response arrives. The original request is retransmitted after a certain timeout without receiving response. To minimize latency in handling incoming requests, a *SIGIO* signal handler is used, that is, when a request message arrives at any socket a *SIGIO* signal is generated to interrupt the system in order to process the request message. Afterward, the handler performs a *select* system call to determine which socket holds the incoming request message. When a handler receives the request message, it performs the specified operation, sends the response message, and returns to the interrupted process.

TreadMarks provides shared memory as linear array of bytes (i.e. 4 Kbytes each). It also provides facilities for creating, destructing, synchronizing processes, and allocating shared memory. *Tmk_malloc()* function is used to create shared memory, whereas allocating memory using *malloc()* function means that the memory is private to the creating process. To maintain consistency in the shared memory, Lazy release consistency model is applied. The system makes use of a multiple writer protocol to reduce the amount of communication involved in implementing the shared memory abstraction. The virtual memory hardware is used to detect accesses to shared memory. TreadMarks provides primitives for applications for synchronization to avoid data races, which occurs when two processes simultaneously access the same shared item and at least one of them is writer. Data races often lead to bugs in the system, since the final outcome of the execution is timing-dependent. Application processes synchronization via two primitives: barriers and exclusive locks.

With this model of lazy release consistency, a process is allowed to buffer multiple writes to shared data in its local memory until a synchronization point is

reached (i.e. barrier). Locks are used to control access to critical regions. *Tmk_lock_acquire()* function is modelled as an acquire, whereas *Tmk_lock_release()* represents release operation. The procedure *Tmk_barrier()* pauses the calling process until all processes in the system have arrived at the same barrier. In other words, it is modelled as a *release* followed by an *acquire*, where each processor performs a *release* at barrier arrival, and an *acquire* at barrier departure. To implement this consistency protocol, the system call *mprotect* is used to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a *SIGSEGV* signal. The signal handler examines local data structure to determine the page's state, and examines the exception stack to determine whether the reference is a read or a write. If the local page is invalid, the handler executes a routine to obtain the essential updates to shared memory from the minimal set of remote machines. If the reference is read, the page protection is set to read-only. Whereas, for a write operation the handler allocates a page from the pool of free pages and performs a *bcopy* to create a *twin* to the page. This twin can be used later to determine the differences between the original page and its twin where the update is done. The same action is taken in response to a fault resulting from a write to a page in read-only mode, and the handler upgrades the access rights to the original page and returns.

The use of multiple-writer protocol allows two or more processors to simultaneously update their local copy of a shared page. Due to the use of lazy release consistency, the propagation of these updates is delayed until the time of an *acquire*. Furthermore, the releaser notifies the acquirer, of which pages have been modified, causing the acquirer to invalidate its local copies of these pages. A processor incurs a page fault on the first access to an invalidated page, and gets only the *diffs* (the differences between the main page and its twin) for that page from the previous releasers. The major benefit of using *diffs* is that they can be used to implement multiple-writer protocols, thereby, reducing the effects of false sharing; additionally, the overall bandwidth requirements can be reduced, since *diffs* are typically much smaller than a whole page. However, there is still a question of what happens if two processes modify overlapping portions of a page without using synchronization. Unfortunately,

TreadMarks' implementation does not check whether a page is modified locally when a global update arrives.

In [Lu H. et al, 1995] experimental results have shown that the separation of synchronization and data transfer and the request-response nature of data communication are responsible for lower performance comparing with a PVM message-passing model.

2.7.1.3 Broadcast Distributed Shared Memory System (BDSM)

This system is designed for common Ethernetworking environment. The use of Ethernet allows the exploitation of hardware broadcast and having a controlled message passing background [Auld P. 2001]. In this system, each user application is run solely on a workstation, and has an associated DSM sub-system that manages the shared memory. The shared memory in each workstation is a complete mirror of the shared memory on each processor (figure 2.5). Writes to memory modify the local copy and arrange to broadcast the updated values to all the other processes.

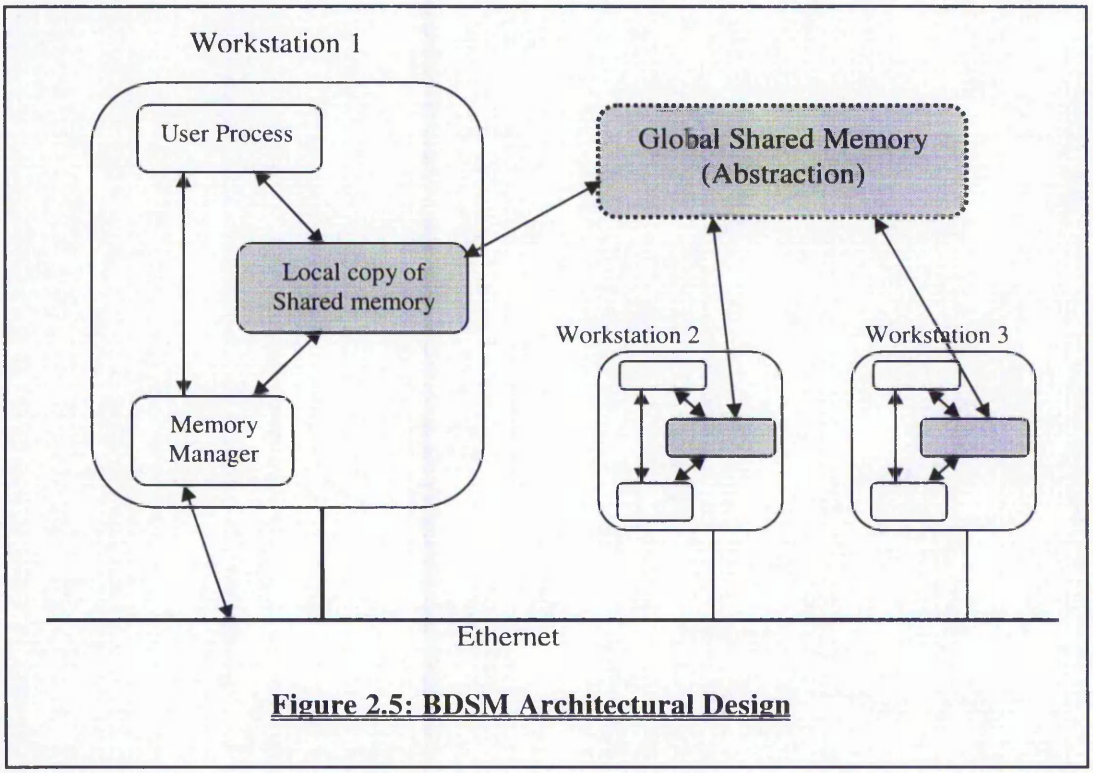


Figure 2.5: BDSM Architectural Design

The memory in BDSM system is maintained as a contiguous collection of discrete locations, where reads and writes operate at this level of granularity. The size of a location is defined by the programmer. The memory manager uses hardware broadcast to propagate updates to all other processes. Updates may be buffered locally to reduce the number of messages sent.

Since hardware broadcast on a single Ethernet segment is subject to message loss, the Pipelined Broadcast Protocol (PBP) library [Auld P. 2001] is used as a communication layer for the system to overcome the problem. The PBP ensures reliable partially ordered delivery of broadcast UDP messages, which are delivered to all receivers in the order sent by the broadcasting process. Incoming updates are applied immediately to the local copy of memory. However, there is no order defined between messages sent by different processes. In effect, a collection of FIFO pipelines is established amongst processes. The PBP library handles group registration and low-level communication. The ordering guarantees made by the PBP provide the basis for the coherence model of the update-based BDSM system.

Therefore, BDSM can be defined as broadcast-based, fully replicated user-level software distributed shared memory system. For maintaining consistency in the shared memory, two forms of process synchronization are employed: global barriers and individual locks. These two schemes of synchronization are implemented in the message exchange protocols, rather than in the context of the system itself. Before any synchronization message is sent, all buffered writes are sent. A barrier is an all-to-all message exchange. Each barrier requires n broadcast messages - where n is the number of local memories. In order for a process to cross a barrier it must receive a corresponding barrier message from each of the other processes. This message is received in FIFO order relative to other messages from the sending process. Once all $n-1$ corresponding barrier messages are received, the waiting process can cross the barrier. The shared memory is coherent at this point as long as there were no data races in the barrier interval. In order to make similar consistency guarantee about locks, a broadcast lock is used. This type of lock requires a process to receive a message from each other before acquiring the lock. Although the only coherence guarantee to programs that are

not properly synchronised is the FIFO ordering, this allows true multi-threaded applications.

Obviously, this system uses consistency model which is somewhat similar to the definition of the release consistency model, however, there is no guarantee that running processes see the same view of the memory. In other words, modifications issued by different processes are not ordered; hence, processes can see different views of the shared memory. Therefore, even after a barrier, it is possible for two processes to have different views of memory. More specifically, a program that allows unsynchronised access to the same locations may not have consistent views of memory across its sub-processes.

To evaluate the performance of the system in comparison with MPI system, an experiment was set using parallel applications for numerical calculations, where pure performance gains were desired. The concern of the experiment was the operations of the DSM system and not the overall performance. For some applications, the BDSM outperformed the common message-passing alternative, MPI. Also, the broadcast update protocol showed effective performance for some situations, and can be viable alternative to page-based DSM or message-passing systems.

2.7.1.4 Brazos

It is presented as a third generation of software DSM systems that use multithreading and apply relaxed consistency models for the shared memory [Speight E. et al, 1997]. It is designed to execute on x86 multiprocessor workstations running Windows NT 4.0, mainly due to the fact that Windows NT has native multithreading support built into the operating system. This operating system provides support for multiple lightweight threads executing within the same process address space. Additionally important, Windows NT implements TCP/IP through the WinSock user-level library. The Win32 API provides a rich set of calls to address threading issues, including support for thread priority manipulation, synchronization, thread context manipulation, and thread suspension and resumption. Processes in Brazos use functions in the WinSock Programming API to communicate with each others within the system.

The Brazos user-level library is statically linked with user application at compile-time, and it provides the interface between user application code and DSM code. The Brazos API includes synchronization primitives in the form of locks and barriers, which can be used to provide synchronization between threads within the same process. Besides, it provides routines for error reporting, statistics gathering, and data output. Brazos includes *service* thread that listens for incoming DSM session requests, authenticates encrypted passwords that users must have prior running a Brazos session. The service thread manages current DSM sessions running on the local machine, and provides a mechanism for the owner to remotely kill a runaway DSM application.

Brazos makes use of multithreading to overlap communication with computation per node. The system utilizes multithreading at both the user level and the system level. Multiple user-level threads allow applications to take advantage of symmetric multiprocessor servers by using all available processors for computation. In the runtime system there are two main threads. One thread is responsible for quickly responding to asynchronous requests data from other processes and runs at the highest possible priority. The other thread handles replies to requests previously sent by the process. This multithreaded aspect of Brazos allows greater amount of computation to communication overlap. The use of separate thread to handle incoming replies allows the system to maintain multiple simultaneous outstanding network requests, which can significantly improve performance. Additionally important, the exploitation of multithreaded DSM algorithms proved significant in hiding the communication latencies [Muller F., 1997].

For maintaining the data consistency in the system, and ensuring that threads do not access stale or out-of-date data that was written by a thread on another machine, Brazos page-based DSM system uses Scope Consistency model [Iftode L. et. al 1996]. Scope model is a relaxed model aims at reducing the present of false-sharing in page-based DSM systems. Recall, this situation occurs when two or more threads modify different parts of the same page of data, while they do not share the same elements in the page. False sharing leads to unnecessary network traffic, and can be a significant problem for DSM systems due to the large granularity of sharing. Scope consistency

guarantees that only data modified within a single scope to be coherent at the end of the scope.

However, Brazos implements a variant of the scope model. The variant is a software-only implementation of scope consistency that requires no additional hardware support. Brazos uses this variant in a conjunction with a distributed management protocol. With a distributed management protocol, each process maintains dirty portions of each shared page of data, requiring processes to communicate with all other processes that have a modified portion in order to bring an invalid page up to date. In order to reduce the inherent huge number of consistency-related messages, and to efficiently implement scope model at software level, Brazos makes use of the multicast primitives provided by the WinSock 2.0 library. In a time-multiplexed network environment such as Ethernet, sending a multicast message is no more expensive than sending a point-to-point message. Moreover, large reductions in both the number of messages sent and the number of bytes transferred to maintain coherence can be achieved by specifying multiple recipients for each message.

Adaptive performance tuning mechanisms can have a beneficial effect on performance when used to tailor runtime management to observed behavior [Amza C. et. al 1997], [Bennett, J et. al 1990]. Brazos employs four adaptive techniques: dynamic copyset reduction, early updates, adaptive page management protocol, and a performance history mechanism.

The system utilised multicast protocol for updating processors with the new data written by another processor. However, multicasting *diffs* causes processors to be interrupted even though these *diffs* are not to be used by the processor until the next invalidation of the page, detracting that from user-code computation time. The dynamic copyset reduction technique is used to ameliorate the effect of this problem by allowing the processes to drop out of the copyset for a particular page, causing them to be excluded from multicast messages providing *diffs* for the page. A process is placed in a list of to-be-dropped processes, if the number of unused *diffs* reaches certain threshold, and it is removed from the list when a page fault occurs. On the other hand, sending indirect *diffs* of pages to processes that have other outstanding requests to other pages,

incurs undesired network overhead. Therefore, Brazos system allows processes to list all these early updated pages to be sent to all other processes, which in turn send all *diffs* of these early updated pages in a single bulk at the next barrier instead of multiple bulks. This technique is called early update mechanism. Processes can return to the default multicast invalidation protocol using the dynamic *copyset* reduction mechanism.

Another adaptive protocol used by Brazos is the page management. With this protocol, a page is managed by either a home-based protocol [Carter JB 1993] or a distributed page protocol according to the observed behavior of the system in order to provide the appropriate management technique for each shared page [Keleher P. et. al 1992]. Also, Brazos incorporates a history mechanism that allows the runtime system to adapt more quickly to the program behavior. Information regarding the performance of the various adaptive protocols is kept in files to be referred to for each application.

Brazos system incorporated number of techniques and mechanism in order to achieve efficient performance. Experimental results showed that Brazos system outperformed TreadMarks system in all the applications that were used in the comparison, which proved the significance of utilizing the discussed techniques in the performance of DSM systems.

2.7.1.5 CLOUDS

So far, the given examples have been software DSM system implemented as user-level routines. However, many other DSM models have been implemented as operating systems via incorporating the DSM mechanism inside or outside the kernel. CLOUDS [Ramachandran U. et. al 1989] is a distributed operating system that incorporates software-based DSM management, and implements a set of primitives in the context of an object-based operating system as well as on top of UNIX. It was implemented on SUN-3 workstations and connected via Ethernet. The resources in the system are viewed as shared objects. The objects are composed of segments. A segment is a logical entity with attributes such as read-only and read-write. Each segment is owned by its creating node, which is responsible for the consistency of the segment. CLOUDS system manages these segments by a distributed shared memory controller

(DSMC). The DSMC manages the segments via two operations: *get* and *discard*. The *get* operation is used to fetch a segment from its owner, whereas *discard* is used to return a segment to its owner. On the other hand, synchronization primitives are provided as separate operations (*P* and *V* semaphore operations), or as combined access and lock operation using get and discard.

A segment is acquired by the get operation in one of four modes: read-only, read-write, weak-read or none. Read-only means non-exclusive access to a segment, that is, the segment will not change until it is discarded by the holding node. Read-writes mode allows exclusive access to the requesting node with a guarantee that the segment will not be thrown away until it is explicitly discarded. When a get primitive is issued with mode read-only or read-write, the local DSMC sends a request to the owner DSMC and suspends the requesting process until the reception of the segment. The segment is kept at the requesting node until it is explicitly discarded. A segment can be read by multiple processes at the same time, but only one process is allowed to perform write operation. On the other hand, weak-read gives non-exclusive access to a segment without guaranteeing that the segment to remain unchanged; therefore, the owner node immediately sends the segment for weak-read access mode. Clearly, having this mode of access, results in inconsistent shared memory. Finally, none mode signifies exclusive access to a segment without guaranteeing that the segment will not be thrown away. If the owner node receives a request for a segment which is currently held by another node for none mode access, the holder is instructed by the owner to forward the segment to the requesting node instantly without performing discard operation (i.e. by the holder).

Objects and threads are the basic building blocks of CLOUDS. Objects are passive entities, and they specify a distinct and disjoint piece of global virtual address space that spans the entire network. An object is an encapsulation of the code and data needed to implement the entry points in the object. On the other hand, threads are the only active entity in the system. When it is created, a thread starts executing an object by entering it through an entry point (i.e. code or data). At run-time, a thread can execute other objects while executing another object. After executing an object, it returns to the

caller object. Therefore, in the course of its execution, a thread traverses the virtual address spaces of the object it invokes.

Ra kernel was designed to run alongside the CLOUDS system. It defines and manages three primitive abstractions: *segment*, *virtual spaces*, and *isiba*. A segment serves as container of data and may be viewed as un-interpreted sequences of bytes, and its contents may only be accessed when the segment is attached to a range of virtual addresses. Collections of segments comprise the Ra virtual spaces where each is associated with a set of windows called descriptor. Each window is a data structure that maps a contiguous piece of the virtual space to a segment. On the other hand, Ra isibas are light-weight processes, and may be used as daemons within the kernel or associated with a Ra virtual space to implement a user process.

The virtual space provided by the Ra kernel is different from the one provided by the architecture of the machine itself. The latter is assumed to be composed of three distinct regions: object, kernel and process. The Ra kernel is mapped in the kernel space of the hosting machine, while a process consists of an isiba and a Ra virtual space that does not contain any code. On the other hand, an object is a Ra virtual space that consists of code and data segments. The code segment of an object has entry points that can be invoked by user processes. The object being executed by a process is mapped into the object space of the machine. On the other hand, the system objects are mapped into the machine's kernel space, and can be installed and removed dynamically. These objects encapsulate necessary and/or useful system resources and resource managers that have direct access to the Ra kernel; however they are outside the kernel. Moreover, they serve as intermediaries between the user objects and the kernel, and they provide system services to user objects. Such objects include resource managers, user-level object support, device drivers, and partitions.

One of the implementations of CLOUDS system was on Ra kernel. In this implementation the DSMC co-operates with remote DSMCs to implement the distributed shared memory primitives. The distributed shared memory (DSM) partition is a Ra partition that provides the kernel with number of operations, such as create/destroy and activate/deactivate segments. The algorithms of DSMC use a simple

transmission protocol for exchanging request/response message reliably. This protocol was called the Ra transaction support protocol TAL/RaTP.

CLOUDS was implemented on top of UNIX; in terms of the DSMC and the DSM partition; for three reasons. The UNIX environment makes it easy to test and verify the DSMC and TAL/RaTP protocols. Besides, the UNIX file system is available for use as permanent store for segments, and Ra can execute on diskless workstation with backing store provided by UNIX machines. Furthermore, bearing in mind that the strength of UNIX is the rich program development environment that is provided, and the strength of CLOUDS is the transparent management of distributed data and computation; therefore, the inter-operability between UNIX and CLOUDS provided by Ra kernel produces the required performance. TAL/RaTP runs a user process that uses SUN's network interface tap (NIT) to receive packets from the network and to route them among a set of clients and servers. The DSMC code is linked-in with the server code that uses the Unix file system to store segments and service requests from Ra DSM partitions, whereas the DSM code is linked-in with client code that uses the system.

2.7.1.6 Orca.

Distributed shared memory mechanism can also be implemented at the level of programming language, since the compiler can detect shared accesses and inserts calls to synchronization and coherence routines into the executable code as in Orca. Orca programming language introduced a new conceptual shared memory model, called the shared data-object model [Bal H et. al 1991]. An object encapsulates data and predefined set of functions. The data in an object can only be accessed by the encapsulated functions.

With orca, objects are distributed in the computing environment upon the creation of a new process. In other words, when a process spawns a child process, it can pass down any of its objects as shared parameters to the child, and that child can pass the objects to its descendants and so on. Therefore, all these processes can share the objects and perform the predefined operations. The compiler distinguishes between two different kinds of operations: read and write. Read does not change the value of a shared

object and is performed locally if a copy exist, whereas, write operation may read or write on the object and effect all the copies all over the system. All writing operations are directed to the primary replica, and then the update is propagated to the secondary replicas. A processor that most frequently changes the object can be assigned as the host of the primary replica, and that assignment is done according to statistics made by the run-time system. The execution time of obtaining the statistics is negligible comparing to the time needed for remote references.

Updates to any object are visible to all processes; therefore a shared data-object is a communication channel between processes. Unlike regular shared variables, access to shared data-object is only through its functions, which are considered as indivisible operations. In short, accesses to the shared objects are automatically synchronised, since simultaneous operation invocations to the same data object can be seen as they were executed one by one, called serialization. Such serialization is enforced by appointing one replica as primary copy. On the other hand, having replicas of shared objects introduces consistency problem. In orca, a 2-phase update protocol is used to have consistent view of all shared objects all over the system. During the first phase, the primary copy is updated and locked, and subsequently, an update message is sent to all processors containing a secondary copy. Before being updated, a secondary copy is locked. A user process that tries to read a locked copy blocks until the lock is released during the second phase, which begins after acknowledging all update messages. At the end of the second phase, the primary copy is unlocked and a message is sent to all processors containing a secondary copy, instructing them to unlock their copies.

To implement this protocol, every processor is associated with one manager process, with the assumption that the manager process and user processes on the same machine can share part of their address space, wherein objects or their replicas are stored. All writes are directed to the manager of the primary object, while reads are performed locally. Each manager process contains multiple threads of control. One thread communicates with remote managers, and the remaining threads are created dynamically to handle write operations. Therefore, multiple writes to different objects can be performed simultaneously, while writes to the same object are serialised. The 2-

phase update protocol guarantees that no process uses the new values of an object while other processes are still using the old value. Processes are allowed to use the new value after the second phase, which is concluded by the completion of updating all replica copies. Concurrent write operations on the same object are serialised by locking the primary copy. However, the next write can only be operated before all secondary copies are unlocked. New requests to update-and-lock a secondary copy are not serviced unless the unlock message generated by the previous write has been handled by its manager and acknowledged. Orca uses multi-threaded managers to prevent deadlock. That is, write-lock operation on a primary copy may block one thread of a manager, but not an entire manager process. Locking a secondary copy always succeeds within a finite amount of time, given that all read operations terminate properly.

Orca is a simple, general purpose programming language based on data-objects intended predominantly for application programming rather than system programming. Parallelism is based on dynamic creation of sequential processes that communicate indirectly through shared data objects. Objects are shared via passing them as shared parameters to a newly created process.

2.7.2 Trends in Improving the Performance of Software DSM Systems.

Building distributed systems on network of workstations with DSM algorithm has been proved as a viable alternative to the traditional message-passing paradigm. The increasing demands of distributed applications require sufficiently high-performance DSM algorithms. Therefore, another research direction has been launched, alongside the developing of new DSM algorithms, to investigate new techniques in improving the performance of DSM algorithms. Among these techniques are: adaptive protocols that adjust to the memory access patterns in distributed applications [Amza C. et. al 1999]; multithreaded DSM algorithms [Mueller F. 1997]; and relaxing consistency definitions to match the needs of distributed and parallel applications.

2.7.2.1 Adaptive Protocols for Software DSM Systems

Many different protocols have been proposed for implementing a software shared memory abstraction on distributed memory hardware environment. The relative performance of these protocols is application-dependent in such a way that the memory access patterns of a distributed application determine which protocols exhibit good performance. Therefore, building distributed systems with different protocols may boost the performance in the sense that the system can choose at run-time the right protocol based on the observed access patterns.

In [Amza C. et. al 1999] experiments were conducted to demonstrate the benefits of having protocols that automatically adapt at run-time to the memory access patterns observed in the applications on the assumption that shared memory access patterns are detected using virtual memory protection scheme in the hosting machine. These protocols implemented the lazy release consistency model, and employed in TreadMarks software distributed system. Particularly, the investigation was mainly in:

- Adaptation between single- and multiple-writer protocols.
- Dynamic aggregation of pages into larger transfer units.
- Adaptation between invalidate and update protocols.

The adaptations are triggered automatically. The run-time system detects certain access patterns and switches between protocols accordingly. Unlike mutli-protocol software implementations [Carter JB 1995], it removes the need of user annotation to select the appropriate protocol in order to improve usability. The choice between the single- and multiple-writer protocols is based on the presence of write-write false sharing and on write granularity. Put in other words, having more than one processor attempting to write on the same page causes write-write false sharing situation that can be avoided by the use of multiple-writer protocol, allowing more than one processor to write on a page simultaneously. While, single-writer protocol is advantageous if a processor writes on large part of a page rather than small part of it, that is, the cost of diffing and twining imposed by the multiple-writer protocol is avoided without much increase in the communication.

Dynamic aggregation is also used to investigate its impact in improving the performance mainly via the reduction the number of update messages exchanged in the system. Dynamic aggregation uses records of earlier accesses by a processor to coalesce page into page groups, in the expectation that those pages will be accessed again by the processor. On the other hand, the choice between invalidate and update protocols is based on whether the destination is to access the modified data before it is overwritten or not. Update protocols send substantially more data, including data that the processor may never access or that may be overwritten by newer data before the processor accesses that data originally sent. Whereas, invalidate protocols only retrieve the data for the pages the processor accesses, but they pay the penalty of the access miss faulty and the round-trip latency to get the modifications. Therefore, a prior knowledge of whether the updated and to-be-transferred data will be overwritten before being read by the processor is crucial in selecting the appropriate protocol.

The experiments indicated that:

- Adaptation between single- and multiple-writer and dynamic aggregation performs well, and in some cases shows substantial performance improvement, and never decreases the performance.
- Adaptation between invalidate and update is less successful, with performance improvements that match dynamic aggregation in some cases and substantial performance losses in others.

The experimental reports concluded that adapting protocols at run-time according to access patterns of the shared memory, can effectively improve the performance only with respect to the nature and the behavior of the distributed applications, as it degrades the performance of others.

Another example of having adaptive protocols to improve the performance of software DSM system is the merit adaptation presented earlier in Brazos system. As a conclusion, adaptation is a promising technique for improving software DSM algorithms especially it adjusts to the current situation of the system with no intervention from the user.

