

LEABHARLANN CHOLÁISTE NA TRÍONÓIDE, BAILE ÁTHA CLIATH | TRINITY COLLEGE LIBRARY DUBLIN
Ollscoil Átha Cliath | The University of Dublin

Terms and Conditions of Use of Digitised Theses from Trinity College Library Dublin

Copyright statement

All material supplied by Trinity College Library is protected by copyright (under the Copyright and Related Rights Act, 2000 as amended) and other relevant Intellectual Property Rights. By accessing and using a Digitised Thesis from Trinity College Library you acknowledge that all Intellectual Property Rights in any Works supplied are the sole and exclusive property of the copyright and/or other IPR holder. Specific copyright holders may not be explicitly identified. Use of materials from other sources within a thesis should not be construed as a claim over them.

A non-exclusive, non-transferable licence is hereby granted to those using or reproducing, in whole or in part, the material for valid purposes, providing the copyright owners are acknowledged using the normal conventions. Where specific permission to use material is required, this is identified and such permission must be sought from the copyright holder or agency cited.

Liability statement

By using a Digitised Thesis, I accept that Trinity College Dublin bears no legal responsibility for the accuracy, legality or comprehensiveness of materials contained within the thesis, and that Trinity College Dublin accepts no liability for indirect, consequential, or incidental, damages or losses arising from use of the thesis for whatever reason. Information located in a thesis may be subject to specific use constraints, details of which may not be explicitly described. It is the responsibility of potential and actual users to be aware of such constraints and to abide by them. By making use of material from a digitised thesis, you accept these copyright and disclaimer provisions. Where it is brought to the attention of Trinity College Library that there may be a breach of copyright or other restraint, it is the policy to withdraw or take down access to a thesis while the issue is being resolved.

Access Agreement

By using a Digitised Thesis from Trinity College Library you are bound by the following Terms & Conditions. Please read them carefully.

I have read and I understand the following statement: All material supplied via a Digitised Thesis from Trinity College Library is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of a thesis is not permitted, except that material may be duplicated by you for your research use or for educational purposes in electronic or print form providing the copyright owners are acknowledged using the normal conventions. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone. This copy has been supplied on the understanding that it is copyright material and that no quotation from the thesis may be published without proper acknowledgement.

A Flexible Framework for Distributed Shared Objects

Stefan Weber

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

October 2001

TRINITY COLLEGE
10 OCT 2002
LIBRARY DUBLIN
Thesis
6970

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.



Stefan Weber

Dated: August 14, 2002

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.



Stefan Weber

Dated: August 14, 2002

Acknowledgements

First and foremost, thanks are due to my two supervisors Brendan Tangney and Paddy Nixon for their endless efforts and infinite patience while supervising me. I'm still surprised that Brendan's hair did not turn gray because of the repeated delays that my difficulties with the writing of this text caused.

Thanks are also due to Vinny Cahill, for his hard work to keep our group running and to provide us with funding for equipment and travels; without his continuous efforts the Distributed Systems Group wouldn't be the same.

I am very grateful to Inmaculada Arnedillo Sanchez and Christian Jensen, who - when I was about to give up in the frequent moments of self-doubt - set me on the right path again.

And finally thanks are due to all members of DSG for their company, their patience with me and their good advice: Stephen, Frank, Mads, Jim, Tim, Tilman, Ray, Rene, Peter, Donal, John, Johan, and many more. "Stick together team!"

At last but not least, I'm very grateful to my family for their constant support and encouragement during the years. This thesis is dedicated to them.

Stefan Weber

University of Dublin, Trinity College

October 2001

Abstract

A distributed shared memory (DSM) system ensures the consistency of shared data in a distributed system while providing the programming paradigm of a single-processor system. Recent DSM systems provide increasingly more support for the sharing of objects rather than portions of memory. However, like earlier DSM systems these distributed shared object (DSO) systems still rely upon a single protocol, or a small set of given protocols, for the sharing of application objects. This limitation prevents the applications from optimizing the underlying communication behaviour on a per-application basis. This results in unnecessary overhead which impacts upon performance which is the main goal of parallelism.

A current trend in software development is toward customizable systems; for example frameworks, reflection, and aspect-oriented programming all aim to give the developer greater flexibility and control over the functionality and performance of their code. The lack of structure in protocols used in DSM systems prevents the introduction of this kind of adaptability into DSM systems.

This thesis categorizes DSM protocols into consistency models and coherency protocols. Consistency models define the order in which accesses to shared data are experienced by nodes of a parallel system. Coherency protocols describe the method by which results of accesses to shared memory are exchange between nodes.

We define consistency maintenance protocols as a combination of consistency models and coherency protocols. One of the contributions of this thesis is the precise definition of consistency models and coherency protocols as components. This definition includes a description of all characteristics of the components and their sub-components. The definition of these components and their interfaces is then used to describe their relationship. This

description is novel to our knowledge.

The definition of this relationship allows the combination of consistency models and coherency protocols in a truly component-based fashion. It allows a consistency model or coherency protocol to be created from subcomponents and be adjusted to changing environments by adjusting the sub-components of the components. This is the fundamental insight that allows us to build systems that can be adapted to application-specific needs.

The definition of the components and their relationships is transferred into a framework that is implemented using Java. This framework provides the developer with interfaces and classes for consistency models and coherency protocols and their sub-components. The interfaces define to which implementations of coherency protocols and consistency models must comply. This maintains the interchangeability of components of the same type in the framework. The classes provided by the framework can be divided into concrete and abstract classes. The concrete classes represent complete implementations of consistency models and coherency protocols. These implementations can be readily used in applications as building blocks for a consistency maintenance protocol. The abstract classes of the framework represent base classes that can be extended to create new implementations of consistency model and coherency protocols in order to adjust a consistency maintenance protocol to application-specific needs.

In order to validate our approach we have implemented a number of applications using the framework. The data used by these applications exhibit different sharing characteristics. The flexibility that is offered by the framework is exploited to adjust the consistency maintenance to suit the sharing characteristics of the individual application. This allows these applications to outperform applications that are based on conventional DSM systems.

Publications Related to this Ph.D.

[1] S. Weber and P. Nixon. An Object Oriented DSM Framework. Proceedings of IEEE Frontiers'96, Annapolis, Maryland, October 1996.

[2] A. Judge, P. Nixon, B. Tangney, S. Weber and V. Cahill. High Performance Cluster Computing, volume 1, chapter 17, pages 409-438. Prentice Hall, 1999.

[3] S. Weber, P. Nixon and B. Tangney. A Flexible Framework for Consistency Management. Journal of Concurrency: Practice and Experience, to be published.

Contents

Acknowledgements iv

Abstract iv

List of Tables xiii

List of Figures xiv

Chapter 1 Introduction 1

1.1 Programming paradigms 1

1.2 Distributed Shared Memory systems 2

1.3 Adaptability 4

1.4 Thesis 5

1.5 Evaluation 6

1.6 Road Map 7

1.7 Summary 7

Chapter 2 Related work 9

2.1 Motivation 9

2.2 Adaptability in DSM systems 11

2.2.1 System adaptability 12

2.2.2 Protocol adaptability 13

2.2.3 Object-based adaptability 13

2.2.4 Runtime adaptability . . . 15
2.2.5 Dynamic adaptability . . . 15
2.2.6 Summary of Adaptability Categories . . . 16
2.3 Types of DSM systems . . . 16
2.3.1 Hardware-assisted shared memory . . . 18
2.3.2 Page-based software systems . . . 23
2.3.3 Object-based systems . . . 29
2.3.4 Summary . . . 30
2.4 Consistency models . . . 30
2.4.1 Strict consistency models . . . 32
Sequential consistency (SC) . . . 32
Atomic consistency (AC) . . . 33
Processor consistency (PC) . . . 33
Cache consistency (CC) . . . 34
2.4.2 Relaxed consistency models . . . 34
Weak Consistency (WC) . . . 35
Release Consistency (RC) . . . 36
Lazy Release Consistency (LRC) . . . 36
2.4.3 Categorization of consistency models . . . 36
2.4.4 Summary of consistency models . . . 37
2.5 Related projects . . . 38
2.5.1 TreadMarks . . . 38
2.5.2 CVM . . . 39
2.5.3 Munin . . . 40
2.5.4 Orca . . . 41
2.5.5 Disom . . . 42
2.5.6 Rabbit . . . 42
2.5.7 Summary of related projects . . . 42
2.6 Summary of related work . . . 43

Chapter 3 A new categorization of consistency maintenance protocols **45**

3.1 Overall structure of consistency maintenance protocols 46

3.1.1 Components of Consistency Maintenance Protocols 47

3.2 Consistency Models 48

3.2.1 Properties of consistency models 49

Examples 50

3.2.2 Primitives in consistency models 50

Description of primitives 51

Semantics of primitives 52

3.2.3 Characterization of consistency models 53

Strict consistency models 53

Relaxed consistency models 56

3.3 Coherency Protocols 59

3.3.1 Reaction on the receipt of primitives 60

Transformation 61

Distribution algorithms 61

3.3.2 Components of Coherency Protocols 64

Memory management 67

Ownership management 70

Distribution management 71

3.3.3 Examples 72

3.4 Complete Examples 72

3.4.1 TreadMarks 72

3.4.2 DASH 74

3.5 Summary 75

Chapter 4 Implementation **76**

4.1 Framework overview 77

4.2 Supporting classes 79

4.2.1 Data Structures 79

Descriptor 80
Intention 82
4.2.2 RMI mechanism 83
4.2.3 Synchronization classes 86
Locks 86
Barriers 90
4.2.4 Execution environment 90
4.3 Consistency model 92
Generic Base-Classes 92
Implementation of a consistency models 92
4.3.1 Overview of the class hierarchy 93
4.3.2 Weak Consistency 96
4.3.3 Modified Weak Consistency (MWC) 97
4.3.4 Lazy Release Consistency (LRC) 97
4.4 Coherency protocols 97
4.4.1 General behaviour of coherency protocols 98
4.4.2 Memory Management 99
Update 100
Example of a memory management class 101
4.4.3 Distribution management 101
Example of a distribution management class 103
4.4.4 Ownership management 103
Example of an ownership management class 105
4.4.5 Implementation of a coherency protocol 105
4.4.6 Class hierarchy of coherency protocols 108
4.4.7 Implemented Coherency Protocols 109
4.5 Deployment of a shared object 110
4.5.1 SharedObject class 110
4.5.2 Initialization 111

4.5.3 Utilization 112
4.6 Summary 113

Chapter 5 Evaluation **114**
5.1 Environment 116
5.1.1 System environment 116
5.1.2 Software environment 118
5.2 The Traveling Salesperson 118
5.3 Matrix Multiplication 120
5.4 The Game of LIFE 123
5.5 Discussion 126

Chapter 6 Conclusion **128**
6.1 Contributions 128
6.2 Future work 129
6.3 Summary 130

Bibliography **131**

List of Tables

2.1 Summary of related projects 43

3.1 Description of TreadMarks 73
3.2 Description of DASH 74

5.1 Evaluation metrics 114
5.2 Best performing CM/CP combinations 126
5.3 Evaluation results 127

List of Figures

2.1 General model of shared memory 17
2.2 Schematic of a bus-based DSM system 19
2.3 2-dimensional mesh of a crossbar 20
2.4 KSR-1 Architecture 22
2.5 Simple example of race condition 31
2.6 History that satisfies processor consistency 33
2.7 History that satisfies cache consistency 34
2.8 Categorization of consistency models 37

3.1 Components of consistency maintenance protocols 48
3.2 Semantics of primitives 53
3.3 Semantics for a seq. cons. model 54
3.4 Semantics for a cache cons. model 55
3.5 Semantics for a proc. cons. model 56
3.6 Semantics for a weak cons. model 57
3.7 Semantics for a release cons. model 58
3.8 Semantics for a lazy release cons. model 59
3.9 Components of coherency protocols 65

4.1 Framework Overview 78
4.2 Interface of the Descriptor class 80
4.3 Interface of the Intention class 82

4.4 Remote method invocation 83
4.5 Example of a remote method invocation 84
4.6 Interface of the Lock class 86
4.7 Interface of the Barrier class 90
4.8 Master/Worker relationship 91
4.9 Consistency Model Class Hierarchy 95
4.10 Interface of the CoherencyProtocol class 98
4.11 Interface of the Memory management class 100
4.12 Interface of the Distribution management class 103
4.13 Interface of the ownership management class 103
4.14 Coherency Protocols Class Hierarchy 108
4.15 Interface of the SharedObject class 111

5.1 Speed-up Curve for TSP 121
5.2 Speed-up Curve for Matrix Multiplication 123
5.3 Communication between Strips 124
5.4 Distribution of Conway's Game of Life 125
5.5 Speed-up Curve for Conway's Game of Life 126

Chapter 1

Introduction

The programming of a single-processor system is a relatively simple task. A single-processor system handles one instruction at a time and stores results of operations in a local memory. All instructions see the same data and a read operation returns the result of the last write operation.

In contrast to single-processor systems, parallel systems process a number of tasks at the same time. These tasks exchange data with one another and need to be coordinated occasionally. The programming of these tasks is being facilitated by programming paradigms that focus on the development of programs for parallel systems.

The two prevalent programming paradigms for the development of parallel systems are message passing and (distributed) shared memory. These two paradigms approach the communication between nodes very differently. Message passing is based on user-involvement in the coordination of communication efforts; distributed shared memory attempts to be transparent to the developer.

1.1 Programming paradigms

Message passing is based on the exchange of messages between nodes. The programmer specifies individual programs that are to run on the nodes of a system and the messages and their distribution among the programs. The messages that are exchanged among nodes

contain only problem-specific data. The coordination of the individual programs is completely in the control of the programmer. The individual program executes on each node like a traditional program. The separation of an application into a number of individual programs makes the execution of the application easy to understand for programmers that are used single node applications.

The message passing paradigm is inconvenient when programming applications with a high amount of communication because all communication between nodes has to be arranged by the developer. The communication between a small set of nodes is not very complex and can be coordinated by the developer without great difficulty; however, it becomes increasingly difficult with a growing number of nodes. In other words, message passing scales very badly towards large parallel systems.

Shared memory offers a simple programming paradigm to the developer. The communication and synchronization between the tasks that are executed on nodes of a parallel system consists of memory accesses. The developer is used to this form of communication from single-processor systems. The system is responsible to provide an image of a single shared memory and the complexity of the underlying system is completely hidden from the developer.

1.2 Distributed Shared Memory systems

Early shared memory systems were implemented as closely-coupled systems where a set of processors share a common main memory. All accesses to shared and non-shared data is served by the same memory banks and the access time of every processor to shared data is identical (Uniform memory access = UMA). The event of distributed parallel systems with support for shared memory introduced an additional layer of indirection for the access of shared data. Shared data are now stored in a set of memory locations that are distributed among the nodes of the distributed system. The time to serve an access to shared data differs depending on the kind and on the origin of the access (Non-uniform memory access = NUMA[89, 30]). An access to locally held shared data is generally served faster than an access to remotely located shared data due to the time spent on communication with remote

nodes.

Shared memory systems based on distributed parallel systems employ protocols that translate memory accesses into messages between the nodes of a system. These protocols are called consistency maintenance protocols. They ensure that memory accesses on one node are visible to other nodes in a predictable manner.

The maintenance of consistency of shared data involves a number of steps. The node where an access originates needs to determine the memory locations that are involved in the access. The access may have to be synchronized with accesses made by other nodes in the system. This may require the exchange of synchronization messages. An access may consist of a retrieval or a modification of the value of a memory location. In case of a modification of the value of a memory location the new contents may have to be distributed among several nodes of the system who posses a replica of the data. This exchange of updates may involve the exchange of messages with one or more of the nodes in the system.

The predictability of the outcome of memory accesses is challenged by the complexity of distributed systems. A distributed system combines a number of individual and independent elements like computing nodes and network components that influence the timing of messages and the execution of operations. These factors complicate a predictable execution of a program in a distributed system.

The time a message takes to be delivered from the sender to the receiver in a distributed system depends on various factors. The sender employs a protocol stack that prepares and sends the message over the wire. A similar protocol stack is employed by the receiver. These protocol stacks handle incoming and outgoing messages and transform them into the format of underlying transport layers such as the physical layer. The execution of these protocols and the transfer of the messages of a phsical medium

The execution of operations in a distributed systems is - similar to the transfer of messages - complex. The concurrent execution of operations in a distributed system has a number of possible execution histories. Every computational node of a distributed system operates at its own individual speed. Operations that are executed on these nodes are performed out of sync with each other.

A protocol defines the synchronization of distributed operations and the handling of messages. This specification guarantees a predictability of the behaviour of memory accesses.

The main issue addressed by most protocols is the amount of data and the frequency of network traffic caused by consistency maintenance. Network traffic has proven to be the major bottleneck in distributed systems despite advances in the speed of transport media and communication protocols. Consistency maintenance protocols aim to reduce network traffic by coupling protocols close to the underlying communication protocol and by gathering messages into blocks of messages.

Recent shared memory systems feature consistency maintenance protocols that facilitate distributed shared memory on various platforms and to adjust systems to various requirements. A number of protocols provide enhancements such as gathering of writes in order to reduce network traffic, increased concurrency to enhance the time spent processing, etc.

Research in the area of DSM systems has produced an abundance of protocols – each with its individual strength and weaknesses. These protocols vary in the concurrency they allow and in the network traffic they cause. However, very few systems provide the flexibility to adjust the consistency maintenance to the needs of an application. Most systems focus on one aspect of distributed shared memory and limit themselves to a single solution that has to serve all applications.

As Munin[14] showed, not all applications exhibit the same access patterns to shared data. Applications share data in very specific ways. This can be exploited if the developer is allowed to provide knowledge about the data and their sharing characteristic. Data structures can hold for example initialization data. This data is distributed at the start of an application and not updated afterwards. The knowledge of this access pattern allows to reduce synchronization efforts in relation to this data.

1.3 Adaptability

Systems that are limited to a single protocol or a fixed set of protocols perform reasonably well for a number of problems but fail to perform adequately when handling other problems. A system that employs a protocol that is optimized for a large number of successive accesses

by individual nodes to shared data may perform poorly when presented with a problem that require individual concurrent accesses by individual nodes.

A number of systems[11, 14, 79] have attempted to adapt their behaviour to the usage pattern displayed by applications. These systems include a fixed set of protocols from which a suitable protocol is chosen according to the needs of the application. The protocol is chosen by a runtime analysis of the sharing characteristics exhibited by an application and a decision process that is coded into the DSM system when it is build.

This movement towards adaptation follows a trend in the general software development towards increased customizability. During recent years a number of techniques that aim towards customizability have found growing acceptance. Object-oriented programming, for instance, has produced the idea of frameworks of components that implement a number of solutions for a problem. These solutions are implemented as components with defined characteristics and interfaces. The components are interchangeable with each other and allow the developer to adapt an existing solution by exchanging a component currently in use by a system against a more suitable component according to the problem at hand.

Aspect-oriented programming and reflection[61] take adaptability a step further by providing developers with mechanisms that allow the separation of functional and non-functional code. The functional code of a system implements the core functionality which defines the system; the non-functional code implements supportive functionality like the persistence of data. The functional and non-functional code are combined by the reflective system to form an executable application.

The separation of concerns allows a developer to create functional code without the need to determine possible changes in the runtime environment. The application can be adapted to its execution environment by providing corresponding non-functional code. This adaptation happens without changes to the application or recompilation of the application code.

1.4 Thesis

In this thesis we argue that flexibility is the key characteristic that needs to be addressed in DSM systems. This characteristic is limited in existing solutions due to the lack of structure

in consistency maintenance protocols. We propose a new characterization for consistency maintenance protocols based on the division of these protocols into consistency models and coherency protocols. One of the contributions of this thesis is the precise definition of consistency models and coherency protocols as components of consistency maintenance protocols. This definition includes a description of all characteristics of the components and their subcomponents. The definition of these components and their interfaces is then used to describe their relationship. This description is novel to our knowledge.

This division into components and a further structuring into subcomponents allow us to adapt a consistency maintenance protocol to the sharing characteristics of a particular application. Every instantiation of a sub-component can be exchanged against another instantiation of the same sub-component. This exchangeability aids the adaptation.

The definition of the components and their relationships is translated into a framework that is implemented using Java. This framework provides the developer with interfaces and classes for consistency models and coherency protocols. The interfaces define methods that coherency protocols and consistency models must provide. The enforcement of these interfaces maintains the exchangeability of components of the same type in the framework. The classes provided by the framework can be divided into concrete and abstract classes. The concrete classes represent complete implementations of consistency models and coherency protocols. These implementations can be readily used in applications as building blocks for a consistency maintenance protocol. The abstract classes of the framework represent base classes that can be extended to create new implementations of consistency model and coherency protocols in order to adjust a consistency maintenance protocol to application-specific needs.

1.5 Evaluation

In order to validate our approach we have implemented a number of applications using the framework. The applications have been chosen according to their sharing characteristics. They represent implementations of general well-known problems that have been used in evaluations of available DSM systems. The sharing characteristics of these applications represent

a selection from the spectrum of possible sharing characteristics. The flexibility that is offered by the framework is exploited to adjust the consistency maintenance to suit the individual sharing characteristic. It is demonstrated how consistency maintenance protocols can be created by combining individual components that are provided by the framework. This shows the general applicability of our approach. A further demonstration of the framework shows how the flexibility provided can be used to create solutions that have not been possible before because of the lack of structure in consistency maintenance protocols.

These implementations demonstrate that the definition of the components of consistency maintenance protocols and the transfer of these definitions into a framework results in a high degree of flexibility. This level of flexibility surpasses the flexibility that is offered by traditional approaches to DSM systems.

1.6 Road Map

The remainder of this text is structured as follows: Chapter 2 gives an overview of the related work in the area of distributed shared memory. This chapter provides the fundamental background information that is used in the subsequent chapter to develop a new characterization for DSM systems. Chapter 3 gives a detailed description of our new characterization of consistency maintenance protocols. This characterization is translated in chapter 4 into a DSM framework. Chapter 5 presents a set of applications that have been implemented using the framework. The performance shown by these applications is used to evaluate the framework and to support our thesis. The thesis concludes with a chapter on the conclusions that can be made from our characterization and from the evaluation of our framework.

1.7 Summary

Distributed shared memory is better suited as programming paradigm for large-scale parallel systems than message passing because of the elimination of the developer from the process of coordinating the communication. This elimination of developer-involvement, however, removes the ability from the developer to adjust the behaviour of a DSM system to the

characteristics of an application. A number of DSM systems have attempted to adapt to the characteristics of an application automatically or allowed limited developer intervention to adjust a DSM system. However, most systems only allow being adapted within certain parameters and prevent a complete adaption of the DSM system. We propose a new characterization of consistency maintenance protocols that contains a description of the main components of a protocol and their interaction. The definition of the components is flexible enough to to encompass all existing protocols and allows components to be adjusted to new requirements. The framework that is derived from this definition demonstrates the implementation of traditional and new protocols.

Chapter 2

Related work

A number of ideas and concepts that are found in distributed shared memory systems can trace their origin in the area of parallel hardware. In order to place the description of our work in context we review the designs and concepts of parallel and distributed systems. This review can also be understood as the motivation in our search for increased flexibility and the resulting definition of our characterization of consistency maintenance protocols.

The first section of this chapter will relate trends in the general area of distributed systems area. These trends can be seen as the background and motivation for the work presented in this thesis. The chapter continues with a description of relevant types of DSM systems and issues of consistency maintenance. This description is followed by a detailed explanation of consistency models. The chapter is concluded with a section about DSM projects that share a close relationship with the work presented in this thesis.

2.1 Motivation

Software systems research has two aims: To improve the performance of software systems and to improve the methods that are used to develop software. These two aims tend to conflict with each other. The first aim favours problem-specific solutions where the second emphasizes generality. The following section will clarify the meaning of performance of software systems, the methods of software development and relate the cause of the conflict between the two

aims.

Performance of software is generally indicated by the speed with which a problem is solved. One of the most influential factors that determine the speed of a solution is the degree of particularity of the implementation for a specific problem. In order to solve a problem most efficiently a program has to be implemented as problem-specific as possible. An implementation has to take advantage of all possible factors that have an influence on the execution speed of the application. These factors include problem-specific information such as value constraints and frame conditions of formulas, information about the hardware that is used as implementation base such as hardware implemented acceleration and information about the language that is used to implement the solution such as the handling of matrices in Fortran.

All these factors create an enormous space of possible solutions where each set of values represents an efficient solution for a particular problem. These solutions are implemented as protocols for various implementations. Each of these protocols has a different emphasis and represents an efficient solution for a particular case. In order to build an efficient application a developer has to choose a number of these protocols and combine them into an application. This combination of protocols implemented for one application can usually not be reused for other applications.

This method – of building extremely problem-specific solutions – conflicts with the quest for ever-faster software development mechanisms. The search for faster software development methods is motivated by industry's desire to speed-up the development of software and to achieve a faster time to the market for their products.

In order to develop software faster, the use of high-level language abstractions and library constructs is suggested. High-level language abstractions reduce the development effort by combining a number of low-level operations. A simple example for such a high-level construct is a function call. A function call appears to the developer as one atomic statement of a program. The compiler translates this statement into a number of low-level operations that handle possible parameters of the function call, save the current processor state and manipulate registers.

However, high-level constructs inhibit the development of highly specialized solutions. Every high-level construct is made up of a number of low-level operations. These operations are the same in every context where the high-level construct is used. Thus an implementation of a high-level construct cannot take advantage of all performance enhancements that may be available.

A possible solution to the conflict between the aim of high-level solution and the use of highly problem-specific code is adaptability. Adaptability is a technique that enables a system to employ high-level solutions while the underlying code is highly problem-specific. An adaptable system contains a number of implementation of protocols with similar functionality. A protocol is chosen to suit a given environment and can be exchanged for another protocol if the characteristics of an environment change.

For example, a chat system may provide the abilities to send and receive messages. Two possible environments for this system can be envisioned: closely-coupled sytems such as clusters and a set of computers connected by a wide-area network. The delivery of messages in these environments differs in terms of reliability. In most closely-coupled systems, the delivery of network traffic is very reliable. The system does not need to ensure that a packet - once it is send - arrives at the receiver in this kind of environment. However, if the system is used in connection with a wide-area network, the delivery of messages becomes more complex and the reliability is often questionable. In this environment, the communication system has to provide additional functionality that ensures the delivery of messages.