2.7.2.2 Multi-threaded Software DSM Systems

In the last decade number of researches have been investigating the applicability of multithreaded algorithms in building distributed systems in order to exploit the potential of multithreading as in Brazos and DSM-Threads [Mueller F. 1997]. In multithreading approach several threads can run under the umbrella of one process, where they share the same resources allocated by the operating system to the process, and therefore saving system resources. Also, using multithreaded algorithms can hide the long communication latencies typically associated with software DSM systems.

DSM-Threads is a software DSM system built to provide a convenient means for a programmer to migrate from concurrent programming model with shared memory (PThreads) to a distributed model with minimal modifications to the application code. The DSM runtime system, per node, consists of a communication server and worker threads. The communication server is a separate thread of control with the highest priority of all threads within the same process. The task of the communication server is to open a communication channel and wait for incoming messages. When a message is received, and according to the type of the message, the communication server may process the request or delegate the message to a worker thread to process it. Worker threads can also be activated via the DSM runtime system to function as user threads or when a page fault occurs. The page fault handler activates a worker thread to perform the task of fetching a page. The worker thread may then send a request to the communication server in the hosting node. The communication server may delegate a worker thread to respond to the request. After fetching the page the worker thread on the requesting host reactivates the user thread.

Having n worker threads delegated to process n pages faults, may cause deadlock situation if a new request arrives to the communication server. DSM-Threads system avoids this situation by assigning messages of the same type to only one worker thread, which requires that the size of the worker threads pool to be at least the number of delegated message types. Moreover, a worker thread is only assigned to a message if there is no worker is yet engaged in processing message of the same type, or if the

number of idle workers exceeds the number of currently unhandled message types. A distributed thread is created by a primitive **dsm_thread_create** that allows the programmer to optionally specify a destination node for the thread. If no destination node is specified, the system has a choice of selecting such a node using a history of load information and CPU throughput. For repeated executions of an application, trace data and thread group information may be used to distribute threads upon creation. More specifically, a small number of threads of a group may be placed on the same node if very frequent page faults occurred for a distributed configuration, i.e. if the overhead of page faults for a distributed execution is likely to exceed the overhead of executing multiple threads on one node. Therefore, repeated traces under different configuration can be used to improve the overall performance gradually by finding better thread distribution. Experiments showed that these method of thread distribution reduced false sharing while improving the overall throughput.

As a conclusion, research reports in multi-threaded DSM systems have proved that it is beneficial to have multiple threads in the system as they can be assigned to different tasks to run at the same time. DSM systems with multithreading algorithms exhibit improved performance in terms of saving system resources, and hiding communication latencies by overlapping communication with computations. However, special care has to be taken to manage and control the overlapping behavior of programming with multiple threads, to avoid the likelihood of memory inconsistency and data races.

2.7.2.3 Relaxing Consistency Definitions

Due to the inherent complexity of implementing consistency models that impose strict constraints which results in poor performance in many cases, new consistency definitions tend to be more relaxed to enhance the performance of DSM systems. For example, although it is programmer-friendly model, the sequential consistency was proved as a source of a serious performance problem [Lipton R. et. al 1988]. Specifically, if the read time is r, and the write time is w, and the minimal packet transfer time between nodes is t, then it is always true that $r + w \geq t$. In other words, for

any sequentially consistent store, changing the protocol to improve read performance makes write performance worse, and vice versa.

As it has been discussed earlier in this chapter, newer definitions of consistency models present weaker restrictions than their predecessors. With relaxed definitions of memory consistency in DSM systems, the number of exchanged messages between different processes is greatly reduced resulting in less communication overhead. For example, in lazy release model, updates are sent only to the process that executes the acquire command, rather than propagating the updates to all processes in the system as in sequential model.

A common approach for weakening consistency models is to use synchronization mechanism, as in lazy release and weak consistency. That is, when a process performs an operation on an ordinary shared data item, no guarantees are given about when this update will be visible to other processes in the system. Models apply such approach vary in when an update is made visible to other processes in the system, however in all cases a process can perform multiple reads and writes in a critical regions without invoking any data transport routine. When a process leaves a critical section, updates are either sent to other processes (i.e. as in release model), or made ready for transfer to any process shows interest in this piece of shared data. However, such relaxation should be at a sensible degree to guarantee the ease of programming, which was the major intention of DSM systems in the first place. The degree of relaxation is greatly influenced by data access patterns, and the purpose on which this data are used. In other words, the significance of implementing a synchronised model in the performance of a distributed shared memory system depends on the nature of the applications that use the system. Nevertheless, implementing relaxed models is often much easier than more restricted models [Tanenbaum A. et. al 2002].

The last two improvement techniques are used in designing and prototyping the new framework, which is presented in the next chapter. As we will see later, these techniques have a significant impact in the performance of the produced system in terms of speeding up the whole system and reducing data retrieval rates.

Chapter Three

DIME-II: NON-LOCKING APPROACH, CONSISTENCY MODEL AND DATA EXCHANGE ALGORITHM – DESIGN & EVALUATION

Building a successful distributed shared memory system depends enormously on the degree of consideration of the design issues at the designing stage. Sometimes this degree varies according to the nature of the distributed application itself, since supporting some features in the application can significantly improve the performance. Also, there are pre-existing requirements that need to be taken into account. For example, our new framework inherits an important feature from DIME software DSM system, which is to support two different data types. These data types naturally exist in Urban Traffic Information and Control system, which is the system that will be used as a case study for the whole project. Other inherited properties from the DIME system are the granularity and the non-locking approach. This thesis refers to the current implementation of DIME as DIME-I, and the new improved framework as DIME-II, and when talking about common features will be using the term DIME.

3.1 DIME and Traffic Control Distributed System

The potential for the enhancement of the performance of current traffic control systems, through supervisory control making use of in-vehicle route guidance, has created the need for a flexible computing environment in which various new applications can be integrated with existing traffic control systems without adversely affecting their performance.

Using network of workstations (NOW) as computing platform for distributed systems has distinctive advantage when building a fault tolerant system. In other words, in the occurrence of machine failure, networked parallel system can continue running with no disturbance and transfer the failed task to be executed on another machine. Moreover, parallel systems can employ more than one server in order to avoid bottleneck. Building parallel and distributed applications on NOW requires a middleware of software that can efficiently manage exchanging messages and data between different applications running on different machines. Traditionally, there are two paradigms in building such middleware in distributed systems: the message passing (MP) paradigm and the distributed shared memory (DSM) paradigm.

Distributed shared memory (DSM) paradigm provides an illusion of one non-physical shared memory in network of workstations where shared data reside in different address spaces. DSM algorithms facilitate accessing the shared memory and exchanging data and messages via normal read and write operations, concealing the interprocess communication and remote memory accesses. Unlike MP paradigm, exchanging complex data between processes in different locations is supported [Protic J. et al 1996].

The **DI**stributed **M**emory **E**nvironment (DIME) system provides an interface between software modules that execute on networked workstations. As first introduced by Argile et al in 1996, DIME is a flexible computing environment that provides a communication harness for the execution of software modules of urban traffic control systems, and allows all these modules to be effortlessly integrated. DIME system has been designed specifically to support vast range of transport telematics applications and it offers a convenient interface to the applications programmer. As it was built as a user-level software DSM system, DIME provides an easy to use communication interface that simply and reliably delivers data and messages to all nodes in the system. This interface was built on top of Transmission Control Protocol/Internet Protocol (TCP/IP). The implementation of the communication interface supports a variety of platforms such as DOS, Windows, UNIX and EPOC (operating system for palm-top computers PSION) [Peytchev E. et. al 1998].

As shown in figure 3.1, DIME system has a role of managing and controlling the shared memory in urban traffic system. It receives access requests to the shared memory from the different modules, and replies appropriately. Typically, SCOOT (Split, Cycle and Offset Optimisation Technique system) module gathers real-time data collected from the traffic network and sends it to DIME, which in turn stores them in the shared memory. These stored data are processed and manipulated by telematics applications (PADSIM, ATTAIN...etc) in order to perform their monitoring and control tasks. The applications communicate with each other through DIME system by saving their outputs in the shared memory, where they are readily available to any requesting application. In brief, all interprocess communication and data exchanges between different modules of the traffic system are delegated to the communication interface of DIME system.

3.2 Types of Data in a Typical Traffic Control System

Building a successful DSM system requires a detailed knowledge of the data transactions in the system; therefore a special consideration of data flows in a traffic control system was taken in the design stage of DIME-I [Peytchev E. 1999]. In a traffic control system there are two kinds of data that can be recognised:

- Dynamic data: It is collected by the real-time traffic control system. It contains all information about traffic counts and local controls as they occur in the traffic network. It is characterised by its high volume - in excess of 120 Mbytes per day per one specific type of message. Besides, this kind of data is updated in a high frequency rate (per second basis).
- Static data: This kind of data is updated in a much longer period of time and its purpose (in general) is to make the results from traffic modules available for reading by the other functional modules in the system.

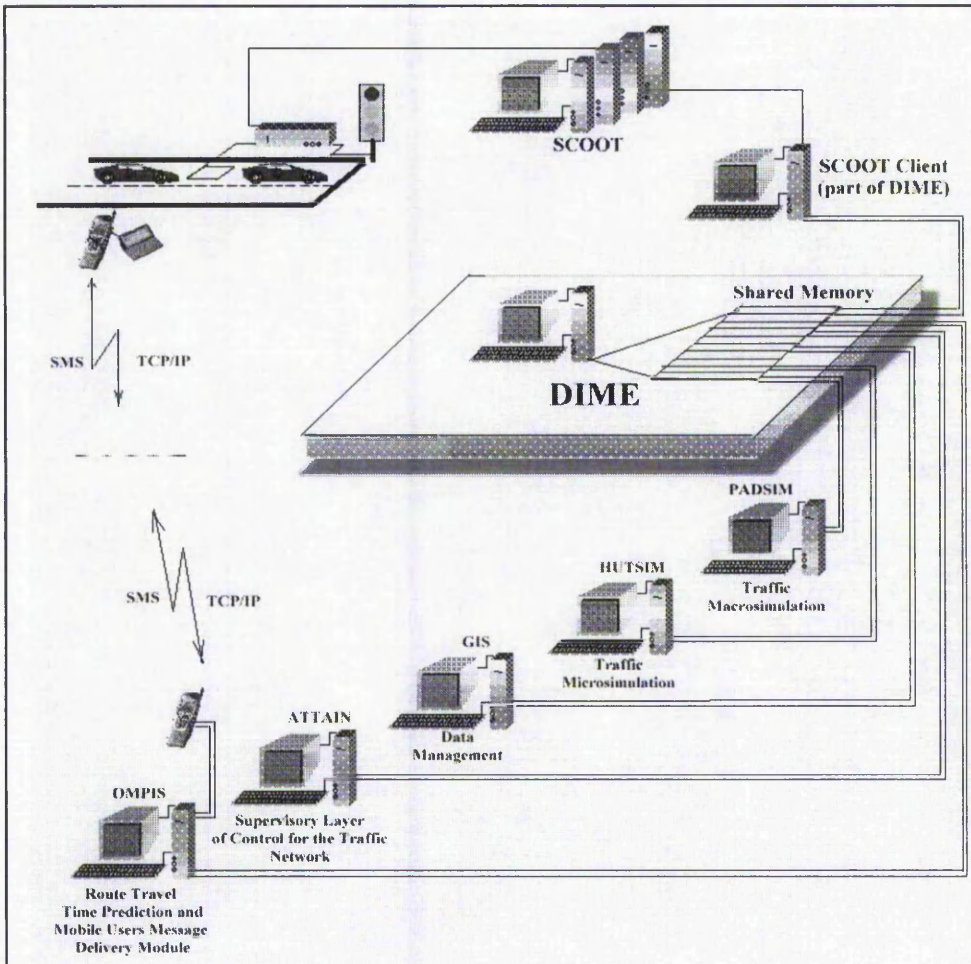


Figure 3.1: DIME System and Traffic Control System

3.3 DIME-I Configuration

DIME system is a user-level software DSM prototype. As in TreadMarks, this system provides user-level runtime library routines. But, unlike IVY, these routines require no modifications on the underlying operating system as they can be used and run on different kind of environments and platforms. This system is designed to work with two functionally different read operations and two functionally different write operations, supporting the two different types of complex data structures [Peytchev E.

1999]. The attributes of a shared element of any data type is user-defined, thus, this system can also be categorised as object-based DSM system. Moreover, DIME is designed to work in a heterogeneous computing environment, however, because it does not facilitate data transactions between different platforms all applications that use DIME system should perform data conversion upon exchanging data between different platforms. Furthermore, new modules can be added to the system at runtime with no disruption to the running modules.

DIME-I system adopts a sequential consistency model and a centralised architecture of the memory manager. In a centralised algorithm all accesses to a shared memory are directed to a central server that controls the whole shared memory. Thereby, DIME-I does not require any hardware routines to detect accesses to pages of shared memory; therefore it can be ported to both UNIX and DOS (or any other similar operating platform) based systems with no modifications to the kernel of the hosting operating system.

In DIME-I system, each user application (module) has an additional component linked to it at compile time, which provides the communication interface via DIME-I APIs with the shared memory system (figure 3.2).

The requests for reading/writing data from/to the shared memory are transferred by the DIME-I libraries over the network to the memory manager task, where they are processed and, subsequently, appropriate replies are sent back. DIME-I system consists of two components: a) The shared memory manager that owns the shared memory (i.e. DIME-I server), and b) the DIME-I communication libraries, which are linked to user applications and the DIME-I server in order to interface to the network [Peytchev E. et. al 1998].

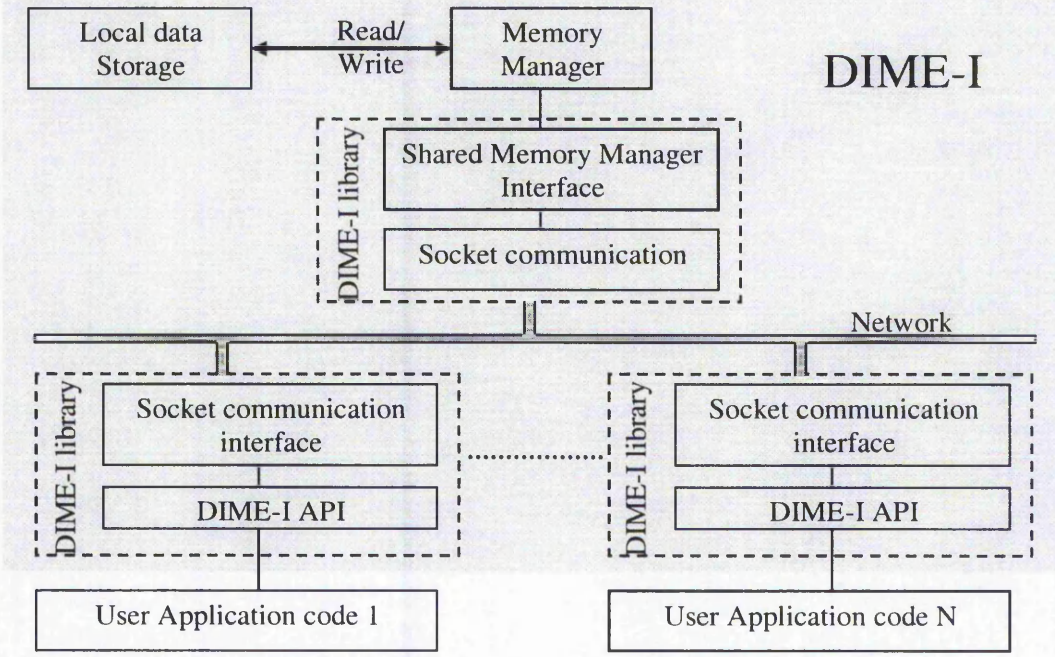


Figure 3.2: DIME-I Configuration

The shared memory manager (SMM) operates on a closed loop basis, continually listening to requests for accessing or maintaining the shared memory. Such requests are issued by the modules by performing appropriate API routines. Concurrent accesses to the shared memory are implicitly synchronised in the operation of the memory manager task, due to the natural sequencing in the underlying network (Ethernet) protocol (TCP/IP). SMM executes the received operations in the order of their arrival, therefore, unlike TreadMarks; it does not require the provision of separate functions for synchronization.

3.4 The Limitations of DIME-I

DIME-I system adopts Sequential consistency (SC) model [Lamport L. 1997], which is the most commonly assumed model by the programmers due to its intuitively expected throughput [Weiwu H. et. al 1998, Hill M. 1998]. However, recall, SC definition is very strict and usually has a problem of poor performance. As shown in

[Lipton R. et. al 1988] experiments proved that any attempt, in a sequentially consistent distributed shared memory system, to change the protocol in order to improve reading operations performance makes writing operations performance worse, and vice versa. Equally important, many performance enhancement techniques, such are prefetching, multiple-issue, and write buffer, are not allowed in a sequentially consistent machine [Tanenbaum A. et. al 2002]. In brief, with the definition of the sequential consistency model, a DSM system will not provide a high-quality performance, and moreover, any attempt to improve the performance using improvement techniques may not be successful or may not even be possible to implement. Therefore, subsequent definitions have been introduced to provide a consistent memory in DSM systems with relaxed or weak constraints in accordance with the nature of the required computing environment, while maintaining the usability and the programmability of the consistency model.

To overcome the poor performance associated with the sequential model, another model has to be used in DIME system that can allow the use of improvement strategies, which in turn can satisfy the demands of the system. Conclusively, the use of a relaxed model capable of supporting the natural existence of two data structures in a typical traffic system; represented by the static and dynamic data; can provide an efficient satisfactory performance. Although many relaxed consistency models have been introduced and implemented in several past and recent DSM systems, they do not provide an explicit support for the two types of data in the traffic system. Considering the nature of the distributed applications, in particular the data usage patterns, can effectively offer the required performance. Therefore, section 3.7 of this chapter presents a consistency model that provides an explicit support for the presence of dynamic and static data structures in the traffic system.

This consistency model has been designed specifically for the traffic system. In this model we have avoided the explicit use of any synchronization mechanism, such as locks and barriers primitives, to avoid the likelihood of slowing down the entire system caused by the competition between different applications and processes to acquire an exclusive access to the shared memory. Excluding the explicit utilization of synchronization mechanisms allows the system to maintain a steady performance. This

purposely designed consistency model relies on the natural sequencing of commands occurring in a TCP/IP-networked environment. In other words, read/write commands in this model are performed in the order they are received, not the order they were sent, and they are processed indivisibly and without interleaving. In accordance with the definition of the sequential consistency model, the processing of read/write operations, atomically, gives the flavor of sequentially consistent shared memory. Such feature makes the new model easy to understand and to use as the programmers of distributed and parallel systems are so accustomed to the semantics of the sequential consistency model. Furthermore, this model is categorised as a non-locking model, and therefore the new framework of DIME system is categorised as a non-locking DSM system.

Another drawback in DIME-I system is that the shared memory is controlled by only one central manager that is responsible for servicing read/write requests from/into the shared memory from all the applications in the system. This central mechanism used by the memory manager is liable to bottleneck problem, because all requests are directed to only one server. Moreover, such mechanism is unreliable, particularly if the server crashes, in which case the distributed modules of the system will lose communication with each other, leading to the failure of the entire system. This problem can be avoided by replicating the shared memory in order to maintain continuity in the process of the distributed system. Additionally important, data replication can reduce the competition between the applications and speed up the entire system.

Data replication can be achieved by replicating the server itself (i.e. mirroring the server) and distributing the workload among several servers. This scheme has disadvantages in two aspects. First, the fact of having several servers of the same task of providing the service to many applications means that the system will have several server workloads and overheads instead of one, which increases the amount of software overhead. Secondly, upon any update made to one shared memory replica, a message-broadcasting process must be performed to update all the existing mirrors of the shared memory in order to make the view of the memory consistent throughout the system and to allow all the applications to have the same versions of data. Undoubtedly, this process

incurs another kind of overhead to the system in terms of flooding the network with messages that contains updates which might not even be needed by all applications.

Some DSM systems provide a replica of the whole shared memory in each machine where an application is running, as in BDSM system [Auld P. 2001]. In this system, each machine runs a subsystem to control a complete mirror of the shared memory. In other words, the same shared memory is replicated in every single machine in the system. Similar to the previous data replication scheme, this kind of systems also suffers from high communication overhead due to the need of broadcasting updates as they occur on the shared memory. BDSM system, in particular, employs a data transmission protocol that uses a fully-replicated broadcast-based algorithm to disseminate messages and updates to all parts of the system. Again, the scheme of fully-replicated broadcast-based algorithm leads to performance degradation as the demands of the applications and messages exchanging increases.

As it will be introduced and described in later sections, the new framework of DIME system also uses a data replication scheme. However, it provides each application only with the data required to perform its native task. The aim is to reduce the time spent by an application to retrieve data from the shared memory. In this framework, the data is kept in the proximity of the application. Unlike the previously described schemes of data replication, this framework provides an application with a copy of the data needed to accomplish that application's task, and therefore, when an update operation is performed on the shared memory, the system propagates the update only to the copies of the shared memory that keep a replica of that particular item. Thus, there is no need to broadcast updates to all data replicas throughout the entire system as some of these replicas may not hold a copy of the updated item, hence, broadcasting updates in this system will be unjustified. Unlike BDSM system, a partially-replicated multicast-based algorithm is needed in the new framework to disseminate messages to update only the shared memory replicas of the applications that process the updated items. It can be foreseen that, this sort of data replication, along with the proposed protocol, saves the network resources and reduces the communication overhead by the reduction of the number of exchanged messages and data.

On the other hand, providing an application with the required data on its machine eliminates the likelihood of having applications competing over the shared memory. In this framework, each application can perform read/write operations on its own replica of the shared memory, and the burden of keeping the shared memory replicas consistent is entirely left to the middleware system. This scheme of data replication allows an application to retrieve the required data from a local repository and with no competition with other applications in the system. Thereby, there is no need for synchronization mechanisms to access the local memory as each application has an exclusive access to its local replica.

In a nutshell, a non-locking partially-replicated DSM algorithm with a relaxed variant of the sequential consistency model is expected to overcome the limitations of the old architecture of DIME system. Furthermore, it is expected to improve the performance of the system by the reduction of the time taken by an application to retrieve data from the shared memory while saving the network resources by exchanging the minimum amount of messages and data. The rest of this chapter presents a new framework for DIME system, which is called DIME-II system. This framework employs a non-locking partially-replicated DSM algorithm

3.5. Design Issues

As mentioned earlier, the aim of the project is to build a DSM framework in which data retrieval rates are reduced in order to save more time for the user application to perform the native tasks. Besides, the new framework is ought to be flexible allowing system re-configurability at start-up time/run-time in order to improve the performance of the system. As we will see in the following sections, this is achieved via the use of a non-locking and partially-replicated algorithm that permits such re-configurability without changing the backbone of the system. Figure (3.3a) illustrates a general view of DSM system as one main server and several applications, whereas; figure (3.3b) depicts one possible architecture of a reconfigured DSM system, which can be structured by redirecting the communication paths of certain applications to take the service from

sources other than the main server (i.e. intermediate servers that are embedded in user applications codes).

In the designing stage of building DIME-II system, number of design issues has been considered in order to provide a structural design capable of delivering an efficient satisfactory performance. As it has been discussed earlier, in the traffic system there are two types of data structures: dynamic and static data [Peytchev E., 1999]. The data representing the dynamic data type is in fact a constant flow of uniform messages issued from the traffic system. To accommodate this type of data, the system needs a formation capable of accepting a number of uniform structures at a time, and at the same time keeping the most recent data only. The implementation of DIME-I utilised circular buffer for this type of data, where each element of the buffer is a user defined message structure and its size depends on the size of the urban traffic network.

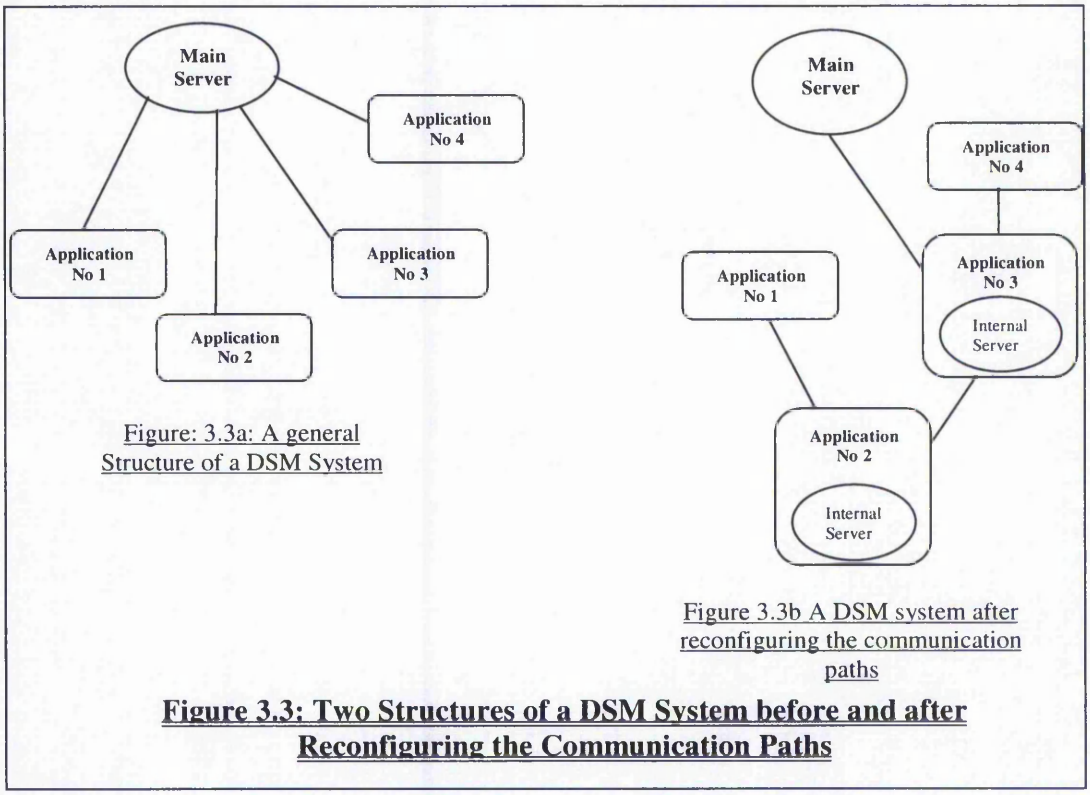


Figure: 3.3a: A general Structure of a DSM System

Figure 3.3b A DSM system after reconfiguring the communication paths

Figure 3.3: Two Structures of a DSM System before and after Reconfiguring the Communication Paths

On the other hand, the relatively static data in the traffic system usually reflects the value of some internal variables in the traffic modules. The volume and the format of this data is usually module dependent since each module has its own internal representation of the traffic. Therefore, DIME-I utilised an array of bytes of user's defined size for static data, as it is the most suitable choice [Peytchev E., 1999]. The implementation of DIME-II system continues using this granularity, since it is suitable and convenient for representing the two types of data of traffic control systems.