The system may have two implementations for message delivery that are each suitable for the corresponding environment. A suitable implementation is chosen according to the environment.

2.2 Adaptability in DSM systems

The movement in the general computer science community towards adaptability is reflected in the DSM area. The subjects of adaptability in DSM systems are protocols that are employed to ensure the consistency of shared data. The following categories of adaptability can be identified:

- System adaptability
- Protocol adaptability
- Object-based adaptability
- Runtime adaptability
- Dynamic adaptability

2.2.1 System adaptability

System adaptability is adopted by DSM systems that employ system-wide a single protocol to ensure the consistency of data. A protocol is chosen before the deployment of the system, i.e., at compile-time. This protocol stays fixed during the lifetime of a system, i.e., until a recompilation of the system.

Systems that provide system adaptability allow protocols to be changed in order to adapt to relatively infrequent changes in the environment, e.g., the adaptation to a new hardware platform. These infrequent changes may require more effort from the maintainer of a system and a reconfiguration and recompilation of the system is necessary to achieve a change.

This type of adaptability is generally facilitated by the definition of interfaces. An interface specifies the access to the implementations of protocols. It defines the means that a system uses to access a protocol and hides the internal functionality of an implementation.

A protocol for a system that provides system adaptability needs to be developed against such an interface. An interface has to accommodate all communication between a protocol and a system. A change in the interface requires changes in all protocols that are to be used with the system. A new protocol that is installed in a system replaces an existing protocol. No two protocols are supported by the system at any time.

This mechanism requires that all conditions of a system and its applications are known prior to its deployment in order to support the correct choice of a protocol. This choice, once made, has to serve all applications that are executed on the system. The replacement of a protocol during the lifetime of a system is not possible.

System adaptability is generally provided by DSM systems that are implemented as part of an operating system [38, 37, 62]. It has its equivalent in the white box approach [60] that is employed in the general computer science community.

2.2.2 Protocol adaptability

Protocol adaptability represents the runtime-equivalent to system-adaptability. It allows a user to choose a system-wide protocol prior to the execution of an application.

This mechanism requires the implementation of a number of protocols in the system prior to its deployment. The number of protocols remains fixed during the lifetime of a system. A system has to be re-deployed if a protocol is to be added to an existing set of protocols. This redeployment is the equivalent of an introduction of a new protocol into a system that provides system adaptability.

A user chooses at compile- or run-time of an application a protocol that is suited to a particular problem at hand. This protocol is then used during the runtime of the system to ensure the consistency of all shared data.

This mechanism usually introduces an intermediate layer through which all protocols are accessible. This intermediate layer represents an additional indirection during the execution of the system. Compared to the introduction of an interface in system adaptability this mechanism needs additional information in order to provide the flexibility between different protocols.

The choice of the protocol is usually implemented as switch or configuration option [93]. Different instances of the system that are executed at the same time can use different protocols.

Protocol adaptability is provided by systems that are implemented as standalone systems or libraries such as CVM [55, 56].

2.2.3 Object-based adaptability

Object-based adaptability can be found in systems that provide the sharing of variably sized regions or objects rather than the sharing of fixed-sized memory pages. These object-based

systems or Distributed Shared Object (DSO) systems allow a developer to define the granularity of sharing.

Johnson et al. developed with the *C Regions Library (CRL)* [50, 51] an all-software DSM system that is implemented as a library for C programs. The system allows the developer to specify any kind of C structure that is to be shared. The structures are kept consistent by a single protocol that reacts on annotations given by the programmer.

Bershad et al. implemented a similar approach with *Midway* [19, 95]. Midway is implemented as a compiler and a runtime environment. Objects in this system are associated with a synchronization object. Accesses to objects trigger a mechanism that invokes the synchronization object and through this the underlying protocol that provides the consistency maintenance.

Both CRL and Midway are object-based systems that provide a single protocol that enforces the consistency among objects in the system. These systems are object-based but do not provide object-based adaptability. Object-based adaptability defines the specification of protocols on a per-object basis. This means that systems that provide object-based adaptability implement a number of protocols that can be assigned on a per-object basis. This mechanism allows the adaptation of a DSO system to the needs of an application in a fine-grained manner.

A number of systems [74, 75, 53] require annotations to the program code by developers. These annotations determine the protocols that are to be used. Orca [11] avoids annotations and achieves object-adaptability through the analysis of source code at compile time. The analysis determines the sharing patterns of individual objects and advises an underlying runtime system on the choice of correct protocols.

Systems that provide object-based adaptability require the implementation of protocols prior to the deployment of a system. The number of these protocols remains fixed during the lifetime of the system.

A closely related variation of object-based adaptability is class-based adaptability. In this type of adaptability a protocol can be specified on a per-class-basis. Every object of a specific class uses the same protocol. This is a little less fine grained than object-based adaptability

but has the advantage that a developer can develop a class library and specify a protocol for a given class; users of this library than do not have to be concerned with the choice of a protocol when they instantiate an object.

2.2.4 Runtime adaptability

Runtime adaptability describes the ability to adapt a system at runtime to the requirements exhibited by an application. This kind of adaptability is achieved by analyzing the runtime behaviour of an application and by changing the underlying protocol according to the results of this analysis.

In contrast to systems with object-based adaptability, runtime adaptability does not require annotations from the developer. However, the analysis of runtime behaviour has a performance cost. This cost has to be outweighed by the gain that the change from one protocol to another can produce.

2.2.5 Dynamic adaptability

Dynamic adaptability represents a combination of object-based adaptability and runtime adaptability. The assignment of a protocol to a particular object can be changed during runtime. This change can be determined either by a runtime analysis of the sharing characteristic of an object or by annotations given by a developer.

Orca provides a runtime environment that monitors the access characteristics of individual objects and decides on the basis of heuristics if a change from one protocol to another would bring a benefit. Additionally, Orca provides a programming language that allows annotations to shared objects. With this mechanism it allows the developer to help the runtime environment to find an appropriate protocol.

This type of adaptability provides the most flexibility but also introduces the largest amount of indirection in comparison to the other types of adaptability. The performance gain that is achieved by exploiting the flexibility has to outweigh the costs for this adaptability.

2.2.6 Summary of Adaptability Categories

Very few DSM systems achieve a high degree of adaptation. This is due to a number of issues. Research implementations of DSM systems are built to prove a very specific point. Very few systems are intended to prove that adaptability is an issue and that a certain adaptation mechanism provides an improvement over existing systems. Another issue is that hardware implementations prevent the introduction of new algorithms. Hardware-based implementations of DSM system have a fixed number of protocols implemented. These protocols can not be modified or extended because of the nature of hardware.

In the following section we will describe a range of types of DSM systems. In the progress of this section it will become clear that the development of the architecture of DSM systems moved from hardware-based DSM systems towards software-based DSM systems that provide increasingly adaptability.

2.3 Types of DSM systems

The general model of a distributed system (as shown in figure 2.1) is made up of a number of processor nodes that are linked by an interconnection.

An interconnection supports the communication of processor nodes with each other and can have a variety of forms. One extreme form of an interconnection is a bus-based architecture where all nodes share a common medium. All nodes exchange messages over this medium and are able to listen to all messages the medium transports. This arrangement has two significant characteristics: 1) every communication affects all nodes and 2) communication of one node with a group of nodes is very cheap. The other extreme form of interconnection is a crossswitch where each node is linked with every other node with a direct, individual connection. This form allows two nodes to communicate without interfering with the communication of other nodes; however, the communication of one node to a number of other nodes is more expensive than in the former case.

Other forms of interconnection attempt to balance between these two extremes. They generally try to balance the connection of all nodes versus the interference of concurrent mes-

sages. An example of this is the arrangement of of one of the extreme forms in a hierarchical order.

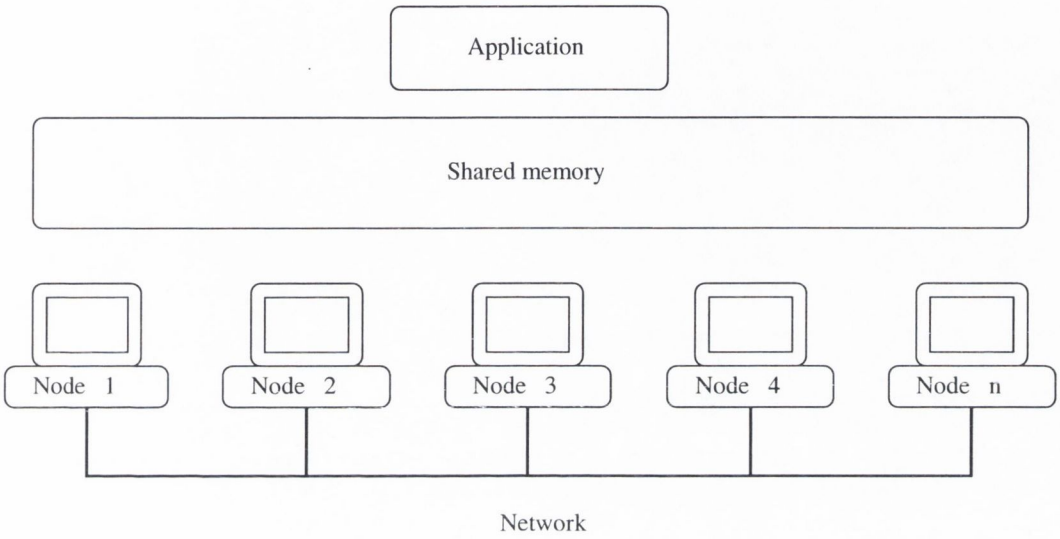


Fig. 2.1: General model of shared memory

A shared memory system introduces a shared state into this model. The shared state is implemented by a shared main memory. The communication between processor nodes and the main memory is transported over the interconnection. In the simplest case a main memory has one connection to the interconnection. The characteristic of the bottleneck increases if either the interconnection or the connection to the main memory supports only a communication with one processor at a time

Caches are introduced into this scenario to limit communication over the interconnection. They are co-located with processor nodes and hold copies of data from a location that has been accessed by a processor node before. In the simplest case, when a processor reads the same location again the cache can provide the data without communicating with the main memory. A write operation to the same location should cause the cache and the main memory to be updated.

The application of caches introduces a characteristic known as cache anomaly: A value written by one processor node can be seen by other processor nodes and by the main memory at different times and a set of accesses to the memory can be seen by different processors in

different orders.

In his paper on sequential consistency [63] Lamport identifies 3 possible scenarios that may lead a processor to read incorrect data:

- A value is written to one memory location and another memory location is read subsequently. The read operation may be served before the write operation has been propagated throughout the system. This may lead to another processor reading the old value of the first memory location. This scenario may cause problems if the read operations are used to determine the entry to a critical section.
- A processor writes a value into its cache where it is stored but not immediately propagated to the main memory. Another processor that accesses the same memory location is provided with a different value.
- A processor writes a value into its cache and the value is propagated to the main memory. Another processor reads the value of the same memory location that has been stored in its cache.

This anomaly can be seen as the source for the development of distributed shared memory architectures. The research in DSM systems has produced a number of different systems with a variety of individual protocols and target areas. Each of these systems implements the abstraction of caches in a different way. The systems can be classified into general categories according to their overall implementation as follows:

- Hardware-assisted shared memory
- Page-based software systems
- Object-based systems

2.3.1 Hardware-assisted shared memory

Hardware-assisted shared memory has been derived from closely coupled shared memory architectures such as Burrough's D825 [7] or Univac's LARC[8]. These architectures feature fixed interconnections that are realized with bus- or crossbar-technologies.

A bus connects a number of nodes with a single communication medium. All nodes share this medium. A message that is created by one node is propagated along the bus to all other nodes. This allows all nodes to listen to all communication on the bus.

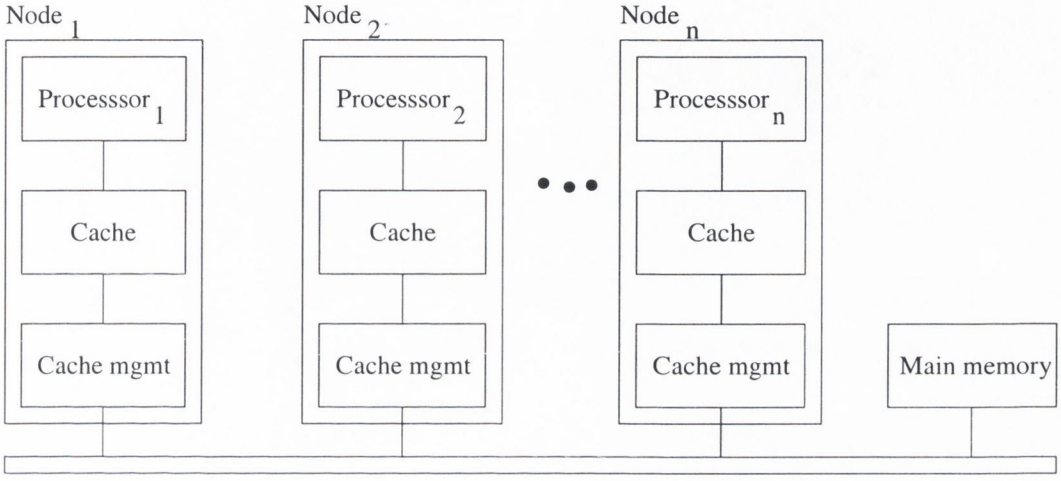


Fig. 2.2: Schematic of a bus-based DSM system

Figure 2.2 depicts a typical model of a bus-based shared memory system. The system consists of a number of processors nodes and a main memory. The processor nodes contain a processor with a co-located cache and a cache management unit. The cache management units and the main memory communicate using the bus.

A read memory access by one of the processor is handled by the cache. If the cache cannot satisfy the access the cache management attempts to retrieve the data from the main memory or from one of the other caches. A number of protocols have been put forward to handle the exchange of data between caches and main memory. These protocols are called snooping protocols because of their characteristic to listen to communication on the bus at all times. Common snooping protocols include the Illinois- [33], Berkley- [32], Firefly- [92] and Dragon-snooping protocols [9].

A crossbar architecture connects every node of a system with every other node with one hop. This results in a 2-dimensional mesh as depicted in Figure 2.3. Every node can communicate with every other node without interfering other nodes; except if these nodes want to communicate with the same node at the same time.

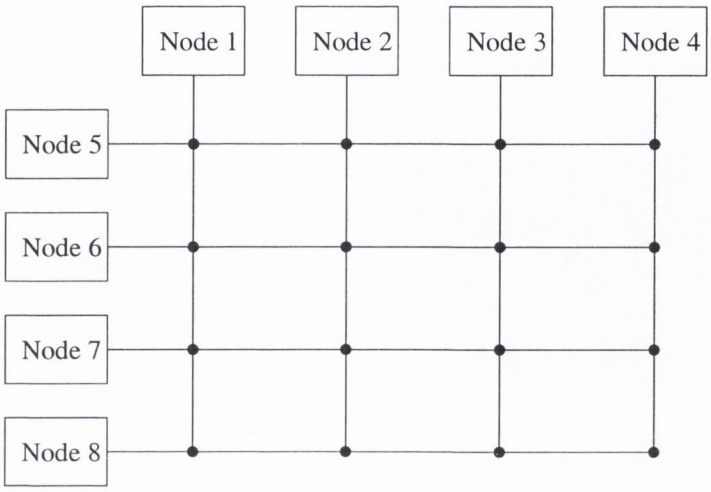


Fig. 2.3: 2-dimensional mesh of a crossbar

Shared memory systems with crossbar architecture place a number of nodes on one side of the mesh and a number of memory modules on the other side. This way, individual memory modules can be accessed without interfering memory accesses to other modules. This enhances concurrency. Introducing caches into this picture is ugly because unlike on bus-based systems nodes cannot monitor accesses by other nodes and thus have to communicate with the memory modules that then might have to invalidate copies of data kept by other processors.

Both, bus- and crossbar-technologies, exhibit limited scalability. Buses limit the number of nodes that can be connected to them because all nodes share the same medium and only one node can send over this medium at any time. Other nodes that want to send something have to wait until a sender has finished and then compete for the right to be the next sender. With an increasing number of nodes, the risk of communication conflicts grows and the delay of communication over the bus increases. Crossbar architectures are limited in their scalability because they require for each node that is added a connection to every other node. The number of connections grows thus exponentially with the number of nodes and make this requirement only manageable for a small number of nodes.

A number of systems have attempted to combine the performance advantage of hardware shared memory system with the flexibility of loosely coupled systems and software shared

memory systems.

The *DASH* system [67, 66, 65, 68, 44] combined bus-based shared memory systems with a general-purpose mesh. The bus-based shared memory systems are SGI Power Station 4D/340. These systems contain 4 processors with co-located caches and a shared main-memory. The consistency of the main-memory and the caches is maintained by the Illinois- or MESI (modified, exclusive, shared, invalid)-snooping protocol.

A memory access is handled by a local cache first. If the local cache cannot satisfy the memory access all caches register the access. A write access to a particular cache line causes an invalidation of cache lines for the same address on other processor caches. On a read of such an invalidated cache line a processor requests an update from the shared memory. DASH combined a number of these hardware-based shared memory systems to a cluster by connecting them with a 2D mesh. The connection among machines is provided by a directory controller (DC). Each node in the mesh holds a predefined address range. The DC keeps track of all memory accesses and resolves all memory accesses to address ranges that are not maintained by the local machine. A memory access to a remote memory address involves communication over the mesh to the node that acts as home for the address range.

The interesting aspect of the DASH system is the combination of bus-based shared memory systems and a general-purpose network mesh. This combination allows the DASH systems to avoid the limitation of bus-based systems and creates a 3-tier structure for memory accesses. This 3-tier structure introduces new possibilities for performance gains by carefully locating the data on nodes that access these data most frequently. Data placement has been an issue of research before but DASH introduced another dimension into the known scenario by having 2 platforms in which the placement makes a difference. By placing the data on a 4-processor node that makes most use of these data another performance gain is possible.

Kendall Square Research Corporation patented the KSR family of multi-processor systems. The *KSR-1* [26, 86] (as shown in Figure 2.4) was the first implementation of systems of this family. Each node in the KSR-1 consists of a 64bit RISC processor with a co-located cache, a 32MB second-level cache, a cache control unit and an interconnect unit. The page size of the second-level cache is 16 kilobyte and the size of a cache-line in the first-level cache

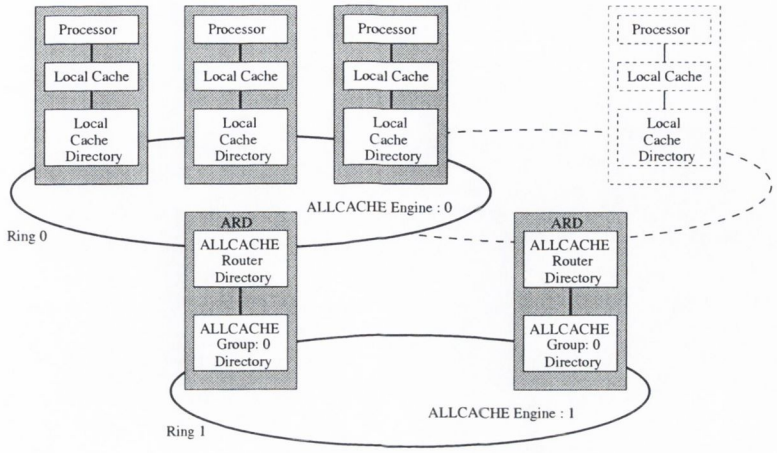


Fig. 2.4: KSR-1 Architecture

is 128 bytes. The processor was clocked at 20MHz and each node had a peak performance of 20 MIPS.

Nodes in the KSR-1 are connected in unidirectional rings with a capacity of 1 Gigabyte/sec. These rings are called ring:0 or leaf rings. Each ring can hold 32 nodes and can be connected to other rings by a second-level ring and thus form a hierarchical structure of rings. Second-level rings are called ring:1 and have a capacity of 1-4 Gigabyte/sec. A configuration of a KSR-1 could range from 8 to 1088 nodes and have a peak performance of 21,750 MIPS.

The memory of the KSR-1 is made up from the second-level caches of all nodes. All data in the system is held in the caches of individual nodes and no node possesses a main memory. This type of memory system is called a cache-only memory architecture (COMA) [89, 46, 84]. The consistency of all caches in the system is maintained by the ALLCACHE memory system. The ALLCACHE system ensures that all accesses to data occur according to sequential consistency.

The ALLCACHE system keeps track of cache lines by maintaining directories of the location of cache lines for each ring. Ring controllers that connect a ring:0 to a ring:1 maintain a directory for all pages in a ring. A memory access that cannot be satisfied by a local cache causes a search for the cache line in the local ring. If the access cannot be satisfied within a ring, a search through all group directories in a ring:1 is invoked.

The interesting aspect of the KSR-1 is again its approach to solve the limited scalability

of a ring structure. The scalability of a ring structure is similar to that of a bus: The round-trip time for a message increases with the number of nodes; the addition of too many nodes impedes the performance of the system dramatically. The KSR-1 avoids this scalability issue by applying a multi-tier structure in form of a hierarchical organization. This approach gives the KSR-1 similar characteristics to the DASH system in terms of data placement: The time of accesses to data increases with the number of tiers that have to be passed. However the organization of nodes into rings differs from bus-based systems in two ways: All messages have to pass all nodes between the sender and the recipient; but compared to bus-based systems not all nodes see a message on the ring. This means that snooping protocols are more complex to implement.

SCI [49] represents a more recent implementation of hardware-based shared memory system. SCI provides an interconnection that is connected to the memory of a node through the Direct Memory Architecture (DMA). The user can share memory between individual nodes by mapping memory from a remote node into the virtual address space of local processes. In the event of a memory access to a remote address an SCI interface identifies the nature of a remote memory access, determines the node that needs to be contacted and realizes the memory transfer.

To summarize this section: Most hardware-based systems employ a single protocol. This protocol cannot be changed during the lifetime of the system. The granularity of sharing is fixed to the size of cache lines and to page sizes. The development of hardware-based DSM systems is mostly concerned with scalability issues.

2.3.2 Page-based software systems

Page-based systems rely on the abstraction of virtual memory [72, 70]. This abstraction is based on the fact that computer systems generally have a larger address space than physical memory is available. In order to use more memory than is physically available a system provides a virtual address space. Accesses to virtual addresses are mapped to physically available addresses. The system has to resolve conflicts when a virtual address is not mapped to a physical address at the moment of an access. This conflict resolution usually involves

loading the memory contents that is expected at the virtual address into physical memory and mapping the virtual address to the physical address where the data has been stored.

The memory address space is divided into memory pages for administrative purposes. A system fills up available physical memory pages until a threshold is reached. A number of memory pages that have been filled may not be accessed very often. The contents of these pages can be moved to a hard disk and their space made available for other pages. The swapped-out pages are brought into the physical memory only when they are accessed again.

Many hardware platforms provide a memory management unit (MMU) to support virtual memory. This MMU handles the translation from virtual to physical addresses. A memory access triggers a function in the MMU that verifies if the memory page referenced by a virtual address is in physical memory. If the page is not in physical memory a function in the operating system is executed to allow the operating system to transfer memory contents from the hard drive into physical memory.

A number of DSM systems use virtual memory mechanisms to regulate the access to shared data. The notification of accesses to invalid memory pages can be used to handle accesses to pages that are held on remote nodes. Instead of causing the system to swap in a memory page from disk the system acquires a valid copy of the page from a remote note.

Systems that use this mechanisms can be implemented in a variety of ways as shown in [72, 19, 82]. The implementation methods can be classified into two main categories: kernel-space and user-space implementations. Kernel-space implementations are more efficient because the access to administrative structures for virtual memory in the kernel does not require context switches. However, kernel-space implementations have the disadvantage that there is one implementation of the system for the whole operating system. Thus, a kernel-space implementation imposes a single protocol on all applications.

A user-level implementation depends on the operating system to provide access to the underlying memory management. The access is provided through a collection of functions that allow the user to provide functions that are invoked when a page fault occurs. This mechanism allows each user to execute its own implementation of shared memory system and enables users to choose their own protocols according to the needs of their applications.

The *Shared Memory Server* implemented by Forin [38] et al for the Mach micro-kernel [24, 25] attempts to combine the advantages of kernel-space implementation and user-level control. Mach exports the interface for virtual memory managers to the user-level and lets a user define individual virtual memory managers. This allows a user to install a memory manager that is suitable for the particular sharing characteristics of an application.

The *TreadMarks* [57, 58, 6] system represents an implementation of a DSM system as a user-level library. The library is based on Unix System V system calls that provide a mechanism to control the access rights to portions of memory. The mprotect() system call provides a mechanism to set the protection bits of virtual pages. This mechanism allows the interception of accesses to invalid pages and the execution of functions that retrieve valid copies of these pages.

The implementation of TreadMarks as user-level library allows users to execute applications independently from each other. A program using TreadMarks is started at one node. This program determines a list of available machines and the execution environment of TreadMarks spawns a copy of the program on each of these machines. This mechanism resembles the *fork*-mechanism of Unix environments with the difference that each copy of the program is executed on a different node. The program use functions provided by TreadMarks to allocate memory on the local machine and to declare it shared. The underlying execution environment of TreadMarks distributes the information about the shared memory sections to the other programs. The consistency of the shared memory is maintained by exploiting the interception of page fault. A page fault causes an execution of a function in the TreadMarks execution environment. This function ensures the consistency of the information that was requested when the page fault occurred.

TreadMarks provides the developer with an implementation of lazy release consistency. The developer needs to annotate the code with synchronization statements that mark locations where the access to shared data needs to be consistent. These annotations are in contradiction with the transparency that makes the shared memory paradigm attractive. Most hardware-based systems allow the programmer to share memory without the need for annotations by providing a strict consistency model because the control over accesses is very

fine grained in these systems. Software-based systems, however, have the problem that monitoring accesses is expensive because with every access operation additional operations have to be executed to ensure the validity of the access. This makes consistency models that do not rely on monitoring individual accesses like relaxed consistency models more attractive to software-based systems. These consistency models, however, depend on information about the beginning and end of critical sections.

Another critical issue for page-based systems is the fixed large granularity of page sizes. This fixed granularity introduces two main issues: false sharing and heterogeneity problems.