Moreover, DIME-II is designed to run on different platforms, and therefore, it can run on a heterogeneous environment. However, DIME-II employs communication protocol that internally exchanges data and messages as arrays of bytes; therefore, software modules have to make their own internal simple conversions. DIME-II is designed with the capability of extension to contain later addition of software modules

3.6 Non-Locking Approach for DIME-II Computing Framework.

To overcome the previously detailed limitations of the current implementation of DIME-I system and in order to improve the performance of the system, DIME-I architecture has to be modified to support the implementation of the non-locking approach, data replication algorithm and per-node multithreading. This framework aims at improving the performance of DIME-I software DSM system mainly by minimizing the time of data retrieval from the viewpoint of user application and to have a flexible reconfigurable design to adjust the system connectivity to the current workload of the network to achieve an optimised performance. With DSM algorithms, distributed applications often waste valuable time while retrieving data from the shared memory, this time is spent by the middleware system during exchanging data and messages between different parts of the system to fetch the requested data. However, with the algorithm presented in this chapter, an application retrieves the required data from a memory associated with that particular application. This intermediate memory contains copies of the data required by that application (not a whole replica of the main memory) and accessed only by that application.

The burden of making the memory consistent all around the system is entirely left to the middleware system and applications can retrieve the data from their associated memories with no competition with each others. Furthermore, the system can provide each application with the required data, which are not necessarily the most recent ones, but these data will be available at anytime they are requested. Thus, user applications will always find the required data without significant delay, bearing in mind that the data is retrieved directly from the intermediate memory. Providing an application with the requested data in a relatively short period of time is considered the one single most important factor of measuring the performance of the system. The motive behind this assumption is that user applications can have more time for performing their native tasks, which time is very often wasted in network communication [Khalil M. et. al 2003b].

Many other different approaches and measurement factors have been introduced and taken into account in past and recent research to measure the performance of DSM systems. For example, Munin [Bennett, J et. al 1990] adopts multiple relaxed consistency protocols in order to achieve good performance through reducing the number of messages exchanged in the network. On the other hand, TreadMarks [Amza C. et. al 1996] adopts the same means, but to speed up the distributed system as a whole.

Having its architecture in figure 3.2, DIME-I can be viewed as a system of three component (figure 3.4), which are: DIME-I-server (the shared memory manager), DIME-I-client (DIME-I APIs), and user applications, wherein all shared data are resident in one machine controlled by one central memory manager. In DIME-I, DIME-I-client acts as an inactive process which is activated by a user application upon performing read/write operations on the central shared memory. Typically, a user application performs reads/writes on the shared memory via calling DIME-I-client routines that contact DIME-I-server to execute the prescribed operation, and then returning to its previous inactive state along with the result of the operation.

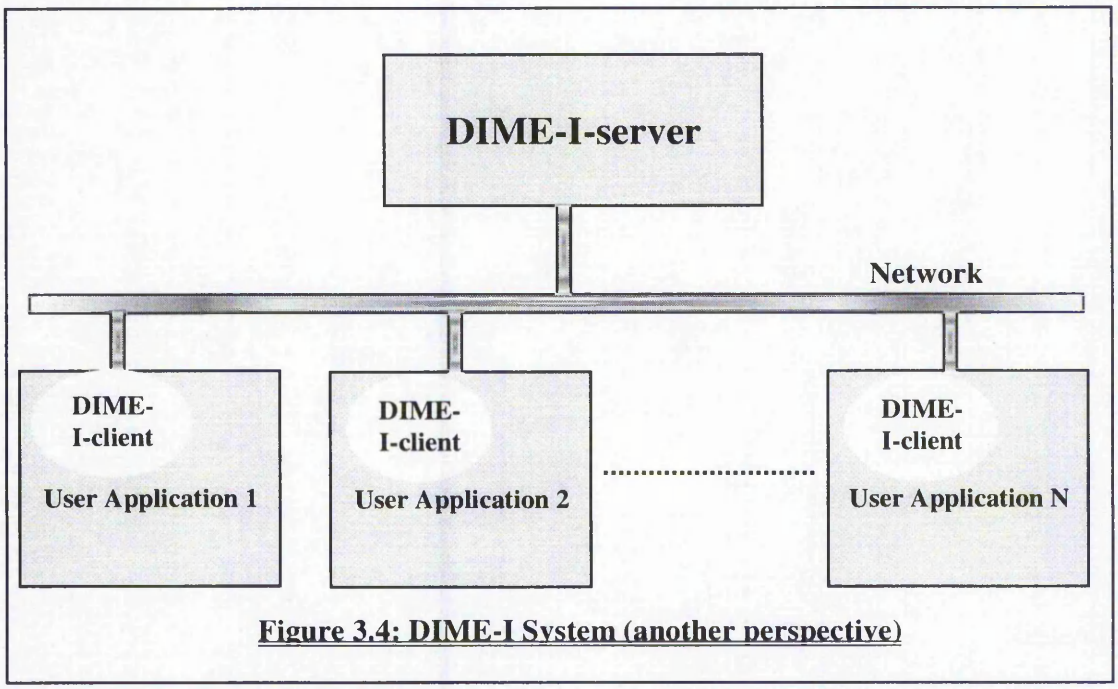


Figure 3.4: DIME-I System (another perspective)

As illustrated in figure 3.5, the process of DIME-I-client is placed in a separate layer along with an intermediate memory holding copies of part of the whole shared memory in terms of data areas and buffers; this new process is called DIME-II-client. The intermediate shared memory contains the data required by the user application that uses the services of DIME-II system through its DIME-II-client. In the architecture, the shared memory consists of two types of data structures: data buffers that represent dynamic data, and data areas representing static data.

Typically, DIME-II-client tasks can be detailed as follows:

- Establishing a persistent connection with DIME-II-server.
- Saving updates in the intermediate memory as they are received from DIME-II-server.
- Sending updates to the server as they are received from its user application.
- Sending data to the user application when requested.

➢ DIME-II-client is responsible for requests retransmission when no acknowledgement has been received as a response from the server in a predefined timeout; moreover, it decides when to stop dealing with the server as it is supposed to be dead.

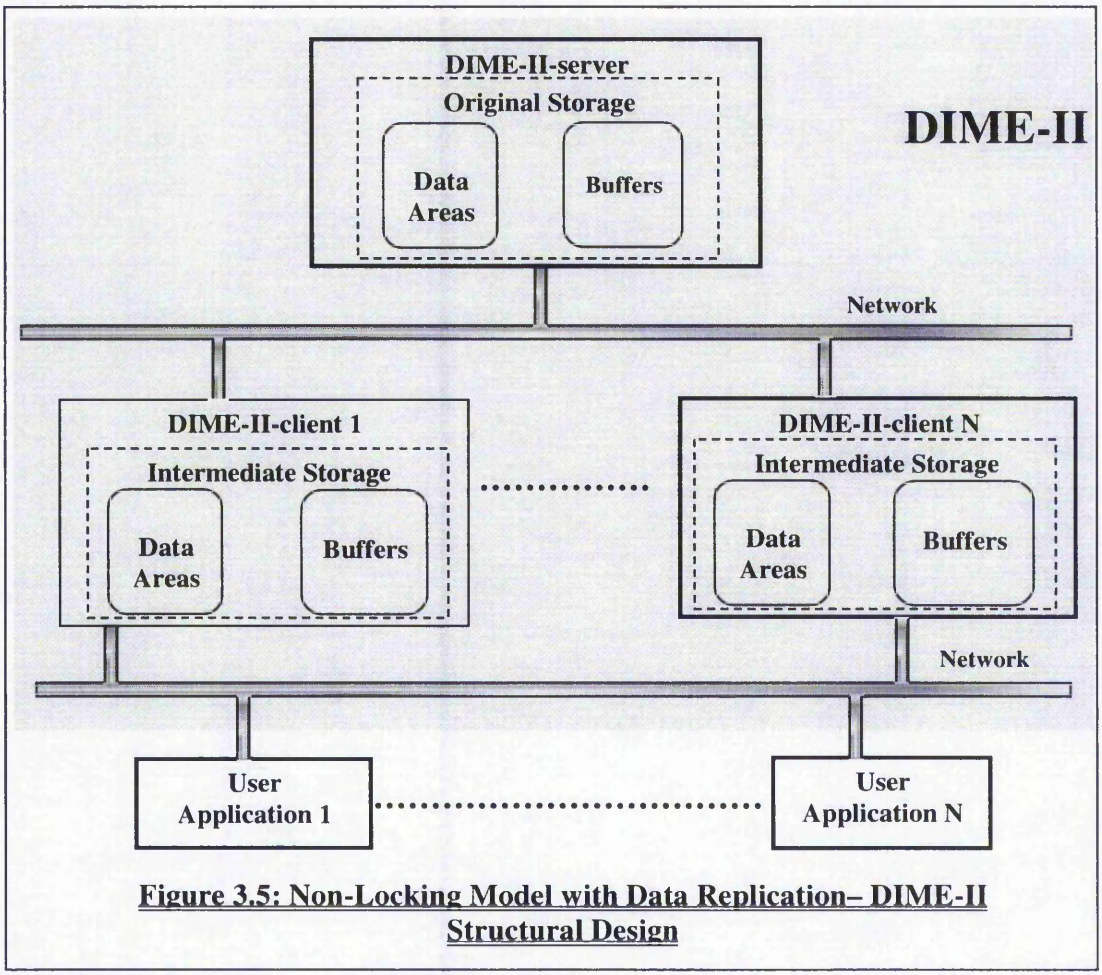


Figure 3.5: Non-Locking Model with Data Replication– DIME-II Structural Design

Thereby, a user application can perform any read or write operation on the intermediate memory rather than dealing directly with the main server, saving valuable time that can be used to perform its native tasks. In this framework, user applications; which are traffic control system modules; perform any operation on the local shared memory leaving the time delay burden of contacting the server to DIME-II-client for making the intermediate memory up to date, and reflecting updates on the original memory.

On the other hand, DIME-II-server takes control over the original shared memory system, and has the role of monitoring any modifications on the central memory in order to keep all intermediate memories throughout the system consistent with the updates - more details about how the whole system works is provided in section 3.9.

Unlike DIME-I, the shared memory is split over the network, and controlled by at least one memory manager. This support of the existence of several data replicas needs a special care to be taken into account to avoid data inconsistency in such architecture, which is described in the next section. One important feature inherited from the DIME-I, is that each read/write command is considered as a single atomic operation, therefore no resources locking takes place and the system relies on the natural sequencing of the commands occurring in TCP/IP-networked environment. In regard with this feature, and because there is no need for any explicit synchronization for an application to have an exclusive access on the shared memory, the improved architecture of DIME (i.e. DIME-II) is categorised as non-locking algorithm.

3.7 Memory Consistency Model for DIME-II.

According to their definitions, we have found that existing consistency models are not applicable to this architecture, particularly because they do not effectively support the nature of the two different types of data structures existing in urban traffic system. Therefore, a relaxed definition of sequential consistency (SC) [Lamport L. 1979] model has been designed to suit the new architecture, as the definition of SC is mostly expected by programmers. This section thoroughly prescribes the relaxed consistency model, and provides figures explaining and proving the semantics of the model.

As presented in [Khalil M. et. al 2003a], to maintain system-wide consistent views of the memory in terms of data area and buffer structures, the constraints below are imposed:

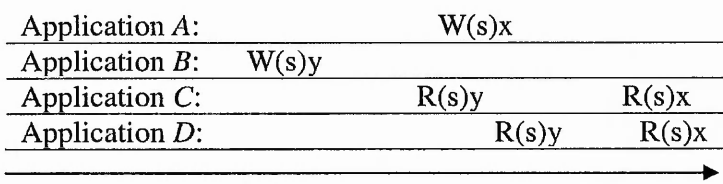
- *Accesses to data store controlled by the server (DIME-II-server) are sequentially consistent, and each read/write is performed as a single atomic operation.*

- *For data buffer; updates to a buffer are first sent to the server before being reflected to the applications that have replica of the updated buffer.*

- *For data area; an intermediate data area can be modified by its user application, locally, and then sends that update to the server, which in turn multicasts the update to the concerned applications*

The first constraint says that DIME-II-clients can access the original storage, controlled by DIME-II-server, in a sequentially consistent manner. In other words, the result of any execution is the same as if the read/write operations by all DIME-II-client on data store were executed in some sequential order and the operations of each individual DIME-II-client appear in the order specified by their program. Therefore, no resource locking takes place and the system will rely on the natural sequencing of the commands occurring in TCP/IP-based network environment (Figure 3.6). Moreover, as the system uses java language for implementation, the unpredicted behavior of threads in the Java Virtual Machine (JVM) has another impact on the sequencing of the commands. As it will be explained more thoroughly in a later section, each application is serviced at the main server by a separate thread which listens on a socket to commands from its application. When a thread has its turn to execute on the processor, it implicitly gains an exclusive access to insert a write operation command to be performed by the main server. This mutual exclusion is done implicitly by the JVM.

This constraint gives an exact and clear idea of how user applications view the shared memory of DIME-II system as a sequentially consistent memory. This direction of having the sense of sequential definition in the system has been followed by most relaxed and weak consistency models that have been presented to produce improved performance in the distributed shared memory prototypes, such as weak and lazy release models.



Where:

W(s)x ≡ Write the value x in the shared item s.

R(s)x ≡ Read the value y from the shared item s.

And,

W(s)y is seen by all applications in the system before W(s)x.

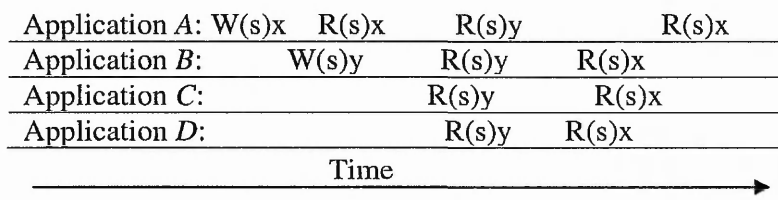
Figure 3.6: The Shared Memory obeys the Consistency Model

According to the second constraint, any update is first sent by a DIME-II-client to the DIME-II-server, which in turn performs it in the original memory and reflects that update to all user applications involved in that modification (i.e. the applications that have replica of the updated piece of memory) (Figure 3.8). This guarantees that updates of a buffer are viewed in the same sequence by all the applications that have copies of that particular buffer. For example, if two applications *A* and *B* send updates of values x and y to the same buffer, respectively, and occurs that the server receives the update y then x, subsequently all the applications in the system will see the value y first, followed by x (Figure 3.6). Thereby, data buffers are seen in the same order throughout the system, which is consistent with the first constraint.

With the third constraint, a user application can perform write operation directly in the intermediate data area and with no competition with other applications. After updating the local replica, the associated DIME-II-client program sends the update to the server which in turn multicasts it to be reflected in other intermediate memories that contain replica of the updated area (Figure 3.9). For instance, a value x is written in certain data area by an application *A*, followed by another update y in the same area but issued by another application *B*. Thereby, since DIME-II-server performs operations in the order they are received (not necessarily the order they were sent), all applications

will see the updates in the same order they are issued from the server. In other words, if *y* is received before *x* then applications will see y before *x* as well (Figure 3.6). However, a critical situation has to be considered at the application that issues the update, as this application will, undesirably, receive updates to this particular area in the same manner. More specifically, the application that has issued the update *x* will receive *y* executing it, and then executing *x* again, which means without considering this situation the application would see *x*, *y* and then *x* again leading to unacceptable inconsistency behavior (Figure 3.7).

To avoid this situation, DIME-II-server attaches to each write update on data area an ID; this ID is already known and saved by the application that originated the update. Each time it receives an update, the originating application compares the received ID with the existing one (if any). If they are not equal it discards the update as it is definitely older than the current one, which is still to be multicast by the server. Otherwise, it removes the current ID, and starts receiving further newer updates. If there is no saved ID, an application applies any received update with no delay (Figure 3.10).



Where:

W(s)x ≡ Write the value *x* in the shared item *s*.
R(s)x ≡ Read the value *y* from the shared item *s*.

Figure 3.7: The view of the shared memory (i.e. data areas) will be inconsistent with the definition of the presented consistency model if DIME-II-client directly applies the received updates

On the other hand, since each application has its own intermediate memory, read operations from either data area or buffer can be performed straightaway on the intermediate storage. That is, an application sends read requests to its DIME-II-client

that accordingly looks up in the memory and responds with the required data if it is available.

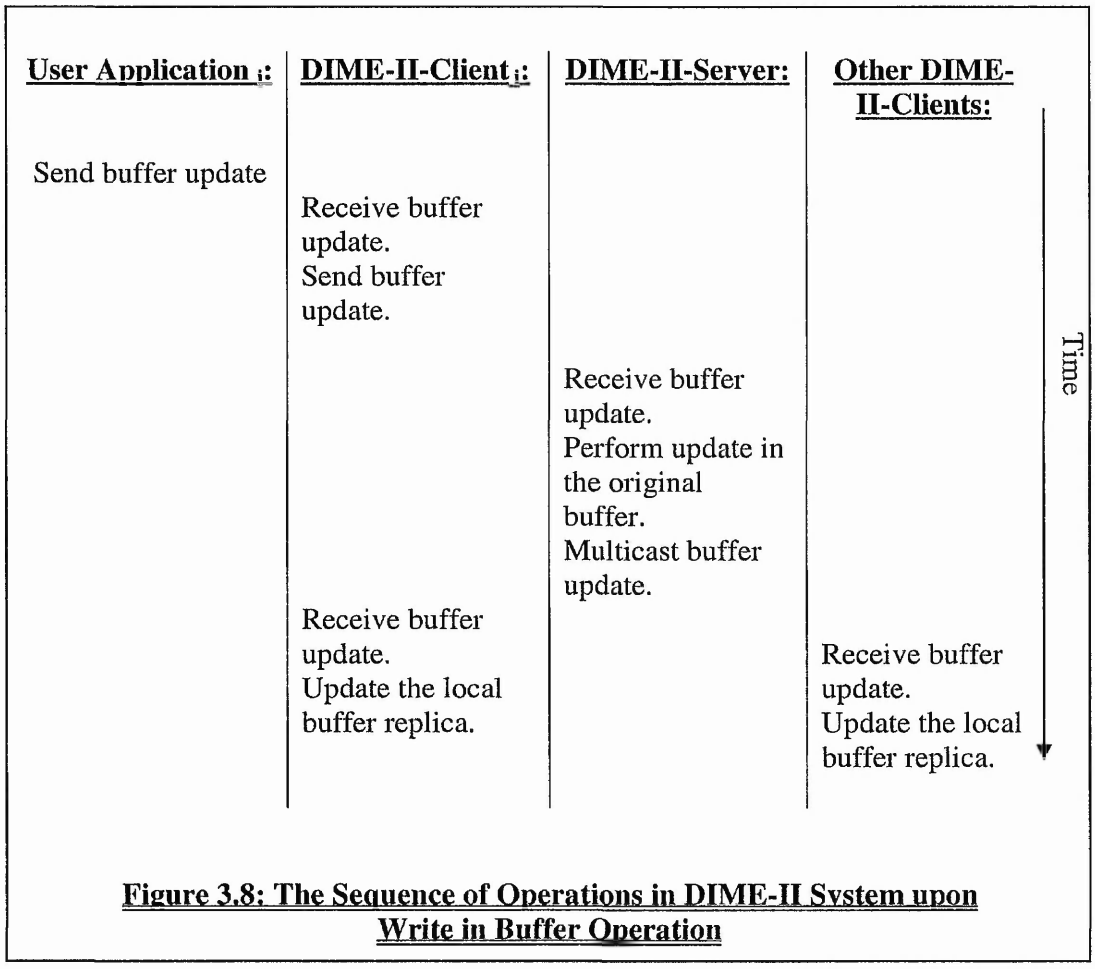


Figure 3.8: The Sequence of Operations in DIME-II System upon Write in Buffer Operation

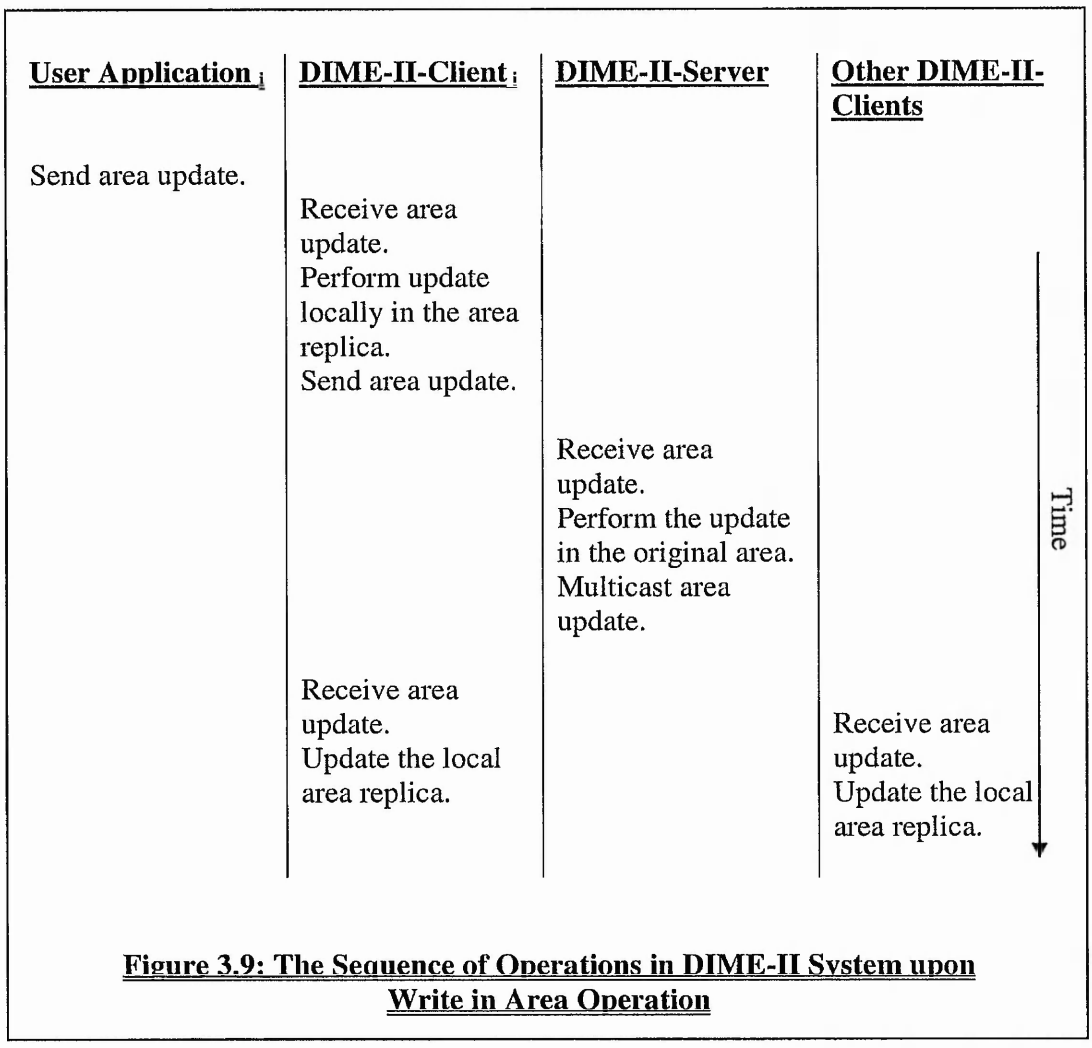
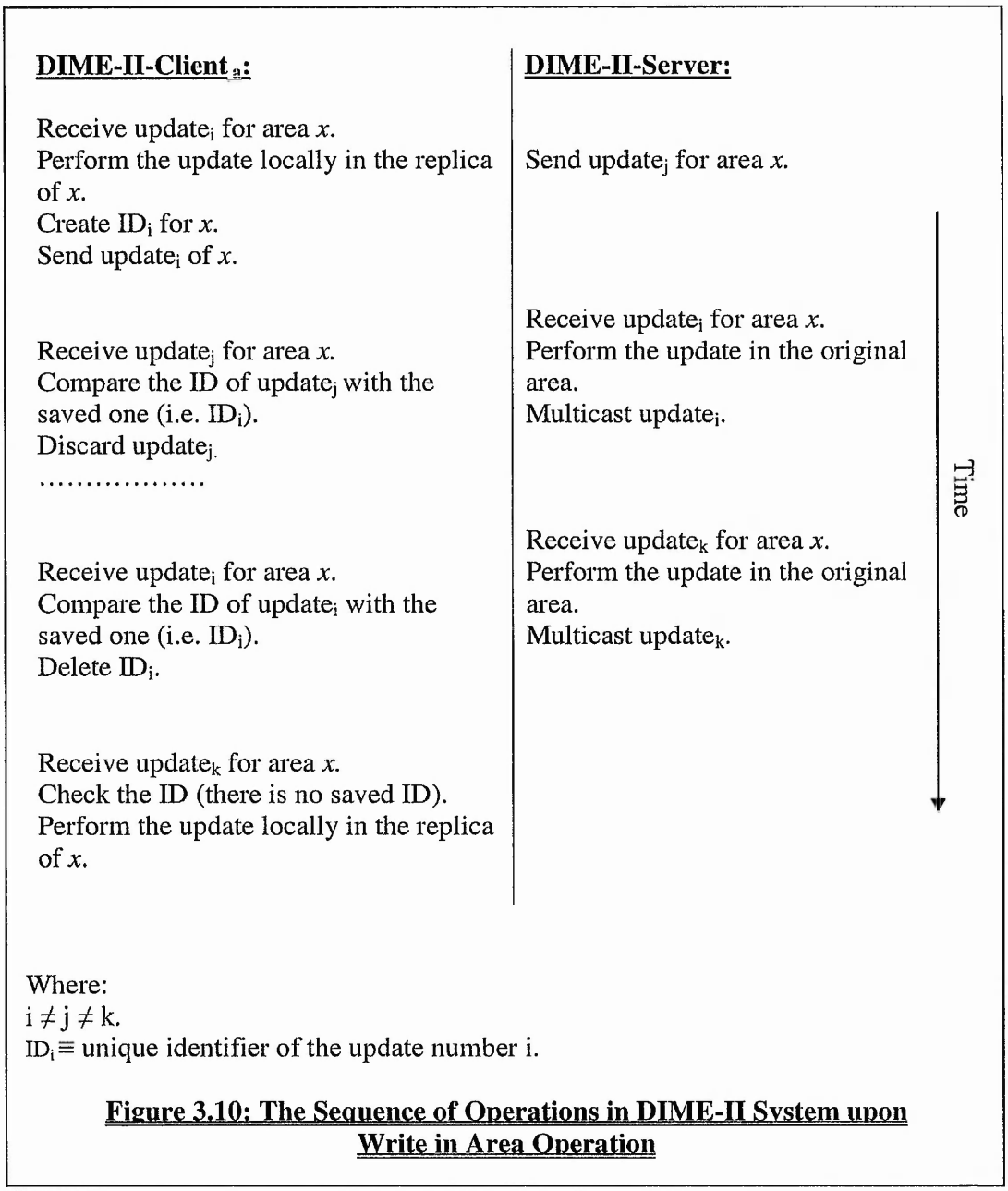


Figure 3.9: The Sequence of Operations in DIME-II System upon Write in Area Operation



Where:
$i \neq j \neq k$.
$ID_i \equiv$ unique identifier of the update number i.

Figure 3.10: The Sequence of Operations in DIME-II System upon Write in Area Operation

3.7.1 Evaluation to the Consistency Model

With the consistency model described in the previous section, it is clear that replicas can temporarily be out of date, which is the expense we always pay for such relaxed models; however, this model can guarantee consistent view of the shared memory all around the system at a point of time, preserving the concept of a distributed memory that is shared as a single non-physical memory. Furthermore, the system can provide each application with the required data, which are not necessarily the most recent ones- i.e. almost certainly can be on the way, but these data will be available at anytime they are requested. Therefore, user applications will always find the required data without significant delay, bearing in mind that they are retrieved directly from the intermediate memory. Providing an application with the requested data in a relatively short period of time allows more time for performing its main task.

This definition differs from the definition of sequential model in its support to data replication and its use of multiple writing mechanisms. These two approaches have been proved as a source for improved performance in distributed shared memory systems. On the other hand, it differs from other relaxed definitions; particularly weak, release and lazy release model; in such a way that it does not require an application to use synchronization mechanisms to have an exclusive access upon performing operations on the shared memory. One important feature in this model is it supports the presence of two different types of data structures of the traffic system, which makes it an application-driven model.

3.8 DIME-II Data Transfer Protocol (DDTP)

Implementing the non-locking approach with the consistency model requires an appropriate communication protocol in order to produce an efficient less complicated distributed middleware. This section presents a proprietary communication protocol designed specifically for DIME-II system. It is a sufficiently simple yet efficient protocol that allows correct implementation to the framework. As the algorithm disseminates updates only to the applications that have replica of the updated shared

data, this protocol ensures correct propagation for the updates to the specified applications.

DDTP protocol is a middleware-level protocol which is used to exchange messages and commands within DIME-II DSM system, allowing user applications to make read and write operations without being aware of the location of the data. DDTP employs positive acknowledgment policy, which means if the destination has not acknowledged the receipt of a packet (command packet) in certain amount of time (timeout); the protocol will re-send the packet for fixed number of times before declaring the destination is dead or offline. The necessity of having positive acknowledgement is explained when the protocol is evaluated later. This protocol encapsulates normal commands (write, read, delete ...etc.), acknowledgements, error and dummy messages, each in a separate unit called command packet, which is represented by array of bytes.

When an update for data area is sent, an identity (ID) is attached with the command packet. This ID allows applications to distinguish between their own updates on one hand, and between others', on the other hand. An ID consists of two parts: the name of the application, to make an update different from other updates of other applications. The second part is the serial number of the command packet itself, to make distinctive updates for the same area within one application. For example, an application sends an update for area and waits for it to be sent back as a confirmation for that update. In the mean time, if it receives any update for the same area, it simply discards it as it is certainly older than the current one (i.e. if it was newer than the expected one, the expected update should have already been received by that time and the ID should have been deleted). After receiving the confirmation it can apply any further updates for the same area. Therefore, this ID helps ensuring consistent view of data area replicas within the system.

DDTP is built on top of TCP/IP, requires no modifications in the underlying communication primitives, and it relies on the natural sequencing of the underlying network. Therefore, DIME-II-server executes write operations in the order they are received. This protocol differs from the protocol used in BDSM system in its use of a

multicast mechanism for propagating updates rather than broadcasting them, saving the resources of the network.

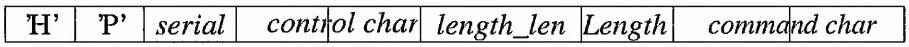
3.8.1 Structures of Command Packets Exchanged between DIME-II-server & DIME-II-clients

Command packets are classified into normal data, acknowledgement, error, redirection, and dummy messages. Each packet contains a control character to make it distinctive among other types of packets. In this section a complete definitions of all classes of command packets and their formats are provided.

3.8.1.1 Normal Data

This packet is sent for performing initiation, creation, destroying, and writing operations on the shared memory. The control character is **ND.**

The structure of the header:



Fields explanations:

HP: the start of the header.

serial: contains the serial number of the packet. This serial number ranges between 1 and 127, which is the range of positive numbers of byte type in java language. When the protocol reaches 127, it starts from 1 again.

control char: specifies the nature of a packet (normal, acknowledgement, error, redirection, or dummy massage), which is *ND* in this case.

Length_len: the length field in the header of the packet is represented by **String** object. The field *Length_len* is simply the length of that object.

length: the length of the packet excluding the length of the header.

c*ommand char:* specifies the nature of the command itself (creating, writing, removing ...etc).

3.8.1.1.1 Initiate DSM

command char: **'ID'.**

Header	*name_len*	*application name*

Description: DIME-II-client sends this packet to DIME-II-server requesting permit for initializing an intermediate shared memory. In other words, it is used to request permission to use the resources of DIME-II system. The command contains the name of the application.

As a response, if the user application is permitted to use the shared memory, DIME-II-server registers the name of the application in the list of the currently connected applications, and sends command packet of the same nature to DIME-II-client accompanied with the permission table (will be defined later) of the requesting user application, otherwise an error message will be sent.

Header	*permission table*

permission table: is an array of bytes as follow:

$SM\_n\_l_1$	SM_name_1	$perm_1$	$SM\_n\_l_2$	SM_name_2	$perm_2$	.....	$SM\_n\_l_n$	SM_name_n	$perm_n$

$SM\_n\_l_i$: is the length of the shared memory name (SM_name_i).

SM_name_i: is a name of shared memory item in the DSM.

$perm_i$: is a permission access code for certain shared item.

3.8.1.1.2 Create Area/Buffer

command char : **'CA'** or **'CB'**, respectively.

Header	*App_name_len*	*App_name*	*SM_name_len*	*SM_name*	*rec._length*	*no of recs*

Description: DIME-II-client sends this packet to DIME-II-server for creating new data area/buffer. This command is performed by DIME-II-client only if the application is permitted to use this particular area/buffer. DIME-II-client

examines the privilege of the application against the to-be-created item from the permission table received earlier with the permission to use the system. The command includes the name of the application, the name of the shared memory to be created, the length of the record, and the number of records if the created item is a buffer.

As a response, DIME-II-server registers this application in the list of applications that have replica of this particular item (i.e. this list is used by the DDTP's multicast algorithm when propagating update to certain shared item). If the area/buffer contains value, it sends a command packet of write operation to DIME-II-client along with the available data as initial value. Otherwise nothing is sent. This response includes the name of the shared memory, the number of the records in the packet, and the actual data.

header	*SM_name_len*	*SM_name*	*no of recs.*	*The actual data*

3.8.1.1.3 Destroy Area/Buffer

command char: **'DA'** or **'DB'**, respectively.

header	*App_name_len*	*App_name*	*SM_name_len*	*SM_name*

Description: DIME-II-client sends this packet to DIME-II-server to destroy an existing data area/buffer (i.e. the to-be-destroyed shared item has to be in the intermediate memory already). This command may result only in deleting the application name from the list of the applications that have replica of that part of the DSM if there is at least another application in the list. But if the application is the last one in the list, the list will be destroyed as well as deleting the shared item from the DSM permanently, as it is not longer needed by the current applications. This command includes the name of the shared memory to be deleted.

No reply from DIME-II-server will be sent in response to this command.

3.8.1.1.4 Write in Area/Buffer

command char : **'WA'** or **'WB'**, respectively.

header	*App_name_len*	*App_name*	*SM_name_len*	*SM_name*	*no_of_rec*	*The update*

Description: After examining that the application is permitted to perform the operation from the permission table, DIME-II-client sends this packet to DIME-II-server to update an existing area/buffer. In accordance to the consistency model presented earlier, the update will first be executed locally (i.e. writing in area).

As a response DIME-II-server sends command packet of the same nature only to the DIME-II-clients associated with the user applications involved in the update. This packet carries update to area/buffer in order to bring the local replicas up to date. This command includes the ID of the update to distinguish between different updates if the updated item is in the area space.

header	*SM_name_len*	*SM_name*	*no_of_recs.*	*ID_ len*	*updateID*	*The actual data*

3.8.1.2 Acknowledgement

Control char: **'AC'**.

'H'	'P'	*serial*	*control char*	*Len_length*	*length*	*serial number of the acknowledged packet*

Description: this packet is sent by DIME-II-server and DIME-II-client upon receiving packet message. This acknowledgement is used as a confirmation for receiving a packet from the other side of the communication channel in order to make the sending side to stop resending it, which complies with the definition of the positive acknowledgement scheme. The serial number of the acknowledged packet is included in the packet to be used by the recipient to recognize which packet is acknowledged.

3.8.1.3 Error Message

control char: **'ER'.**

'H'	'P'	*serial*	*contr*	*ol char*	*Length_Len*	*Length*	*error code*	*explanation _length*	*explanation*

Description: this packet is sent by DIME-II-server to DIME-II-client when an error occurs.

An error message contains a code number to explain the nature of the error. There are three different codes. Code 0, means that the application attempting to register in the system is not permitted to use the shared memory, whereas, code 1, means the application is already registered and is using the distributed shared memory (i.e. an attempt to register an application twice). Finally, when an error message sent with a code number of 2 that means there is an error in creating a new area/buffer. DIME-II system sends this message when an application attempts to register in the list of an existing item giving either incorrect record length or incorrect buffer length. Beside the error code, sometimes this message contains more explanation for the error. For instance, when there is an error in creating a shared item, the server sends the name of the item to the creator as an explanation.

3.8.1.4 Redirection Message

control char : **'RC'.**

'H'	*'P'*	*serial*	*contr*	*ol char*	*Length_ len*	*Length*	*No_of_servers*	*DIS servers_addresses*

Description: This message is used by the DIME-II-server to reconfigure the system using the reconfiguration algorithm (to be explained later in chapter four). The message contains the number of available servers and their addresses (IP addresses and port numbers).

Apart from the acknowledgment message the DIME client does not send any further message. However, when the new server starts to provide the service

to the redirected application it sends its name (the name of the new server) to the server to be used for later purpose (refer to chapter four).

'H'	'P'	*serial*	'A'	'M'	*Length_len*	*Length*	*Servername_length*	*DIS servers_name*

3.8.1.5 Dummy Message

control char: **'DM'.**

'H'	'P'	*serial*	*control char*

Description: this packet is exchanged by DIME-II-clients within the system for the purpose of calculating the round-trip time between each pair (will be explained further in chapter four).

3.8.2 Evaluation to the Communication Protocol DDTP

DDTP protocol is built on top of the transmission protocol TCP/IP. TCP/IP and most communication protocols rely on some algorithms to estimate the round-trip time, in order to handle the problem of messages loss, which partly make a protocol reliable. However, research has shown that the common approaches to estimating round-trip times for the TCP/IP are inaccurate if datagrams are lost or round-trip times are highly variable [Jain R., 1986], [Zhang L., 1986]. Therefore, it can be concluded that there is a possibility of message loss with TCP/IP. This kind of behavior is not tolerated in the traffic system and, certainly, most distributed systems do not permit that, as it is certainly crucial that every module in the system receives all the updates. For this reason, the DDTP employs positive acknowledgement mechanism to make sure that even if the TCP/IP fails to transfer a message, this problem is handled by resending the message until it is acknowledged by the other side of the communication channel. It is predictable that the cost of calculating the round-trip time in the DDTP is negligible, at least if compared with the cost of losing a message.

However, to avoid this problem, the common approaches have to be used once again to measure the round-trip time in order to estimate the appropriate timeout. After implementing DDTP in DIME-II, it has been noticed that there is a large number of unnecessary retransmissions of data and messages in DIME-II system. In the next

subsections this problem is explained and solved using some relatively new approach called karn's algorithm [Karn P. et. al 1991].

3.8.2.1 Formulating the Problem

The DDTP protocol uses TCP algorithm to predict the next round-trip time by sampling the behaviour of messages sent and averaging those samples into a smoothed round-trip time estimate, SRTT. The new SRTT is computed by the formula:

$$SRTT_{i+1} = (\alpha \times SRTT_i) + (1 - \alpha) \times s_i \qquad [1]$$

Where:

$SRTT_i \equiv$ The current estimate of the round-trip time.

$SRTT_{i+1} \equiv$ The new computed value of the round-trip time.

$s_i \equiv$ Sample of round-trip time.

$\alpha \equiv$ Constant between 0 and 1 that controls how rabidly the SRTT adapts to change.

The retransmission time-out, RTO, is the amount of time the sender will wait for a given message to be acknowledged, and is computed from $SRTT_i$ using the formula below:

$$RTO_i = \beta \times SRTT_i \qquad [2]$$

Where:

$\beta \equiv$ Constant greater than 1, chosen such that there is an acceptably small probability that the round-trip time for the datagram will exceed RTO_i .

A crucial decision for computing adequate RTO is an accurate measurement of the true network round-trip times. In other words, using an appropriate sampling method for the round-trip time gives accurate RTO. There are different sampling techniques for measuring the RTT each has its advantages and disadvantages. DDTP used a sampling technique that computes the RTT from the most recent transmission of a message. Put in

other words, when an application sends a message the time of sending this message is recorded by the DDTP, and it waits for an acknowledgement from the other side of the communication. Whenever an acknowledgement is received, DDTP records this time and uses it with the first recorded time to compute the new sample, and the SRTT is then computed. Finally, the RTO is computed to predict the time slot where the next message is expected to be acknowledged. However, if an acknowledgement arrives after the RTO has elapsed and a retransmission has been sent, the new sample will be incorrect, as according to the method, it will be computed using the time of sending the second transmission of the message with the time of receiving the acknowledgement of the first transmission, which is incorrect.

The implicit assumption in this method is that the RTO is accurate; if a datagram has to be retransmitted then previous transmissions have almost certainly lost. This assumption is often false. If the RTO is smaller than the true round-trip time, acknowledgment for previous transmissions may arrive after a retransmission [Karn P. et. al 1991]. In other words, if an acknowledgement has not arrived by the expiration of the RTO, it is highly likely to come very shortly afterwards. This method gives bad samples of round-trip time if the delay of the acknowledgement is due to network delay rather than datagram loss. Therefore, SRTT drops leading to inaccurate RTO resulting in numerous unnecessary retransmissions and the network bandwidth is wasted. Therefore, using only this method by the DDTP caused the large number of unnecessary retransmissions of messages in DIME-II system.

3.8.2.2 Back-off Technique

Every TCP implementation increases the RTO by some factor before retransmitting the unacknowledged data. Some implementations simply double the RTO for each consecutive attempt. If the larger RTO expires before the retransmitted data is acknowledged, the RTO is increased further. This technique is known as *back-off*. This technique is essential in keeping the network stable when sudden overloads cause messages to be dropped [Jacobson V., 1988]. Whenever the overload condition

disappears, datagram loss stops and the TCPs reduce their RTO to their normal SRTT-based value.

3.8.2.3 Karn's Agorithm

It is another sampling method based on avoiding any sample that is contaminated by retransmission ambiguity. It ignores any round-trip time sample for messages that have been retransmitted. The rule of the algorithm is:

> *When an acknowledgement arrives for a datagram that has been sent more than once (i.e. retransmitted at least once), ignore any round-trip measurement based on this datagram, thus avoiding the retransmission ambiguity problem. In addition, the backed-off RTO for this datagram is kept for the next datagram. Only when it (or a succeeding datagram) is acknowledged without an intervening retransmission will the RTO be recalculated from SRTT.*

This algorithm guarantees measuring SRTT from only good samples of round-trip time and, at the same time; it uses the back-off technique to adapt to the current state of the network. Karn's algorithm ensures that the new accurate round-trip measurement will be taken and fed into the SRTT estimate regardless of any sudden increase of round-trip delay.

Karn's algorithm has been heavily used in TCP implementations and achieved good results on perhaps the worst medium ever used to pass IP datagrams: amateur packet radio [Karn P. et. al 1985].

3.8.2.4 Implementing Karn's Algorithm in the Communication Protocol

To avoid the large amount of unnecessary retransmissions of messages in DIME-II system, karn's algorithm has been used in the communication protocol DDTP. How quickly the SRTT converges to the new round-trip time depends on the back-off algorithm and the SRTT smoothing algorithm. The communication protocol implements the algorithm with the suggested algorithms and values in formula 1. It uses the formula

number (1) for calculating SRTT, and back-off algorithm with factor of 2. It uses typical values of $\alpha = 0.875$ and $\beta = 2$ in the SRTT smoothing algorithm.

3.8.2.5 Experimental Results

To evaluate the impact of karn's algorithm on the number of exchanged messages in DIME-II, several experiments were launched. The experiments used two kinds of simple codes to evaluate the performance of the communication protocol. The first code, *writer*, continuously performs write operations on the shared memory, whereas, the second code, *reader*, keeps reading from the shared memory. As it can be perceived, such codes make a significant overhead on the network by making it busy all the time, which is due to the fact that the read/write operations require a persistent communication and data exchanging between DIME-II-server and its DIME-II-clients. The duration of the experiment was 24 hours. In the experiment, DIME-II system was considered as two sides of communication: DIME-II-client and DIME-II-server, each side executes *send* and *receive* operations.

These experiments were conducted on the university LAN during a working day (i.e. including peak and off-peak times). The DIME-II-server was placed on UNIX-based machine, and the applications were executed on three Windows2000-based machines. The machine where the server runs was linked to the rest of the machines by five switches. The evaluation was based on different workloads; each workload executes only one writer and different number of readers.

The first experiment intended to count frequencies of message retransmissions in DIME-II. The results showed that the maximum number of data retransmissions at DIME-II-client was only about 0.5% of the total number of messages sent from DIME-II-client. At DIME-II-server, the maximum number of data retransmissions was only about 0.15% of the total messages sent by DIME-II-server. Therefore, in the whole system a maximum of 0.28% of the total of messages exchanged in 24 hour experiment was retransmissions of data, which is trivial.

Another experiment was for specifying when the number of data retransmission reaches its maximum level. It showed that in the set of time (2.00pm – 6.00pm) the number of message retransmission reached its maximum levels due to network overload (i.e. the peak time in the university network).

Therefore, it can be concluded that the communication protocol of DIME-II, using karn's algorithm, exchanges the least number of messages and data. Therefore, the network bandwidth is saved by this algorithm in DIME-II system.

3.9 An Implementation for DIME-II Framework

The first decision has to be taken at this stage is where to locate the process of DIME-II-client in the framework, as the general structural design of the DIME-II has not identified its location. The implementation presented here chooses to place DIME-II-client process at the same machine as its associated user application, in order to achieve the sought goals in terms of reducing the data retrieval time from the viewpoint of the user applications. Therefore, slight modification to the framework in figure 3.5 has to be made as illustrated in figure 3.11.

For implementing this framework, two separate executables have been coded for DIME-II-server and DIME-II-client [Khalil M. et. al 2003c]. Both make use of the Java programming language. Java's multithreaded support is essential for the successful programming of the DIME-II software. The presented implementation exploits the potential of multithreading as it has shown improved performance in DSM systems by hiding the long communication latencies typically associated with software DSM systems [Speight E. et. al, 1997] [Mueller F., 1997].

The DIME-II software reflects the new architecture with the new consistency model and the DDTP communication protocol. The produced software is described comprehensively in the following subsections in terms of DIME-II-server and DIME-II-client.

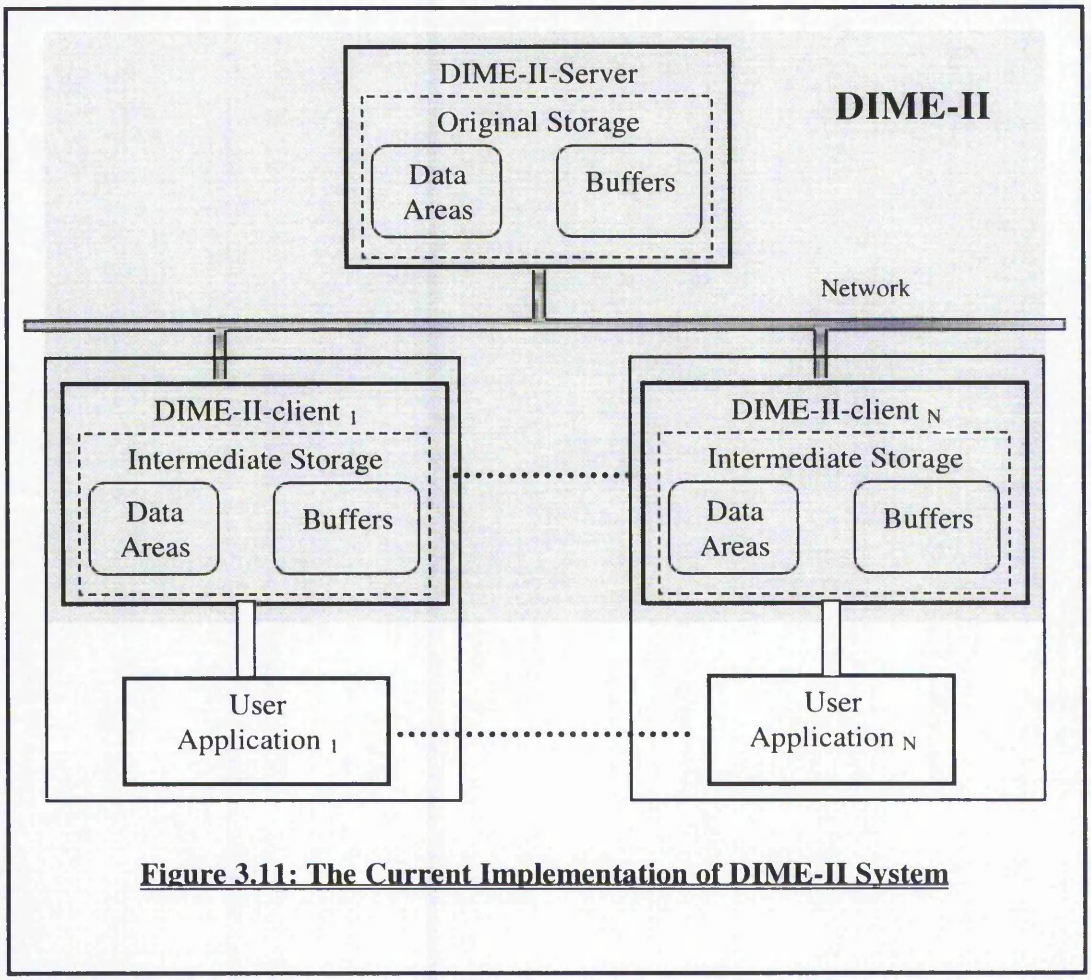


Figure 3.11: The Current Implementation of DIME-II System

3.9.1 DIME-II-server

DIME-II-server executes command packets in the order they are received (not necessarily in the order they were sent). In accordance with the atomicity of DIME-II system, each command is performed as an indivisible operation. In other words, there is no interleaving when DIME-II-server is performing a command.

DIME-II-server keeps a list of applications that are permitted by the administrator of the system to use DIME-II system. This list is used when a request for using and initiating the shared memory is received. If the application name is in the list then the permit is sent along with the permission table that contains the privileges of the application to use the shared memory. Otherwise, an error message is sent. Also, DIME-

II-server keeps a list for each shared item available in the shared memory, which contains names of user applications that currently have replica of it. When it receives a request for creating shared item, DIME-II-server allocates space for the item in the DSM only if it has not been created yet, and the name of the requesting application is added to the list of the applications that have the replica of that shared item. In the case of write operations, it sends updates only to the applications that have replica of the updated value using the relative list of applications. In other words, unlike BDSM [Auld P. 2001], DIME-II system employs a multicast-based algorithm to disseminate updates to the application that are involved in the write operation. On the other hand, when it receives a request for the deletion of a shared item, DIME-II-server deletes the name of the application from the list of that item. The item is removed permanently only if the requesting application is the last one in the list.

In order to improve the performance of DIME-II-server, and to make benefit of overlapping communication with computation, numbers of threads are used. Each application is serviced by separate thread that listens to its requests and inserts them in a queue of command packets waiting for process. This thread is called C*lientService* and is created when DIME-II-server receives request from a user application to use the shared memory, along with an empty queue to be used later for saving commands that that are to be sent to the user application. This thread keeps checking this queue of command packets and sends any available command to the user application. *ClientService* thread contains another thread, called *ListenToPacket,* which continuously listens to command packets sent from its user application and inserts them in a queue to be processed later. This queue is processed by another thread called S*equencer.*

The *sequencer* is the only thread that can read from that queue and perform operations on the shared memory in DIME-II-server, and it is called so because the sequence of its reading and execution of commands will be the order of updates appearance to all applications in the system. After processing an operation, the *sequencer* passes an appropriate command packet to certain *clientServices* through their command packet lists, which in turn send the command to their applications. Commands are executed by the *sequencer* in the order they are received. This order depends on two

factors: the natural sequencing of messages occurring in the underlying TCP/IP-based network, and the unpredicted behavior of threads in the java virtual machine (JVM). Recall, each application is serviced at DIME-II-Server by a thread called *ClientService.* When a *ClientService* thread has its turn to execute on the processor, it gains an exclusive access to the queue of commands, processed by the *sequencer*, before it inserts a command in that queue. This mutual exclusion is done implicitly by java language as there is no need to perform any special command to have an exclusive access to the queue.