False sharing[22, 34] occurs when two independent data items are placed within the same page. A write operation to one data item causes generally all other copies of this page to be invalidated. In the case of two data items accessed by two nodes concurrently over some time this can lead to a "ping-pong" effect in which the page is transferred back and forth between the two processors involved. TreadMarks solves the problems by applying a method developed first in Munin [14]: Each processor copies a page before an update is allowed. This "twin" of the original is kept until the critical section of accesses is finished. Then the twin and modified page are compared and the modifications or "diffs" are exchanged between interested nodes.

The fixed granularity of memory pages also causes problems when different architectures are combined in a heterogeneous environment. An architecture can implement either little or big endian byte-order and can implement any size for a memory page. The combination of different architectures in a heterogeneous environment makes an exchange of memory page a non-trivial issue if the byte-order and page size is different among the architectures involved.

Page-based systems are generally built from sets of workstations that are interconnected by general-purpose networks. These general-purpose networks exhibit an increased latency and reduced bandwidth compared to tightly-coupled solutions such as bus-based DSM systems. In general DSM systems have to balance the locality of data against the communication cost that are connected with having these data local to a processor. The limitation of the interconnections forces page-based systems to avoid communication even more than hardware-based DSM systems and turns the balance towards the reduction of communication.

Ivy [71] is one of the first DSM systems based on a virtual memory mechanism and represents a typical and rather simplified approach to this kind of DSM system implementation.

The introduction of multiple copies of shared data items and the change of the node that hold the most up-to-date copy resulted in a discussion of ownership. Li and Hudak [73] discuss a number of possible ownership management schemes. These schemes determine the node that manages the access to a page. Ownership management schemes can be coarsely categorized as fixed ownership management and dynamic ownership management.

Fixed ownership management determines the ownership of a page and the ownership of the page stays fixed with a node during the runtime of the system. All write accesses to a given page have to be made through the owner; nodes that are not owner of a page never get direct write access to this page. This mechanism simplifies the algorithm that is needed to keep the page consistent i.e. it is easy to linearize the access to a page. Everything is handled by one node thus all write accesses can be easily put in sequential order by this node. However, this simplified mechanism introduces a bottleneck and reduces the concurrency of write accesses. Because all writes have to go through the owner write can become blocked there.

Dynamic ownership management determines the ownership of a page during the runtime of a system. This allows the system to place a page where it is accessed most frequently. The mechanism introduces the problem of finding the owner of a page. These algorithms can be subdivided into two categories: Centralized and distributed ownership management.

Centralized dynamic ownership management assigns a management node to every page. The node keeps track of where the page has moved and as central information hub for the page. The node can determine access rights of the nodes that content for a page and can linearize the write accesses.

Distributed ownership managements can be classified into dynamic management and broadcast management.

In a system with *dynamic distributed ownership* management every node that accesses a page keeps a reference to the "probable" owner of that page. This information is updated every time more accurate information about an owner becomes available. If the information

about an owner of a page turns out to be false, it provides at least a "hint" where to find the owner of that page. Li and Hudak show that, in the worst case, the number of messages to locate a page depends on the number of times a page was searched for and on the number of processors contending for the page. The worst case according to Li and Hudak is O(p + K log p) where p is the number of processors and K is the number of times the page was searched.

The use of broadcast in distributed ownership management is similar to the use of buses in hardware-based shared memory systems. A faulting processing node broadcasts a request to find the present owner of a page. Tannenbaum et al [91] show with Orca that broadcasts in certain systems can be inexpensive when supported efficiently by hardware. However, broadcast is very expensive if it is not support by the underlying network infrastructure.

Mirage [36, 35] introduced a timed ownership for pages. A system provides every node with a time similar to traditional CPU time slice in multi-process environments. Every node can hold a page for a given time. If a request for this page is received during this time the time-slice is completed and the ownership of the page is then transferred to the requesting node.

The Mirage system is implemented as an extension of the Locus operating system on VAX 11/750s. The Locus system is a System V compliant Unix implementation. The page size in this system is 512 bytes. Every page in the system is part of a larger segment. Segments can be attached into the address space of a process.

Single-address space systems [94, 87, 47, 27, 29] can be based on a virtual memory abstraction similarly to page-based systems. However, page-based systems provide each process on a node with its own address space. A page on a node may be accessed by two programs. The memory manager on this node has to maintain the state of the page for both programs and resolve any conflicts between these programs.

A single-address space system provides a single-address space for all interested nodes and all programs on these nodes. Every object in a single-address space is known by its address. This requires a very large address space.

2.3.3 Object-based systems

A third class of DSM systems includes object-based systems. These systems use the abstractions of object-oriented languages as a basis.

One of the important aspects of object-orientation in this context is the encapsulation of data in objects. The data items that make up an object can only be accessed through methods that are provided by the object. This allows the programmer to control the access to these data items and to introduce instruction for the consistency maintenance if needed. Thus an object-oriented DSM system can be implemented without relying on support from the underlying operating system.

Another important aspect of object-based systems is the definition of granularity. Object-based systems place the definition of granularity in the hands of the developer by enabling the developer to define objects and thus the size of the shared data items. Thus the developer can introduce structures of varying sizes rather than memory pages with a fixed size that is determined by the underlying operating system and hardware. This means that the granularity of sharing can be adapted to individual applications.

A number of object-based systems make use of inheritance in order to introduce support for consistency maintenance, either in a base-class or in a sub-class. In the first case, all objects that are to be maintained by this mechanism inherit from the base-class. In the second case, a base-class implements the functionality of an object. This functionality is then extended through inheritance by the functionality for consistency maintenance.

A number of systems have explored different aspects of object-based systems:

ORCA [11, 12] chooses a suitable mechanism for sharing on a per-object basis. The system introduces its own language. The compiler for this language generates information about the access characteristics of every object at compile time and analyzes these heuristics at runtime. On basis of this analysis either replication or migration is chosen as sharing mechanism for an object. This combination of compiler and runtime support allows a per-object adaptation transparent to the developer.

Midway [19, 95] uses the encapsulation of object-orientation to activate the synchronization of shared objects. Every object is combined with an individual lock. A method call to

an object triggers an acquisition of the related lock. The acquisition of a lock causes both the lock and updates to shared object to be transfered to the local node. This allows the implementation of a transparent sharing mechanism without the necessity to introduce a compiler or runtime environment.

Rabbit [53] intercepts method calls and introduces functionality that support the maintenance and synchronization of copies. The functionality is implemented in components that are implemented as separate objects and can be modified by the developer. This allows the developer to adjust the synchronization and consistency maintenance to the need of an application.

2.3.4 Summary

The categories of DSM systems listed above are arranged after increasing adaptability. The development of this increasing adaptability reflects almost the chronological development of DSM systems. Early DSM systems were implemented fully in hardware and therefore difficult to adapt. More recent systems tend to be implemented in software and allow the developer to adapt the consistency maintenance mechanisms.

The following section will describe consistency models in detail. These models represent one of essential parts of DSM systems and are a key issue in achieving adaptability.

2.4 Consistency models

A consistency model defines the order in which accesses to shared data are experienced by participating nodes. In a single-processor system a read operation is assumed to return the last written value to a memory location. This assumption fails to hold in a distributed environment for a number of reasons.

In a single-processor system the order of instructions specified in a program identifies exactly the order in which these instructions are executed. In a distributed environment a number of programs are executed at the same time. The instructions of these programs are executed concurrently and individual instructions have individual execution times. This

makes the order in which instructions are executed difficult to determine. Thus, a program order like it exists in single-processor systems is not given in a distributed environment.

In a single-processor system it is easy to ensure that a write operation has concluded before a read operation is executed. In a distributed environment the atomicity of an operation is not given. A write operation to a memory location is first processed by the local node, then transmitted to all interested participants and where it is processed again by the individual nodes. The steps of this process are executed in a non-atomic manner and the result of a read operation to a memory location depends on the progress of the last write operation. This phenomenon is known as a race condition.

A developer intuitively expects the behaviour of single-processor computer. Programs that have been developed with the assumption of such behaviour will perform poorly on multiprocessor computers and may deliver incorrect results.

```
Process 1                       Process 2

write(A, 1)                     write(B, 1)
if (read(B)=0) {                if (read(A)=0) {
   critical section                critical section
   }                               }
else                            else
   resolve conflict                resolve conflict
```

Fig. 2.5: Simple example of race condition

Lamport[63] gave a simple example of a race condition (shown in figure 2.5) in which two concurrent processes are expected to synchronize by using two shared variables. Each process writes into its own variable and then checks the status of the variable of the other process. A process may enter a critical section if the other process hasn't written to its variable yet. The execution of these two processes in a distributed environment may result in both process entering their critical sections at the same time.

Consistency models describe the behaviour that shared memory computers can exhibit with respect to the propagation of memory updates. The behaviour of consistency models

is generally described by histories of accesses that are valid or invalid by a given consistency model.

In the following we will explain a number of consistency models in an informal way using histories of accesses to demonstrate the definitions of individual models. Formal definitions of consistency models can be found in [83, 4, 3].

2.4.1 Strict consistency models

Strict consistency models describe consistency models that see a single memory access as the granularity to order. An access to shared memory is either a read operation or a write operation. Each of these operations is considered separate and atomic. Separate in this context means each operation is considered on its own and not in connection to previous or subsequent operations. The operations are assumed to be atomic in the sense that if a processor issues an operation, no other operation is issued from this processor before the operation is completed. This ensures the program order among operations for individual processors.

Sequential consistency (SC)

The first published description of a consistency model was formalized by Leslie Lamport [63]. He describes requirements for **sequential consistency** as follows:

The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This definition highlights two issues that are intuitively expected from memory systems: The linearizability of accesses and the execution of memory accesses in program order. The execution of memory accesses in program order is ensured if the operations comply with the requirement for atomicity. The linearizability of accesses describes the history of accesses. Linearizability is fulfilled if all accesses can be seen in a single sequential order from all processors.

Sequential consistency represents a very strict consistency model in that all processors

have to agree on an order in which all accesses are seen. This requirement can lead to a lot of communication because all processors need to be informed of accesses. The strictness of this model also reduces the concurrency of accesses to shared data because processors cannot access uncorrelated locations without interrupting other processors. For a discussion of efficient implementations of sequential consistency models (see [3]).

Atomic consistency (AC)

Atomic consistency [48], is a subtype of sequential consistency. It defines that all nodes see all accesses at exactly the same time. This definition represents the most intuitive consistency model as it comes closest to the behaviour of a single-processor system. It requires however the synchronization of all processors at the same time. This presents an even stricter requirement than sequential consistency and reduces the concurrency of accesses even further.

Processor consistency (PC)

Processor consistency was defined by Goodman [42]. It specifies that:

- A multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear [to any other processor] in the sequential order specified by its program.

This means that a number of accesses that are performed by one node are seen in the same order by all nodes. Accesses from two different nodes, however, can be seen in different order by individual nodes. Figure 2.6 shows a history of accesses that satisfies processor consistency.

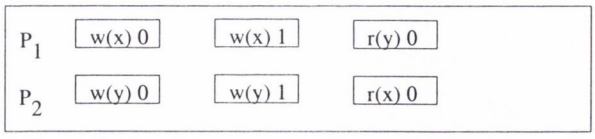


Fig. 2.6: History that satisfies processor consistency

Ahamad et al [4] give a formal definition of Goodman's specification of processor consistency. This definition extends Goldman's specification by introducing an order of accesses

to individual memory locations among processors. This extension differentiates the formal description from Goodman's [42] and Gharachorloo's et al [40].

The history as depicted in figure 2.6 does not satisfy this definition because the accesses to x can not be formed into a sequential order. A history as shown in figure 2.7 satisfies this definition of processor consistency because all accesses to individual memory locations can be seen in a sequential order.

Cache consistency (CC)

Cache consistency (or coherence) [42] defines a weaker model than processor consistency by limiting the order of accesses to individual memory locations. Processors have only to agree on an order on a per-location basis and do not consider accesses that are otherwise related.

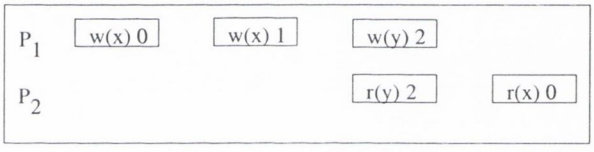


Fig. 2.7: History that satisfies cache consistency

This definition allows histories such a depicted in figure 2.7. This history is not linearizable with respect to the program order as one would expect the write operations to be executed before the read operations. However, as the operations issued at each processor are not related because they address different locations the program order of these operations can be ignored and the read operations can be executed before the write operations. This allows highly concurrent access to shared data that are unrelated without causing overhead; however, programs that rely on sequential consistency might fail.

2.4.2 Relaxed consistency models

All consistency models that have been described above order each access separately. These consistency models are commonly called strict consistency models [81]. Strict consistency models have been developed for closely coupled shared memory systems. The event of loosely coupled DSM systems such as networks of workstations lead to the development of relaxed

consistency models. Relaxed consistency models order accesses to shared data into blocks of accesses.

These types of consistency models require developers to annotate their programs in order to identify the critical sections that enclose the accesses to shared data. The beginning of a critical section is marked by an acquire operation and a section is terminated by a release operations. These annotations allow the mechanisms to gather the results of accesses and distribute them as one message. This reduces the network traffic and the synchronization overhead. Synchronization in these consistency models occurs only at certain points and not after every access.

Weak Consistency (WC)

Weak consistency was defined by Dubois et al [31]. It defines that accesses to shared data are weakly consistent if all blocks of accesses are sequentially consistent.

Gharachorloo et al. [41] describe different classes of access to shared data. These classes include ordinary and synchronization accesses. Synchronization accesses identify points in the program at which shared data should be synchronized. Ordinary accesses do not have this property.

These classes of accesses are used to give the conditions for weak consistency as follows:

1. before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and

2. before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and

3. synchronization accesses are sequentially consistent with respect to one another.

They continue to define a variation of weak consistency in which synchronization accesses have to be processor consistent with respect to one another instead of sequentially consistent as specified in point 3).

Release Consistency (RC)

Release consistency represents a refinement of weak consistency. It was defined by Gharachorloo et al and implemented in the DASH system.

The conditions for release consistency given in [41] are as follows:

1. before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and
2. before a release operation is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed, and
3. synchronization accesses are sequentially consistent with respect to one another.

This change of point 2) from the definition of weak consistency allows release consistency to perform accesses at a second processor as soon as a release operation is executed at a processor. This change enhances the concurrency of the model and gives release consistency an advantage over weak consistency.

Lazy Release Consistency (LRC)

Lazy release consistency was defined by Keleher et al [57] and was first implemented in the TreadMarks systems [58]. It defines that the results of accesses to shared data are exchanged at the occurrence of an acquire operation.

Entry consistency [18] is very similar to lazy release consistency. It was defined in the same period and its only variation is the fixed relation of locks and shared data. Entry consistency was implemented in Midway, an object oriented DSM system that does not rely on page-based systems mechanisms (see 2.3.2). The synchronization mechanism is used in order to determine the locations that have been accessed.

2.4.3 Categorization of consistency models

The consistency models described above can be categorized into strict and relaxed models as is depicted in figure 2.8.

- Strict consistency models
  - Atomic consistency [76]
  - Sequential consistency [63]
  - Processor consistency [42]
  - Causal consistency [5]
  - Cache consistency [42]
- Relaxed consistency models
  - Weak consistency [31]
  - Release consistency [41]
  - Lazy release consistency [57]
  - Entry consistency [18]

Fig. 2.8: Categorization of consistency models

A similar categorization has been made by Mosberger [81]. In his categorization, consistency models are divided into uniform and hybrid models. Hybrid models are defined as consistency models with different access categories. Access categories determine a division of access operations into several layer like synchronizing and non-synchronizing memory accesses. Uniform consistency models do not distinguish between access categories. In the description of the primitives we distinguish between access and synchronization operations. This distinction postpones the decision about the implementation of the synchronization operation and leaves the implementation based on special memory operations as one possibility.

2.4.4 Summary of consistency models

The previous sections gave descriptions of the most common consistency models. Each description identified the model in an informal way and gave a reference to where a formal definition of the model can be found. The descriptions were followed by a categorization of consistency models into strict and relaxed consistency models.

The following section describes a number of DSM systems that share a close relationship

with the work presented in this thesis.

2.5 Related projects

The quest for flexibility in DSM systems has produced a number of systems. In this section we describe DSM projects that are closely related to our work. Each project is given in an abstract, followed by a list of unique characteristics and a description of the project's contribution to the area. The descriptions are concluded with a paragraph about the relationship between the project and the work described in this thesis.

2.5.1 TreadMarks

TreadMarks represents a classical page-based system. It is implemented as a user-space library. The library makes use of standard System V system calls that provide access to virtual memory mechanisms of the underlying system. In this way the abstraction of virtual memory is used as described in section 2.3.2.

TreadMarks' major contribution is that it was the first implementation and evaluation of lazy release consistency. This consistency model allows TreadMarks to outperform other system on some benchmarks [57]. This performance advantage is the result of reduced network traffic compared to other consistency models. (Eager) Release consistency as described in TreadMarks [58, 54] transfers update notifications to all interested nodes after every release operation. TreadMarks' lazy release consistency implementation avoids sending messages until an acquire operation by another processor.

TreadMarks provides two synchronization mechanisms: locks and barriers. Locks are based on acquire and release operations. These operations map onto the synchronization operations for properly labeled programs specified by Gharachorloo et al. [41]. A barrier in TreadMarks is modeled as a release operation that is immediately followed by an acquire operation. Each node performs a release operation at the arrival at a barrier, and an acquire operation at the departure from the barrier.

TreadMarks implements single- and multiple-writer protocols. One version of these

multiple-writer protocols is based on barriers. Processors meet at a barrier and exchange diffs. These diffs are merged at each processor. Race conditions have to be resolved by the programmer.

The TreadMarks system has been ported to a variety of platforms such as SPARC, DECstation, HP and SGI platforms. It supports networking hardware for Ethernet and ATM. The Ethernet support uses packet-based UDP/IP and the ATM based implementation is based on connection-oriented AAL3/4. These communication mechanisms provide no guarantee for the delivery of messages; thus they are extended by TreadMarks with user-level protocols that ensure reliable delivery.

Maintaining twins and creating diffs has its cost in effort for copying and comparison and in storage for the twin. These costs have to be outweighed by the reduction of network traffic.

ADSM [79, 80] extended TreadMarks into the area of adaptable DSM systems. Similar to "Adaptable TreadMarks", it implements protocols for single-writer and multiple-writer. The adaptation is achieved by a dynamic assignment of protocols at runtime based on an analysis of the sharing characteristics of a shared data item.

TreadMarks and its successors proposed a variety of new mechanisms that extended the knowledge about the adaptability of DSM systems based on the VM abstraction. The description of these mechanisms is integrated into our characterization as presented in the next chapter.

2.5.2 CVM

CVM [55, 56] is not directly based on TreadMarks implementation; however, it represents an object-oriented recode of TreadMarks.

It implements a shared memory system based on a user-level library. The library provides the developer with primitives for synchronization and allocation of shared memory. The primitives are based on abstractions for memory pages and coherency protocols. The implementation of the library can thus be modified to work with different protocols. Memory that is allocated with the provided primitives is consistent at synchronization points that use the given synchronization primitives. The user can choose a consistency model from a set

of given consistency models by providing parameters to a program at the beginning of its execution.

CVM comes very close to our vision of a complete customizable system by opening the components of the system to the developer. It lacks however the possibility of using a combination of different consistency models for different types of shared data. We see this as a major advantage in our system and a necessity in order to be able to provide fine grained control over the sharing characteristics.

2.5.3 Munin

Munin [14, 16, 15] is a classical distributed shared object system based on a page-based DSM system. Each object in the system is either broken up into a number of page-sized objects or placed together with other objects into one memory page.

Munin provides the developer with a set of annotations for shared objects. These annotations represent access patterns for shared objects and aid the choice of parameters for protocols that underlie each object. Every protocol implements a sharing characteristic. The characteristics include: read-only, migratory, write-shared, producer-consumer, reduction, result and conventional. All protocols are based on release consistency.

The contribution of Munin lies in the identification of access patterns and the description of protocol parameters. The protocol parameters can be seen as questions that determine the access pattern that is best suited to a particular object. This choice of access patterns on a per-object basis is unique and has not been achieved by any other system.

An additional contribution of Munin is the introduction of 'diffs'. Diffs are changes made to a memory page. Before an access to a memory page takes place a copy of this page is made. When a critical section ends and a notification is to be created the original copy and the page are compared and only the difference between them (the *diff*) is exchanged with other interested parties. This avoids false sharing and allows a number of writers to change objects in a page independently and concurrently.

Munin differs from our framework in that it provides a limited set of algorithms. These algorithms are fixed in the system and cannot be adjusted to application needs, as is made

possible with our framework. All types provided by Munin can be implemented using our framework.

2.5.4 Orca

Orca [11, 12, 91, 10] is an object-based DSM system based on a language, a compiler and a runtime system. The Orca language is an object-oriented language that encapsulates the attributes of an object and restricts access to these attributes to the methods provided by an object. Every operation (method call) is seen as the granularity of accesses to order. These operations are seen as atomically and are ordered according to sequential consistency. Results of accesses to shared data are distributed using an update protocol that is based on broadcast. Update messages can either include function or data shipping meaning that either new values for objects are distributed or operations and their parameters are distributed and executed on remote locations.

Orca's compiler allows the system to analyze the code at compile time. It provides the runtime module with access patterns on a per-object basis. This information together with information gathered during the execution is used to determine the optimal placement strategy for individual objects. An object can either be replicated among interested processors or migrate to the location where it is used most.

The main contribution of Orca is its proof that an all-software system can outperform system based on hardware support. This performance advantage is due to the increased knowledge about the usage pattern exposed by shared data based on compile time analysis and the analysis of accesses during runtime.

Orca provides a set of protocols that are chosen by the system at runtime without involvement of the programmer. This mechanism relieves the programmer from annotating the code and makes the system completely transparent to the programmer. However, the fixed set of protocol prevents the system to be adjusted to an application without changing the complete system. Furthermore, the complete transparency of the system removes the possibility for the programmer to implement knowledge that might not be recognized/determined by the compile-time or runtime analysis.

2.5.5 Disom

Disom [43] presents an object-oriented framework for object sharing. Classes are made sharable by inheriting from particular super-classes. These super-classes define methods that facilitate the exchange of updates among interested nodes. On top of these methods different consistency models may be implemented.

The framework provides the developer with an implementation of entry consistency and additional consistency models can be added. The details of communication can be modified by the developer and adjusted to the individual application.

Disom differs from our framework in two major ways: the limitation of sharing types to classes and the restriction to a single communication protocol. We see it as essential to be able to share instances of the same class with different sharing characteristics. Furthermore the developer should be able to change to a suitable communication mechanism whenever necessary.

2.5.6 Rabbit

Rabbit [53] is similar to DISOM in that it provides an object-oriented framework for object sharing. The framework provides an interface for consistency maintenance and the developer can - based on this interface - define application specific consistency maintenance algorithms. The methods defined by the interface are called by the underlying system whenever a sharing event occurs. This mechanism allows the developer to implement sharing characteristics adjusted to the individual class.

The difference between Rabbit and our framework is, as with Disom, the limitation of sharing characteristics to the class level and the limitation of the underlying coherency protocol.

2.5.7 Summary of related projects

This section described the projects that have had the most influence in the work that is presented in later chapters. The projects exhibit a wide variety of aspects of DSM systems. They provide varying degrees of transparency and divers support for adaptability.

Project	Influential characteristic
TreadMarks	Adaptation between single writer / multiple writer protocols
CVM	Implementation of a set of consistency models
Munin	Introduction of a number of protocols to represent individual sharing characteristics
Orca	Dynamic adaptation between replication and migration
Disom	Combination of objects and protocols
Rabbit	Implementation of application-specific consistency maintenance mechanism

Table 2.1: Summary of related projects

Table 2.5.7 shows the projects and their most influential characteristic. TreadMarks' most influential contribution - in context of this thesis - is the investigation of single writer/multiple writer protocols and the change at runtime between them. CVM represents on of the systems that implement a number of consistency models and offer the developer to choose a suitable model at runtime. Munin introduced a set of protocols to represent the sharing characteristics of individual shared data items. Orca has been noted here for its dynamic adaptation to the runtime behaviour of an application by choosing between replication and migration. Disom represents one of the projects that provide the ability to implement consistency models on a per class-basis. The implementation of an application-specific consistency maintenance mechanism is Rabbit's most influential characteristic.

All these characteristics have had an impact on the characterization of consistency maintenance protocols that is presented in the following chapter.

2.6 Summary of related work

This chapter presented the work in the area of distributed shared memory related to this thesis. It provides the background information for the coming chapters.

The chapter started with a motivation for the work presented in this thesis. This motivation explained the trend towards increased flexibility in software systems This trend towards

flexibility was shown as a general trend in software development and then closer examined in the relation to the development of DSM systems. This examination led to the definition of a number of types of adaptability that are implemented in consistency maintenance protocols.

The subsequent section gave a description of the possible types of DSM systems and explained the issues involved in the designs of these systems. One of the major issues in the design of DSM systems are consistency models. The overall structure of these models was described and the characteristics of a number of consistency models were explained in detail.

The chapter concluded with a review of projects that were most influential to the development of our characterization of consistency maintenance protocols and the design of our system.

Chapter 3

A new categorization of consistency maintenance protocols

This chapter contains the proposal of a new characterization of consistency maintenance protocols. The chapter begins with an introduction of the overall structure of consistency maintenance protocols. This overview presents the components and their general interfaces. The introduction is followed by a description of the individual components and their sub-components. This description builds the foundation of our approach to provide flexibility. The definition of components and their characteristics allows a consistency maintenance protocol to be build from a set of components and to adjust this protocol by replacing individual components against components with the same external functionality but different internal characteristics.

The description of the individual components is followed by a description of the interaction of the components and summary of the overall appearance of resulting consistency maintenance protocols. The chapter concludes with a set of examples that demonstrate how existing solutions can be described with the help of this new categorization.