Employing several threads allows dividing the task of the DIME-II-server into a number of sub-tasks to be executed at the same time, enhancing the functionality of the DIME-II-server. The speed of the sequencer performing commands in the shared memory (read & write) is much higher than the speed of the underlying network and therefore this is not a cause for bottleneck problem. Figure 3.12 illustrates a general view of DIME-II-server.

As mentioned earlier, in addition to the main task of controlling the shared memory, DIME-II-server holds number of permission tables. These tables prescribe levels of access that each user application has on the shared memory. The level of access is either no access, read only or read/write. DIME-II-server disseminates certain permission table to a DIME-II-client upon initiating a process of traffic module. This permission table is used by DIME-II-client upon performing any operation on the shared memory to examine the privileges of the application beforehand.

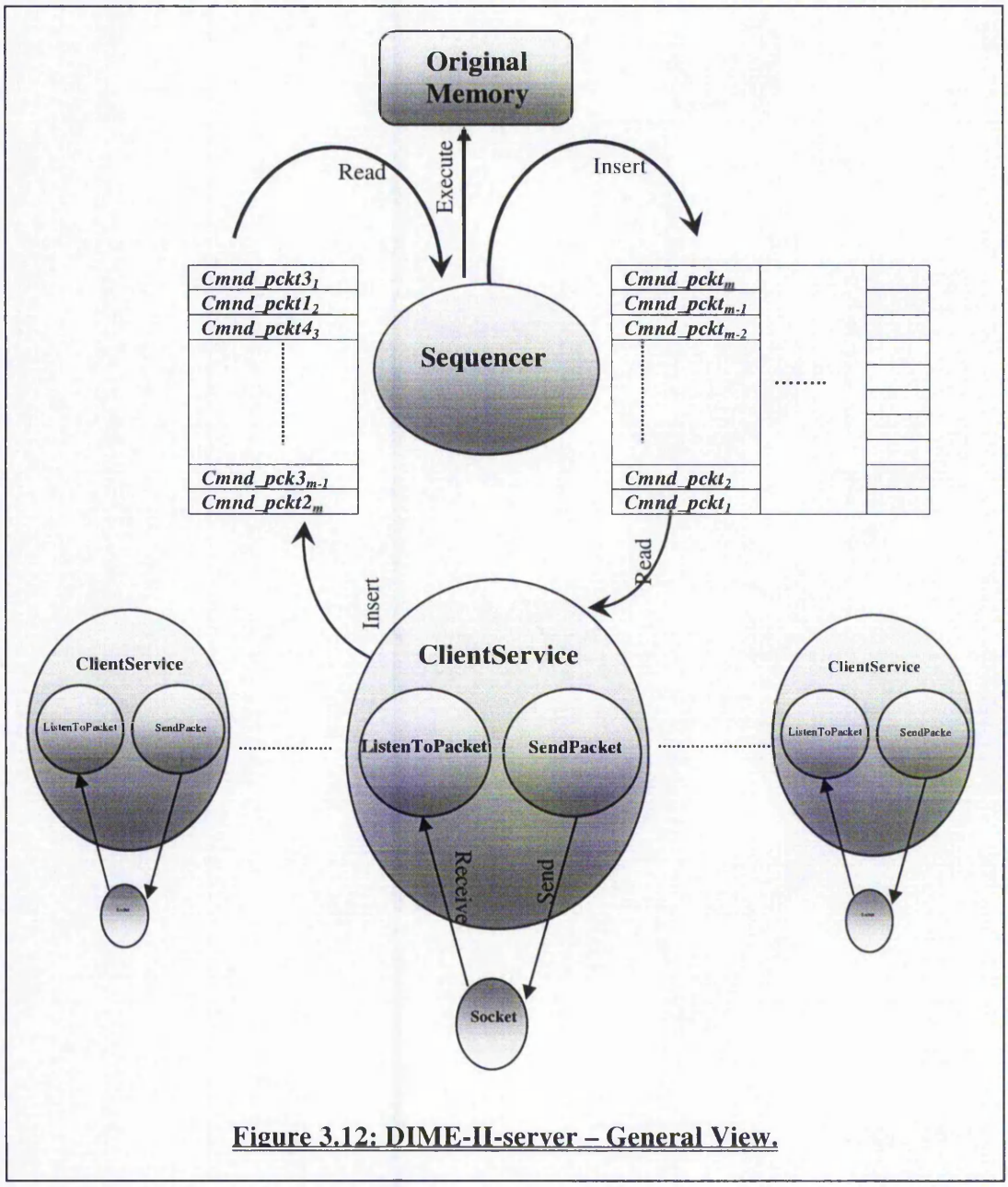


Figure 3.12: DIME-II-server – General View.

3.9.2 DIME-II-client

DIME-II-client keeps a copy of the permission table of its user application. It checks the privilege of its user application upon performing any operation on the intermediate memory. An operation on the shared memory is processed only if the application is permitted to do so, otherwise an error message is sent back. For write operation, DIME-II-client applies the update locally in the case of area writing, and then the update is sent to the DIME-II-server.

At DIME-II-client side, there is a thread that continually listens to messages from DIME-II-server and acknowledges them. The main task of this thread is to receive new updates from the DIME-II-server and then update the intermediate memory accordingly. Thereby, DIME-II-client can guarantee the consistency of the local replicas of the shared memory. On the other hand, any read operation can be performed directly on the available local memory with no need to contact the main shared memory controlled by the DIME-II-server over the network. This locality of reference is advantageous in saving network bandwidth, and reducing time of data retrieval for user application.

Therefore, this implementation of the framework can improve the performance of the whole system in many aspects: saving network resources; reducing data retrieval from user application viewpoint; performing number of tasks per-node simultaneously; and at the same time maintaining consistency by a simple straightforward model. Unlike TreadMarks system, this implementation of DIME-II does not require any synchronization mechanism for a user application to have an exclusive access to the shared memory, since each user application is associated with an intermediate memory where all its operations are performed via its DIME-II-client. Similar to this implementation, in BDSM system each user process has an associated DSM subsystem that manages the shared memory, however, each user process in BDSM has a complete copy of the shared memory where it processes all reads and writes locally. Also, unlike our system, in DIME-II all writes to memory modify the local copy and arrange to broadcast the updated values to all the other processes. Another major difference with

the presented implementation is that, BDSM allows only one user process to be executed on a workstation [Auld P. 2001].

3.9.3. User's Interface of DIME-II Software

The produced system provides number of functions to be used by distributed system programmers for performing different essential operations on the distributed computers shared memory. These functions are:

- Initializing intermediate shared memory. User application calls this function to get permission for initializing the shared memory and start using the DSM. If the application has a permission to use the system, a permit will be sent along with a permission table. This permission table contains names of shared items the application is permitted to use, and the access privileges for each item.
- Creating data area/buffer. This function is called to create a new shared memory item. A shared memory item is created only if the creating user application is permitted to use it.
- Writing in area/buffer. A user application invokes this function to update an existing shared item. This operation is performed only if the user application has enough permission to update that item.
- Reading from area/buffer. The required data are sent to the requesting application along with the number of the sent values. Zero is sent if the requested item is empty. This operation is performed locally.
- Destroying area/buffer. Performing such operation results in removing certain shared item from the intermediate memory of the requesting application.

3.10 Evaluating the Performance of DIME-II System in Comparison with DIME-I System

Experiments were launched to quantify the performance of DIME-II system in comparison with the old system DIME-I. Both systems employ a non-locking algorithm and no synchronization mechanisms are needed in both systems for having exclusive access to the shared memory, but unlike DIME-I, the non-locking approach allows applications in DIME-II system to have replicas of the shared memory locally. This approach is expected to speed up data retrieval rates in DIME-II system. Therefore, in these experiments the data retrieval time from the viewpoint of the applications is considered as a major measurement factor for the performance. The motive behind this consideration is that the user application can have more time for performing its native tasks, which time is very often wasted in network communication.

3.10.1 Experiment Benchmark

The experiments used two kinds of simple codes to compare the performance of the two systems. The first code, writer, continuously makes write operations on the shared memory, whereas, the second code, reader, keeps reading from it. As it can be perceived, such codes make a significant overhead on the system, and make the network busy all the time, as the read/write operations require a persistent communication and data exchanging between DIME-II-server and its DIME-II-clients. Additionally, it can provide a good idea about how the system performs with highly demanding applications.

The codes were executed on three Windows2000-based machines, while the server was based on a UNIX-based machine. The experiments were conducted on the university LAN at different times of the day (i.e. included peak and off-peak time). The evaluation was based on different workloads; each workload executes only one writer and different number of readers. Workload 1 means one reader and one writer, and workload 2 means two readers and one writer, and so on.

3.10.2 Evaluation and Results Summarization Scheme

To evaluate the performance of the DIME-II system with DIME-I, a method of confidence interval is used [Weisong S. et. al 1998]. The basic idea behind the use of this method is that a definite statement can not be made about the characteristics of all DSM systems, but a probabilistic statement about the range in which the characteristics of most systems would drop can be made. The variety of applications determines that it is not possible for one system to be better than others in all cases [Adve S. et.al 1996]. To explain the basics of the method, assume that there are two bounds c_1 and c_2, wherein a mean of population of observations, μ, drops by a high probability, $1 - \alpha$. It can be said that the population mean is in the interval (c_1, c_2):

$$\text{Probability } \{c_1 \leq \mu \leq c_2\} = 1 - \alpha$$

The interval (c_1, c_2) is called the confidence interval for the population mean, α is called the significance level, and $100\ (1 - \alpha)$ is called the confidence level.

The confidence interval method uses the central limit theorem. This theorem states that the sum of large number of independent observations from any distribution tends to have a normal distribution. The 100(1-α)% confidence interval for an experiment of less than 30 samples (i.e. observations) is given by the formula:

$$\left(\bar{x} - t\,[1 - \alpha/2;\, n - 1]\; s / \sqrt{n},\ \bar{x} + t\,[1 - \alpha/2;\, n - 1]\; s / \sqrt{n} \right)$$

where:

$\bar{x} \equiv$ is the sample mean.

$s \equiv$ the sample standard deviation.

$n \equiv$ the sample size.

$t\,[1 - \alpha/2;\, n - 1] \equiv$ the $(1 - \alpha/2)$-quantile of a t-variate with n-1 degree of freedom.

The interval is based on the fact that for samples from a normal population N (μ, σ^2), ($\bar{x} - \mu$) / (σ - $\sqrt{n}$) has a N (0, 1) distribution and (n – 1) s^2 / σ^2 has a chi-square distribution with n – 1 degrees of freedom, and therefore, ($\bar{x} - \mu$) / $\sqrt{(s^2/n)}$ has a t distribution with n – 1 degrees of freedom.

There are two types of confidence interval methods: paired, and unpaired. The former is used for evaluation when there is one-to-one correspondence between the tests made on the compared systems; otherwise, the unpaired scheme is used. Our evaluation uses the paired confidence interval method, as the two systems employ the same paradigm of user-level software distributed shared memory, with the difference only in the framework. Furthermore, the same benchmarks are used in the executions of the two systems.

3.10.3 Comparing the Performance in Terms of Data Retrieval Rates from the Viewpoint of System Modules

The separate executions of the two systems with the codes on different workloads yield the results shown in table 3.1. The results have shown that with DIME-II system, an application can retrieve more data from the shared memory comparing to the old implementation of DIME-I. With DIME-II, data retrieval rate ranges between 5.3 Kilo Byte/ second and 26.3 Kilo Byte/ second, whereas, with DIME-I, it has a range of 1.7-2.7 KB/second. However, the performance chart of DIME-II (figure 3.13) shows that the performance of the system deteriorates sharply as the number of the running applications increases. Therefore, the next experiment is to find out the reason of such undesirable scalability (section 3.10.4).

3.10.3.1 Summarizing and Evaluating the Results using Confidence Interval Method

In this evaluation, the steps of the paired confidence interval method are literally followed.

- The differences between the two systems are:

23.6 20.8 19.2 13.9 10.7 10.6 3.6

- Sample mean = 14.6
- Sample variance = 49.2
- Sample standard deviation = 7.0
- The 0.95-quantile of a t-variate with 6 degrees of freedom is 1.9.
- 90% Confidence interval for difference = $14.6 \pm t\sqrt{(49.2/7)} = 14.6 \pm 2.65 \times 1.94 = (9.459, 19.74)$

Workload	DIME-I	DIME-II
Workload1	2.7	26.3
Workload2	2.5	23.4
Workload3	2.4	21.6
Workload4	2.3	16.3
Workload5	2.3	12.9
Workload6	2.0	12.6
Workload7	1.7	5.3

Table 3.1: The Performance in DIME-I and DIME-II Measured as Data Retrieval Rates (kilobytes/second)

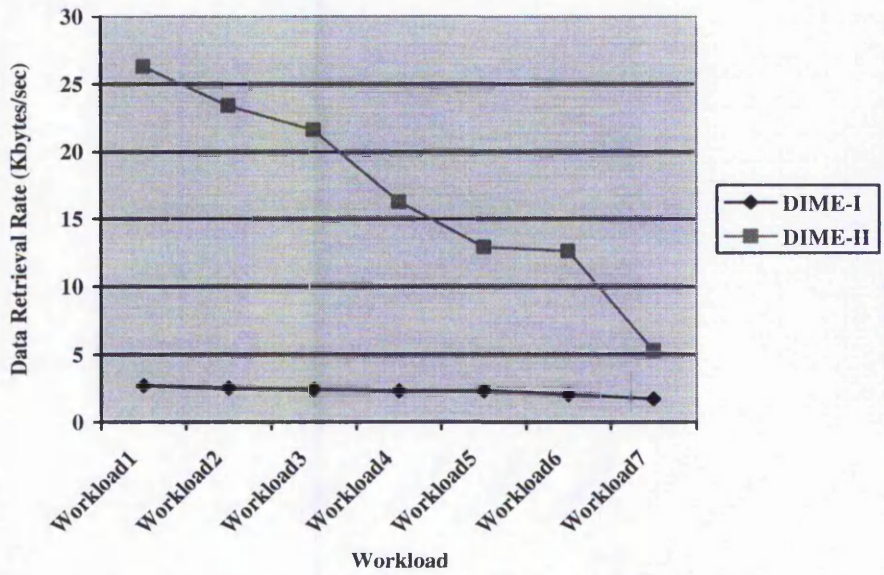


Figure 3.13: The Performances of DIME-II in Comparison with DIME-I

According the paired confidence level method we can draw the conclusion that with 90% confidence level DIME-II is better than DIME-I considering that the performance is measured as data retrieval rates from the viewpoint of the system modules. However, the results show that, unlike DIME-II, there is no great variation in the rates using DIME-I system as the number of applications increases. Therefore, DIME-I scales better than DIME-II.

3.10.4 The Time (in milliseconds) DIME-II Server spent in listening to Messages from the Network

In order to find out the reason of the high variation in data retrieval rates in DIME-II system, and based on the fact that the time of computation is far less than the time of communication, the time that DIME-II-server spent on the socket listening and waiting for messages from the network is roughly measured (table 3.2).

This experiment shows that DIME-II-server approximately spends 72%-94% from the execution time listening to the network-depending on the state of the network, which means it has few time to execute commands on the shared memory. Put in other words, as the number of applications in the system increases the rates of data retrieval decrease due to the fact that DIME-II-server always has the same few amount of time to share out between the increased numbers of applications. In general, implicit waiting is one of the factors that degrade the performance of parallel-processing systems [Lai A. et. al 1992].

Workload	Time (millisecond)	Time%
Workload1	238747	79.6%
Workload2	269753	89.9%
Workload3	282269	94.1%
Workload4	217572	72.5%
Workload5	243709	81.2%
Workload6	274755	91.6%
Workload7	234964	78.3%

Table 3.2.: The Time DIME-II-server Spent Listening to Messages from the Network

Although multithreading is used essentially to hide the latency of communications in DIME-II DSM system, this experiment shows that as the size of the system (i.e. the number of connected and running applications) increases, the increased number of spawned threads leads to the appearance of the latency again and consequently the undesirable scalability. Recall, in DIME-II system, DIME-II-server spawns two threads upon registering an application to use the system; these two threads are dedicated to respond to the requests of that application. Unfortunately, creating two threads for every application in the system is inevitable in order to provide applications with recent updates as they occurred on the server (i.e. to maintain consistency all over the system), which makes the network overloaded with messages that carry updates. In java-multithreaded systems, the behavior and throughput of threads can not be predicted, even though we can arguably say that the workload of the threads that respond to different applications in the system is the same. Though, as the size of the system becomes large the likelihood of performance deterioration in DIME-II system becomes unavoidable. Therefore, in order to improve the performance of DIME-II, another

technique is needed to reduce the negative effects of multithreading and boost the scalability of the system. An enhancement technique that works alongside other approaches and algorithms used in this system is presented in chapter four.

3.10.5 The Time (in milliseconds) an Application is blocked while performing Read Operation on the Shared Memory

These experiments are for measuring the time that an application remains blocked while performing read operation on the shared memory in terms of data areas and buffers. In particular, read operation time is namely measured to verify whether the new framework with the adoption of a non-locking model realizes the sought goal of minimizing the time of data retrieval from the viewpoint of the applications in the system.

The results showed that DIME-II provides data from the shared memory in a time far less- and even negligible (less than a millisecond) - than its analogous DIME-I (tables 3.3 & 3.4). Although DIME-I scales better than DIME-II, due to the locality of reference in DIME-II and the data are retrieved from a local storage, DIME-II system still provides data in less than a millisecond allowing applications to perform more read operations on the shared space and to retrieve more data. Moreover, in DIME-II, retrieving data from the shared memory takes less than 40% out of the total execution time, whereas DIME-I data retrieval operations take about less than 80% from the execution time (tables 3.5 & 3.6). Therefore, using DIME-II systems, an application is allowed more time to perform its native tasks realizing the main objective of the new framework.

Workload	DIME-I			DIME-II		
	Time	*Number of Commands*	*Time / Command*	*Time*	*Number of Commands*	*Time / Command*
Workload1	143177	710	201.66	104191	6825596	0.02
Workload2	140791	672	209.51	107947	7024775	0.02
Workload3	139187	677	205.59	66391	3996488	0.02
Workload4	141740	701	202.34	47032	1810092	0.03
Workload5	134709	635	212.01	35840	1534828	0.02
Workload6	129535	592	218.93	29720	1228711	0.02
Workload7	119940	513	234.03	15150	376023	0.04

Table 3.3: The Time (milliseconds) an Application Remains Blocked while Performing Read Operation on Data Areas

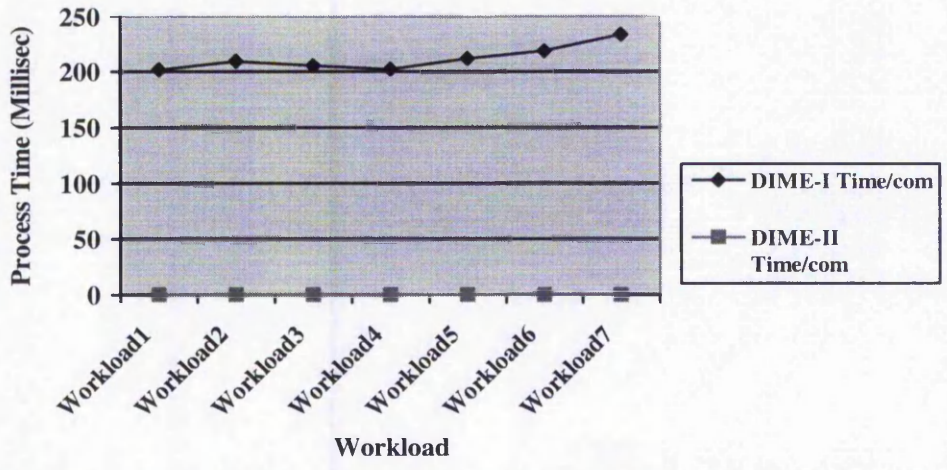


Figure 3.14: The Time (milliseconds) an Application Remains Blocked while Performing Read Operation on Data Areas

Workload	DIME-I			DIME-II		
	Time	*Number of Commands*	*Time / Command*	*Time*	*Number of Commands*	*Time / Command*
Workload1	143119	710	201.58	2070	116181	0.02
Workload2	140418	672	208.95	1685	10379	0.16
Workload3	128808	642	200.53	1882	9599	0.20
Workload4	116730	624	186.99	1654	7221	0.23
Workload5	119638	608	196.64	1442	5742	0.25
Workload6	109724	543	202.01	1623	5561	0.29
Workload7	98175	467	210.14	918	2324	0.40

Table 3.4: The Time (milliseconds) an Application Remains Blocked while Performing Read on Data Buffers.

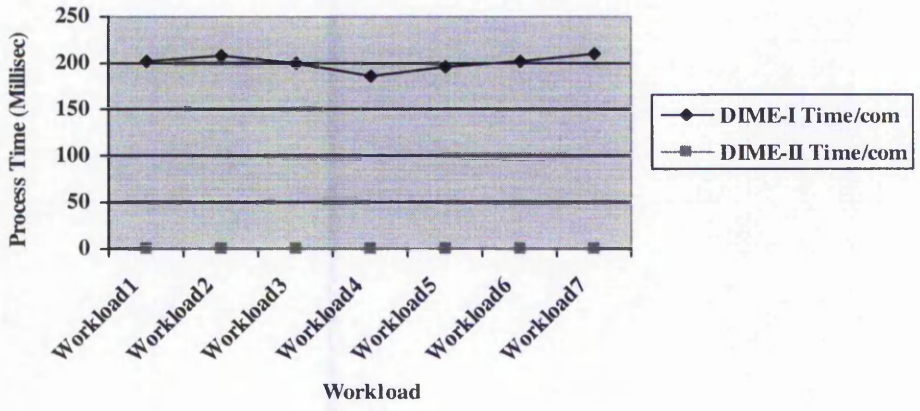


Figure 3.15: The Time (milliseconds) an Application Remains Blocked while Performing Read on Data Buffers.

Workload	DIME-I	DIME-II
Workload1	47.7%	34.7%
Workload2	46.9%	36.0%
Workload3	46.4%	22.1%
Workload4	47.2%	15.7%
Workload5	44.9%	11.9%
Workload6	43.2%	9.9%
Workload7	40.0%	5.1%

Table 3.5: The Time (in percentage) of Data retrieval Operations on Data Areas

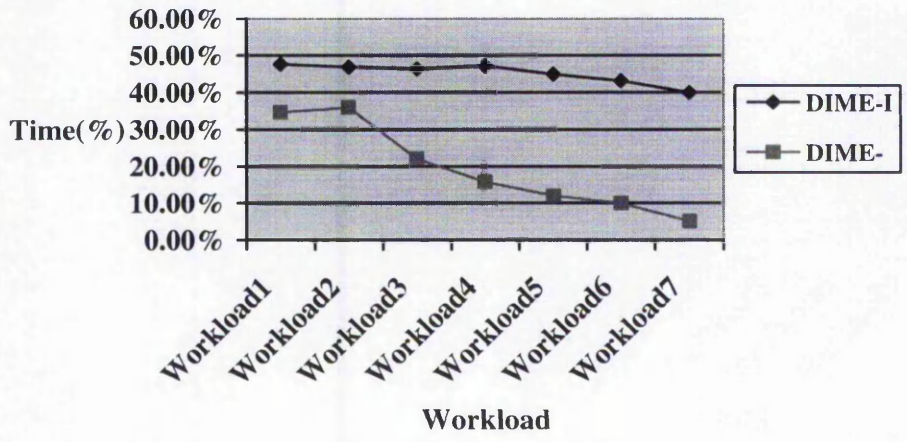


Figure 3.16: The Time (in percentage) of Data Retrieval Operations on Data Areas

Workload	DIME-I	DIME-II
Workload1	47.7%	0.7%
Workload2	46.8%	0.6%
Workload3	42.9%	0.6%
Workload4	38.9%	0.6%
Workload5	39.9%	0.5%
Workload6	36.6%	0.5%
Workload7	32.7%	0.3%

Table 3.6: The Time (in percentage) of Data Retrieval Operations on Data Buffers

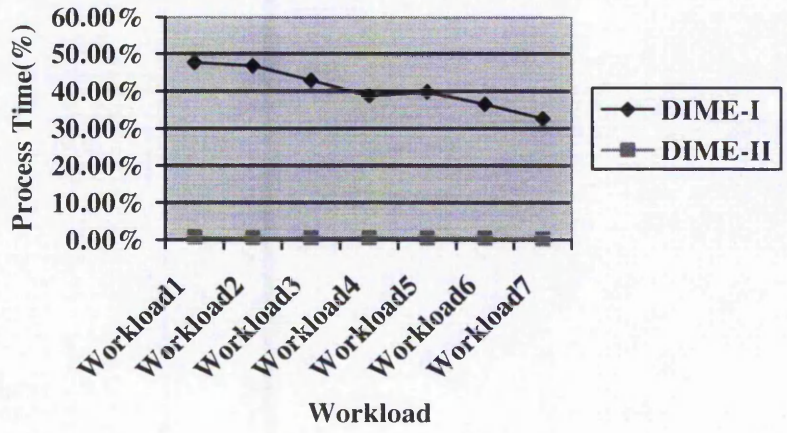


Figure 3.17: The Time (in percentage) of Data Retrieval Operations on Data Buffers

3.10.6 Conclusions in Bullets

1. The framework of DIME-II with the non-locking approach, locality of references, and the relaxed consistency model minimizes the time of data retrieval from the viewpoint of the application in the system.

2. In DIME-II, an application can have more time to execute its native tasks.

3. DIME-I scales better than DIME-II in the sense that its performance in terms of data retrieval time does not change significantly.

4. DIME-II has less scalability due to the fact that DIME-II-server has to maintain replicas of the data required by the applications and with increased number of applications the server must maintain increased number of replicas which lead to the great variation in the data retrieval rates in the system.

Chapter Four

PERFORMANCE OPTIMIZATION IN DISTRIBUTED SHARED MEMORY SYSTEMS – HEURISTIC ALGORITHM

Building distributed systems on network of workstations with DSM paradigm has been proved as a viable alternative to the traditional message-passing paradigm. The increasing demands of distributed applications require sufficiently high-performance DSM algorithms. As mentioned in an earlier chapter, another research direction has been launched, alongside developing new DSM algorithms, to investigate new techniques in improving and enhancing the performance of DSM algorithms. Such techniques can be called complementary techniques as they are used in conjunction with DSM algorithms. This direction of research has presented a wide spectrum of techniques to optimize the performance of distributed and networked systems at different levels of optimization.