3.1 Overall structure of consistency maintenance protocols

In the description of the related work we described various aspects of DSM systems. A number of projects – some of which have been described in the previous sections – have concentrated on improving a single aspect or a limited set of aspects. These projects [58, 11] have built DSM systems as singular units that implement all aspects of a DSM system in a single design block or block of code. These singular entities need to be separated into components in order to achieve a flexible, adaptable DSM mechanism.

The literature of DSM systems does not offer a complete and coherent description of the overall structure of consistency maintenance and the interaction of the components that ensure consistency maintenance. The following text gives a definition of what we understand by consistency maintenance and explains the overall layout of consistency maintenance mechanisms. This is followed by an explanation of the individual components and their characteristics. The section concludes with a description of the interaction of the components.

Shi et al. [85] investigate the interaction between consistency models and coherency protocols. The conclusion of their investigation is that a consistency model is a combination of an event ordering mechanism and a coherency protocol. This conclusion contradicts our approach that holds that a consistency maintenance algorithm is a combination of implementations of consistency models and coherency protocols. However, closer examination shows that the difference between the two forms is mainly in the use of terminology. The term "consistency model" used in [85] matches the term consistency maintenance protocol used here and the term "event ordering" matches the term "consistency models" used in this text.

Consistency maintenance encompasses all actions and communications necessary to provide shared memory. The provision of shared memory in general does not imply that nodes have a copy of shared data items or that copies of shared data items are kept up-to-date. It only denotes that at some point in time the memory content of one node is accessible to another node. The specification of consistency maintenance actions identifies how data are shared and if copies of shared data item are kept on different nodes in a system.

The actions of consistency maintenance include the modifications of memory locations on local and remote nodes, the synchronization of accesses to memory locations and the

communications between processor nodes. The communications between processor nodes can be subdivided into the synchronization messages to other nodes, distribution of management information about the shared data (e.g., location of the master copy) and the distribution of update information.

3.1.1 Components of Consistency Maintenance Protocols

A consistency maintenance protocol can be split into two distinct components: consistency models and coherency protocols. These components interact using a small interface and represent a classical separation of policies and mechanism [69].

As laid out in the previous section consistency models specify when copies of shared data are brought up-to-date. The bringing up-to-date does not necessarily include the exchange of the updates but may consist only of the invalidation of out-of-date copies. A consistency model ensures that the outcome of a program is predictable.

A coherency protocol in turn specifies how copies of shared data are kept up-to-date. This includes the exchange of update and invalidation messages, the maintenance of local copies and the administration of copies.

A high-level description of the interface between consistency models and coherency protocols can be given by three primitives (Fig 3.1): read, write and synchronize.

A consistency model issues a **read** primitive when an application attempts to access a memory location. This read access signifies to the underlying coherency protocol to provide the value that is stored in the local copy of shared data. It does not involve any synchronization with remote nodes. However, if the coherency protocol is based on an invalidation mechanism it might be necessary to communicate with other nodes in order to obtain an up-to-date copy of the shared data.

A **write** primitive is invoked when an application wants to transfer a new value to a memory location. The coherency protocol accesses only the local copy of shared data. Again it does not involve necessarily involve any communication with other nodes.

A **synchronize** primitive is issued when a consistency model wants to achieve a synchronization with other nodes that hold copies of the shared data. This activity involves the

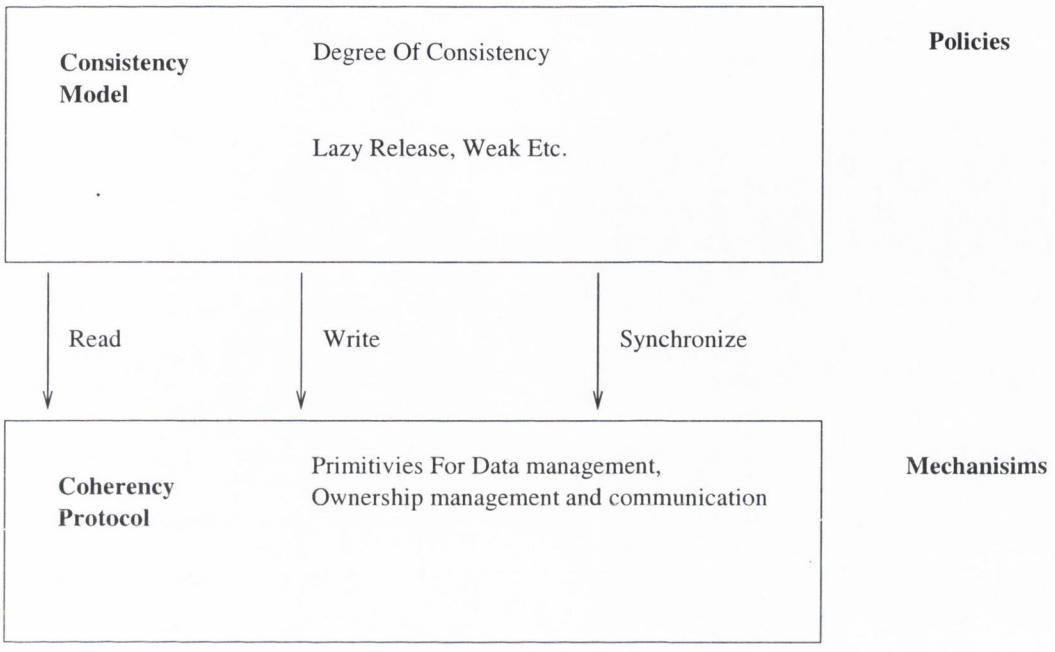


Fig. 3.1: Components of consistency maintenance protocols

notification of other nodes.

The following section will describe in detail the characteristics of consistency models and the influence that these characteristics have on the above explained primitives. This is followed by a description of coherency protocols and the actions that are taken by these protocols on receipt of the primitives.

3.2 Consistency Models

A consistency model, as described in 2.4.1, defines how access to shared data is seen by interested parties. A number of different models have been proposed in the literature [52, 3], including *sequential*, *processor*, *release* and *lazy release* consistency. In general, implementations of consistency models can be characterized by four properties, namely: order of access; concurrency; atomicity and scope.

3.2.1 Properties of consistency models

The **order of access** defines the sequence in which accesses are seen by interested parties. This property represents the fundamental characteristic of a consistency model. For example processor consistency requires that accesses to a shared data item, originating from node A, are seen in the same order by every other node in the system. However, accesses originating from two different nodes can be seen in different orders. Sequential consistency goes further to require that accesses from different nodes be ordered across a system.

Concurrency of access defines whether or not nodes can concurrently access data and if so in which modes that access can take place. Read operations can usually be executed concurrently, with each other, without causing problems, while under certain circumstances writes may also be executed concurrently, for example if the writes are to non-overlapping data. In all there are four possible combinations for concurrent access, namely: EREW (exclusive read, exclusive write); CREW (concurrent read, exclusive write); ERCW (exclusive read, concurrent write) and CRCW (concurrent read, concurrent write).

Scope determines the set of data that is kept consistent. This set can range from a single memory location over a couple of related locations to the whole set of shared memory locations. This property is defined generally by the implementation of a consistency model. In Midway [18] only those shared data items that are explicitly locked are guaranteed to be kept consistent while in TreadMarks [6] explicit locking is not necessary as the scope of consistency extends (transparently) to all shared data.

Atomicity defines whether the propagation of updates is done on a per access basis or whether several local updates can be done before a batched update is sent out. For example sequential consistency requires that updates be on a per access basis while release consistency allows updates to be batched. This property divides consistency models into strict consistency models and relaxed consistency models. Strict consistency models (as described in 2.4.1) enforce a propagation of updates after every access; relaxed consistency models gather accesses before engaging an update propagation.

Examples

We argue that the four properties, just described, are sufficient to describe the implementation of any consistency model. Space is not sufficient here to give an exhaustive proof of this assertion but by way of illustration the following list shows how a representative sample of systems from the literature can be described.

- The implementation of sequential consistency, as described by Mizuno *et al* [77], can be characterized as follows: all accesses are ordered sequentially, the concurrency is CREW, the scope embraces all shared data and atomicity is on a per access basis.
- The derivation of lazy release consistency described by Amza *et al* [6], can be described as follows: a lock mechanism is used to force all accesses to happen in a sequential order, the concurrency is EREW, the scope embraces all shared data and atomicity is at the granularity of a batch of accesses.
- The derivation of lazy release consistency described by Keleher [55] is similar to that of Amza. The scope embraces all shared data, the concurrency is CREW and the atomicity is at the granularity of a batch of accesses. The difference, from Amza, is that a barrier mechanism forces no ordering among the accesses to data. This relaxation is only possible if no data races are expected in applications.

3.2.2 Primitives in consistency models

The properties described in the previous section give a general notion of the implementation of a consistency model. This notion can be applied to the primitives given in 3.1.1. This combined description gives a definition of a consistency model that fits into the general framework described in 3.1.1.

In order to allow the combination of the primitives and properties, the primitives need to be identified in more detail. This description of the primitives differs slightly from the description of the general interface between consistency model and coherency protocols given in figure 3.1. The behaviour of the primitives is specific to the individual consistency model depending on the definition of the above describe properties. Their general meaning however,

remains the same throughout all consistency models and meets the implementation of the coherency protocols.

The following section contains a classification of the primitives and provide a description of their general behaviour. The behaviour of the primitives in the individual consistency model is described by the properties explained above.

Description of primitives

The primitives can be divided into three general classes: access, synchronization and update operations.

Access operations *Access operations* describe the ways in which cached data can be accessed by a program. From the view of the consistency model, these operations are atomic and executed instantaneously. The scope of the access operations is the local copy of the shared data. Thus, there is no communication with other nodes connected with the execution of access operations.

The access operations can be subdivided into *Read* and *Write* primitives. The *Read* primitive returns the value of a given memory location. The individual consistency model defines whether the Read primitive is exclusive or can occur concurrently to other access operations. Additionally, the individual consistency model describes whether the returned value is the most recent written one, or an older value may be returned. The *Write* primitive fills a memory location with a given value. The individual consistency model has to define whether the written value is propagated directly after it is transferred to the local copy, or a set of written values is allowed to be pipelined and propagated later.

Synchronization operations *Synchronization operations* are used to ensure the order of accesses. They consist of an opening and a closing primitive called *Lock* and *Unlock*, respectively. These two primitives surround a block of operations that is to be synchronized. The operations are similar to traditional locks, semaphores or monitor mechanisms. They can be implemented by using a software synchronization service, or build with hardware-supported synchronization mechanisms.

Some systems may be based on a barrier mechanism. In this case, the opening primitive is implicitly assumed and the block of operations is identified by the last closing primitive.

The scope of the synchronization operations differs among the consistency models. Depending on the scope of consistency the scope of the synchronization is limited to a local node or comprises a set of interested nodes. Additionally the synchronization can apply for all shared data or limited to a set or single shared item.

Update operation The *update operation* includes the *Update* primitive. This primitive invokes the propagation of modifications to other interested nodes. The propagation tells the underlying coherency protocol that modifications have been made. The further handling of the propagation depends on the coherency protocol. It does not necessarily imply a transfer of values to other interested nodes. The propagation can be a transfer of values to a central node or a transfer of invalidation messages to interested nodes.

Semantics of primitives

An overview of the semantics of the primitives is given in Figure 3.2.

The possible operations shown in Figure 3.2 as *Operation* can be one of the three sub-types: *AccessOp* for access operation, *SyncOp* for synchronization operation and *UpdateOp* update operation. These sub-types can be further divided into terminal primitives. An access operations represented by AccessOp can be either a *Read* or *Write* primitive. A synchronization operation represented by *SyncOp* can be either a *Lock* or an *Unlock* primitive. The update operation represented by UpdateOp is an *Update* primitive. Access operations can be combined to an operation block represented by *AccessOpBlock*. This operation block can be one or more access operations.

These primitives can be used in conjunction with the properties described in section 3.2.1 to define individual consistency models.

```
Operation:=       AccessOp
                | SyncOp
                | UpdateOp

AccessOpBlock:=   AccessOp
                | AccessOp AccessOpBlock

AccessOp:=        Read
                | Write

SyncOp:=          Lock
                | Unlock

UpdateOp:         Update
```

Fig. 3.2: Semantics of primitives

3.2.3 Characterization of consistency models

As described in 2.4, consistency models can be distinguished in two major categories: strict and relaxed consistency models.

Using the primitives that we defined in 3.2.2, we can now identify *strict consistency models* as models that order every access operation to shared data. After each access operation, an update primitive is issued and modifications are propagated among interested nodes.

Relaxed consistency models, in turn, can be defined as models that combine a number of access operations into an access block. The consistency of shared data in these models is only guaranteed inside a block that is surrounded by synchronization primitives.

Strict consistency models

Strict consistency models describe the order of access operations where every access operation is viewed as an individual unit. After every access operation the shared data is assumed to be consistent. This assumption requires that whenever a modification has been made the change

has to be propagated to other interested nodes. The difference between the individual models is defined by the strictness of the order and the scope of consistency.

Atomic consistency Atomic consistency [76, 48, 83] describes the strictest consistency model. All accesses to shared data have to be seen at all nodes in the same order as they are issued by the nodes *in real time*. From this definition follows that all nodes have to agree upon one execution history. Thus the scope of consistency comprises all interested nodes. The large scope combined with the strict order makes this consistency model very demanding according to system resources such as network bandwidth and processing time.

Sequential consistency A simple conservative sequential consistency model, as described in section 2.4.1 can be realized by arranging the given primitives as depicted in Figure 3.3. In this model the primitives *Read* and *Write* are combined with a pair of synchronization operations *Lock* and *Unlock*. Every *Write* primitive is also followed by an *Update* operation to propagate the modifications immediately after the *Write* operation took place and before other operations are allowed to perform.

```
SeqRead:=   Lock  Read   Unlock

SeqWrite:=  Lock  Write  Update  Unlock
```

Fig. 3.3: Semantics for a seq. cons. model

The scope of consistency is all interested nodes. The data scope consists of all shared data. From these parameters follows the global characteristic for the synchronization operations in this consistency model. The synchronization operations have to ensure on all interested nodes that no conflicting accesses to the data are executed at the same time. This behaviour can be achieved by blocking every access to the shared data while performing one access, or by processing the accesses in a sequential order in a central place.

A possible implementation of this structure could be a central host to which all interested

nodes send their updates using a synchronous transfer mode. The central host performs one update after the other and thus serializes the accesses implicitly. By using a synchronous transfer mode the nodes sending updates have to wait until the central node sends an acknowledge message. This implementation, however, would allow little concurrency and would not scale well.

Cache consistency Cache consistency [42, 4] describes a consistency scheme similar to sequential consistency, but limits the scope of the consistency to a single data item. Interested nodes have to agree upon a single execution history for accesses to the individual data item, but do not have to synchronize accesses to different data items. Following this definition accesses to different data items may be seen by different nodes in a different order.

The separate synchronization for each data item allows greater concurrency compared to sequential consistency. Algorithms, however, that were developed with uniprocessor behaviour in mind, might perform incorrectly. In a two-process mutual exclusion protocol like depicted in figure 2.5 both processes are able to enter the critical section. This behaviour is possible because cache consistency does not enforce the order of statements given by the program. Thus the read operations of both processes might bypass the write operations and allow both processes to enter their critical section at the same time.

In addition to the described behaviour the separate synchronization of all data items might increase the efforts connected to the synchronization. The time to process the synchronization and the network traffic caused by the synchronization have a negative impact on the performance of the system.

```
CacheRead:=   Lock  Read  Unlock

CacheWrite:=  Lock  Write  Update  Unlock
```

Fig. 3.4: Semantics for a cache cons. model

The semantics for cache consistency as depicted in figure 3.4 are equivalent to the semantics proposed for the sequential consistency model. The difference between the two models

is like described above the function of the synchronization primitives. The synchronization primitives for cache consistency separately order accesses to every single data item.

Processor consistency Goodman [42, 4] defined processor consistency as:

- a multiprocessor is said to be processor consistent if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program.

```
ProcRead:=   Lock  Read  Unlock

ProcWrite:=  Lock  Write  Update  Unlock
```

Fig. 3.5: Semantics for a proc. cons. model

A combination of primitives to satisfy the definition is given in Figure 3.5. This combination is equivalent to the combination of primitives that has been given for sequential consistency. The difference between the two definitions lies in the definition of the scope of consistency. For processor consistency, the scope of consistency is limited to the node that is executing the access. The data scope encompasses all shared data.

These definitions gives the synchronization operations for this model a local characteristic. They ensure that accesses to the data are performed and seen by other nodes in the order specified by the program. They do not ensure a global serialization of all accesses.

Relaxed consistency models

Weak consistency The model for weak consistency [31] represents the simplest relaxed consistency model. This model distinguishes two forms of memory accesses: ordinary and synchronizing accesses. Synchronizing accesses in this definition are used to order the processing of ordinary accesses. They can be represented in our description by synchronization operations.

In the description for this model we distinguish between two different blocks of access operations: blocks with ordinary accesses, and blocks with synchronized accesses. Blocks with ordinary accesses are allowed to perform after all prior blocks with synchronized accesses have been performed. Blocks with synchronized accesses are allowed perform after all prior blocks with ordinary accesses have been performed. The performance of blocks with synchronized accesses takes place in a sequential order. This definition differs from the conditions given by Dubois [31] and Gharachorloo [41]. The resulting behaviour, however, is the same.

```
WeakOp:=        OrdBlock
              | SyncBlock

SyncBlock:=     Lock AccessOpBlock Update Unlock

OrdBlock:=      AccessOpBlock Update
```

Fig. 3.6: Semantics for a weak cons. model

A combination of primitives to satisfy weak consistency can be given as depicted in 3.6. The possible operations, represented by *WeakOp*, can be a block with ordinary accesses *OrdBlock*, or a block with synchronized accesses *SyncBlock*. The block with synchronized accesses contains a pair of synchronization operations. This pair embraces a block of accesses and an Update operation. All accesses from outside the synchronized block to shared data are blocked by the synchronization operations until the access block and the Update operation have been performed. The access operations inside the access blocks are assumed to be performed processor consistent. The data scope comprises of all shared data.

Gharachorloo [41] proposes to handle synchronizing accesses either sequential or processor consistent order. This distinction can be transferred to the description based on access blocks. The synchronized access blocks may be handled either sequential or processor consistent. The handling determines the scope of synchronization. This scope may be restricted to a single node or comprise a set of interested nodes.

Release consistency Release consistency [41] was proposed as an extension to weak consistency. It relaxes the concurrency constraints of weak consistency by allowing a synchronized access block to begin being processed while a prior block finishes being processed. All access operations have to be performed before a closing synchronization operation starts being processed. Thus all operations have been performed. Weak consistency allows opening synchronization operations to perform as soon as a closing synchronization operations of a prior block starts being processed.

Gharachorloo et al [41] defined release consistency as a system that complies to following conditions:

- Before an ordinary access is allowed to perform with respect to any other node, all previous opening synchronization operations must be performed, and
- before a closing synchronization operation is allowed to perform with respect to any other node, all previous ordinary accesses must be performed, and
- accesses in a synchronized block are processor consistent to one another.

```
AccessOp:=      SyncBlock
              | OrdBlock

SyncBlock:=     Lock AccessOpBlock Update Unlock

OrdBlock:=      AccessOpBlock Update
```

Fig. 3.7: Semantics for a release cons. model

A combination of primitives to satisfy release consistency is given in figure 3.7. This combination of primitives is equivalent to the combination given for weak consistency. The difference separating the two models is the definition of the relationship of the update and synchronization operations. In the release consistency model the opening synchronization operation is allowed to perform as soon as the closing synchronization operation of the prior synchronized block starts being processed.

Lazy release consistency Lazy release consistency [57, 54] defines a consistency model derived from release consistency. This consistency model postpones the propagation of updates until the following synchronized block on another interested node is entered. This modification the propagation until it is absolutely necessary.

The conditions for this mode were defined by Keleher [54] as:

- Before an ordinary access is allowed to perform with respect to another node, all previous opening synchronization operations must be performed with respect to this other node, and
- before a closing synchronization operation is allowed to perform with respect to any other node, all previous ordinary accesses must be performed with respect to this other node, and
- accesses in a synchronized block are sequentially consistent to one another.

```
AccessOp:=      SyncBlock
              | OrdBlock

SyncBlock:=     Lock Update AccessOpBlock Unlock

OrdBlock:=      AccessOpBlock Update
```

Fig. 3.8: Semantics for a lazy release cons. model

A combination of primitives that ensures lazy release consistency is depicted in 3.8. This combination is similar to the combination given for release consistency. The difference between the two models is the placement of the Update operation. The Update operation in a lazy release consistency model is executed before an synchronized operation block is entered.

3.3 Coherency Protocols

As described above, a coherency protocol maintains the copies of shared data that are kept in a system. It underlies the consistency model and is layered on top of the available communi-

cation mechanism. A coherency protocol is triggered by the primitives issued by the overlaid consistency model. Different coherency protocols may implement a different behaviour according to their environment.

The behaviour of a coherency protocol has to be consistent with the definition that is expected by the consistency model. The coherency protocol has to ensure that the invocation of primitives has no side effects that undermine the working of the consistency model.

3.3.1 Reaction on the receipt of primitives

The following paragraphs describe the general reaction of coherency protocols on the reception of a primitive.

A **Read** primitive invokes a coherency protocol to return the requested value of a data item. The value has to be the last written value to this data item from the view of the coherency protocol. The view of the coherency protocol is *different* from the view of the consistency model. The coherency protocol limits the verification of the data item to the local copy. The coherency protocol has to ensure that the local copy contains valid data. In the case it has been invalidated, the protocol may have to retrieve an update for the data from a remote node. The data is returned after the value is ensured to be valid.

A **Write** primitive causes a coherency protocol to store a given value into a given data item. This action might include an update of a list of modified data. The primitive does *not* include a propagation of the modified data to other interested nodes.

A **Synchronize** primitive invokes the primary task of a coherency protocol. This task is the propagation of the modifications. The propagation consists of three steps: the construction of a message containing the propagation, the distribution of the message, and the incorporation of the modifications at other nodes. The steps can be divided into two major categories: the conversion of a propagation into a message and back, and the distribution of a message.

The reaction to these primitives involves the two main actions of coherency protocols: transformation and distribution.

Transformation

The transformation of propagations into messages depends on two parameters: the kind of the propagation, and the granularity of the shared data. The *kind of propagation* describes whether a message contains invalidation or update information for the remote copies. Depending on this information, a message includes either a description of the modified data items or the new values for data items. The two kinds of propagation are described in the literature under the terms write-invalidate and write-update, respectively [20, 57].

The kind of propagation also determines whether a message contains a whole unit of shared data, e.g. a memory page in a page-based systems, or a description of the difference between the "original" data and the modified data. The summary of the differences between original and modified data is described in the literature as diff [17, 59].

The *granularity of the shared data* describes the size of of data that is viewed as modified when a modification has taken place [72, 78]. In page-based systems the granularity could comprise a memory page. Whenever a byte in a page is modified, the page is viewed as modified.

Depending on the *kind of propagation* and *granularity* the coherency protocol has to create a message from the modifications that have been made to the local copy of shared data. This message is then propagated using a distribution algorithm.

Distribution algorithms

Distribution algorithms interface the consistency maintenance protocol to an underlying transport mechanisms. A distribution algorithm has to deliver propagation messages from one node to other interested nodes. Algorithms and infrastructures for the distribution of propagation messages have been subject of prior research work [72, 16] and have been described in detail.

The form of distribution depends on two parameters: the location of copies of shared data and the management scheme for these copies. The location of the copies describes the set of nodes that hold a consistent copy of the shared data. This parameter implies also the number of consistent copies. The management scheme for the copies describes the way of the

distribution and how a node with a consistent copy can be found.

The next sections give an explanation of the different possible locations. These explanations are followed by a set of examples. These examples show a small selection of possible distribution algorithms.

Single Copy at a fixed position The simplest form a distribution algorithm keeps one centralized copy of the data in a fixed position. Updates only have to be propagated to this central copy. Whenever a node intends to work on consistent data it has to consult this copy. The consultation can result in a copy of the data or in the latest updates. This centralized structure has the advantage of simplicity. The nodes do not have to change the destination of their propagations during runtime. The disadvantage is the central structure itself. Like all centralized systems it introduces a potential bottleneck and is susceptible for faults. The bottleneck is introduced by the central node because all propagations have to be routed to it. The sensitivity for faults roots from the fact that a fault occurring to the central node affects all interested nodes.

These characteristics determine the algorithm for situations where the data is mainly accessed by one node. This node hosts the data and has thus local access to the data. Network traffic only occurs at occasional accesses by other nodes.

Single copy migrating among nodes The central model can be enhanced by allowing the consistent copy to move among the interested nodes. The migration of the copy requires more administrative efforts. An interested node can not just assume to find the consistent copy at a fixed node. A mechanism has to be implemented that allow to find out where the consistent copy is held. For every shared data item a reference to the location of the consistent copy is necessary. This track-keeping increases the complexity of the algorithm.

Li and Hudak present in [72] a set of management algorithms targeting this issue. The management algorithms handle the dynamic ownership of the shared data in different ways. The paper proposes a classification of algorithms into centralized and distributed management algorithms.

In centralized algorithms, the copy management is fixed to one node. The copy man-

agement keeps track of all shared data. A node that is looking for a consistent copy of the data can contact this central management for information. This scheme has similar characteristics to the above described scheme for fixed location of consistent copies. The network traffic accumulates at one node. This characteristic opposes the advantages of distributed architectures.

Distributed management algorithms target the disadvantages of the centralized structure by spreading the administrative workload among nodes. The partitioning of the workload can be handled either statically or dynamically. A system that statically distributes the administration assigns the management of a part of shared data to a fixed node. A system that dynamically distributes the administration has to keep a reference to the location of the data item. This reference has either to be updated whenever the data item is moved or points to a location where the data item recently resided. In the first case the management algorithm has to contact all interested nodes and inform them about the changed location. This scheme allows an interested node to find rapidly a consistent copy of the data. The disadvantage of this scheme is a large amounts of network traffic because whenever a data item moves all interested nodes have to be notified. In the second case an interested node has a reference to a possible location of the data item. If the data item has moved from this location a reference to the new location is left behind. With this scheme, an interested node has to trace a chain of references to find a consistent copy of the data item. The advantage of this scheme is its simplicity at the time when a data item moves. Only the two nodes, that are involved in the movement, have to update their references.