Optimization levels range from client interface, through middleware and servers, to the communication infrastructure. In these techniques a variety of criteria are examined, including time, space and quality of service. Among these techniques are: adaptive protocols that adjust to the memory access patterns in distributed applications [Amza C. et. al 1999]; per-node multithreading [Mueller F. 1997]; relaxing consistency definitions to match the needs of the distributed applications; load balancing…etc.

This chapter takes us to the broad world of performance optimization by presenting some common strategies and techniques, besides, introducing a novel heuristic algorithm for enhancing the performance of DSM systems. Generally speaking, this algorithm reconfigures the system at start-up or run-time by adding up intermediate servers to support the main server supplying the service to the currently connected

applications. More specifically, the algorithm optimizes the performance of DSM systems by improving the system connectivity via reconfiguring the construction of the system while preserving its backbone (i.e. the DSM algorithm and the consistency model). This reconfiguration is based on the current state of the network which is evaluated by calculating the round-trip times between different components of the system. Therefore, with this algorithm, the organization of the network; in terms of its size, topology and the current workload; has a great influence in the new improved reconstruction of the network connectivity of DSM systems. It has to be mentioned that, this algorithm aims, overall, at scaling up the overall performance of the system via maximizing the data retrieval rate at application level.

4.1 Related Research

4.1.1 Load Sharing and Load Balancing Policies

In distributed systems, number of processors is utilised where preferably no processor should remain idle while others are overloaded. To efficiently utilize the power of computation provided by the computing environment in distributed systems, many approaches and algorithms have been introduced to uniformly distribute the workload over all processors. These approaches are generally categorised into two: load sharing and load balancing. The purpose of load balancing, in general, is to divide the work evenly among the processors; whereas, the purpose of load sharing is to ensure that no processor remains idle when there are other heavily loaded processors in the system [Karatza H et. al 2001].

4.1.1.1 Load Sharing Algorithms

To redistribute tasks between processors in order to ensure that no processor remains idle, the load distribution activity is initiated in two different ways. With sender-initiated algorithms, the activity is initiated when an over-loaded node (sender) attempts to send task to another under-loaded node (receiver). On the other hand, receiver-initiated algorithms trigger the activity when an under-loaded node (receiver) requests a task from an over-loaded node (sender).

For load sharing, there are policies that use information about the average behavior of the system and ignore the current state. Such policies are called static policies. Policies that react to the system state are called adaptive or dynamic. The function of dynamic policy is important in distributed systems as it is designed to distribute the workload among processors and improve the overall performance at run-time with the consideration of the current state of the system. Static policy is characterised by its simplicity as it does not require the maintenance and the processing of system state information. On the other hand, dynamic policies tend to be more complex as they require information of the system's current state when making transfer decisions. Although they are associated with complexity, dynamic policy gives better overall performance than the achievable by static policies [Karatza H et. al 2001].

Karatza H et. al 2002 studied the effect of different models of load sharing in the performance of heterogeneous distributed systems. In heterogeneous distributed systems, normally, processors operate at different speeds, therefore; jobs are expected to be executed at different times. This particular study aimed at exhibiting the effect of various models on the job performance of distributed systems where half of the total number of the processors has double the speed of the others. Part of the jobs is dedicated to fast processors, whereas the rest are generic and can be individually assigned to any processor.

The models used in the experiment are: probabilistic, shortest queue, least expected response time, and the migratory version of each one. With the probabilistic model, dedicated jobs are dispatched randomly to fast processors, whereas generic jobs are sent to slow processors all with equal probability. Jobs are dispatched in a First Come First Served manner. For this model, the scheduler creates only a small amount of overhead when generating random numbers. In the model probabilistic with migration, the original probabilistic model was modified to include job migration. In this model, when a fast processor becomes idle and generic jobs are waiting in the queues of slow processors, a job migrates from the most heavily loaded slow processor to the idle fast processor. This model employs receiver-initiated algorithm for activating load distribution.

Shortest queue model assigns dedicated and generic tasks to the shortest queue of fast and slow processors, respectively. This method requires knowledge regarding half of the queues on job arrival. The modified version of this method, which is called shortest queue with generic job migration, behaves the same way as the original one with one difference of migrating generic jobs. The modified model migrates generic jobs the same way as the model of probabilistic with generic job migration. On the other hand, in least expected response time model, dedicated jobs join the shortest queue of fast processors, while generic jobs are assigned to the slow/fast processor that offers the least expected response time. This policy requires information about the queue lengths of the processors as well as additional knowledge about the time dedicated jobs have been waiting in a queue. Finally, the last model used in the comparative study was the migratory version of the least expected response time model.

In these experiments, all migratory versions generated a non-trivial amount of overhead as the scheduler requires additional load information to decide when a fast processor becomes idle after a job departure. Additionally important, migratory versions employ non-preemptive algorithm. That is, migratory models transfer only queued jobs, as the jobs being executed are complex to migrate. The complexity of transferring executing jobs is due to the fact that the transfer will involve the memory associated with the migrated job. That does not ignore the fact that migration in itself is a complex process, but migrating active process is much greater than migrating non-active process.

The experimental results showed that the migratory versions of each strategy improved the performance of the generic jobs, due to the fact that when generic jobs migrate they are served by fast processors and therefore, having shorter response time. On the other hand, the delay time of dedicated jobs increases when they arrive at a fast processor that is serving a generic job. The shortest queue strategy and its migratory version exhibited better overall performance, in terms of mean response time of all jobs, than all other models. Moreover, the advantage of the two policies becomes more significant at high loads. Finally, the results proved that the shortest queue with generic jobs migration is the best model when the performance and fairness of individual job class are essential.

In [Karatza H et. al 2001] a new epoch load sharing strategy was presented. With this policy, workload is uniformly distributed among workstations using job migration, which takes place only at the end of predefined intervals called epochs. The scheduler starts collecting information at the end of the epoch. The scheduler collects the information about the status of all workstation queues, evaluates the mean of all queue lengths and places processor queue length into increasing order in a table. After collecting the information, jobs are transferred from the most heavily loaded processors to the lightly loaded ones. This process continues until either all processors have queue lengths equal to the mean or some of them differ at most by one job.

The aim of this model is to reduce the number of times that global system information is needed to make allocation decision while obtaining good overall performance. This strategy was expected to exhibit less overhead in terms of information collection. The epoch algorithm, with different epoch sizes, was compared with some of the previously described models, such as the migratory probabilistic and shortest queue models. The comparison concluded that for all level of migration overhead with different workloads, epoch model with different epoch sizes involved much less overhead, in terms of collecting global system information, than the shortest queue policy and the migratory probabilistic method. Also, the performance of epoch strategy with small epoch sizes performed comparably to the performance of shortest queue method.

4.1.1.2 Load Balancing Strategy

As mentioned earlier, this strategy keeps the workload on the processors in a distributed system environment evenly distributed in order to utilize the available computing power in the best possible way. The balancing problem arises from the fact that in distributed shared memory environment there are processors with different speeds that perform tasks to completion in different execution times. Therefore, some processors can be under-loaded or even idle, while the rest of processors are flooded with tasks, leading to imbalanced system execution.

A load balancing strategy is either centralised or distributed. Unlike distributed strategies, centralised strategies suffer from bottleneck, and they are usually less reliable. On the other hand, although they distribute the load among processors, distributed algorithms may consume the network bandwidth by exchanging huge amount of messages. The selection of the class of the algorithm depends on the underlying architecture and the topology of the systems, as well as the inherent behavior of the parallel applications. Also, there are algorithms that divide the load appropriately to each processor according to prior information about the application. Such algorithms are called static. However, the performance of static algorithm is not guaranteed, especially, when computations are varying unpredictably from time to time. On the contrary, dynamic algorithms assume no prior information, but instead they adapt to the current state of the system. Dynamic algorithms outperform the static ones when fluctuations in execution of applications are modest. But, no dynamic algorithm can effectively offer fast response to frequent variation, due to the inherent overhead of information collection.

The problem of load imbalance became more observable after the introduction of multithreading programming, as load imbalance is a natural inherent from multithreaded processes. In multithreaded system, a number of threads, each of which is designated to perform a piece of computation of a parallel program, will usually be assigned to every processor. Although multithreading aims at providing efficient performance, thread assignment to processors can provoke load imbalance when computation is not decomposed appropriately and accurately [Lai A. et. al 1992]. In other words, multithreaded programming suffers from load imbalance, which happens during execution when there is a difficulty in distributing the balance statically or dynamically. Lai et. al 1992 presented a method that it is dedicated to multithreaded DSM systems for tackling this problem. The method called Dependence-Driven Load Balancing (DDLB). This algorithm adopts distributed and dynamic methods. The motive behind designing this method is to increase the processor utilization of a system by redistributing the load among processors, besides, optimizing the average response time of a system.

DDLB was implemented in a DSM system called cohesion [Shieh K et. al 1995]. The system uses release consistency model with barriers for achieving consistency in the system. The model was embedded in the thread control system of cohesion. The thread control system is responsible of scheduling and managing the threads in cohesion system. DDLB model considers three common policies: transfer policy, location policy and selection policy. The first one is to decide the best instant during the execution course to initiate a load balancing activity. DDLB initiates the activity when the processor becomes idle, specifically, when all threads on the same node have arrived at the barrier before those on other processors.

The second policy, location policy, decides the destination of a thread. In DDLB, the under-loaded processor polls the overloaded ones. For instance, when a processor becomes idle, it contacts the overloaded processors, and a thread is migrated from the first one that responds. The polling phase stops when the load within each phase is balanced and the states of all processors in the system become under-loaded. On the other hand, selection policy is used to select the appropriate thread to migrate.

In DDLB, the dependency between threads is considered in the selection stage. The policy considers the relationships (i.e. threads that share the same piece of information) between pairs of threads in the same nodes, which is called intra-dependency, and different nodes, called inter-dependency. The importance of this consideration is that false sharing can be reduced to the extreme level if the migrated thread has low degree of inter-dependency and high degree of intra-dependency. In other words, if the migrated thread has high intra-dependency, it becomes inter-dependency after migration and then causes false sharing. As it can be predicted from the previous discussion, a thread can migrate from one node to another continuously and therefore, the system is liable to thrashing situation. To overcome the problem, DDLB limits the number of migrations of a thread. If a thread exceeds this number, the system does not allow it to be migrated any more, even though the system is out of balance. However, other threads can be selected for load balancing. Experimental results on the DDLB policy concluded that the inclusion of threads dependence in the selection policy is significant and necessary.

4.1.2 Communication Minimization Strategies

As mentioned in the previous sections, load sharing and balancing can be performed via thread migration from one node to another to achieve the sought improvement in DSM systems. For example, if threads on certain processors share data on a specific page, sharing traffic can only be eliminated by placing these threads on the same node. However, to recall, migrating thread to another node may adversely deteriorate the performance of DSM system if the thread has high inra-dependency. Therefore, attention has to be paid in remapping threads at run-time. Creating good mapping of threads to nodes requires several distinct steps [Thitikamol K. et. al 1999]. First, load distribution of a given mapping must be evaluated, which generally requires a way of estimating threads' computational capacities. This distribution must take into account both parallelism and load balance of the system. On the other hand, the communication cost of thread mapping has to be evaluated in order to reduce the likelihood of incorrect mapping that result in an increasing communication in the system.

The authors believed that, in general, parallelism maximization and communication minimization should be considered together. For instance, a system of four threads; P_1, P_2, P_3 and P_4; that are distributed across the network every two threads on one node (P_1 and P_2 on one node, P_3 and P_4 on another). Seemingly, if each thread has the same amount of work, this system is balanced. However, if each thread communicates with the neighbors (i.e. P_1 communicates with P_3, and P_2 communicates with P_4), the communication is clearly not optimal. In other words, the system is balanced at the expense of high communication overhead. For better performance, threads can be remapped by placing P_1 and P_3 on one node, and P_2 and P_4 on the other. Therefore, the performance can be improved by reducing the communication without affecting the load balance.

An approach presented in that paper aimed at maximizing parallelism, minimizing load imbalance and also minimizing communication. To achieve this complicated objective, three distinct tasks were considered in the designing stage. The

first one was to determine the number of nodes that will result in the greatest speedup, which can be done by the help of an initial guess provided by the user at startup time. Throughout the execution, processor efficiencies are measured by calculating the proportion of time spent waiting on communication and synchronization. These efficiencies are compared with two certain thresholds. According to the chosen thresholds a decision can be made to run the application on fewer nodes or more. The second task was to minimize load imbalance by adjusting the number of threads per node. The last task was to minimize communication by taking sharing into account when mapping threads to node.

Although addressing all these tasks simultaneously can make the complexity of the required algorithm unmanageable, they all had to be considered because they are all interrelated. Since the amount of communication can affect an application's efficiency, the mapping of threads to nodes could affect the number of nodes of which the best performance is achieved. The approach uses the number of pages shared across node boundaries as a predictor of the amount of communication that a mapping of threads to nodes will produce. And the correlation of a pair of threads was defined as the total of number of pages shared between the threads. Also, the cost of a given mapping of threads to nodes is defined as the sum total of all threads-pair correlations for which the component threads are on distinct nodes. The overall evaluation is that the approach exhibited promising results toward achieving minimised communication while obtaining good overall performance with thread migration.

4.2 New Optimization Strategy - Round-Trip Time-based Adaptive Algorithm

DIME-II distributed shared memory system, presented in the previous chapter, adopts a framework of two levels of storage space (figure 4.1). The higher level of storage, called the original memory, contains the original copies of the shared memory of the system and controlled by the DIME-II-server. The lower level of storage space, called the intermediate memory, keeps copies of part of the original memory needed by the creating application. This intermediate memory resides on the machine where that

the creating application is running and controlled by a DIME-II-client. DIME-II-client acts as a server at this level of control as it keeps the intermediate memory consistent with the original memory, and at the same time it provides services to its application. Reading and writing operations are performed on the shared memory in DIME-II system according to the sequential consistency variant model purposely designed for this framework (refer to section 3.7).

Furthermore, in this system, each application communicates with the system via its DIME-II-client, and the DIME-II-client takes the burden of sending/receiving updates to/from the main DIME-II-server. It can be concluded that the structure of the DIME-II system is flexible in the sense that DIME-II-clients can take the service from other servers resides in between DIME-II-clients and the main server, as long as these intermediate servers maintain tunnels for communication with the main server. Such flexibility is an important feature to implement the idea of the strategy.

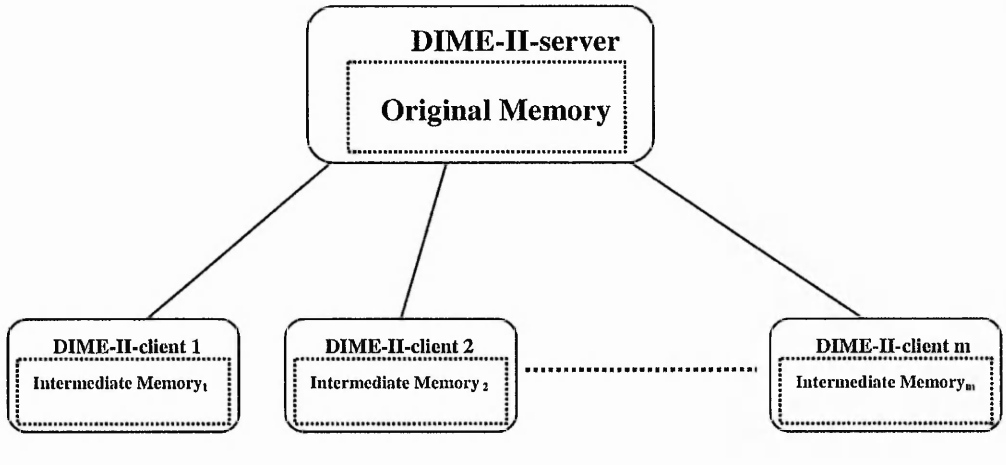


Figure 4.1: DIME-II's Levels of Storage Space

The experiments in chapter three (section 3.10) have proven that, with this framework, DIME-II system outperforms DIME-I DSM system, as it allows user applications to retrieve more data from the system. However, DIME-II does not scale quite well - without reconfiguration - as the number of applications increases. An

experiment in section 3.10.4, have shown that as the number of applications increases, the load on the main server of DIME-II system increases leading to significant performance deterioration.

The round-trip time-based algorithm scales up the performance of the system by adjusting to the demands of the applications and to the current state of the network. This algorithm aims at optimizing the performance of DSM systems by reconfiguring the system connectivity (i.e. the communication paths connecting different parts of the system) to adjust to the current state of the network while preserving the main structure of the system. This reconfiguration can be achieved by initiating another level of service that can supply the running applications with the service, rather than having these applications take the service directly from the main server, which alleviates the load on the main DIME-II-server. In the presence of intermediate level of control, the communication paths of applications can be diverted from the main server and connected to nearby intermediate servers. The location of an intermediate server can be identified according to round-trip times between different parts of the system allowing the system to adjust to the current state of the system. Having several levels of control also allows more applications to be added to the system while maintaining good performance (i.e. good scalability) [Khalil M et al, 2004].

The rest of this chapter presents different implementations for the strategy:

- Statically-initiated Intermediate Servers (SIS), which implements the strategy at start-up time.
- Dynamically-initiated Intermediate Servers (DIS), which implements the strategy at run-time when there are certain number of applications (i.e. three applications) are connected to the main server.
- Heuristic algorithm that implements the strategy at run-time by reconfiguring the system connectivity regardless of the number of connected applications.

4.2.1 Implementing the Strategy

The extra level of control or service consists of number of intermediate servers and memories. Each intermediate server supplies the service to some of the applications that are connected to the system, and therefore, the memory associated with it will contain only the data needed by these applications. Therefore, as it is the case at the level of DIME-II-client, even this lower level of the structure of the shared memory of DIME-II system will contain only part of the original memory.

When it receives a connection request an intermediate server registers the name of the new application in the list of currently connected applications (i.e. acts the same as the DIME-II-server upon accepting connection requests). If the request is the first one, the intermediate server sends a message to register its name in the DIME-II-server as an active server. Each intermediate server has a unique distinct name. This name is needed to be registered in the main server to ensure that the intermediate server gets the updates as they occur in the main server.

When an item is created for the first time in an intermediate server, it sends a creation command to the DIME-II-server, which in turn registers the name of the intermediate server in the list of the applications that have replica of the that particular item. The DIME-II-server registers the name of intermediate server as a copy holder for the shared item instead of the application that actually issued the command in the first place. In fact, at the higher level of control, DIME-II-server does not consider the applications at the lower level, but their intermediate server instead. DIME-II-server deals with active intermediate servers as normal applications, thus, when DIME-II-server receives, for instance, an update for a shared item in the shared memory, it disseminates that update to the intermediate servers that have replica of the updated item.

Intermediate servers respond to operation requests on the shared memory and keep the intermediate storage consistent with the main storage. Intermediate servers respond to requests the same way as DIME-II-clients, only with minor differences. More specifically, when an intermediate server receives a write command it forwards it to the

main server without performing it on the intermediate memory, regardless of the type of the written item (i.e. data area or buffer). However, it performs updates received from the main server only. Therefore, although several intermediate servers may take place in the system, the main server (i.e. DIME-II server) still operates as the only sequencer for all write operations in the system, complying with first constraint of the employed consistency model. This decision is very important in implementing this strategy, as it avoids the likelihood of inconsistency in the presence of intermediate servers within the system. Intermediate servers have a major difference from DIME-II-clients in providing services to more than one application simultaneously.

The location of intermediate servers lies between DIME-II-server and DIME-II-clients (figure 4.2). That gives three levels of memory structure in DIME-II of multi-level storage spaces. 1. The main (original) memory, which is controlled by DIME-II-server. 2. Intermediate memories (level 1) controlled by intermediate servers. 3. Intermediate memories (level 2), which are under the control of DIME-II-clients.

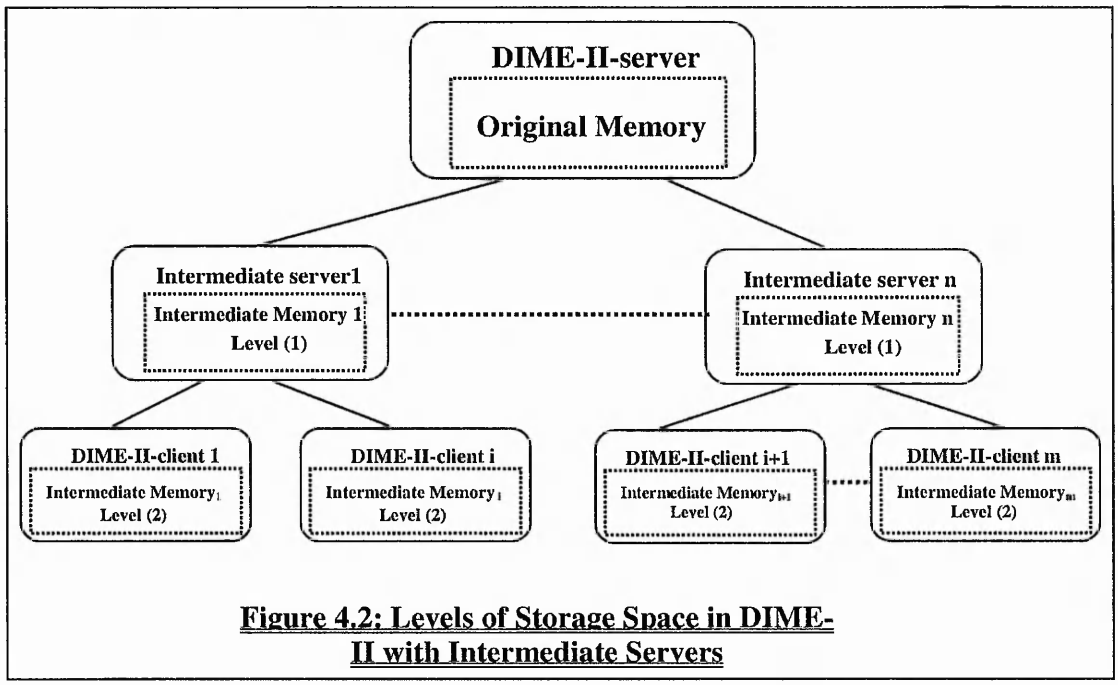


Figure 4.2: Levels of Storage Space in DIME-II with Intermediate Servers

As previously mentioned, to implement this strategy, another crucial decision has to be taken carefully of when to initiate this level and where to place the new intermediate server and memory.

4.3.1 Statically-initiated Intermediate Servers (SIS) – Start-up Time Initiation

As the aim is the improvement of the performance of distributed shared memory systems by reducing the overload at the main server, an intermediate server and an associated memory can be placed in a location nearby specific applications. For instance, if the main server resides in a far-off location, remote applications can have intermediate server, which is connected to the main server, in their LAN. Remote applications can get the service of DIME-II DSM system via an intermediate server associated with a storage space. The intermediate server communicates with the main server (i.e. DIME-II server) on behalf of the remote applications. Therefore, DIME-II-server will give the service to only one application (i.e. the intermediate server) instead of several distant applications. These distant applications are represented in the DIME-II-server by the intermediate server, and therefore DIME-II-server will have less overhead. Such servers are called statically-initiated intermediate servers (SIS), because they are initiated at setup time rather than run time.

To examine the impact of having intermediate level of control on the performance of DIME-II system, experiments have been launched to compare the performance of the system with and without statically-initiated intermediate servers. Again, applications that read and write continuously on the DIME-II system are used. Those applications are placed on the same LAN with the intermediate server, whereas DIME-II server is located on a machine that was connected with the applications' LAN by five switches. The user applications are executed on Windows2000-based machines, while the servers are executed on Unix-based machines. The experiments have been executed with different workloads (Table 4.1). Each workload executes only one writer and different number of readers. Here, workload 1 means one reader and one writer, and workload 2 means two readers and one writer, and so on.

The results show that, having a static intermediate server does not improve the performance when there are only two applications in the system (Table 4.1). However, the impact of the presence of the intermediate control level becomes observable as the number of application increases (Figure 4.3). Therefore, statically-initiated intermediate servers can significantly improve the performance of DSM system even when the number of applications is not large. Such static servers can be used more effectively when having, for instance, DSM system that is distributed in a very wide area (i.e. different cities or may be countries) (figure 4.4).