Several consistent copies Single copies of shared data are only consistent at one node. Highly frequent concurrent accesses to this kind of data may result in a high amount of network traffic. The ownership of these data items has always to be transfered among interested nodes. This disadvantage may be avoided by introducing several consistent copies.

The introduction of several copies enhances the locality of data to nodes. A node with a consistent copy needs less communication when it accesses shared data. An access to shared data can perform immediately without causing a transfer of ownerships to the node.

The maintenance of several copies may increase network traffic and administrative efforts. The interested nodes need to keep all other interested nodes informed about modifications to the shared data. This task may be solved by keeping a list of interested nodes. When a modification is made, all nodes in this list have to be notified. The notification of several nodes may increase the network traffic. In point-to-point networks, one message is needed for each node. In networks that support broadcasting to multiple nodes, the network traffic may be reduced. In these networks, the distribution of a modification to several nodes consists of only one message. This one message is delivered to all interested nodes and hence reduces the number of necessary messages.

3.3.2 Components of Coherency Protocols

Figure 3.9 shows the general structure of coherency protocols. A coherency protocol has to react to invocations that it receives from programs on the local node and has to propagate and process update information.

As shown in Figure 3.9, a coherency protocol consists of a combination of three components: memory, ownership and distribution management. These components implement the reactions to primitives that have been illustrated above. The tasks of transformation and distribution are split up between these components.

- The **memory management** component controls access to local copies of shared data, access to which can be divided into access events and update management. Access events are either read or write operations that are issued by local applications. Update management describes the process of generating update information for copies of shared data on remote nodes or incorporating update information from remote nodes.

  The memory management component is responsible for the management of only the local copy of a data item, and it carries out the two major functions of transformation: the processing of access operations and the handling of updates.

- The **ownership management** component is responsible for liaising with other nodes to keep track of the current "ownership" of the data and a variety of algorithms have

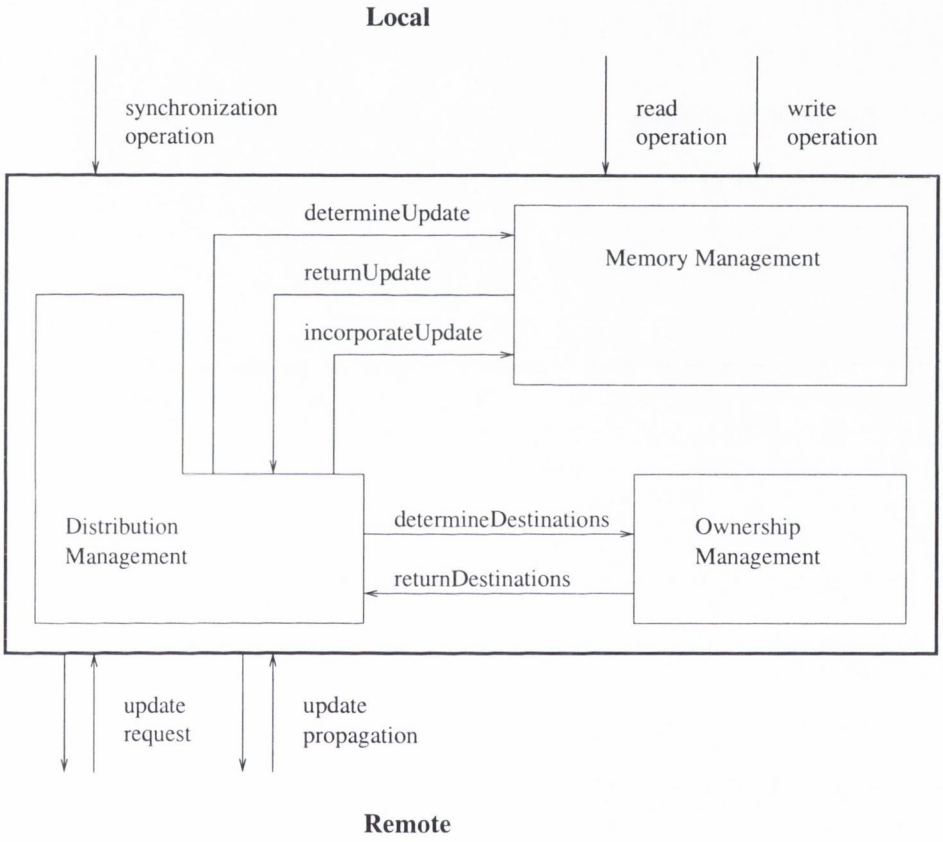


Fig. 3.9: Components of coherency protocols

been proposed in the literature, e.g. *static central ownership* [71] and *directory-based ownership* [26].

The ownership management component is responsible for the administration of the ownership of copies. Its main role is to identify the location of other copies of shared data and maintain a global view of the sharing. The interface to the ownership management component offers three methods: *registerCharge* to announce that a local copy is being shared; *getUpdateSource* to determine the source of updates; *getUpdateDestinations* to determine the destination of updates. The exact definition of these methods depends on the distribution management and communication mechanism used.

- The **distribution management** component is responsible for management of synchronization messages from remote nodes and it in turn generates either update propagations or update requests.

  An update propagation is sent to remote nodes to inform them about changes to shared data. It contains an update of the shared data, that was retrieved from the memory management component, along with any protocol-related data, like timestamps or address information. On receiving an update propagation from a remote node the distribution management component has to determine if the update is relevant to its local copy and, if necessary, to ask the memory management component to incorporate the new update of the data.

  An update request is a request to the distribution management components on remote nodes to propagate modifications from their local copies. It may contain a set of protocol-related information, and is sent to nodes that are determined by the ownership component. The request is answered by an update propagation.

  The distribution management component is responsible for the exchange of data between copies. It is linked closely to the communications mechanism. The distribution management component provides methods to send and request updates from remote nodes. The method to send updates - *sendUpdate* - takes as parameters an update and a set of destinations. The updates are sent to the addresses of the given destinations.

The method to request updates - *requestUpdate* - takes as parameters a descriptor and a set of sources. The sources are contacted in turn to retrieve updates from them according to the given descriptor. Every distribution management component, when started, listens to a given address for incoming requests and updates.

Memory management

The memory management handles accesses to the local copy of shared data. This includes the processing of read and write accesses from local applications, the creation of update and/or invalidation data and the incorporation of such updates from other nodes. These functions build the frame that implements the action that has been described as transformation above.

A protocol needs to maintain certain structures in order to provide coherency. These structures depend on the kind of propagation and the granularity of shared data items.

The characteristic that has the most influence on the design of memory management component is the kind of propagation. A protocol that implements an update protocol requires relatively few structures. These structures hold information about the memory locations that have been changed by accesses. These structures are coarse-grained if the protocol implements the exchange of complete copies of shared data e.g. the structures only enumerate memory pages that have been changed if the granularity of the coherency protocol is the size of virtual memory pages. Much finer-grained structures are needed if the protocol implements the exchange of diffs and registers all changes to copies individually.

Alternatively, the protocol may take copies of original shared data items before an access is allowed. These copies can be compared to current copies of these items when update information is request.

The employment of an invalidation protocol requires additional structures that contain information about the validity of memory locations. These structures need to be updated whenever invalidation or update information from a remote nodes is received.

The methods by which the above listed functions modify the necessary structures determines the characteristic of a memory management component. The following paragraphs give short descriptions of the individual functions and their relationship to the structures of

a memory management component:

- A *read access*, as described above, is issued to obtain a value of a memory location. The memory management component retrieves this value from the local copy of the shared data and returns it to the consistency model. In the case of an invalidation protocol, the memory management component has to determine the validity of the addressed memory location and to issue an update request to the distribution management if necessary.

  This mechanism can be achieved in a number of ways. Underlying all approaches is a local storage that maintains the local copy of shared data. This local storage may be implemented as general memory mechanism that directly handles accesses to memory banks or as an intermediate mechanism that allows the interception of accesses and represents a layer between the application and the underlying low-level memory mechanism. The intermediate layer may constitute a virtual memory mechanism that is provided by an operating system or a user-level library.

  This interception mechanism can then be used to determine access violations, to validate memory contents or to synchronize with other copies of shared data. In the case of a memory component the validation of memory content is the most important task for this interception mechanism.

  The difficulty in the description of a read access is the differentiation between design issue and implementation detail. A memory mechanism can be implemented in a number of ways. To the outside world this memory mechanism appears to be working in a certain manner. This manner is the design issue. It does not depend on how the memory mechanism achieves its goals as long as its behaviour is perceived in the manner.

- A *write access* overwrites an old value at a memory location with a new value.

  The memory management component needs to maintain information about the changes that a write access performs. In order to do so, the component can either make a copy of the original data or maintain a list of memory locations that have been changed.

- The *creation of an invalidation* is invoked when an invalidation is to be sent to other interested nodes. The invalidation message includes a description of the memory locations that have been changed. The description is either a list of shared memory items that have been changed or a more detailed description of the individual memory locations that have been changed.

  The information about the changes that have been made through write accesses is used to create this description. A protocol that implements the copying of data before write accesses implements the creation of invalidation descriptions by comparing the original data with the up-to-date data of the local copy of shared data. A protocol that maintains lists of the memory locations that have been changed by write accesses analyzes these lists in order to create invalidation information.

- The *incorporation of an invalidation* involves the update of internal structures that contain information of the validity of memory contents. In the case of the management of complete copies of shared data items the memory management modules maintains information about the validity of complete shared memory items such as pages. If the management of shared data items is based on individual memory locations the structures that signify the validity of individual memory locations needs to be updated.

- The *creation of an update* includes the copying of modifications into a message. The format of this message depends on the kind of propagation that is supported by the protocol. The message contains either a complete copy of the data items that have been changed or the values of individual memory locations. The generation of a list of individual values can be achieved in two ways:

  1. Through a combination of an original copy of shared data items that has been taken before modifications were made and the current data items.
  2. Through a list of data items that have been changed in the course of the process. This list has to be maintain while write access happen.

- The *incorporation of an update* involves the incorporation of updates from remote nodes

into a local copy. The update information from remote nodes contains the values of data items that have been changed on a remote nodes. These values are copied into the local copies of the shared data items.

These functions determine the character of a memory management component. The definition of these functions is independent from the definition of components for ownership management or distribution management. This ensures the modular character of the components.

Ownership management

The ownership management component keeps track of the current "ownership" of shared data items. The ownership management determines the set of nodes that holds copies of shared data items. These copies can either be consistent or inconsistent. The ownership management has to distinguish between these two types. The set of nodes can encompass all nodes of a network, a subset of nodes or just one node.

The mechanism behind the ownership management is based on four functions: the management and determination of the source of propagations and the management and determination of the destinations of propagations.

- The functions to *determine sources and destinations* describe the operations that determine the sources and the destinations for propagations. The sources that are obtained by these functions is used by the distribution management to request invalidation or update information. The destination information are used to distribute invalidation or update information.

- The functions to *update sources and destinations* describe operations that are used to maintain the ownership information that is held by the management module. These functions are generally invoked by information obtained from other nodes.

The mechanism that is used to keep this information up-to-date depends on the underlying communications system. This system can be based on one of two paradigms: multicast and

unicast. The choice of the communications paradigm influences the design of the ownership management and its functions.

The term "unicast" in this contents relates to a mechanism where a message is delivered from one node to another without being received by any other node. An ownership management that is based on this mechanism

The term "multicast" identifies a mechanism that delivers a message to a set of nodes that have subscribed to a list. The message is sent only once from the sender and routing mechanisms direct the message or copies of the message to nodes that have subscribed. This mechanism involves the distribution of 1-n messages. In this description we include "broadcast" under the term multicast. Broadcast describes a mechanism where a message is delivered to all nodes that are listening to a communications medium. This mechanism distributes only one copy of a message. An ownership management system based on multicast has the advantage that the information about the ownership is

Multicast can be simulated using unicast by maintaining lists of nodes that are the receivers of messages. This mechanism has the disadvantage that n number of messages are sent for n nodes on a list.

Distribution management

The distribution management has two major functions: To send and to receive messages. These messages can either be invalidation messages or update messages.

- The *sending of a message* involves the distribution of data that is supplied by the memory management component to other nodes. The destination of a message is determined by the ownership management component.

- The *receipt of a message* triggers either an incorporation of an update or an invalidation by the memory management component. The distribution management has to decide whether the message receive contains an update or an invalidation.

The distribution management can be based on one of two communication paradigms, similarly to the ownership management: multicast or unicast. Depending on the chosen

paradigm the distribution management needs to adjust to deliver messages to groups of interested nodes.

3.3.3 Examples

We believe that this model is general enough to characterize available coherency protocols. As in the previous section, we will give examples of existing representative coherency protocols.

- The coherency protocol found in KSR-1 [26] can be described as follows: The granularity of consistency maintenance provided by the memory management is 128 bytes, a cache-line of local caches. The memory management uses an invalidation mechanism to notify holders of other copies. The ownership management in this scheme is directory based. The distribution management is based on a proprietary interconnection mechanism.

- TreadMarks' coherency protocol as described in [6] can be characterized as follows: The memory management is page based, it employs an invalidation mechanism and propagates diffs of changes when an update is requested. The ownership management employs a migrating-home algorithm. Nodes acquire a lock and subsequently become the "home" of the data until another node acquires a lock. The distribution management implements a light-weight, operation-specific protocol based on UDP/IP or AAL3/4. This protocol has to ensure message arrival because the underlying protocols do not guarantee reliable delivery.

3.4 Complete Examples

This section gives two complete descriptions of existing DSM systems, TreadMarks and DASH. The descriptions are based on the components that have been defined in the previous sections.

3.4.1 TreadMarks

The description of TreadMarks is based on the version of TreadMarks presented by Keleher et al [57]. This description uses locks as synchronization mechanism and should not be confused

Consistency model	Lazy release consistency

Semantics	`AccessOp: SyncBlock`
	`OrdBlock`
	
	`SyncBlock: Lock Update OpBlock Unlock`
	
	`OrdBlock: OpBlock Update`

Order of accesses	Sequential
Concurrency	CREW
Scope of consistency	All shared data
Atomicity	Blocks of accesses
Coherency protocol	
Memory management	Diffs - virtual memory mechanism
Kind of propagation	Invalidation with update-messages using diffs
Granularity	Page-based
Ownership management	Dynamic distributed
Distribution management	Point-to-Point

Table 3.1: Description of TreadMarks

with the version presented in [6] that uses a barrier mechanism for synchronization.

Table 3.1 gives an overview of the description of TreadMarks using the definitions from the previous sections.

TreadMarks implements a lazy release consistency model. The definition of this consistency model has been given in 2.4.2. The order of synchronized access blocks is sequential; the order of ordinary access blocks is not defined. Data can be read concurrently and write accesses to data items are exclusive. The scope of consistency encompasses all shared data and the atomicity of accesses is defined by the access blocks.

The coherency protocol in this system is an invalidation protocol that is based on a virtual memory mechanism. The granularity is defined by the virtual memory mechanism to the size of a memory page. The distribution management is based on point-to-point communication. The underlying implementation is based on UDP with an extension that provides delivery guarantees.

3.4.2 DASH

Table 3.2 gives an overview of the description of DASH using the definitions from the previous sections.

Consistency model	Release consistency

Semantics	`AccessOp: SyncBlock`
	`OrdBlock`
	
	`SyncBlock: Lock OpBlock Update Unlock`
	
	`OrdBlock: OpBlock Update`

Order of accesses	Sequential
Concurrency	CREW
Scope of consistency	All shared data
Atomicity	Blocks of accesses
Coherency protocol	Two-layered, Multiprocessor systems connected by a 2D mesh
Memory management	Cache-lines
Kind of propagation	Complete copies
Granularity	Cache lines
Ownership management	Directory-based
Distribution management	1st layer: Bus, 2nd layer: 2D-mesh

Table 3.2: Description of DASH

DASH implements a consistency model that provides release consistency as defined in 2.4.2. The order of synchronized access blocks is sequential, ordinary access blocks are not ordered. The data can be read concurrently and the write access to the data is exclusive. The scope of consistency encompasses all shared data and the atomicity of access is defined by the access blocks.

The Update operation in a lazy release consistency model is executed before an synchronized operation block is entered. The synchronization mechanism based on locks.

The coherency protocol implemented in DASH is structured in two levels. The first level is implemented as a bus-based system with 4 processor; similar to the one shown in figure 2.2. This system is based on cache-lines and implements an invalidation protocol. The second

level is implemented by an extension to the bus-based system. This extension connects a set of multiprocessor nodes with a 2-dimensional mesh. The extension implements a directory-based mechanism that attempts to resolve requests for cache lines if they can not be satisfied within a node.

3.5 Summary

This chapter introduced a new characterization of consistency maintenance protocols. The characterization identifies the two main components of consistency maintenance protocols: consistency models and coherency protocols.

The chapter continued with a definition of the components. The definition includes a description of the interface between the components and an explanation of the internal characteristics of the components.

The description of the components are followed by examples of existing protocols. The examples present a description of these protocol with the use of the definition of the components.

Chapter 4

Implementation

In chapter 2, we gave an overview of related work in the area of DSM that provides the background for our work. The subsequent chapter presented our characterization of consistency maintenance mechanisms. The present chapter describes the realization of this characterization in form of an object-oriented framework.

This framework represents a proof-of-concept implementation that demonstrates the viability of our approach. The implementation in form of a framework was chosen because a framework is well suited to reflect the structure and interconnection of the components that represent a consistency maintenance mechanism and allows to evolve the different components in form of class hierarchies.

The following section explains the choice of a framework as form of the implementation in detail and gives a high-level overview of the components of the framework. The second section describes a number of supporting classes. These classes are used by the components of the framework to communicate with one another and to synchronize their activities. The third section lays out the class hierarchy for the implementation of consistency models and gives an overview of the consistency models that have been implemented. The fourth section discusses the class hierarchy for coherency protocols and discusses the sub-components that are used to build a coherency protocol. The chapter concludes with a section that shows a step-by-step example of the use of the framework.

The explanation of the individual classes and their implementation is in some cases demon-

strated by the use of program listings. These listings show only excerpts of the full implementations and omit such functionality as the handling of exceptions and errors. The interfaces of the classes are presented in UML.

4.1 Framework overview

Booch [23] defines a framework as consisting of "a collection of classes together with a number of patterns of collaboration among instances of these classes". The choice to implement our characterization in form of a framework is motivated by this definition and is based on two main reasons:

1. The core components of the characterization - consistency models and coherency protocols - lend themselves to the description as class hierarchies. The instances of consistency models and coherency protocols have a number of individual and common characteristics. In an object-oriented development method, the common characteristics of related classes can be implemented in base-classes that form the basis of a class hierarchy. The leafs of the class hierarchy implement the individual characteristics of the components.

2. The implementation of the characterization as a framework allows to model the relationship between the components. The components communicate by invoking methods of interfaces that are defined by the base classes of the class hierarchies.

The framework that is described in this chapter is implemented in Java and consists of a set of class hierarchies for consistency models and coherency protocols and a number of supporting classes.

Figure 4.1 gives a high-level view of the main components of the framework and their relationship. The lowest level represents communication mechanisms. These mechanisms can be group communication mechanisms, broadcast mechanisms or unicast mechanisms that offer communication services to the higher-level components. The communication services are used by the instantiations of coherency protocols and by mechanisms for concurrency controls.

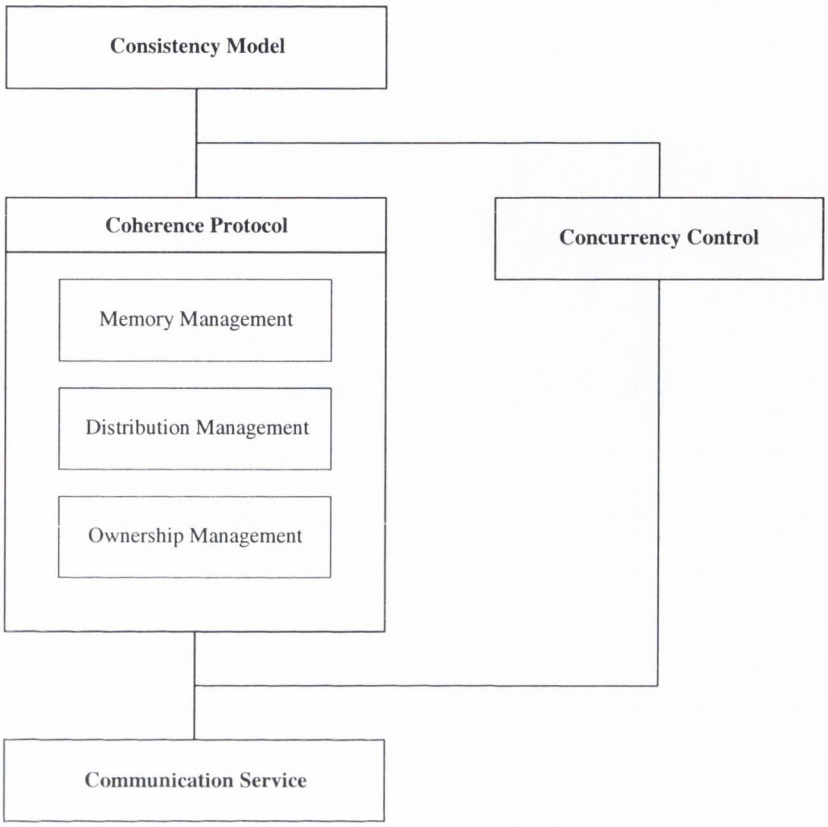


Fig. 4.1: Framework Overview

The implementations of coherency protocols combine instantiations of memory management, distribution management and ownership management components and form an underlying layer for the consistency model implementations. The instantiations of consistency models are based on coherency protocols for the management of the copies of shared data and on concurrency control mechanism for the coordination among the accesses to shared data.

The classes for coherency protocols and consistency models form the focus of the framework. The class hierarchies of these components will be described in detail in the following sections.

The communication mechanisms are provided by a number of classes of the JDK [2] and by third-party frameworks. The JDK provides interfaces to Posix sockets. These sockets are used in the framework for UDP and TCP/IP point-to-point communication and for UDP multicast communication. Third-party frameworks such as LRMP [64] and IBus [88]

provide reliable group-communication for specialized coherency protocols. The use of these communication mechanisms will be explained in detail in the description of the classes that make use of these mechanisms.

The concurrency control components implement distributed synchronization mechanisms such as barriers and locks. These components are not the focus of the framework and are briefly described among the supporting classes in the following section.

4.2 Supporting classes

The framework provides a number of classes to support the consistency maintenance mechanism. The core of the consistency maintenance mechanism is provided by classes that represent consistency models and coherency protocols. These classes could be used individually and separately from the framework; however, they need to exchange data with one another, require synchronization mechanisms for various forms of consistency models and are easier to use when accompanied by the supporting classes. The framework implements this support in a number of classes. The classes represent the data that is exchanged between the core classes and an RMI mechanism. The RMI mechanism in turn is the foundation of a set of synchronization mechanisms and an execution environment. These classes are not essential for the maintenance of consistency but facilitate the distribution of an application and are necessary in the context of this chapter for the later description of the employment of the consistency maintenance mechanism.

The following sections describe the data structures and the RMI mechanism. This description is followed by an explanation of the synchronization mechanisms and the execution environment.

4.2.1 Data Structures

The data structures described in this section are exchanged between the classes for consistency models and the components of coherency protocols. They provide a general communication mechanism that binds the individual components of the framework together. The structures

can be divided into the following classes:

- Descriptor
- Intention

Descriptor

A descriptor provides a basic mechanism to give a reference of a shared data item or any part of it. It holds references to memory locations in a local copy of shared data. These references may describe a single location such as a single field or memory address, or a set of locations. A location signifies the smallest sharable unit, e.g. a memory address or a field. The implementation of references depends on the implementation language and environment. An implementation in C/C++, for example, may favour references in the form of low level addresses whereas the implementation in Java, described here, favours field identifiers and array indices.

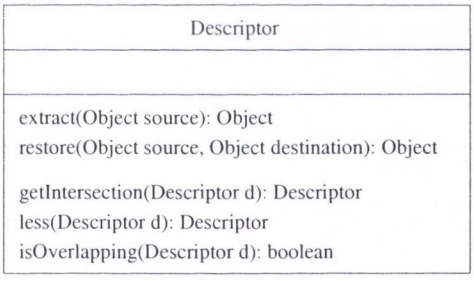


Fig. 4.2: Interface of the Descriptor class

The methods of a descriptor can divided into two categories: methods for object manipulation and methods for comparison. The methods for object manipulation allow information that is described by the descriptor to be extracted from a given object and to be restored to a given object. The methods for comparison provide a mechanism to determine the difference and the intersection between two given descriptors.

The framework provides a class for generic descriptors that can be used to specify locations within any object. This class specifies locations within an object using fieldnames and indeces.

```
public Matrix {
  int data[][];
  public Matrix() {data=new int[50][50];}
  }

GenericDescriptor descr;
Matrix source;
Matrix destination;
Object copy;

source= new Matrix();
destination= new Matrix();

d= new GenericDescriptor();
d.addIdentifier("data");
d.addIndex(12);
d.addRange(5,14);

copy= descr.extract(source);
descr.restore(destination, copy);
```

Listing 4.1: Usage of a generic descriptor

Listing 4.1 shows the usage of a generic descriptor in combination with a matrix object. The matrix object consists of an integer array called "data" that holds the contents of the matrix. In the subsequent code, a generic descriptor is created that describes a number of cells in the matrix. The contents of these cells is copied from the matrix object into a destination object. This object can then be transferred into a destination matrix.

The construction of a generic descriptor as shown in Listing 4.1 required 4 steps. The extraction and restoration of data that uses this descriptor uses introspection. These factors make the use of a generic descriptor slow and awkward to use.

An application developer may choose to implement an application-specific descriptor that is derived from the *descriptor* class and implements constructor and resolution methods that exploit the knowledge about the classes of an application. These application-specific descriptors can offer easy-to-use constructors and improve the performance of the extraction and incorporation of information.

For example, a descriptor for a matrix object can leverage the knowledge about the definition of the matrix class to offer an easier to use interface. A descriptor specific for the

matrix can reduce the constructor to a line as this:

```
d= new MatrixDescriptor(12,12,5,14);
```

Intention

An intention represents a combination of an identifier and a descriptor. An identifier specifies an operation that is to be executed. The descriptor, as described previously, describes a location or locations in a shared data item that is effected by the operation. Based on this information intentions can determine if they conflict with any other given intention.

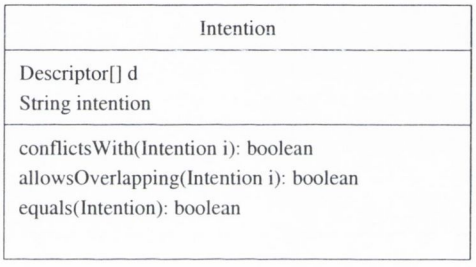


Fig. 4.3: Interface of the Intention class

Figure 4.3 shows the definition of the *intention* class. The intention class contains an array of descriptors and an identifier that classifies the intention. The methods of this interface provide one main function: the comparison with other intentions. Each class of intentions defines in these methods the compatibility between its instances. For example, the *conflictsWith* method determines if an intention conflicts with another given intention.

The framework features three predefined classes that indicate read, write and combined read/write access to data:

- ReadIntention
- WriteIntention
- AccessIntention

These classes define the methods of the *Intention* interface to match their specific need. For example, the *conflictsWith* method should return *true* if an intention conflicts with a given

intention. The *ReadIntention* class defines this method to return *true* if a given intention is of the type *WriteIntention* or *AccessIntention* and the descriptors of the intention and the given intention overlap; otherwise this method returns false.

An application developer can extend the range of intention with application-specific intention by deriving a class from the *Intention* class and implementing the methods according to the needs of an application.

4.2.2 RMI mechanism

The framework features a light-weight mechanism for remote method invocation (RMI). A RMI mechanism allows the invocation of methods of remote objects by providing a proxy object that relays method invocations from the local node to the remote node..

Figure 4.4 depicts the general scenario for remote method invocations. Two objects - A and B - are located on two separate hosts X and Y. A third object called stub- or proxy-object is located on host X as well and provides a connection to object B on host Y. The stub object has the same interface as the remote object and receives method invocations from object A. These method invocations are translated int the stub object into messages to the remote object B. The remote object handles these messages as method invocations and may return a return value in a reply message.

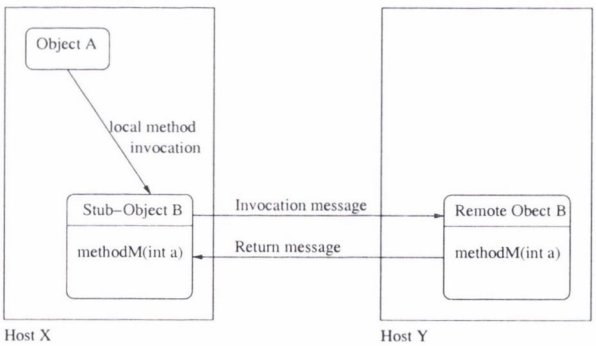


Fig. 4.4: Remote method invocation

The RMI mechanism that is featured in the JDK uses TCP/IP as transport mechanism. This transport mechanism provides reliable message delivery and has advantages when transferring messages over unreliable networks or over long distances. TCP/IP is connection-based

and needs additional messages to setup and terminate connections between nodes.

The framework is intended to be used in small-area networks and closely-coupled systems. In these environments, a number of efforts that are undertaken to provide reliable delivery pose an avoidable overhead. The RMI mechanism of the framework therefore is based on UDP. This transport mechanism is light-weight, simple and asynchronous.

The RMI mechanism provides two main classes: RMIObject and Stub. Both classes communicate with one another using a third class called *RMIMessageHandler*. The message handler implements a multiplexer that listens to a number of sockets and delivers incoming message to message listeners.

The *RMIObject* class provides the base-class for all RMI objects. It defines a constructor that connects an instance of this class to a RMI message handler. The constructor also takes an object as identifier for the RMI object. This identifier is registered with the RMI message handler which in turn registers it together with connection information of the message handler at a registry. The *Stub* class provides a constructor that connects an instance of this class to a RMI message handler on a remote node. The constructor takes an object as identifier for the RMI object. It retrieves information connected to the identifier from the registry and connects to the corresponding RMI message handler.

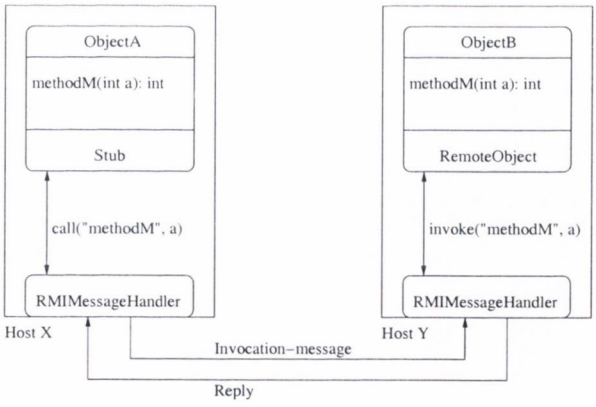


Fig. 4.5: Example of a remote method invocation

The most significant method of the *Stub* class is the method *call*. This method initiates the transfer of a message to the remote object. The message is sent asynchronously and the *call* method offers through a parameter the choice to wait for the return of a reply or to

```
public ClassA extends Stub {
  public int methodM(int a) {
    Object[] parameter= {new Integer(a)};
    Object result;

    result= call("create", parameter, true);
    return (((Integer) result).intValue());
    }
  }

public ClassB extends RMIObject {
  public int methodM(int a) {
    return (a*a);
  }

/* Node Y:*/
ClassB objectB;
objectB= new ClassB("remoteObject");

/* Node X: */
ClassA objectA;
objectA= new ClassA("remoteObject");
objectA.methodM(10);
```

Listing 4.2: Example of two classes using RMI

continue the execution of the program.

Figure 4.5 shows a scenario with an object *ObjectA* that relays invocation of the method *methodM* to the remote object *ObjectB*. The code for the classes of these two objects is given in listing 4.2. The class for *objectA* inherits from the class *Stub*. The methodM of *ClassA* invokes the *call* with the name of the method to invoke on the remote node and the parameter for the invocation. The *call* method passes this invocation on to the RMIMessageHandler. The message handler relays the message containing the invocation to the remote node. The remote node examines the message and invokes the method *methodM* of object *objectB*. The result of this invocation is returned to the message handler on node X which in turn returns it to the invoking method. The method *methodM* interprets the result of the invocation and returns the result as integer value.

The following section describes the implementation of synchronization mechanisms that represent the concurrency control component of the framework. The implementation of these components demonstrates also the use of the RMI mechanism.

4.2.3 Synchronization classes

The framework provides two types of synchronization mechanisms: locks and barriers. Each type defines a base class that defines the access to the synchronization mechanisms. The implementation of the synchronization mechanisms inherit the interface from the base-class and implement their individual mechanism behind this interface. This separation of interface and implementation allows a class to switch from one synchronization mechanism to another without the need to modify the code for the invocation of the synchronization mechanism.

The following section describes the definition of the interface for locks and the implementation of a locking mechanism based on the RMI mechanism that was described in the previous section. The subsequent section presents the interface for barriers and demonstrates the implementation of a barrier mechanism.

Locks

A lock implements a synchronization mechanism for mutual exclusion as described in [13]. The lock mechanism presented here is based on the intention described in section 4.2.1. The main function of the lock is the comparison of intentions against one another.

An intention defines methods that determine if a given intention conflicts with another intention. The lock implementation uses these methods to determine if a request for a lock conflicts with a lock request that has been granted already. If a lock request conflicts with a granted lock, the request is blocked until the conflicting lock is released.

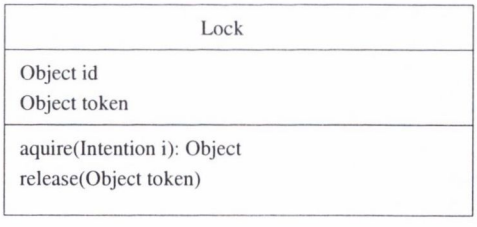


Fig. 4.6: Interface of the Lock class

The interface shown in figure 4.6 defines two main methods *acquire* and *release*. Classes that inherit from this interface implement these methods specific to their environment. For

```
public class LocalLock implements Lock {
  int token;
  public LocalLock() {token= 0;}

  public synchronized Object acquire(Object intention) {
    while(i==1) wait();
    i= 1;
    }

  public synchronized void release(Object token) {
    i= 0;
    signal();
    }
}
```

Listing 4.3: Implementation of a local lock

example, a class that implements a locking mechanism for processes on individual nodes formulates the methods to synchronize with a local token (as shown in listing 4.3).

The framework implements a simple locking mechanism based on the RMI mechanism described in the previous section. The lock uses a centralized server that serializes the requests for locks. The implementation of the locking mechanism consists of two classes: CentralLock and CentralLockServer. The class *CentralLock* implements the stub code for the locking mechanism and is instantiated by nodes that need access to the locking mechanism. The class *CentralLockServer* implements the server code for the locking mechanism. This class is instantiated once in every execution environment.

The *CentralLockServer* maintains a hashtable for locks and their state. The state of the locks is changed by two methods: *acquire* and *release*. These methods implement the comparison of intentions as described above. A local or remote node can access the state of a lock by invoking the methods of the *CentralLockServer* through an instance of the *CentralLock* class.

Listing 4.4 shows an excerpt of the implementation of the *CentralLock* class. This class is derived from the *Stub* class and implements the *Lock* interface. It inherits the call method from the *Stub* class and uses this to call methods of the LockServer.

The *init* method of the class invokes the *create* method of the lock server. The parameter of this method call is the identifier under which the lock is known in the execution environ-

```
public class CentralLock
  extends Stub
  implements Lock, Serializable {

    public CentralLock(Object id) {
      super("LockServer");
      this.id= id;
      init();
      }

    protected void init() {
      Object[] parameter= {id};

      call("create", parameter, true);
      }

    public synchronized Object acquire(Object intention) {
      Object[] parameter= {id,intention};
      Object token;

      token= call("acquire", parameter, true);
      return token;
      }

    public synchronized void release(Object token) {
      Object[] parameter= {id, token};

      call("release", parameter, false);
      }
    }
```

Listing 4.4: CentralLock implementation

```
CentralLock lock;
Object token;

lock= new CentralLock("ObjectA");

token= lock.acquire(new AccessIntention(new CompleteDescriptor()));
if (token!=null) {
  ... perform critical section ...
  lock.release(token);
  }
else
  ... resolve conflict ...
```

Listing 4.5: Utilization of a lock

ment. The *acquire* method invokes the *acquire* of the lock server and supplies it with the identifier of a lock and the intention that is connected to the critical section for which the lock is acquired. The *acquire* method then waits for the return of a token from the lock server. The *release* method returns this token to the lock server once the critical section for which the lock was acquired is passed.

The *release* method is generally invoked once a critical section has been passed. The method executes asynchronously and does not block the execution of a process because it is not necessary to wait for the completion of the release operation.

Listing 4.5 demonstrates the deployment of an instance of the CentralLock class. An instance of a lock is created with an identifier. The instance connects to the central lock-server and sends the identifier as identifier for the lock it wants to interact with. The server determines if a lock with the same identifier already exists. In case the lock exists, the server connects the instance to this lock; otherwise the server creates a new lock.

After the lock has been initialized, it can be acquired with the *acquire* method with a given intention. The execution of a process is suspended, if the lock can not be acquired immediately. Once a lock is acquired, the process that is in possession of the lock can enter a critical section. After the critical section has been processed, the lock can be released using the *release* method. The release of a lock may start a previously suspended processes

Barriers

The barrier class implemented in the framework provides a synchronization mechanism to coordinate a number of concurrent processes.

The barrier is given a number that specifies the number of process to expect. A participating process calls the *wait* method. The method call is blocked until all participating processes have called in. When all processes have called in, the calls are unblocked and the mechanism is reset to its original state. Figure 4.7 shows the interface of the barrier class.

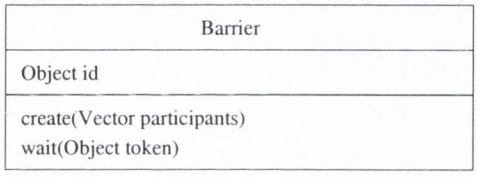


Fig. 4.7: Interface of the Barrier class

The framework provides an implementation based on the RMI mechanism similar to the *CentralLock* class. The centralized barrier implementation is implemented as an RMI object that registers the number of participants.

A class called *CentralBarrierServer* implements a central point for the coordination of distributed objects. This class is accessed by instances of the class *CentralBarrier*. This class implements the *Barrier* interface and maps it into calls to the central barrier mechanism.

4.2.4 Execution environment

The framework implements an execution environment that allows an application to be executed in a distributed manner.

The assumption of the framework is that an application is to be executed using a number of nodes connected by a network. The application is started at one node and initiates the distribution and execution of work at other nodes. In order to be able to distribute the work, the application has to be split into a number of discrete work-packages. These work-packages are distributed among the available nodes and executed in an order specified by the application. During their execution the work-packages communicate with each other using

message passing and shared objects.

The execution environment divides available network nodes into two sets: master and worker nodes (see figure 4.8). Nodes at which applications originate are called master nodes; nodes that are available to perform tasks for an application are called worker nodes.

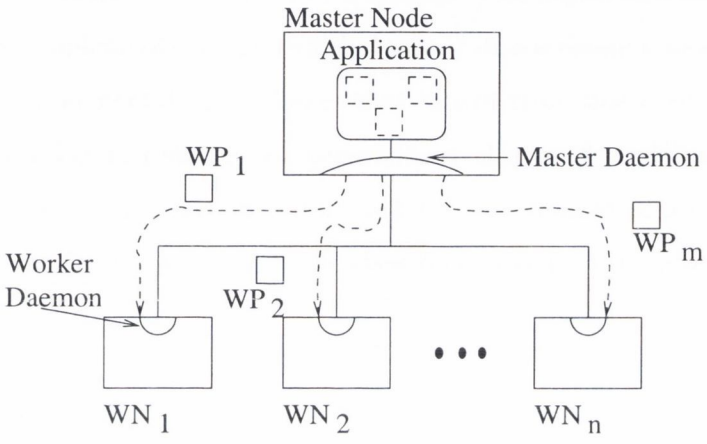


Fig. 4.8: Master/Worker relationship

Each worker node runs a daemon process. This daemon process registers at its startup time with a registry. The registry provides a general name service that is implemented as a distributed hashtable. A daemon process that starts up creates a connection point (e.g.. a socket address) at which it will wait to receive work and registers this connection point with a well-known name with the registry. A master discovers available workers by consulting the registry. The registry entry with the well-known name provides the master with connection points of available workers.

The distribution of application components can take place in various forms. The framework provides a set of mechanisms in order to facilitate well-known distribution paradigms such as Single-Program-Multiple-Data (SPMD) and Multiple-Program-Multiple-Data (MPMD). The algorithms for the distribution are implemented in a daemon process that is run on the master node. The user has the choice of using the provided daemon processes, to derive from these daemon processes or to implement application-specific distribution algorithms.

4.3 Consistency model

The implementation of consistency models that is presented in this section follows on directly from our characterization of consistency maintenance protocols as the high level policies which define the degree of consistency provided in a system. The implementation is arranged in a class hierarchy that implements the general behaviour of consistency models in a base-class and specializes the implementation in classes that inherit from this base-class

The following section introduces the generic base-class and explains the specialization of the sub-classes. This explanation is followed by an example of an implementation of consistency model and by an overview of the class hierarchy that describes the implemented consistency models.

Generic Base-Classes

The framework provides a generic base-class for consistency models. This base class defines the common interface for all consistency models and supports four fundamental operations to access shared data: read; write; begin and end. The access operations are mapped to corresponding operations provided by an underlying coherency protocol while the synchronization operations are mapped to corresponding operations provided by concurrency control. The implementation of this class is shown in Listing 4.6.

Two classes inherit from the generic base-class: *StrictConsModel* and *RelaxedConsModel.* These classes specialize the behaviour of the base-class. The *StrictConsModel* redefines the methods *read* and *write* by wrapping every access into invocations of a synchronization mechanism. The *RelaxedConsModel* defines two new methods: *begin* and *end.* These methods are used to annotate the beginning and the end of a section that includes accesses to shared data.

Implementation of a consistency models

This section describes the implementation of a relaxed consistency model. The model represented here is a weak consistency model that is based on locks. The implementation is based on the definition presented in chapter 3.2. The

```
public abstract class ConsistencyModel {
  protected Object id;
  protected Object charge;
  protected CoherencyProtocol[] cohProtocol;

  public ConsistencyModel() {
    id= null;
    charge= null;
    cohProtocol= null;
    }

  public void write(Descriptor descriptor, Object value) {
    cohProtocol[0].set(charge, descriptor, value);
    }

  public Object read(Descriptor descriptor) {
    return cohProtocol[0].get(charge, descriptor);
    }
}
```

Listing 4.6: Implementation of the ConsistencyModel class

Listing 4.7 shows the implementation of a weak consistency model. The class re-implements the methods *begin* and *end*. The *begin* method acquires a lock for the location that is specified by the given intention. It then invokes the underlying coherency protocol to ensure the consistency of the local copy of a shared data item. The *end* method invokes the propagation mechanism of the underlying coherency protocol and then releases the lock that has been acquired in the *begin* method.

4.3.1 Overview of the class hierarchy

Figure 4.9 shows an overview of the class hierarchy that has been implemented for consistency models.

All consistency model components are derived from a single base class (level 1 in figure 4.9). This base class defines the common interface for all consistency models and supports four fundamental operations to access shared data: *read*; *write*; *begin* and *end*. The access operations are mapped to corresponding operations provided by an underlying coherency protocol while the synchronization operations are mapped to corresponding operations provided by concurrency control.

```
public class WeakConsistency extends ConsistencyModel {
  public void begin(Intention intention) {
    Object[] buffer={intention, null};

    buffer[1]= lock.acquire(intention);
    token.addElement(buffer);
    if ((intention instanceof ReadIntention) ||
    (intention instanceof AccessIntention)){
      cohProtocol.ensureConsistency(charge, intention.getDescriptor());
      }
    }

  public void end() {
    Object[] buffer=(Object[]) token.elementAt(token.size()-1);;
    Intention intention=(Intention) buffer[0];

    if ((intention instanceof WriteIntention) ||
    (intention instanceof AccessIntention)){
      cohProtocol.syncPropagate(charge, intention.getDescriptor());
      }
    lock.release(buffer[1]);
    }
  }
```

Listing 4.7: Implementation of Weak Consistency

In the framework, consistency models are sub-divided into two major categories - strict and relaxed, see level 2 in figure 4.9. Strict requires that updates be on a per access basis while relaxed consistency there is more flexibility in how updates are done.

These two sub-categories are represented in the inheritance hierarchy by two classes. The class for strict consistency redefines the access operations inherited from the base class so that they interact with a given ordering mechanisms, e.g. locking, as necessary. The class for relaxed consistency defines two new operations. These two operations are used for annotation in the program to signify the beginning and the end of a set of access operations.

The next level (level 3) in the inheritance hierarchy consists of ready-to-use implementations of various consistency models and This is the aspect of the framework used by an application programmer. These classes implement the specific characteristics of an individual consistency model and is currently populated with the most widely used models. User defined consistency models can be inserted into the framework at this level.

In general the 4th, or base, level of the provides information on *required* or *supporting*

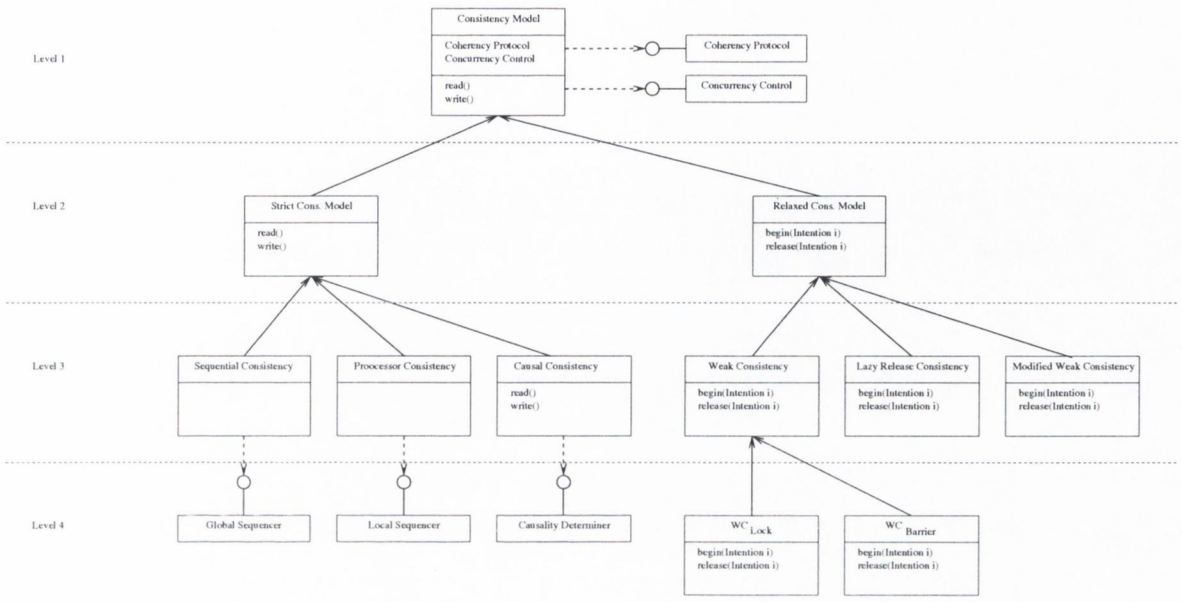


Fig. 4.9: Consistency Model Class Hierarchy

infrastructure. In the case of the strict models, the base level indicates a requirement for certain facilities not explicitly provided by the framework. Specifically, in the case of sequential consistency, there is a requirement for a global sequencer. This sequencer has to ensure that all access operations are executed sequentially in a global context. The framework indicates the requirement for such a sequencer but does not prescribe how it is provided. In this particular case the global sequencer could be provided as part of a coherency protocol which supports ordered accesses, by a concurrency control component, or by the application programmer as a stand alone component. The requirements for processor, causal, or any other model can be deduced from the framework in the same way.

The implementation of the consistency model framework is made up of generic super-classes at the base level and a small set of consistency models - **Lazy Release Consistency** and a number of variations on **Weak Consistency**. These are described in detail below.

The generic super-classes provide the connection between the actual implementation of a consistency model and supporting components. The supporting components are represented by underlying coherency protocol and concurrency control components. The base class for consistency models stores references to these components for use by the derived classes.

A superclass for relaxed consistency models is derived from the base class explained above. This class defines the interface for all relaxed consistency models. The interface consists of two methods to mark the beginning and end of a critical section; *begin* and *end*. The *begin* method takes as an argument an *intention* containing a description of the region of shared data which will be accessed during the critical section. The *end* method indicates the end of the *intention*. Both methods have to be defined in derived classes to model the individual behaviour of a relaxed consistency model.

4.3.2 Weak Consistency

Two minor derivations of weak consistency have been implemented, both are based on the class for relaxed consistency models.

WC_{Lock} is based on a concurrency control component that provides a mutex lock mechanism. The *begin* method announces the beginning of a critical section by acquiring a lock from the concurrency control component. The intention given as parameter to the method is used in the acquiring process as the parameter for this action. This way the concurrency control mechanism can determine if the intended action conflicts with another ongoing action. The *end* method ends the critical section by notifying the coherency protocol to publish changes made to the data. After the changes have been published the lock is released.

$WC_{Barrier}$ is based on a barrier mechanism. The implementation assumes that changes to the shared data do not affect each other; or if they do that they are synchronized by the developer. The *begin* method stores the given intention for later use in the *end* method. The *end* method publishes the data described by the descriptor of the intention. After the data has been published, the method waits at a barrier until all participating parties have published their changes to the data.

4.3.3 Modified Weak Consistency (MWC)

A modified version of weak consistency has been implemented to facilitate the particular sharing characteristics of the Life application, explained later, which requires the use of two coherency protocols. This requirement can be met in two ways. By using two consistency models where each consistency model is connected to a different coherency protocol or by implementing a consistency model that is based on two coherency protocols. We adopted the second approach and use a modified weak consistency model which takes as parameters two coherency protocols. The first exchanges data between a central copy and others, while the second exchanges data between all copies of shared data.

The location of the data exchanged in the Life application does not change between iterations so the descriptors have been embedded into the consistency model and are initialized at startup time. The *begin* method has been redefined not to change these descriptors and the *end* method has been defined to invoke the coherency protocols to publish the updates. Each coherency protocol is invoked with a corresponding descriptor.

4.3.4 Lazy Release Consistency (LRC)

Similar to the implementation of weak consistency two minor derivations of lazy release consistency have been implemented, LRC_{Lock} and $\text{LRC}_{\text{Barrier}}$.

To model the behaviour of lazy release consistency the classes redefine the *begin* method to ask the coherency protocol to ensure that the local copy is up to date. The *end* method tells the coherency protocol which data items have been changed during the critical section.

4.4 Coherency protocols

This section describes the implementation of coherency protocols and their subcomponents.

The task of a coherency protocol is the exchange of update information among nodes that are sharing data. As described in chapter 3.3, a coherency protocol can be assembled from three components: distribution management, memory management and ownership management.

The following sections describe the base-class for coherency protocols, the base-classes for the individual components and an implementation of an example. The description of coherency protocols is concluded with a presentation of the class hierarchy of implemented coherency protocols and a short description of each coherency protocol.

4.4.1 General behaviour of coherency protocols

There are two flavours of coherency protocol: update, or push, protocols that send out updated data as soon as the change is made; and invalidate, or pull, protocols that mark the data as invalid requiring it to be refreshed on a subsequent access. Coherency protocols also differ in the way they distribute data which can be done either by sending a complete copy of the data or by sending the incremental changes only.

The coherency protocols implement a general interface (as shown in Figure 4.10) that allows the consistency model to communicate with different coherency protocols. The core methods of this interface are *forceUpdate* and *ensureConsistency*. The method *forceUpdate* distributes updates. First it determines what needs to be updated via the memory management component. Then it asks the ownership management component for the destination for these updates. The update is sent to its destination by using the sendUpdate method of the distribution management component. The *ensureConsistency* method asks the ownership management component to determine the source for updates. This source is then queried using the *requestUpdate* method of the distribution management component. The returned update is then incorporated into the local copy by the memory management component.