Workload	**Data retrieval rates without SIS in KB/sec**	**Data retrieval rates with SIS in KB/sec**
Workload1	134	124
Workload2	112	121
Workload3	80	118
Workload4	71	102
Workload5	58	89

Table 4.1: The Performances of DIME-II with and without Static Intermediate Server (SIS) Measured as Data Retrieval Rates (kilobytes/second)

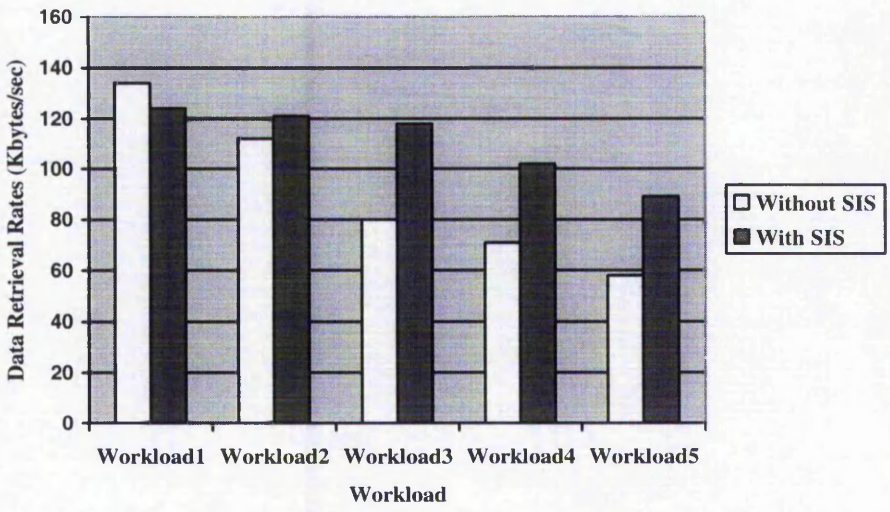


Figure 4.3: The Performance of DIME-II System with and without Statically-initiated Intermediates Servers

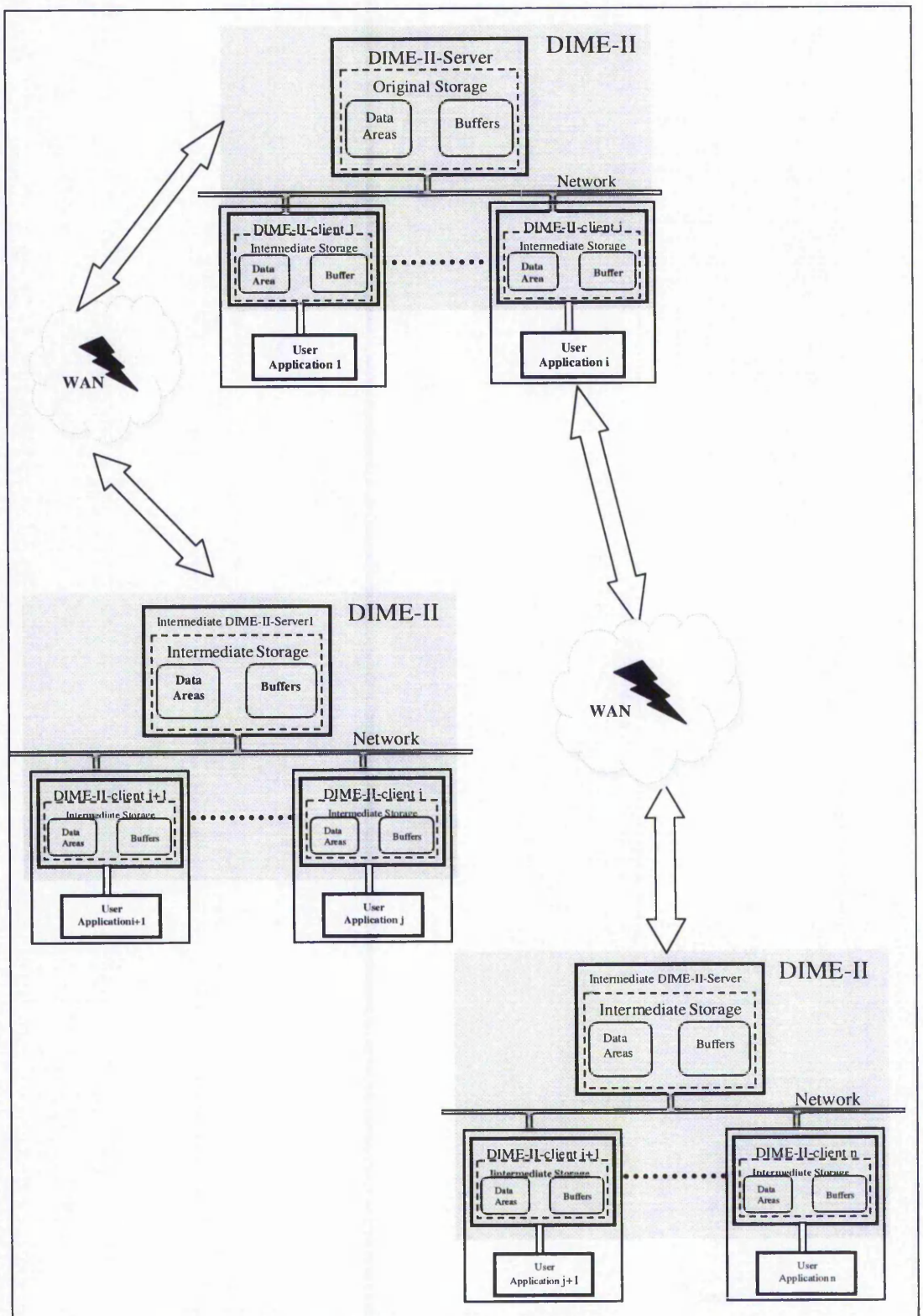


Figure 4.4: DIME-II Architecture with Intermediate Servers

4.3.1 Dynamically-initiated Intermediate Servers (DIS) – Run-time Initiation

Although the previous implementation of the algorithm has achieved a satisfactory improvement in the performance DIME-II DSM system, this implementation is not capable of adjusting the system to the current state of the network due to it is applicability only at set-up time (i.e. before the execution of the remote applications). Another important factor to consider is that, to achieve the required performance improvement in DSM systems with the static scheme, the administrator of the system must have a prior knowledge of the location of the remote applications in order to determine the location of the intermediate server.

Nevertheless, the encouraging results from the implementation of SIS within DIME-II system persuaded us to take the idea one step further - to examine the benefits of initiating intermediate servers at run time and without user's intervention. The user intervention in the SIS implementation is represented by the identification of the location of intermediate servers by the administrator prior executing remote applications. In this section we present an algorithm that can initiate intermediate servers dynamically and when the system has three applications, as we have noticed that the DIME-II system starts to give an undesirable performance with more than three applications (chapter three, section 3.10.3).

In DIS implementation, when the DIME-II-server is supplying the service to three applications, it can direct any further connection requests from new applications to get the service from the nearest intermediate server in order to reduce the load on the main server. In this algorithm, each DIME-II-client in the system has an embedded intermediate server that is ready to provide the service to the requesting applications. DIME-II-server keeps records of the available intermediate servers; it gets the IP address and the port number where an embedded server is listening when it receives a connection request. Hence, in the absence of any static intermediate servers, there will be intermediate servers of number equals to the number of the applications that are currently connected to the system.

When it receives a connection request from an application, DIME-II-server sends message to that application to get the service from an available intermediate server. This message contains IP addresses and port numbers of all intermediate servers in the system. The application, via its DIME-II-client, sends dummy messages to the intermediate servers to calculate the round-trip time (RTT) between the application and each of the servers. Assuming that the server of the shortest RTT is the nearest one, the application can take the service from the intermediate server of the shortest RTT. This is based on the results of the evaluation of SIS servers, that the performance is much better when an application gets the service from a nearby intermediate server. Such servers are called dynamically-intermediate servers (DIS) as they are initiated at run time.

To examine the effectiveness of the algorithm, the same benchmark of the previous experiments is used. This time workload 1 contains 4 applications, five applications in workload 2, and so on (Table 4.2). The results have shown that the algorithm performs very well regardless of the workload. Therefore, initiating intermediate servers that are embedded within DIME-II-client code is a practical means to produce an improved performance in DIME-II DSM system (figure 4.5), especially when the main server is overloaded with applications requests (i.e. the overhead in this evaluation is measured by the number of applications where DIME-II system performance degrades). Furthermore, it has been proved that round trip times can be used to identify the location of the intermediate server, where an application is preferred to get the service from. Figure 4.6 shows some of the architectures of DIME-II system produced by the algorithm during the experiments, two are without DIS (figure 4.6a, 4.6b), and the rest are with DISs.

Workload	Data retrieval rates without DIS in KB/sec	Data retrieval rates with DIS in KB/sec
Workload1	87	104
Workload2	70	99
Workload3	57	85
Workload4	50	74

Table 4.2: The Performances of DIME-II with and without Dynamic Intermediate Servers (DIS) Measured as Data Retrieval Rates (kilobytes/second)

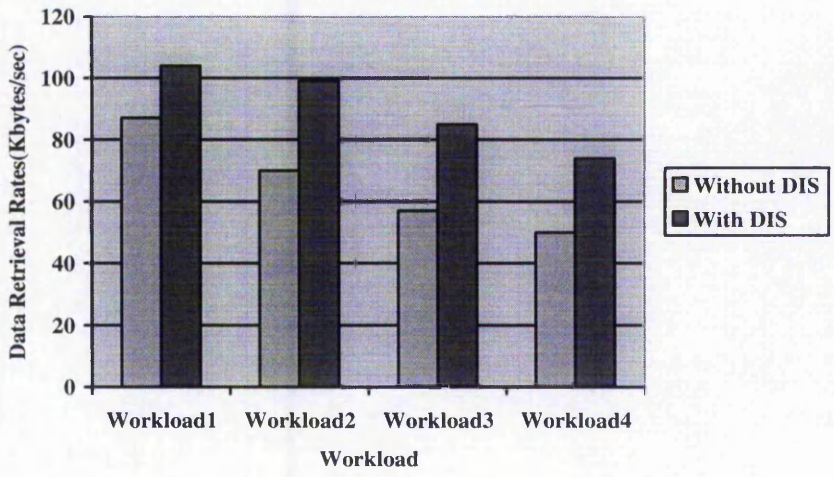


Figure 4.5: The Performances of DIME-II with and without Dynamic Intermediate Servers (DIS)

This section has proved that having dynamically-initiated servers can provide the distributed system with a better performance; however, the DIS strategy is limited to redirecting only newly connected applications. Besides, it still has a user intervention of specifying the maximum number of applications that are allowed to get a direct service from the main server. On the light of that, this strategy can be extended once more, to allow the system to reconfigure its own structural design by adding number of intermediate servers and redirecting the connected applications to get the service from the available servers. Thus, the system is allowed to decide, solely, how many applications are permitted to get the service from the main server. Again, this decision depends on the current state of the network and the system connectivity.

It is always difficult to reconfigure DSM systems at run time to scale up the system, as it usually involves high software and communication overhead. However, having such algorithm is always an attractive solution, as the system can be monitored all the time and reconfigured when certain overhead is reached. Thus, it provides the using system a better scalability and, of course, an optimised performance. It has to be emphasised that, algorithm is ought to be simple yet efficient to avoid the inherent overhead, which is likely to adversely degrade the performance.

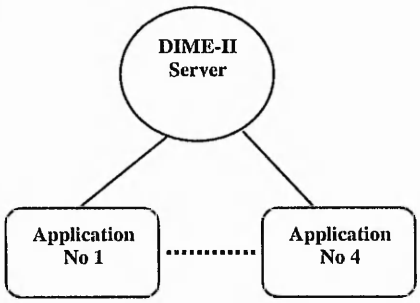


Figure 4.6a: Data Retrieval
Rate of 87 Kbytes/sec

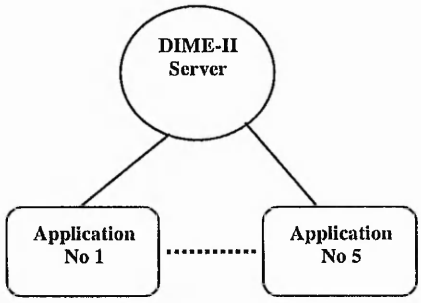


Figure 4.6b: Data Retrieval
Rate of 70 Kbytes/sec

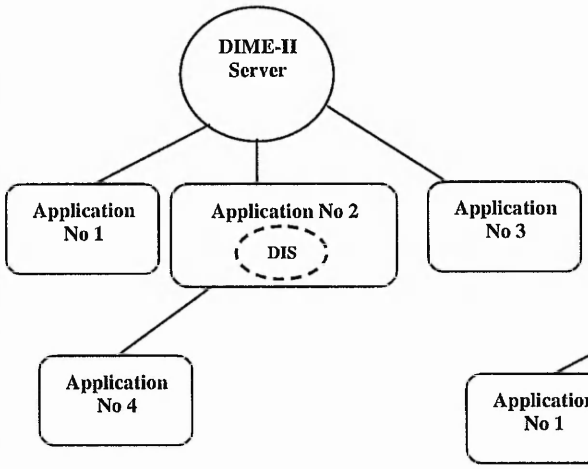


Figure 4.6c: Data Retrieval
Rate of 104 Kbytes/sec

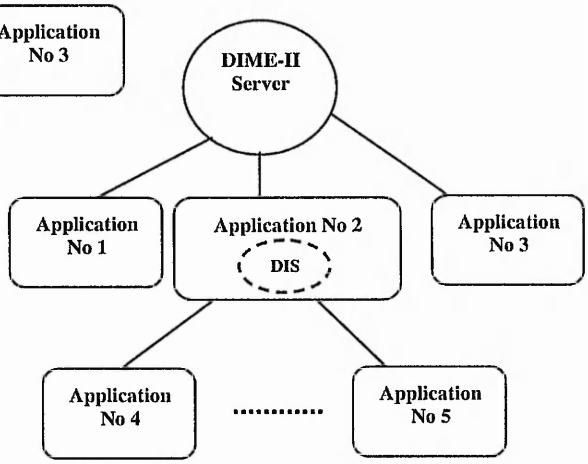


Figure 4.6d: Data Retrieval
Rate of 99 Kbytes/sec

Figure 4.6: Some of the Architectures of DIME-II Produced during the Experiments

4.3.2. Heuristic Algorithm for Optimised DSM Systems

Having potential servers readily available for supplying services to user applications on demand, can be exploited to reconfigure the system at run time. When DIME-II-server reaches certain overhead, it can create an intermediate level of control to scale up the system and maintain the desired performance. Before designing the algorithm, number of experiments has been carried out to examine the performance of different architectures of the DIME-II system with several intermediate servers. Figures 4.7, 4.8 and 4.9 present different architectures of DIME-II system with intermediate servers, which their locations are identified at the set up time of the experiments (not dynamically). It can be seen that, regardless of the architecture, DIME-II system has better performance with intermediate servers. In terms of data retrieval rate, the performance of DIME-II system can be increased by approximately 20%, as in 4.7d, and by about 43%, as in 4.8d, and by 87%, as in 4.9c. Bear in mind that, the architectures used in the experiments are only few instances from the possible wide range of architectures.

Therefore, having an algorithm capable of adjusting DSM systems to the current state of the network at run time by adding up intermediate level of control and reconfiguring the system connectivity can effectively improve the performance. However, such algorithm should be simple, as much as possible, to avoid high software and communication overhead, which is likely to adversely degrade the performance. This requirement has been satisfied in the adaptive algorithm as we will see from the experiments later in this section. The rest of the chapter gives a detailed description of the algorithm and evaluates it.

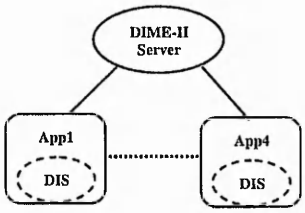


Figure 4.7a: Data Retrieval
Rate of 72 Kbytes/sec

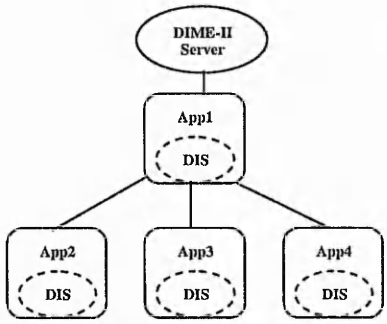


Figure 4.7b: Data Retrieval
Rate of 72 Kbytes/sec

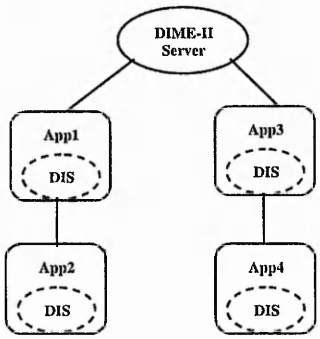


Figure 4.7c: Data Retrieval
Rate of 80 Kbytes/sec

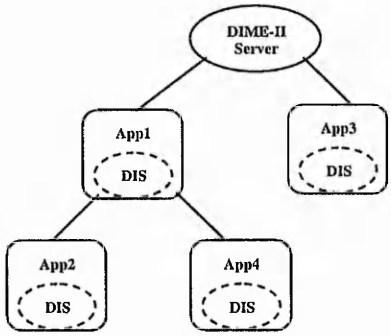


Figure 4.7d: Data Retrieval
Rate of 87 Kbytes/sec

Figure 4.7: Different Structures of DIME-II with Intermediate Servers (4 applications)

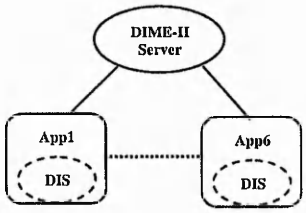


Figure 4.8a: Data Retrieval Rate of 60 Kbytes/sec

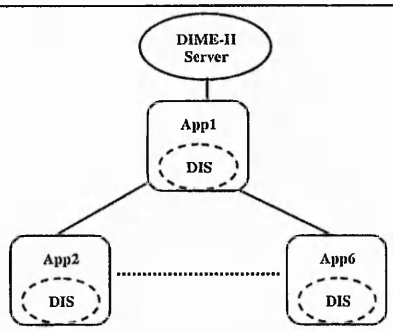


Figure 4.8b: Data Retrieval Rate of 81 Kbytes/sec

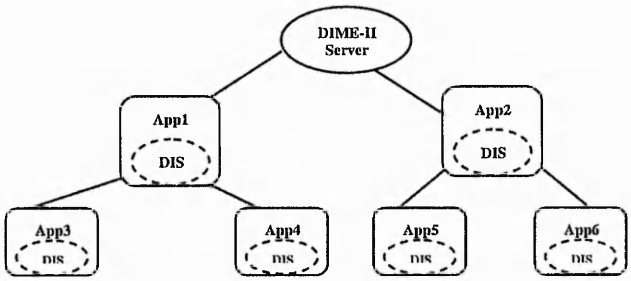


Figure 4.8c: Data Retrieval Rate of 79 Kbytes/sec

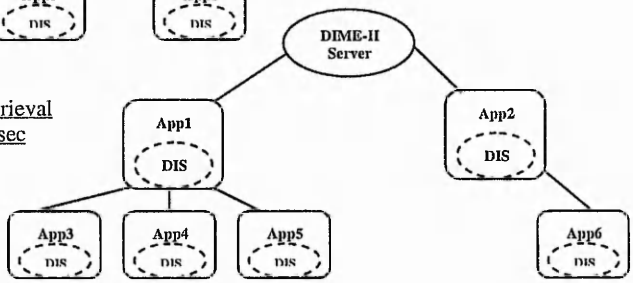


Figure 4.8d: Data Retrieval Rate of 86 Kbytes/sec

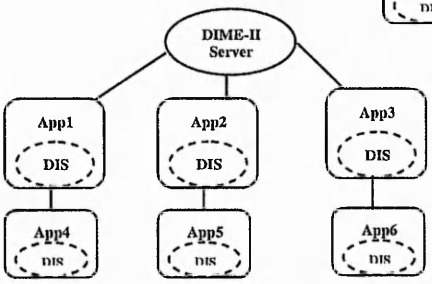


Figure 4.8e: Data Retrieval Rate of 75 Kbytes/sec

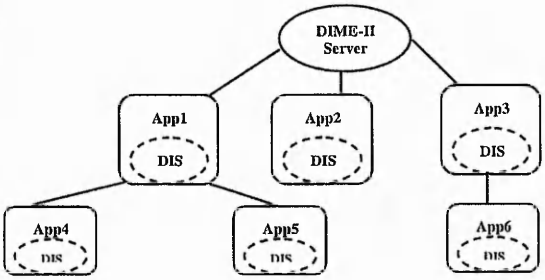


Figure 4.8f: Data Retrieval Rate of 76 Kbytes/sec

Figure 4.8: Different Structures of DIME-II with Intermediate Servers (6 applications)

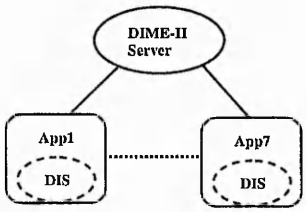


Figure 4.9a: Data Retrieval Rate of 47 Kbytes/sec

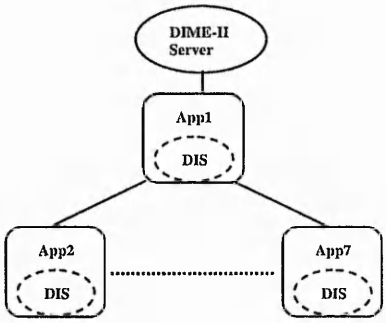


Figure 4.9b: Data Retrieval Rate of 77 Kbytes/sec

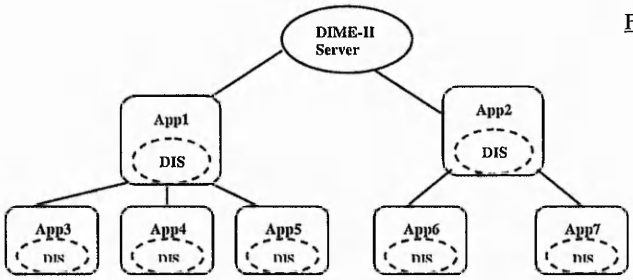


Figure 4.9c: Data Retrieval Rate of 88 Kbytes/sec

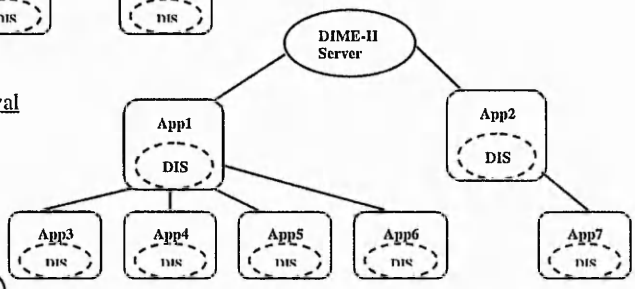


Figure 4.9d: Data Retrieval Rate of 83 Kbytes/sec

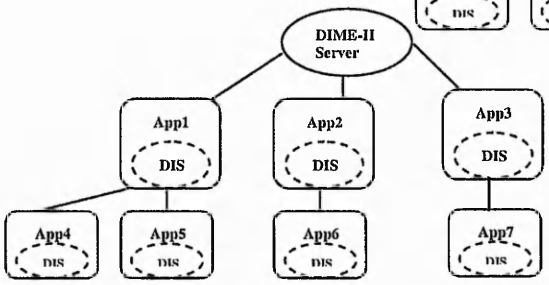


Figure 4.9e: Data Retrieval Rate of 70 Kbytes/sec

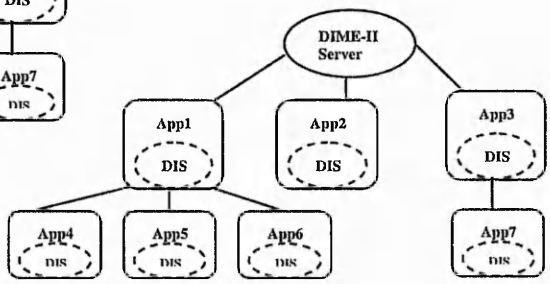


Figure 4.9f: Data Retrieval Rate of 75 Kbytes/sec

Figure 4.9: Different Structures of DIME-II with Intermediate Servers (7 applications)

To reconfigure the system at run time, DIME-II-server starts contacting the running applications one after another for reconfiguration by issuing reconfiguration messages. This message contains the IP addresses and the port numbers of the potential intermediate servers in the system. When an application receives this message, it identifies the location of the nearest intermediate server using RTTs as previously descried. Afterward, the application and its embedded intermediate server redirect their communication channel to take the service from the identified server. DIME-II-server does not issue another reconfiguration message until it is acknowledged that the application has redirected to another server. This acknowledgement is sent by the intermediate server chosen by the redirected application for service provision. When it is acknowledged, the name of the redirected application and its intermediate server will be removed from the list of applications sand servers in DIME-II-server, as they will not be available at this level of control any more. And therefore, the address of the redirected server will not be included in further reconfiguration messages. Furthermore, the application associated with DIME-II-client where the intermediate is running, will be redirected to take the service from the local intermediate server instead of the DIME-II-server, and therefore, its name will also be removed from the list of DIME-II-server as there is no need to send message to it to reconfigure.

During the execution of the algorithm, the system can continue running without any disruption to the executing applications allowing the overlapping of the reconfiguration process and the ongoing computations. Therefore, the impact of the reconfiguration process is reduced to its minimal. With the usual benchmark used throughout the thesis, DIME-II system has been examined with and without the algorithm with different sizes (i.e. the number of applications) as illustrated by figures 4.10, 4.11, 4.12, and 4.13. The architectures appeared in the mentioned figures represent the different structures for DIME-II DSM system produced by the algorithm at run-time.

Although the algorithm has produced a limited number of architectures during the course of the experiments, the round-trip time-based adaptive algorithm has shown a significant success in reconfiguring the system connectivity at run time and achieving better performance every time the system is executed, that is regardless of the produced

architecture. I have to mention that, the variation of the produced data retrieval rates; for example, with 4 applications the rate is 134 Kbytes/sec in figure 4.10, whereas in figure 4.7 the rate is 72Kbytes/sec; is due to execution of the experiments at different times of different network loads during the day.

Although it may not produce the architecture in which the system can have the optimal performance; for example, when there are 6 applications in the system, with intermediate servers the performance can be improved by about 43% (figure 4.8), whereas with the algorithm the performance is improved by only 13%; the algorithm still gives an optimised performance without disturbing the execution of the system. Furthermore, due to the simplicity and the non-disruptive behavior of the algorithm, the duration of the reconfiguration process has not exceeded 50 seconds in the worse case (i.e. when having 10 applications). The duration of the process varies due to the state of the network and the unpredicted behavior of threads in java programming language. Another reason behind the success of the algorithm is the overlapping of reconfiguration process and the ongoing computations.