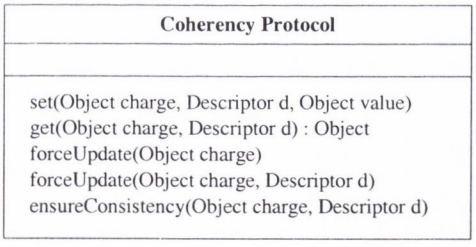


Fig. 4.10: Interface of the CoherencyProtocol class

4.4.2 Memory Management

A memory management component is responsible for the management of the local copy of a shared data item and carries out two major functions: the processing of access operations and the handling of updates.

The processing of access operations comprises read and write operations to the local copy. The synchronization of concurrent accesses is handled by the consistency model that invokes the coherency protocol.

Write operations cause the memory management component to update a location in the local copy with a given data value. It has to resolve a given descriptor into a location and write the given data into this location. This definition provides support for single data values as well as for a combination of objects such as arrays or compound objects. The write operation can be used to keep a list of changes made on a local copy. This list can contain only the locations changed or all values changed.

Read operations cause the memory management to return the value of a shared data item. This operation can involve the retrieval of update information if the protocol is an invalidation protocol and the information that is encountered in the shared data item has been invalidated.

A memory management component provides methods to determine local updates and to incorporate remote updates. These methods - *determineUpdate* and *incorporateUpdate* - use the descriptor parameters to extract data from, and incorporate data into, a local copy. The *determineUpdate* method asks a given descriptor to extract the data it describes from a local copy and gives an update containing the extract and the descriptor pair back as result. The *incorporateUpdate* method takes a descriptor from a given update and asks the descriptor to restore the data connected to it into the local copy.

Each memory management component defines a corresponding class that holds the update information that is exchanged between interested nodes.

MemoryManagement
addCharge(Object charge, Object id) removeCharge(Object charge) determineUpdate(): Update determine(Descriptor[] d): Update get(Descriptor d): Object set(Descriptor d, Object value) commitChanges(Descriptor[] d)

Fig. 4.11: Interface of the Memory management class

Update

The modifications by write operations are made to local copies and subsequently propagated to be applied to copies at other interested node. The information that is exchanged by interested nodes is represented by an *update* class.

The internal structure of an update depends on the following characteristics:

- An update may contain either a complete copy of a shared data item or hold a set of values that have been changed since the last synchronization event.
- The internal structure of the data section depends on the implementation language and the level at which memory can be accessed. The data may for example consist of memory pages that are platform specific or of some form of encrypted data that could be platform independent.
- An update also depends on the protocols implemented by the distribution management component. In the case of an invalidation algorithm it may hold no data at all, in which case the descriptors of this message point to the locations that have to be invalidated.

These characteristics directly correspond to the explanation given in section 3.3.2.

The framework features a generic interface for all update classes. This interface does not define any methods but serves as a common type identifier for derived classes. This type

identification is used by two interfaces that handle the the creation and incorporation of updates.

An implementation of an update generally consists of a collection of descriptors and data. It may hold a number of descriptors that point to various locations in local copies of shared data items. These descriptors describe the locations from which data has been taken to be propagated. The data part of an update holds copies of data items that have been changed.

Example of a memory management class

Listing 4.8 presents the implementation of a memory management class that creates updates with complete copies of the information that has been changed. This class is called *CopyMemoryMgr* and works together with a *CopyUpdate* class. The *CopyUpdate* class inherits from the *Update* class and holds complete copies of the fields that have been changed. By defining the *CopyUpdate* class the *CopyMemoryMgr* class implements an update protocol. The two most important methods of the memory management class are the *incorporateUpdate* method and the *determineUpdate*. The *incorporateUpdate* method takes an instance of *Update* as parameter. After it has determined that the update is an instance of *CopyUpdate*, it copies the contents of the update into the local of shared data items. The *determineUpdate* takes a number of descriptors and creates a *CopyUpdate*. The *CopyUpdate* contains copies of the data items that are described by the given descriptors and the descriptors themselves.

4.4.3 Distribution management

The distribution management component is responsible for the exchange of data between copies. It is linked closely to the communications mechanism. The distribution management component provides methods to send and request updates from remote nodes. The method to send updates - *sendUpdate* - takes as parameters an update and a set of destinations. The updates are sent to the addresses of the given destinations. The method to request updates - *requestUpdate* - takes as parameters a descriptor and a set of sources. The sources are contacted in turn to retrieve updates from them according to the given descriptor. Every distribution management component, when started, listens to a given address for incoming

```
public class CopyMemoryManagement implements MemoryManagement {

  public void incorporateUpdate(Update update) {
    CopyUpdate cupdate;
    Enumeration descriptors;
    Enumeration data;
    Descriptor des;
    Object ditem;

    if (update instanceof CopyUpdate) {
        cupdate= (CopyUpdate) update;
        for(descriptors= cupdate.getDescriptors(),
              data= cupdate.getData();
            descriptors.hasMoreElements() &&
              data.hasMoreElements();) {
          des= (Descriptor) descriptors.nextElement();
          if (des!=null) {
            ditem= data.nextElement();
            des.restore(ditem, charge);
          }
        }
    }
    else {
      throw (new UnsupportedUpdateType(update));
    }
  }

  public Update determineUpdate(Descriptor[] descriptor) {
    CopyUpdate update;
    Object data;

    update= new CopyUpdate();
    for(int i= 0; i<descriptor.length; i++) {
      if (descriptor[i]!=null) {
        data= descriptor[i].extract(charge);
        if (data!=null)
        update.addData(descriptor[i], data);
        }
      }
   return update;
  }
```

Listing 4.8: Initialization of CopyMemoryManagement

requests and updates.

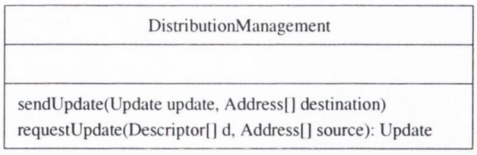


Fig. 4.12: Interface of the Distribution management class

The framework implements a number of distribution management classes that inherit this interface. An example of such a class is presented in the following paragraphs.

Example of a distribution management class

The framework provides a distribution management class based on TCP/IP sockets. Listing 4.9 presents the two methods: *sendUpdate* and *requestUpdate*. The *sendUpdate* method transmits a given update to a number of destinations that are specified by given addresses. The *requestUpdate* method sends a request to a number of given sources and waits for replies.

4.4.4 Ownership management

The ownership management component is responsible for the administration of the ownership of copies. Its main role is to identify the location of other copies of shared data and maintain a global view of the sharing.

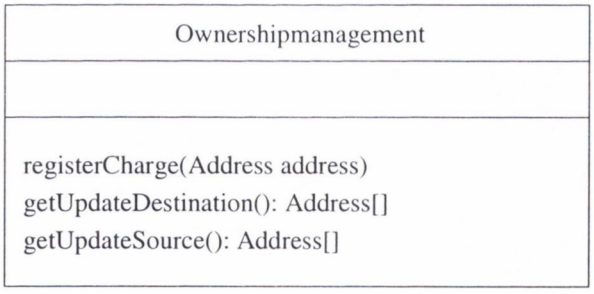


Fig. 4.13: Interface of the ownership management class

The interface of the ownership management component (as shown in figure 4.13) offers three methods: *registerCharge* to announce that a local copy is being shared; *getUpdateSource*

```
public class TCPDistributionMgr implements DistributionManagement {

  public synchronized void sendUpdate(Update update,
                                      Address[] destination) {
    Message message;

    messageNo++;
    message= new SyncUpdateMessage(messageNo, update);
    for(int i= 0; i<destination.length; i++) {
      multiplexer.send((SocketAddress) destination[i], message);
      }
    }

  public synchronized Update[] requestUpdate(Descriptor[] descriptor,
                                             Address[] source) {
    RequestMessage message;

    requestNo++;
    message= new RequestMessage(requestNo, descriptor);
    replyBarrier= new Barrier();
    msgListener.addReplyBarrier(requestNo, replyBarrier);

    for(int i= 0; i<source.length; i++) {
      multiplexer.send((SocketAddress) source[i], message);
      }
    replyBarrier.expect((expectAll) ? source.length : 1);
    msgListener.removeReplyBarrier(requestNo);

    return null;
    }
}
```

Listing 4.9: Initialization of the TCPDistMgr class

to determine the source of updates; *getUpdateDestinations* to determine the destinations of updates. The exact definition of these methods depends on the distribution management and communication mechanism used.

Example of an ownership management class

The framework provides a number of ownership management classes. One of these classes provides ownership management based on a static home-based mechanism. An up-to-date copy of every shared data item is kept at a specific fixed node. All updates are only propagated to this node. The ownership of the shared data item is fixed to this node for the runtime of an application. Listing 4.10 shows the implementation of the key methods of the ownership class.

4.4.5 Implementation of a coherency protocol

This section describes the implementation of coherency protocol. A coherency protocol assembles instantiations of a memory management class, a distribution management class and an ownership management class. The task of the coherency protocol is to route information between the instances of the management components. The coherency protocol described here is called HB_{TCP}. This protocol combines an instance of the *CopyMemoryMgr* class, an instance of the *CentralHostOwnershipMgr* class and an instance of the *TCPDistMgr* class.

Listing 4.11 shows the initialization of the coherency protocol. The protocol creates instances of the management components and provides them with links to one another. The memory management component is provided with a reference to the local copy of a shared data item and takes care of its maintenance. The distribution management component is provided with a reference to the memory management component and uses the reference to supply the memory management with incoming update information. The ownership management component is provided with the object identifier and with an address at which the distribution management is expecting update information.

```
public class SingleCopyOwnershipMgr implements OwnershipManagement {

  Address homeAddress;
  Object objectID;
  transient Address incomingAddress;

  public void addCharge(Object objectID) {
    RegistryStub registry;

    this.objectID= objectID;
    registry= new RegistryStub();
    homeAddress= (Address) registry.retrieveEntry(objectID);
    }

  public void registerCharge(Address address) {
    RegistryStub registry;

    registry= new RegistryStub();
    if (homeAddress==null) {
      registry.registerEntry(objectID, address, false);
      homeAddress= address;
      }
    else {
      WrapperObject wrapper= new WrapperObject(objectID, "clients");
      registry.addEntry(wrapper, address);
      }
    incomingAddress= address;
    }

  public Address[] getUpdateDestinations() {
    Address[] result= {homeAddress};

    if (isHome()) {
      RegistryStub registry;
      WrapperObject wrapper= new WrapperObject(objectID, "clients");
      Vector entries;

      registry= new RegistryStub();
      entries= (Vector) registry.retrieveEntry(wrapper);
      result= new Address[entries.size()];
      entries.copyInto(result);
      }
    return result;
    }

  public Address[] getUpdateSource() {
    Address[] result= {homeAddress};
    return result;
    }
}
```

Listing 4.10: Initialization of SingleCopyOwnershipMgr

```
public class HomeBasedProtocolTCP implements CoherencyProtocol {

  TCPDistributionMgr distributionMgr;
  CopyMemoryManagement memoryMgr;
  CentralHostOwnershipMgr ownershipMgr;

  public void addCharge(Object charge, Object objectID,
                        boolean initialized) {
    Address address;

    memoryMgr.addCharge(charge, initialized);
    distributionMgr.initialize(memoryMgr, memoryMgr);
    ownershipMgr.addCharge(objectID);

    address= distributionMgr.getAddress();
    if (!ownershipMgr.isInitialized()) {
      ownershipMgr.registerCharge(address);
      }
    else {
      ownershipMgr.setIncomingAddress(address);
      }
    }


  public void forceUpdate(Object charge, Descriptor[] descriptor) {
    if (!ownershipMgr.isHome()) {
      distributionMgr.sendSyncUpdate(memoryMgr.determineUpdate
      (descriptor), ownershipMgr.getUpdateDestinations(), true);
      }
    }

  public void ensureConsistency(Object charge, Descriptor descriptor) {
    Descriptor[] d= {descriptor};
    Update[] update;

    if (!ownershipMgr.isHome()) {
      if (!memoryMgr.isInitialized(descriptor)) {
        update= distributionMgr.requestUpdate
        (d, ownershipMgr.getUpdateSource(), false);
        if (update!=null) {
          for(int i= 0; i<update.length; i++) {
            memoryMgr.incorporateUpdate(update[i]);
          }
        }
      }
    }
  }
}
```

Listing 4.11: Initialization of HBtcp

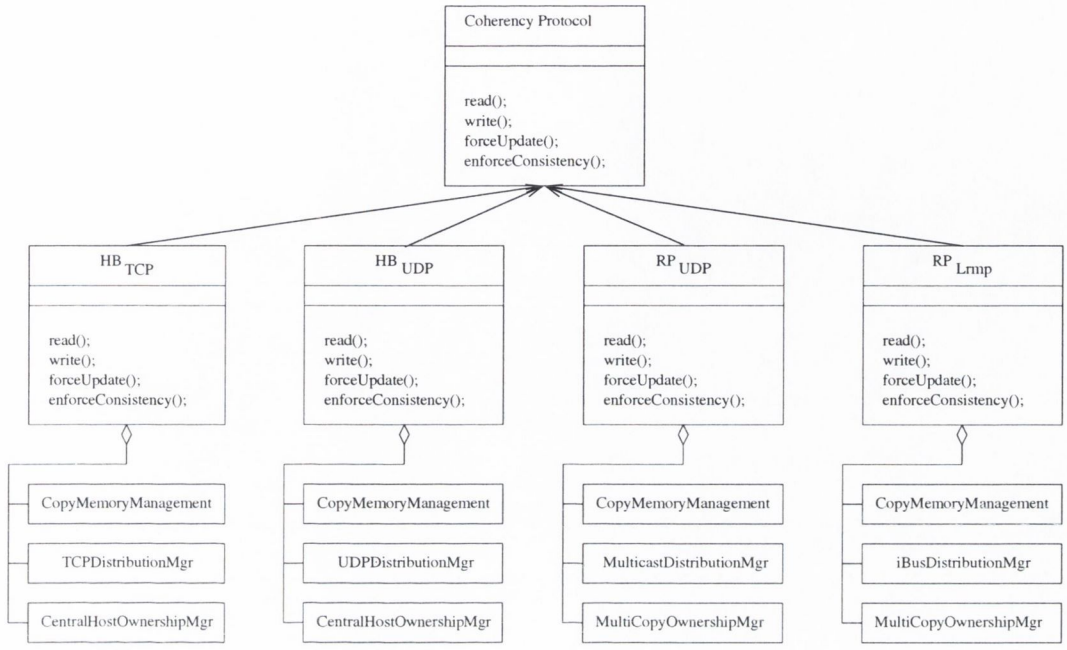


Fig. 4.14: Coherency Protocols Class Hierarchy

4.4.6 Class hierarchy of coherency protocols

As shown in figure 4.14 the coherency protocol component is built from subcomponents for memory management, ownership management and distribution management. The methods defined by the general interface are used as wrappers around the components to provide the individual behaviour of a particular coherency protocol. Hence each of the models (HB_{TCP} to RP_{Lrmp} in figure 4.14) provides exactly the same set of methods, but each individual model implements these differently through a different combination of subcomponents.

By way of an example HB_{TCP} in figure 4.14 implements a coherency protocol which is an aggregation of *CopyMemoryManagement*, *TCPDistributionMgr* (which is itself an aggregation of a TCP communication manager and a generic distribution manager) and *CentralHostOwnershipMgr*. This structure and its supporting elements are described in following section.

4.4.7 Implemented Coherency Protocols

As stated above coherency protocols are made up of memory management, distribution, and ownership components with the later two interfacing with the communication service. For the applications discussed in chapter 5 a number of different coherency protocols have been implemented, all using the same memory management component. They differ primarily in the communication service they use which in turn results in different distribution and ownership components.

Of the four protocols implemented the first two are based upon a centralized, or home based, scheme while the second two are based upon keeping peer-replicas up to date. In each case the code dependent upon the individual communication mechanism has been encapsulated in the distribution management and ownership management components.

HB_{TCP} uses TCP sockets as the communication mechanism between the central copy and the others. The ownership management registers the incoming address in a registry. Ownership managers of other copies retrieve the location of the master copy from this registry and provide the distribution management with this address for updates.

HB_{UDP} uses UDP sockets as communication mechanism. The ownership management of this protocol is the same used in HB_{TCP}.

RP_{UDP} uses multicast over UDP to keep replicated copies up to date. The ownership management registers an object identifier together with a multicast address at a registry. Nodes interested in a given object retrieve a multicast address corresponding to an object identifier. Updates and requests are exchanged with every interested node using this multicast address.

RP_{Lrmp} uses *Lrmp* as the communication mechanism.*Lrmp* is a lightweight reliable multicast protocol [64]. The protocol is based on UDP multicast and relies on the underlying communication mechanism to support multicast. *Lrmp* implements a slim layer for reliability on top of UDP multicast. The ownership management is the same as used by RP_{UDP}.

RP_{iBus} uses a group communication mechanism called IBus. This group communication mechanism provides group management, multicast features, and delivery guarantees. The group management protocol is completely implemented in software. This allows the protocol to provide group communication without multicast support from the underlying communication system. The ownership management is again the same as used by RP_{UDP}.

4.5 Deployment of a shared object

This section describes the deployment of an object as a shared object. A shared object in this context is one of many instances of a class that is instantiated in the above described execution environment and is identified by a specific identifier.

A shared object is instantiated like any Java object. In order to become a shared object, an object needs to be connected to an instance of a consistency model and a coherency protocol. This connection can be established by a programmer or by deriving the class of the object from a class that is provided by the framework. This class defines methods that establish the connections for the consistency maintenance mechanism and provide a connection to the underlying consistency model. Once the connection to the underlying components has been created, the state of the object is maintained by these components.

The following example demonstrates the deployment of an object that represents a matrix. The *Matrix* class (as shown in listing 4.12) inherits from the class *SharedObject*. The following section describes the *SharedObject* class. This description is followed by an explanation of the initialization process and the utilization of a shared object.

4.5.1 SharedObject class

The *SharedObject* class provides the developer with a simple way to share an object.

Figure 4.15 shows a simplified version of the interface of the *SharedObject* class. The figure omits a number of methods that do not affect the sharing of an object. The methods that are shown in the figure can be divided into three groups: setup, synchronization, and

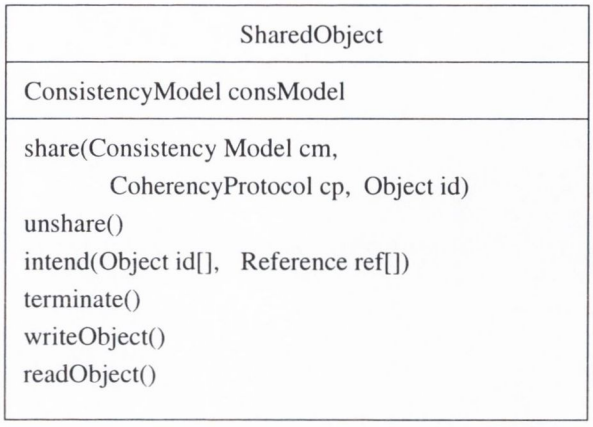


Fig. 4.15: Interface of the SharedObject class

support methods. The setup methods *share* and *unshare* establish the connection between the shared object and the instances for consistency models and coherency protocols. The synchronization methods *intend* and *terminate* publish an intention throughout the underlying consistency maintenance mechanism. The support methods *writeObject* and *readObject* implement mechanisms to distribute either the complete state of an associated object or only a few selected fields of such an object.

The *share* method has three parameters: An identifier, a reference to an instance of a consistency model and a reference to an instance of a coherency protocol. This method stores the consistency model for future reference in a variable and initializes the consistency model with the coherency protocol

4.5.2 Initialization

Listing 4.12 shows a part of the implementation of the Matrix class that demonstrates the initial setup of an object with implementations of a consistency model and a coherency protocol. The method *multiply* creates first an instance of a consistency model and a coherency protocol and then call the method *share* with these two components as parameters. The first parameter of the method-call is an identifier under which the object will be known in the distributed context.

```
public Matrix extends SharedObject {

  public Matrix multiply(Matrix B) {
    Matrix C;
    ConsModel cm;
    CohProtocol cp;

    cm= new WeakConsModel();
    cp= new HomeBasedProtocolTCP();
    this.share("A", cm, cp);

    cm= new WeakConsModel();
    cp= new HomeBasedProtocolTCP();
    this.share("B", cm, cp);

    C= new Matrix(A.getRows(), B.getColumns());
    cm= new WeakConsModel();
    cp= new HomeBasedProtocolTCP();
    this.share("C", cm, cp);
    ...
    for(i= 0; i<C.rows; i++) {
      node[random].multiplyRow(A, B, C, rows);
      }
  }
}
```

Listing 4.12: Combination of object and framework

4.5.3 Utilization

Listing 4.13 present an example that demonstrate the utilization of a shared object. The example assumes that the matrices A, B, and C have been shared. The method is intended to calculate a row of a result matrix on a worker node. The method needs to read from a given row of matrix A and all data from matrix B in order to perform this calculation for a given row of result matrix C.

The access to a shared object is synchronized by two methods *intend* and *terminate*. The *intend* method takes two parameters: an indicator of the intended action and a reference to the area that will be accessed during this action. The method converts this information into an intention and provides it to the underlying consistency model. The consistency model in turn may invoke the coherency protocol to ensure the consistency of the data.

The example shows the invocation of the *intend* method for each of the matrices. After the *intend* methods have been performed the method has the necessary access rights to perform

the calculation. The method terminates the intentions, once the calculation of the row has been performed. The termination of the intention is relayed to the underlying consistency model and the consistency model may invoke the coherency protocol to propagate any updates that were performed.

```
public void multiplyRow(Matrix A, Matrix B, Matrix C, int index) {
  A.intend("read", new MatrixReference(A, "row", index));
  B.intend("read", new MatrixReference(B, "All", 0));
  C.intend("write", new MatrixReference(C, "row", index));
  ...
  /* calculate row */
  ...
  C.terminate();
  B.terminate();
  A.terminate();
  }
```

Listing 4.13: Usage of shared object

4.6 Summary

In this chapter we presented the implementation of our framework and its supporting structures.

The beginning of the chapter introduced the supporting structures. These structures form the communication mechanism through which the components exchange information.

The description of the supporting structures was followed by two sections that detailed the class hierarchies for consistency models and coherency protocols. These class hierarchies form the heart of the implementation.

The chapter concluded with a description of the runtime environment that allows the distribution of work tasks. The description explained the assumptions about the general network environment and included details the communication between masters and workers.

Chapter 5

Evaluation

This thesis defines the crucial parts of existing DSM systems - consistency models (CMs) and coherency protocols (CPs) - as components and details their characteristics and interfaces. By doing so, it identifies a set of possible combinations of CMs and CPs. The establishment of the set allows to map all existing solutions and to identify possible unexplored CM/CP combinations.

The implementation of CM/CP interfaces and characteristics that is given in this thesis allows to explore the behaviour of combinations. Combinations can be specified as consistency maintenance algorithms on a per-object basis. This ability can be leveraged to adjust the DSM systems to application-specific semantics and thus improve performance.

Goals	Evaluation metric	Experiment
Flexibility on object level	Support of various protocols	TSP
Adaptability per object	Support of an access characteristic on a per-object basis	Matrix Mult.
Fine-grained adaptability per object	Support of several access characteristics per object	Game of Life

Table 5.1: Evaluation metrics

Table 5.1 gives an overview of the issues that are examined in this evaluation. The first issue is the protocol adaptability of the system. The evaluation of this attribute is performed by examining an application that employs a single protocol to maintain shared data. The

second point is the adaptability of the system on a per object-basis. This characteristic is evaluated with the means of an application that maintains a number of shared data items that exhibit different sharing characteristics. The third attribute of the system is a fine-grained adaptability per object. This attribute is evaluated by examining an application that shares a data item that exhibits a set of sharing characteristics.

The applications implemented are textbook parallel ones including the Traveling Salesperson, Matrix Multiplication, and Conway's Game Of Life. The applications were chosen for the different sharing characteristics they exhibit.

The Traveling Salesperson is representative for the class of applications where all communication between worker nodes is restricted to a single data structure. In these applications the data structure is used to synchronize between the individual workers and to represent their common state. This application demonstrates the sharing of a single variable. It shows the varying performance depending on the protocols chosen as underlying sharing mechanism.

Matrix multiplication is a typical example for applications where a set of structures with input data are used to calculate the contents of a result structure. The access characteristics of these structures require different sharing protocols [14] to model the read and write access that are performed on these structures. This application demonstrates the sharing of several data structures and shows how performance is influenced by individual protocols on a per-object basis.

The calculation for Conway's Game Of Life represents a group of applications where a shared data structure is accessed differently by the nodes that are involved in a calculation. The access characteristics of these structures call for the ability to apply individual protocols to model the individual access characteristics. This type of applications includes all types of matrix calculations where the matrix is divided into tiles that are calculated by individual nodes and where nodes that are calculating adjunct tiles incorporate results from their neighbours.

This application demonstrates how different sharing characteristics exhibited by a single data structure can be modeled. This example is particularly interesting as it shows how different sharing characteristics of one data structure can be represented by a combination of

one consistency model and two coherency protocols. This combination is made possible by the flexibility and extensibility of the framework.

In the next section we describe the environment in which the performance evaluation took place. This section is followed by a description of the individual applications and a discussion of their performance. The chapter concludes with a discussion of the evaluation of the framework.

The following two sections describe the environment in which the applications were executed. The first section describes the general system environment including a description of the machines that were used and the infrastructure that connected them. The second section describes the software environment that provided a mechanism to run the applications in a distributed manner.

5.1 Environment

This section presents the environment in which the evaluation was performed. The presentation is divided into the presentation of the system environment and the presentation of the software environment. The system environment describes the hardware setup and system software that was used as basis for the performance evaluation. The software environment describes the software layer that was used above the system software. It encompasses the framework classes that support the distribution of an application and provide synchronization for concurrent parts of an application.

5.1.1 System environment

The framework was evaluated on two separate clusters of workstations; a cluster of 16 Ultra-SPARC Suns and a cluster of 16 PC workstations.