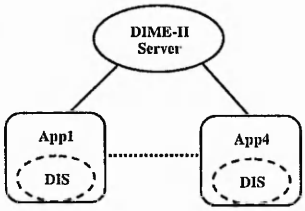


Figure 4.10a: Data Retrieval
Rate of 135 Kbytes/sec

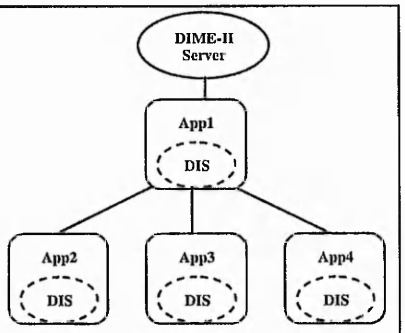


Figure 4.10b: Data Retrieval
Rate of 168 Kbytes/sec

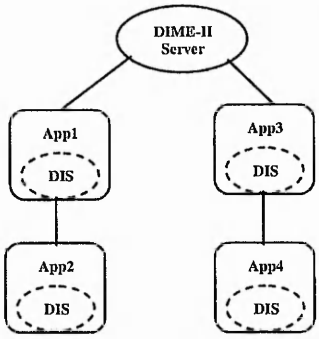


Figure 4.10c: Data Retrieval
Rate of 156 Kbytes/sec

Figure 4.10: Different Structures of DIME-II with Intermediate Servers Produced by the Adaptive Algorithm (4 applications)

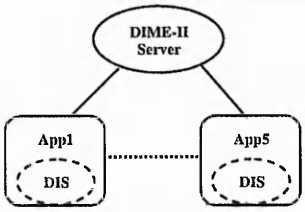


Figure 4.11a: Data Retrieval
Rate of 88 Kbytes/sec

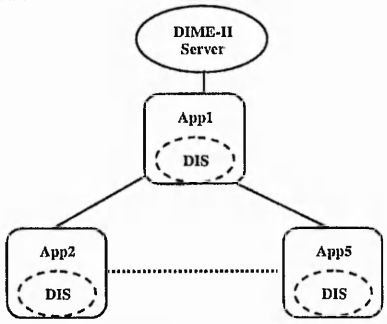


Figure 4.11b: Data Retrieval
Rate of 137 Kbytes/sec

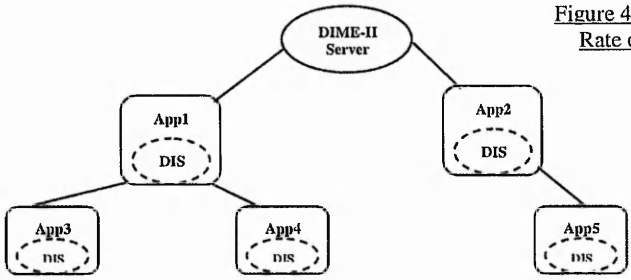


Figure 4.11c: Data Retrieval
Rate of 124 Kbytes/sec

Figure 4.11: Different Structures of DIME-II with Intermediate Servers Produced by the Adaptive Algorithm (5 applications)

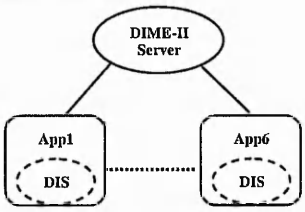


Figure 4.12a: Data Retrieval Rate of 90 Kbytes/sec

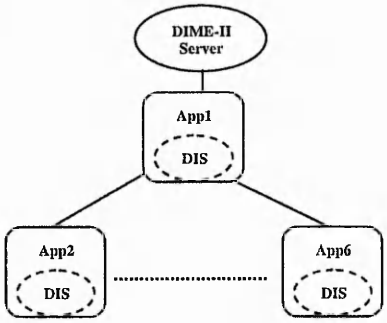


Figure 4.12b: Data Retrieval Rate of 102 Kbytes/sec

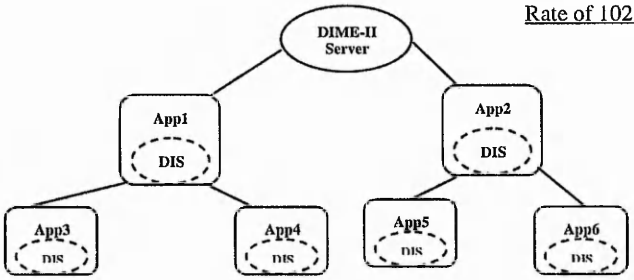


Figure 4.12c: Data Retrieval Rate of 97 Kbytes/sec

Figure: 4.12: Different Structures of DIME-II with Intermediate Servers Produced by the Adaptive Algorithm (6 applications)

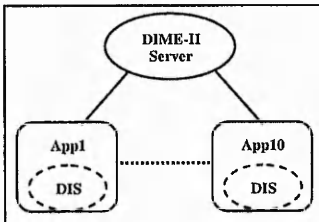


Figure 4.13a: Data Retrieval Rate of 44 Kbytes/sec

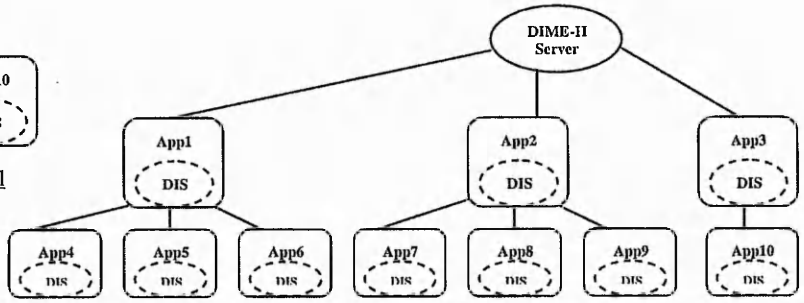


Figure 4.13b: Data Retrieval Rate of 59 Kbytes/sec

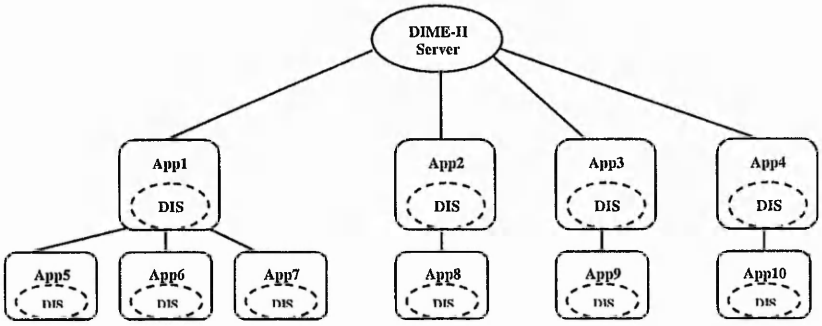


Figure 4.13c: Data Retrieval Rate of 58 Kbytes/sec

Figure: 4.13: Different Structures of DIME-II with Intermediate Servers Produced by the Adaptive Algorithm (10 applications)

Chapter Five

CONCLUSIONS AND FUTURE RESEARCH

5.1 Conclusions

Distributed shared memory (DSM) paradigm provides an illusion of one non-physical shared memory on a network of workstations where shared data exist in different address spaces. In this thesis, we have provided an extensive study for the concept of DSM paradigm, its design issues, levels of implementation, and we have also identified some improvement techniques that are used to bridge the performance gap between DSM systems and message-passing systems. The study has focused on software-oriented DSM systems as they provide a viable solution for building fast computing environments for parallel applications on the commodity workstations without incurring the cost of additional hardware. Also, software support for DSM is generally more flexible and easy to program than hardware-oriented counterparts, and it enables better tailoring of the consistency mechanisms according to the data usage patterns and the application's behaviour. We have also concluded that the degree of considering the design issues of DSM systems varies from one implementation to another according to the nature of the distributed application, and can significantly affect the performance of the produced framework.

In this thesis we have achieved the two main objective of the project. First, we have designed, developed and implemented a distributed shared memory framework that employs a partially-replicated non-locking DSM approach to minimize data retrieval rates from the viewpoint of the distributed applications that use the framework. This method allows an application in a distributed environment to perform read/write operations on the shared memory in its proximity in a relatively short time allowing it more time for performing its native tasks. Secondly, we have developed a flexible architecture for the framework to allow the system to reconfigure its communication paths in order to improve and optimize the performance of the system.

In the process of building the framework we have accomplished the following:

- Categorizing DSM systems into locking and non-locking classes.
- Developing a partially-replicated non-locking algorithm for prototyping the new framework.
- Developing a relaxed variant of the sequential consistency model designed specifically for maintaining consistent memory view in the framework.
- Designing and implementing a middleware-level, multicast-based algorithm for data and messages exchange to reliably disseminate updates to different copies of the shared memory throughout the system.

One of the main contributions of this study is the categorization of distributed shared memory systems into two groups: locking and non-locking DSM systems. This categorization is based on the fact that some of the consistency models -used by distributed systems to ensure consistent view of the shared memory- explicitly use synchronization mechanisms such as Barriers and Locks, whereas the rest are not characterised by this feature. In non-locking algorithms, operations are considered as atomic processes, and performed indivisibly. We have concluded that, the use of synchronization mechanisms can result in poor performance, which encourages us to avoid explicit use of such mechanisms in our system, and therefore the developed framework is categorised as a non-locking DSM prototype.

On the basis of these findings, we have developed a system that employs a non-locking approach as well as other well-known techniques such as multithreading and data replication to provide an optimised performance. This system is called DIME-II as it is a revised version of DIME DSM system. DIME-II system is implemented at user level which does not require any changes in the lower levels of the system of the machine (compiler and operating system). We have integrated number of improvement techniques in the framework aiming at allowing the applications to have more time to perform their native tasks via minimizing the data retrieval rates within the system. User applications' execution time is very often wasted in network communication. We have

assumed that the speed of data processing is greater than the speed of exchanging data and messages over the network.

To maintain consistency in the physically distributed shared memory of the framework, we have defined a consistency model. This model is designed specifically to support the two types of data structures that comprise the shared memory in the traffic system, and it has a flavor of sequential model as it is the most intuitive definition for programmers. Unlike sequential consistency, the presented consistency definition is advantageous in such a way that it supports the concepts of locality of references and multi-reading/multi-writing, which are used in the framework.

In the course of building the framework, we have developed a communication protocol to allow data and messages exchange between different parts of the system. This protocol, called DIME-II Data Transfer Protocol (DDTP), is a middleware-level protocol that disseminates updates only to the applications that have replicas of the updated shared data, reducing the number of messages exchanged in the system and therefore saving network resources. The main feature of this proprietary protocol is that it is built on top of the TCP/IP requiring no modifications in the underlying communication primitives.

We have exploited the potential of per-node multithreading when implementing the framework in order to enhance the functionality of the produced system by hiding the communication latencies by overlapping processes of the system. To evaluate and analyze the performance of the DIME-II DSM system in comparison with the old DIME-I system, we have used experimental studies. Experimental results have shown significant improvement in the DIME-II system in terms of minimizing the time of data retrieval, from the viewpoint of the applications of the system. However, the new framework does not scale quite well due to the high overhead at the server side of the system. However, the flexible architecture of the framework of DIME-II allows system reconfiguration to improve the scalability of the system and optimize the performance.

To overcome the undesirable scalability of the DIME-II system, we have developed a novel heuristic algorithm. This strategy optimizes the performance of DSM

systems via reconfiguring the system connectivity while preserving its backbone and without disturbing the execution of the running applications. The state of the network is evaluated by calculating round-trip times between different components of the system. This non-disruptive algorithm reconfigures the communication paths of the system at run-time by inserting an intermediate level of control and redirects applications that are connected to the main server to get the service from this level of control in order to alleviate the overhead on the main server. In this heuristic algorithm, applications are, individually, allowed to identify the location of the nearest intermediate server to take service from. This location identification is accomplished by calculating the round-trip times (RTT) between each application and all available intermediate servers, and the server with the shortest RTT is chosen to supply that application with the service. Therefore, the structure of the system produced by the algorithm depends on the current state of the system networking.

Apparently, the reconfiguration process adds an extra software and communication overheads to the system, in terms of exchanging dummy messages between different extremes of the system for measuring the round-trip times in order to evaluate the network overhead, and also the process of calculating the round-trip times and deciding which intermediate server is close to an application to take the service from. However, the experiments have shown that the improvement of the performance of the system outweighs this overhead, especially, the whole process of the algorithm overlaps with the operations and the ongoing computations of the system. Finally, the evaluation of the algorithm has confirmed that by using calculated round-trip times between different extremes of the system; the performance can be optimised as the execution of the system progresses via changing the communication configuration of DSM systems adjusting to the current state of the system networking.

5.2 Future Research

The framework presented in this thesis has shown effective impact on the performance of distributed shared memory systems. However, all the experiments have been conducted on LANs and extended LANs. Considering the effect of network latencies in the performance of DSM systems, it would be interesting to see the impact of the framework on DSM systems run on WANs; especially the framework has taken this issue into account mainly via the utilization of per-node multithreading technique and multiple levels of control and memory management.

Also, the adaptive algorithm has exhibited significant enhancement in the performance of the distributed system via reconfiguring the construction of the system. Nevertheless, there is still a room for improvement. This technique can be improved in three possible ways.

- During the course of evaluating the adaptive algorithm, it has been shown that, when there are 10 applications connected to the system, the reconfiguration process of the system takes up to 42 seconds to produce new architecture that can provide an optimised performance. That means it takes less than 5 seconds for an application to get connected to an intermediate server for the service. Let's assume that it is required to spend only 5% of the time between successive reconfiguration processes (reconfiguration interval) in performing the optimization process. Therefore, if there are 10 applications connected to the main server before the reconfiguration takes place, the interval time of the process will be 1000 seconds. This percentage could be included as a parameter to decide the exact time of triggering the process (i.e. the process intervals). However, this parameter, alone, is not enough for this crucial decision, as the system may arrive at the interval without having overhead that can justify performing the process of reconfiguration. But still, this parameter can be decisive as it is not always wise to spend a lot of time of the system execution in too many reconfiguration processes.

- Also, including the execution time of an application could be vital in reducing the overhead of the reconfiguration procedure. On the assumption that an application may not live for long time, excluding this particular application from being reconfigured with other applications might be worthwhile in reducing the time of the reconfiguration process to the minimum. However, in the absence of a prior knowledge of the lifetime of an application, the elapsed time of the application could be used instead. The elapsed time can only be used if we assume that, for instance, an application of elapsed time less than quarter of the average of elapsed execution times of other applications, is considered as a short-life application, and then can be excluded from the process. Of course that does not mean removing the application from the system, but it could still reduce the time of the reconfiguration procedure and therefore reducing its overhead.

- Furthermore, as it is shown in chapter four, there is a wide range of architectures of DIME-II DSM system with intermediate servers that can give performances better than the ones produced by the adaptive algorithm. It is an interesting research issue to investigate new analytical solutions or approaches that are capable of producing the best possible architecture with the best possible performance.

Finally, the evaluation of the heuristic strategy presented in this work, has been given to at most 10 distributed applications. It might be more practical to have an algorithm capable of predicting the behavior of the strategy prior to its implementation, especially in critical applications or distributed systems with large number of applications.

Reference

[Adve S. et. al 1996]

S. Adve, A. L. Cox, S. Dwarkadas, R. Rajamony and W. Zwaenepoel, *A comparison of entry consistency and lazy release consistency implementations,* Proceedings of the 2nd High Performance Computer Architecture Conference, Pages 26-37, February 1996.

[Amza C. et. al 1999]

C. Amza, A.L. Cox, S. Dwarkadas, J. Li-Jie, K. Rajamani and W. Zwaenepoel, *Adaptive Protocols for Software Distributed Shared Memory*, Proceedings of the IEEE, Special Issue on Distributed Shared Memory, Vol.87, No.3, Pages 467-475, March 1999.

[Amza C. et. al 1997]

C. Amza, A. Cox, S. Dwarkadas and W. Zwaenepoel, *Software DSM Protocols that Adapt between Single Writer and Multiple Writer*, The Third International Symposium On High-Performance Computer Architecture, San Antonio, TX, Pages 261-271, February 1997.

[Amza C. et. al 1996]

C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu., R. Rajamony, W. Yu and W. Zwaenepoel, *TreadMarks: shared memory computing on networks of workstations*, IEEE Computer, Vol. 29, No.2, Pages 18-28, February 1996.

[Argile A. et. al 1996]

A. Argile, E. Peytchev, A. Bargiela and I. Kosonen, *DIME: A Shared Memory Environment for Distributed Simulation, Monitoring and Control of Urban Traffic*, Proceedings of European Simulation Symposium ESS'96, Genoa, ISBN 1-565555-099-4, Vol.1, Pages 152-156, October 1996.

[Attiya H. et. al 1994]

H. Attiya and J. Welch, *Sequential Consistency versus Linearizability*, ACM Transactions on Computer Systems, Vol. 12, No. 2, Pages 91-122, May 1994.

[Auld P. 2001]

P. Auld, *Broadcast Distributed Shared Memory*, PhD Thesis, Department of Computer Science, The college of William and Mary, USA, 2001.

[Bal H. et. al 1991]

H. Bal and A. Tanenbaum, *Distributed Programming with Shared Data*, Computer Languages Journal, Vol. 16, No. 2, Pages 129-146, 1991.

[Bennett J et. al 1990]

J. K. Bennett, J. B. Carter and W. Zwaenepoel, *Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence*", Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), Seattle, Washington, USA, Pages 168-176, March 1990.

[Bisiani R. et. al 1990]

R. Bisiani and M. Ravishankar, *PLUS: A Distributed Shared-Memory System,* Proceedings of the 17th Annual International Symposium on Computer Architecture, Pages 115-124, June 1990.

[Carter JB 1993]

J. B. Carter, *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency,* PhD thesis, Rice University, Houston, Texas, September 1993.

[Carter JB 1995]

J.B. Carter, J.K. Bennett and W. Zwaenepoel, *Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems,* ACM Transactions on Computer Systems, Vol. 13, No. 3, Pages 205-243, August 1995.

[Carter JB 1995]

J. B. Carter, *Design of the Munin distributed shared memory system*, Journal of Parallel & Distributed Computing, Vol.29, No.2, USA, Pages 219-27, September 1995

[Cox A. et. al 1994]

A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony and W. Zwaenepoel, *Software Versus Hardware Shared-Memory Implementation: A Case Study,* Proceedings of the 21st Annual International Symposium on Computer Architecture, Pages 106-117, April 1994.

[Delp G., 1988]

G. Delp, *The Architecture and implementation of Memnet: A High-Speed Shared Memory Computer Communication Network,* PhD thesis, University of Delaware, 1988.

[Dubois M. et. al 1988]

M. Dubois, C. Scheurich and F. Briggs, *Synchronization, Coherence, and Event Ordering in Multiprocessors*, IEEE Computer Journal, Vol. 21, No. 2, Pages 9-21, February 1988.

[Fleisch BD et. Al 1989]

B. D. Fleisch and G. J. Popek, *Mirage: a Coherent Distributed Shared Memory Design,* Operating Systems Review, Vol.23, No.5, Pages 211-223, 1989.

[Frank S. et. al 1993]

S. Frank, J. Rothnie and H. Burkhardt, *The KSR1: Bridging The Gap between Shared Memory and MPPs*, Proceedings of Computer Conference, San Francisco, CA, USA, Pages 285-294, February 1993.

[Gharachorloo K. et. al 1990]

K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*, Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, Washington, USA, Pages 15-26, June 1990.

[Herlihy M. et. al 1991]

M. Herlihy and J. Wing, *Linearizability: A Correctness Condition for Concurrent Objects*, ACM Transactions in Programming Languages and System, Vol. 12, No. 3, Pages 463-492, July 1991.

[Hill M., 1998]

M. Hill, *Multiprocessors Should Support Simple Memory Consistency Models*, IEEE Computer Journal, Vol.31, No.8, Publisher: IEEE Computer Society, USA, Pages 28-34, August 1998.

[Hutto P. et. al 1996]

P. Hutto and M. Ahmad, *Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories*, Proceedings of the 10th

International Conference on Distributed Computing Systems, IEEE Computer Society Press, Pages 302-311, June 1996.

[Iftode L. et. al 1996]

L. Iftode, J. P. Singh and K. Li, *Scope Consistency: A Bridge between Release Consistency and Entry Consistency*, Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, Pages 277-287, June 1996.

[Jacobson V., 1988]

V. Jacobson, *Congestion Avoidance and Control*, Proceedings of the ACM SIGCOMM 88, Stanford, CA, Pages 314-329, August 1988.

[Jain R., 1986]

R. Jain, *Divergence of Timeout Algorithms for Packet Retransmissions,* Proceedings of the Fifth Annual International Phoenix Conference on Computers and Communications, Scottsdale, Ariz, Pages 174-179, March 1986.

[Karatza H et. al 2001]

H. Karatza and R.C. Hilzer, *Epoch Load Sharing in a Network of Workstations,* Proceedings of the 34th Annual Simulation Symposium, IEEE Computer Society Press, SCS, Seattle, Washington, Pages 36-42, April 2001.

[Karatza H et. al 2002]

H.D. Karatza and R.C. Hilzer, *Load Sharing in Heterogeneous Distributed Systems*, Proceedings of the Winter Simulation Conference, ACM, IEEE, SCS, San Diego, California, Pages 489-496, December 2002.

[Karn P. et. al 1985]

P. R. Karn, H. E. Price and R. J. Diersing, *Packet Radio in the Amateur Service*, IEEE Journal of Selected Areas in Communications, Vol. SAC-3, No. 3, Pages 431-439 , May 1985.

[Karn P. et. al 1991]

P. Karn and C. Partridge, *Improving Round-Trip Time Estimates in Reliable Transport Protocols*, ACM Transactions on Computer Systems, Vol. 9, No. 4, Pages 364-373, November 1991.

[Keleher P. et. al 1992]

P. Keleher, A. Cox and W. Zwaenepoel, *Lazy Release Consistency for Software Distributed Shared Memory*, Computer Architecture News, Vol.20, No.2, Pages 13-21, May 1992.

[Khalil M. et. al 2003a]

M. Khalil and E. Peytchev, *Traffic Telematics Computing Framework Based on Non-Locking and Housekeeping Distributed Shared Memory Algorithm*, Sixth United Kingdom Simulation Society Conference (UKSIM2003), Emmanuel's college, Cambridge, UK, Pages 201-206, April 2003.

[Khalil M et. al 2003b]

M. Khalil and E. Peytchev, *An Approach for Improving the Performance of Software Distributed Shared Memory Systems*, PREP2003, Exeter, Pages 135-136, April 2003.

[Khalil M et. al 2003c]

M. Khalil and E. Peytchev, *DIME-II: A Computing Framework for Traffic Systems*, 17th European Simulation Multiconferenc (ESM2003), Nottingham, Pages 595-600, June 2003.

[Khalil M et. al 2004]

M. Khalil and E. Peytchev, *A Strategy For Tuning The Performance Of Distributed Shared Memory Systems*, Seventh United Kingdom Simulation Society Conference (UKSIM2004), St Catherine's College, Oxford, UK, Pages 118-123, March 2004,

[Lamport L. 1979]

L. Lamport, *How to make a Multiprocessor Computer that correctly executes Multiprocessor Programs*", IEEE Transactions on Computer, Vol. C-29, No. 9, Pages 690-691, September 1979.

[Lai A. et. al 1992]

A. C. Lai, C. K. Shieh and Y. T. Kok, *Load Balancing in Distributed Shared Memory Systems*, The 1997 IEEE International Performance, Computing, and Communications Conference, Pages 152-158, February 1997.

[Lenoski D. et. al 1992]

D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz and M. Lam, *The Stanford Dash Multiprocessor*, IEEE Computer Journal, Vol. 5, No 3, Pages 63-79, March 1992.

[Li K. 1988]

K. Li, *IVY: A Shared Memory Virtual Memory System for Parallel Computing*, Proceedings of the 1988 International Conference on Parallel Processing, Pages II-94 - II-101, 1988.

[Lipton R. et. al 1988]

R. Lipton and J. Sandberg, *PRAM: a Scalable Shared Memory*, Technical Report CS-TR, Princeton University, Pages 180-188, September 1988.

[Lu H. et. al 1995]

H. Lu, S. Dwarkadas, A.L. Cox and W. Zwaenepoel, *Message Passing versus Distributed Shared Memory on Networks of Workstations*, Proceedings of the 1995 ACM/IEEE Supercomputing Conference (IEEE Cat. No. 95CB35990). ACM Part, New York, NY, USA, Vol.1, Pages 865-906, 1995.

[Mueller F. 1997]

F. Mueller, *Distributed Shared-Memory Threads: DSM-Threads: Description of Work in Progress,* Proceedings of the Workshop on Run--Time Systems for Parallel Programming, Pages 31-40, April 1997.

[Nitzberg B. et. al 1991]

B. Nitzberg and V. Lo, D*istributed shared memory: A survey of issues and algorithms*, IEEE Computer, Vol. 24, Pages 52-60, August 1991.

[Peytchev E. et. al 1998]

E. Peytchev and A. Bargiela, *Traffic Telematics Software Environment*, Simulation Technology: Science and Art 10th European Simulation Symposium ESS'98, San Diego, CA, USA, Pages378-82, 1998.

[Peytchev E. 1999]

E. Peytchev, *Integrative Framework for Discrete Systems Simulation and Monitoring*, Ph.D. thesis, Department of Computing, The Nottingham Trent University, Nottingham, England, February 1999.

[Protic J. et. al 1993]

J. Protic and M. Aleksic, *An example of Efficient Message Protocol for Industrial LAN*, Microprocessing and Microprogramming, Vol. 37, Pages 45-48, January 1993.

[Protic J. et. al 1996]

J. Protic, M. Tomasevic and V. Milutinovic, *Distributed Shared Memory: Concepts and Systems*", IEEE, USA, 1996.

[Ramachandran U. et. al 1989]

U. Ramachandran and M. Khalidi, *An Implementation of Distributed Shared Memory*, Distributed and Multiprocessor Systems Workshop, Pages 21-38, 1989.

[Rzeczkowski W et. al 1980]

W. Rzeczkowski and K. Subieta, *LINDA-a data base management system II*, Informatyka, Poland ,Vol.15, No.12, Pages 8-10, December 1980.

[Shieh K et. al 1995]

C. K. Shieh, A. C. Lai, J. C. Ueng, T. Y. Liang, T. C. Chang and S. C. Mac, *Cohesion: An Efficient Distributed Shared Memory System Supporting Multiple Memory Consistency Models*, Aizu International Symposium on Parallel Algorithms/Architecture Synthesis, Pages 146-152, February 1995.

[Speight E. et. al 1997]

E. Speight, J. Bennett, *Brazos: A Third Generation DSM System*, Proceedings of the 1st USENIX Windows NT Symposium, Pages 95-106, August 1997.

[Tanenbaum A. et. al 2002]

A. Tanenbaum, M. Steen, *Distributed Systems: Principles and Paradigm*, New Jersey, Prentice Hall, 2002.

[Thitikamol K. et. al 1999]

K. Thitikamol and P. Keleher, *Thread Migration and Communication Minimization in DSM Systems*, Proceedings of the IEEE, Special Issue on Distributed Shared Memory, Pages 487-497, March 1999.

[Weisong S. et. al 1998]

S. Weisong, H. Weiwu, T. Zhimin, *Using confidence Interval to Summerize the Evaluating Results of DSM Systems*, Technical Report, Center of High Performance Computing, Institute of Computing Technology, Chinese Academy of Sciences, September 1998.

[Weiwu H. et. al 1998]

H. Weiwu, S. Weisong, T. Zhimin, *A Framework of Memory Consistency Models*, Journal of Computer Science & Technology, Publisher: Science Press, China, Vol.13, No.2, Pages 110-124, March 1998.

[Wittie L. et. al 1989]

L. Wittie and C. Maples, *MERLIN: Massively Parallel Heterogeneous Computing*, International Conference on Parallel Processing, Vol. I: Architecture, Pages 142-150, 1989.

[Zhang L., 1986]

L. Zhang, *Why TCP timers don't work well*, Proceedings of ACM SIGCOMM '86, Pages 397-405, August 1986.

[Zhou S. et. al 1990]

S. Zhou, M. Stumm, T. McInerney, *Extending Distributed Shared Memory to Heterogeneous Environments*, Proceedings of the 10th International Conference on Distributed Computing Systems, IEEE, Pages 30-37, 1990.