The first cluster consisted of 16 Ultra-SPARC workstations Ultra 1 Model 170 with "Model 170 UltraSPARC" processors clocked at 167MHz. The workstations were connected by 100Mbps Ethernet and part of a larger general purpose network. The applications were executed during a time of low network traffic.

The operating system on these workstations was Solaris 2.7. Sun's JDK 1.1.6 [2] was used for the measurements and configured to use native threads. The virtual machines on the individual nodes were started with 10Mb pre-allocated memory. A virtual machine would start under normal circumstances with 2Mb of memory. When an application needs more memory for objects the virtual machine allocates more memory. The memory allocation during execution time can been influenced by various parameters e.g. memory contention of the system, transfer of memory contents to swap space, etc. The memory was pre-allocated before the execution of application in order to prevent interference from memory allocation during the execution time.

The measurements were taken at a time of relative idleness of the network. The remaining network traffic should have no effect on the measurements. However, it was discovered that the network was configured as one Ethernet network with more than 2000 computers connected. The individual sub-networks of the network were connected by switches. These switches allow data from certain network protocols to be transmitted over the whole network. This configuration allowed individual nodes to disrupt the communication over the network. A complete disruption of all communication occurs relatively seldom; an incident in a month. These incidents are easily identified and the cause can be corrected. However, misconfiguration of some of the connected computers cause small packet-storms. These packet-storms consist of small packets that are broadcasted to discover addresses in an Ethernet network. A misconfiguration of a computer can cause these packets to be sent in bursts. These bursts are sent at nondeterministic times by various computers on the network. They do not cause disruptions of the normal network traffic but the measurements are influenced. This influence causes the results to become nondeterministic and a fluctuation in the standard deviation.

The PC cluster consisted of uni-processor workstations with Intel Pentium II clocked at 450MHz with 512KB cache and 256Mb RAM. The workstations were connected by 100Mbps Ethernet. The cluster was disconnected from the college network in order to exclude network traffic that could interfere with the measurements.

The operating system on the workstations was a RedHat Linux 5.2 release with a Linux kernel 2.2.12. The Java development kit (JDK) used for the measurements was Blackdown

JDK 1.1.7 version 3 [1]. This JDK is based on Sun's JDK source and adapted to the GNU/Linux operating systems. It implements native thread support for the Java virtual machine (JVM) on Linux. The virtual machines were started with 10Mb pre-allocated memory to prevent by memory allocation during the runtime of measurements.

5.1.2 Software environment

The software environment consists of the framework and the execution environment that have been described in the previous chapter.

The applications that are examined in this evaluation have been distributed using the master-slave mechanism provided by the framework. Every worker node in the network executes a slave daemons. The slave daemons register with a registry and then wait for work. The applications disassemble the calculations into a number of work packages that are distributed among the waiting slave nodes.

5.2 The Traveling Salesperson

The Traveling Salesman Problem (TSP) represents a class of routing problems that search for a shortest path. Routing problems in general attempt to find a route that connects a number of given points in a specific way. A valid route has to connect these points by complying with a set of requirements. A variety of routing problems have been identified in the literature [90].

Routing problems differ in the requirements that a valid route needs to satisfy. These requirements control the calculation of a route and the validity of a connection between two points. For example, a valid route may have to connect all given points or a subset of these points and the starting and end points of a route may be fixed or freely chosen by an algorithm. A connection between two points is associated with a set of weights that represent for example the length of a connection between two locations, the costs for traveling this connection or other penalties for the use of this connection.

The Traveling Salesman Problem is a simple instance of these routing problems and has been used in a variety of evaluations of DSM systems. The points that are to be connected

in this problem represent towns that a salesman has to visit on a sales tour. The connections between towns are given as distances; where every distance between two towns is represented by a value. This creates a diagonal matrix of values with towns as specifier for rows and columns and distances as values for the individual matrix cells. A salesman has to visit every town exactly once and as an additional requirement in this instance has to return to his starting point. This requires that all possible routes have to be calculated and to be compared with one another in order to find the shortest route.

The algorithm that has been implemented to solve this problem is a branch-and-bound variant that uses a priority list. The nodes of a cluster are divided into a master node and a number of worker nodes. The master node calculates an initial "best route" following the shortest connection from city to city starting with a random city. This route represents an approximation to the best route that is fast to calculate and that may be used to eliminate other routes. This route is given to all workers as initial best route. In addition to an initial "best route", the master creates a priority list of sub-routes. Sub-routes are the possible combination of the first 4 cities of all possible routes. The priority list contains these sub-routes ordered after their length.

Each worker is assigned a route from the priority list. A worker produces complete routes from the seed-sub-route by adding the distances for new connections. The length of these sub-routes is compared with the length of the known "best route". When a sub-route is already longer than the known "best route" the sub-route and all resulting complete routes are eliminated from the calculation. This reduces the number of calculations for routes by eliminating branches of the tree that represents all possible routes. The "best route" is updated if a complete route produced by a worker is shorter than the known "best route".

The implementation of our algorithm has three shared data items, the "best route", the matrix of distances and the list of initial sub-routes.

The "best route" is - once initialized by the master - mostly read by workers. The discovery of a route that is shorter than the known "best route" and a subsequent update of the "best route" is relatively rare. This results in limited updates of the shared data and thus in low network traffic. The matrix with distances between points and the priority list

containing the initial sub-routes are shared as source structures. They are initialized once by the master at the beginning of an execution and subsequently read by workers. The matrix of distances is read frequently and replicated completely at workers nodes in order to minimize network traffic. The list of sub-routes is read sparsely by workers and is replicated only upon request.

The performance of this application has been measured with the 'best route' being shared with different CM/CP combinations.

WC_{Lock}-RP_{Lrmp} combines a weak consistency implementation based on locks with a replication protocol. The replication protocol uses Lrmp to keep all workers informed about the "best route".

WC_{Lock}-RP_{UDP} differs from the previous combination in the replication protocol which employs UDP multicast instead of Lrmp. This combination takes advantage of the small size of the shared data item. The update messages fit into on packet of the underlying system. Furthermore is the group management and the guarantee of deliveries left to the underlying system.

WC_{Lock}-HB_{TCP} employs a home-based protocol to distribute the updates to the "best route".

Figure 5.1 gives the speed-up curves for three such combinations. The best performance is achieved by WC_{Lock}combined with either of the replication based coherency protocols. It is interesting to note that there is little difference between the UDP based protocol and the lightweight reliable multicast one when running on a isolated high-speed network.

5.3 Matrix Multiplication

Matrix multiplication represents a class of problems that combine a number of source matrices to calculate a result matrix. The value of a cell of a result matrix is determined through a calculation that involves the combination of a number of cells from a set of source matrices. The calculation that is performed to determine the result can be very simple or very complex;

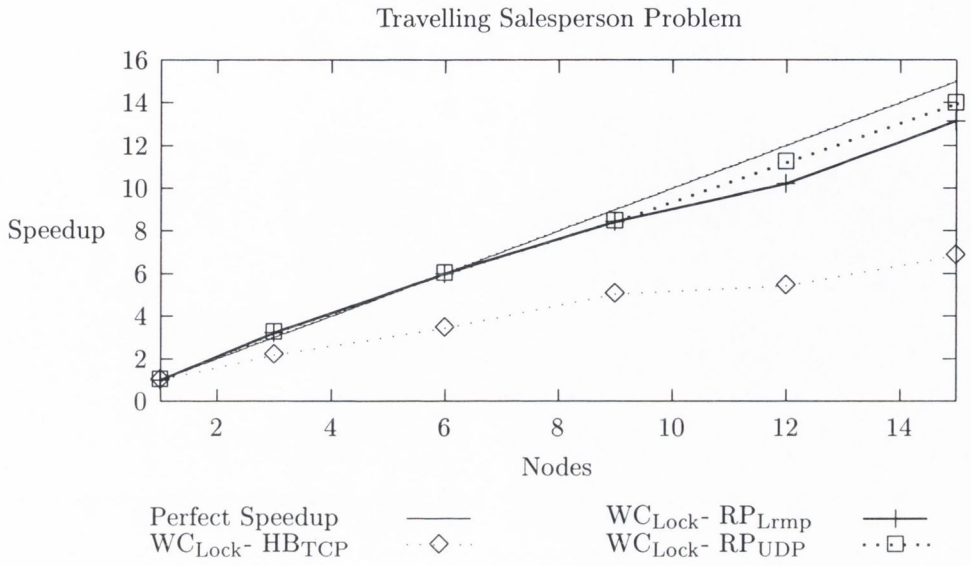


Fig. 5.1: Speed-up Curve for TSP

the access characteristic of these calculation, however, remains the same through the whole class of problems. This class of problems is very common in scientific applications and the access characteristic are repeated in these applications.

Matrix multiplication, of the form $C = A \times B$, is a simple illustration of the above describe class of problems. Elements in a row in the result matrix C are determined by multiplying elements of a row of matrix A with corresponding columns of matrix B: $C[k][l] = \sum A[i][l] \times B[k][i]$. This calculation involves read accesses from the two source matrices A and B and write accesses to the result matrix C.

The implementation of matrix multiplication in this evaluation was parallelized using the "give me work" paradigm. The master node initializes matrices A and B and creates an empty result matrix C. Available workers are assigned rows in the result matrix that need to be calculated. On completion of a computation of a row a worker writes results into the result matrix and then contacts the master looking for more work.

Each of the three structures shared in the application exhibits different sharing characteristics and needs an individual CM/CP combination. The calculation of a row in the result matrix requires a worker to have access to a corresponding row in matrix A and a full copy of

matrix B. Thus a master and workers share full copies of B while A is only partially shared, in that workers only need access to selected rows. In our implementation complete copies of matrices A and B are distributed from the master during the initialization of the workers and shared using a weak consistency model.

The sharing characteristic of matrix C has a major influence on the performance of the application. Unlike the best route in the previous example the access to this structure is more ordered. The contents of the matrix is changed during the calculation by the workers and is read at the end of the calculation by the master. Three different CM/CP combinations were used with matrix C.

WC_{Lock}- RP_{Lrmp} combines a weak consistency implementation with a replicating protocol. The replicating protocol is based on Lrmp. This combination propagates updates to the result matrix when a row has been completed. The update is multicasted to all interested nodes.

This case represents a traditional configuration in which all interested nodes are kept up-to-date during the calculation. It represents a base line to which all following measurements are compared.

WC_{Lock}- HB_{UDP} combines a weak consistency implementation with a home-based protocol. The home-based protocol is based on UDP. Updates to the result matrix are sent to the home of the data - in this case the master - using point-to-point UDP.

This configuration represents an optimization over the previous one in that the updates are sent only to one node. This configuration assumes a reliable underlying network as UDP only provides best-effort guarantees of delivery.

$LRC_{Barrier}$- HB_{TCP} is a combination of lazy release consistency and a home-based protocol. The home-based protocol is based on TCP. The updates to result matrix are kept at the worker nodes until the master attempts to read the matrix. A read attempt causes all workers to propagate their updates to the master.

This combination represents a solution that is more suited to a less reliable communications medium.

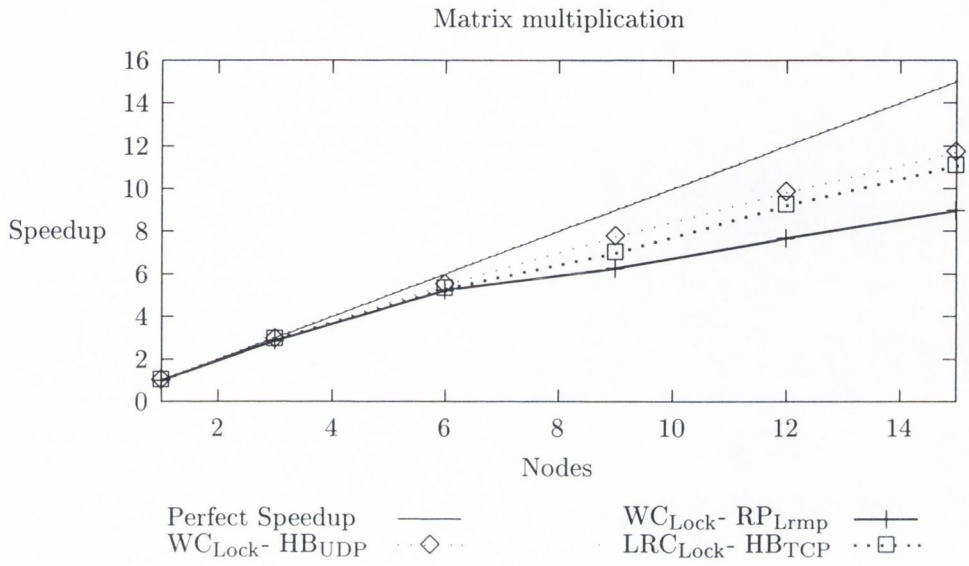


Fig. 5.2: Speed-up Curve for Matrix Multiplication

Figure 5.2 shows the speedups curves obtained. The combination (WC_{Lock}- HB_{UDP}) gives the best performance with ($LRC_{Barrier}$- HB_{TCP}) following closely behind. While for this application (WC_{Lock}- RP_{Lrmp}) gives the worst performance.

The combination (WC_{Lock}- RP_{Lrmp}) introduces overhead through the communication between worker nodes.

TCP has slight disadvantages over UDP in terms of overhead through additional measurement to ensure delivery guarantees. The combination LRC implementation delays all communication till the end of the calculation. In our case this is an advantage; in cases where communication can be handled separately from the calculation the combination (WC_{Lock}- HB_{UDP}) may have a higher advantage through concurrent calculation and communication.

5.4 The Game of LIFE

Conway's "Game Of Life" [39] represents a class of applications that maintain shared data items that are accessed during a calculation in a number of different ways by different nodes. The data items consist of a number of overlapping areas that each exhibit a different sharing characteristic.

The separation of these areas is often not possible due to overlap and functional characteristics that bind these areas close together. An example for such functional binding are the cells of a matrix that are updated sequentially by some nodes and sparsely read by other nodes. The cells could be arranged in structures so that cells that exhibit similar access characteristics are co-located and supported by the same protocol. However, this would influence the addressing of cells and increase the complexity to implement well-known algorithms.

This type of problem can be found in the scientific and economic applications where one set of nodes performs continuous calculations on a structure, while a second set of nodes uses the results of these calculations for another purpose. The "Game of Life" represents a simple example of this type of application.

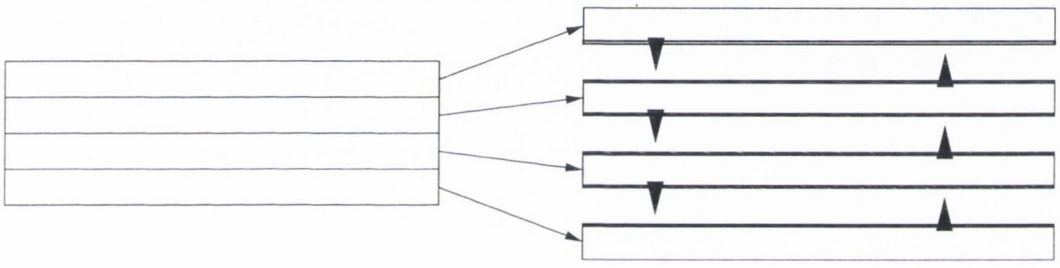


Fig. 5.3: Communication between Strips

The "Game Of Life" involves populating a two dimensional grid with a pattern of full and empty cells. Successive iterations of the grid are produced in which the cells live or die in the next generation depending upon the number of immediate neighbours they have. Life can be parallelized by dividing the grid into a number of strips and processing each strip in parallel.

Access to neighbouring rows in adjacent strips is required when calculating the values of cells on the edges, thus this application requires worker-worker communication as well as master-worker. The state of the grid is displayed by a master. In each iteration the master receives a full copy of the data but as it is too time consuming to display every iteration, it takes longer to display the data than it does to calculate it, only every 10th iteration is displayed.

The "Game Of Life" has the interesting property that the optimum coherency protocol

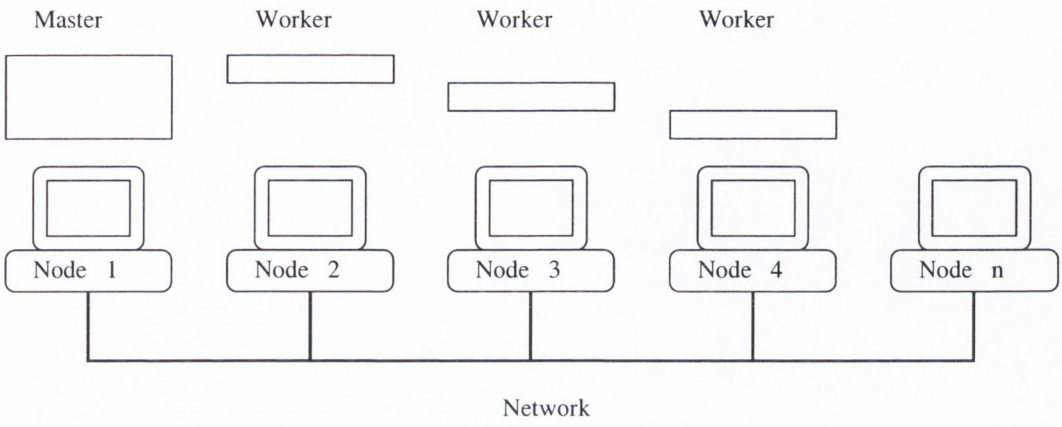


Fig. 5.4: Distribution of Conway's Game of Life

for sharing data between workers is different than the one for sharing data between workers and the master.

Hence different versions of Life were implemented.

$WC_{Barrier}$-RP_{Lrmp} uses a weak consistency model $WC_{Barrier}$ combined with a replication protocol RP_{Lrmp} for the sharing between all nodes, i.e. for both worker-worker sharing and worker-master sharing.

This combination represents a traditional solution that shares the complete structure with a single protocol. As a result all data is multicasted among all nodes.

$mWC_{Barrier}$-HB_{TCP}/RP_{Lrmp} uses a modified weak consistency model $mWC_{Barrier}$ combined with two coherency protocols HB_{TCP} and RP_{Lrmp}. HB_{TCP} is used for sharing between worker and master while RP_{Lrmp} was used to distribute the edges of strips between workers.

This combination adapts the consistency maintenance mechanism to the sharing characteristics of the individual areas. The edges that connect the strips of the individual workers represent one area. These edges are multicasted among among the workers. The complete strips that are calculated by the workers are sent to the master via TCP.

The resulting speed-up curves are depicted in figure 5.5. It shows that the application scales good with the first combination of protocols. A collection of 15 nodes gives a speed-up

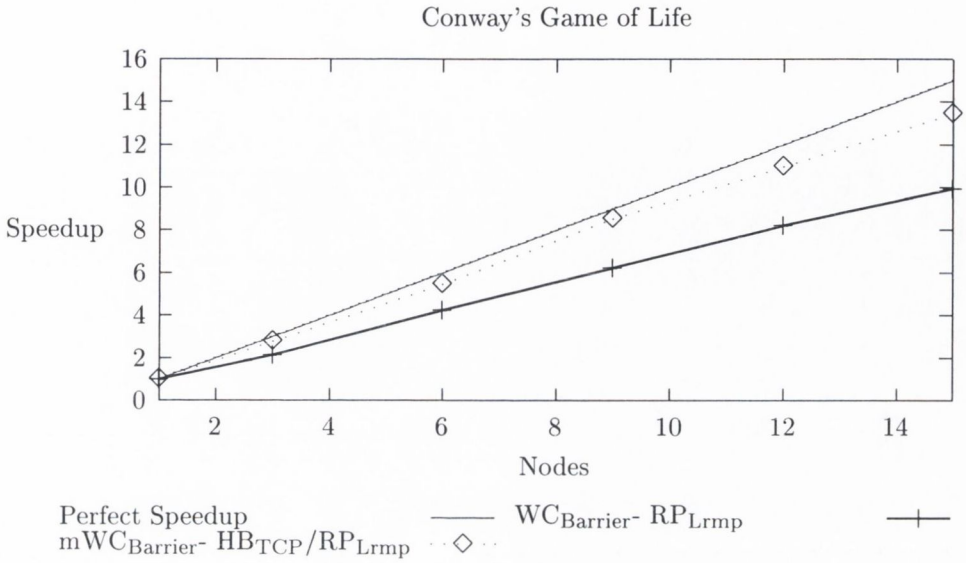


Fig. 5.5: Speed-up Curve for Conway's Game of Life

of 10. The second combination shows an even better performance with a collection of 15 node giving a speed-up of 14.

5.5 Discussion

The framework allows the developer to exploit application-specific semantics, and thus to improve performance, by enabling different consistency models and coherency protocols to be used within a single application. No single combination of consistency model and coherency protocol is best for all applications so the approach being advocated, of a customizable framework, would appear to be of potential benefit to application developers.

TSP	WC_{Lock}	-	RP_{UDP}
Matrix Mult.	WC_{Lock}	-	HB_{UDP}
Game of Life	$mWC_{Barrier}$	-	HB_{TCP}/RP_{UDP}

Table 5.2: Best performing CM/CP combinations

As shown in table 5.2, each application type has its individual best performing CM/CP combination. This can be seen as indication that no single protocol is suitable for all types of access characteristics.

The implementation of the "Traveling Salesperson Problem" shows more explicitly that no single protocol is sufficient for all environments. The transport protocol based on UDP shows advantages in isolated local-area networks. This protocol, however, is not suitable in networks with higher failure rates and needs to be exchanged against a protocol with delivery guarantees. This emphasizes that the importance of support for a variety of protocols.

Matrix multiplication shows that the application of individual CM/CP combinations on a per-object basis can result in performance gains. This confirms the necessity for support of a number of protocols at the same time.

The application of Conway's "Game of life" exhibits the advantage of our approach. The flexibility of the framework allows the exploitation of sharing characteristics. Comes close to the performance of an application based on message passing.

Goals	Evaluation metric	Result
Flexibility on object level	Support of various protocols	100% gain
Adaptability per object	Support of access characteristics on a per-object basis	33% gain
Fine-grained adaptability per object	Support of several access characteristics per object	40% gain

Table 5.3: Evaluation results

As table 5.3 shows, the flexibility provided by the framework can be exploited to gain performance advantages over traditional solutions. Especially the results of the application "Game of Life" demonstrate the advantage of the adaptability of the framework over traditional solutions.

Chapter 6

Conclusion

The focus of this thesis is the flexibility and adaptability of protocols in distributed shared memory systems. The work presented here analyzed existing protocols and examined their internal structures. This led to a new categorization of consistency models and coherency protocols. This categorization forms the basis for a framework of consistency models and coherency protocols. It allows applications that are build from the framework to be adjust the behaviour of the consistency maintenance protocols to their needs.

This chapter presents the conclusion of this thesis. It summarizes the contribution of this work and presents possible directions of future work.

6.1 Contributions

As described in the review of related work in chapter 2, existing solutions for consistency maintenance in distributed shared memory systems lack flexibility and adaptability. We see the cause for this deficiency in the definition of consistency maintenance protocols. The definition misses to identify the parameters and subcomponents of a protocol. The identification of parameters and subcomponents enables a protocol to be adapted; however, due to the lack of this identification very little flexibility and adaptability is found in existing systems.

A first achievement of this thesis is the classification of adaptability types that has been given in section 2.2. This classification details a set of possible types of adaptability that can

be implemented in a consistency maintenance protocol.

The challenge then was the separation of consistency maintenance protocols into consistency models and coherency protocols. The definition of these components and the interface between them represents the main contribution of this thesis. The clear definition of these components and their subcomponents allows an implementation of a variety of different well-specified components and their combination to individual consistency maintenance protocols. These protocols have individual characteristics and are suitable for specific access characteristics of individual shared data items.

The definition of consistency models as a combination of primitives with additional description of characteristics allows the direct comparison of implementations that follow these definitions. Previous implementations of consistency models are difficult to compare because . The definition allows implementations to be compared easily and differences in implementations to be identified.

The definition of the interface of coherency protocols and the components that provide the necessary functionality represents a new perspective on coherency protocols and the transport mechanism of consistency maintenance protocols in general. Traditional systems implemented the transport mechanism as a singular unit that was difficult to adapt. The definition given here allows to assemble a mechanism from components and adapt it to individual circumstances by selecting suitable components.

Finally, the implementation and performance chapter present proof of concept implementations based on a small set of consistency models and coherency protocols. Using these we demonstrate the improvements in performance to be accrued from tailoring the combinations of consistency models and coherency protocols to suit the application's semantics.

6.2 Future work

Future work will involve the implementation of more consistency models and coherency protocols. The population of these models and protocols will allow to adapt the mechanisms even closer to the optimal solution.

An increased number of applications will give us further indications of the advantages

and disadvantages of the flexibility of the framework and highlight the points where the discrepancy between indirection and performance gain is critical. These points represent where the design needs to be improved.

It is also planned to investigate the performance of the framework on different network environments ranging from GRID solutions over wide area networks (WANs) to loose-coupled, very high speed clusters based on the Scalable Coherent Interface (SCI) [45, 49], Gigabit Ethernet [28] or Myrinet [21].

The underlying structures of coherency protocols represent another issue that needs to be explored further. These structures include primitives that are needed to provide complete support for coherency protocols. The development of these primitives should ease the implementation of the framework on various hardware platforms.

Finally, the research into the runtime change from one consistency model to another presents another challenge. This issue has been largely ignored as many systems only provide one model or a small set of statically assigned models. In order to achieve dynamic adaptability, it appears necessary to define the exact behaviour of a system at the time of a change between consistency models.

6.3 Summary

This chapter summarized the work presented in this thesis and provided a perspective on open issues and possible future work.

The chapter started by relating the presented work to the motivation and goals described in the abstract and the chapter on related work. This was followed by a description of the contributions of this thesis. The chapter concluded with a list of open issues that are the foundation for future work.

Bibliography

[1] Blackdown Java-Linux port of JDK 1.1.7 to Linux, September 1999. `http://www.blackdown.org/`.

[2] Java Development Kit Version 1.1.6, September 1999. `http://java.sun.com/products/jdk/1.1/`.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[4] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency (extended abstract). In *Proc. of the 5th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'93)*, pages 251–260, June 1993.

[5] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming casual distributed shared memory. In *Proc. of the 11th Int'l Conf. on Distributed Computing Systems (ICDCS-11)*, pages 274–281, May 1991.

[6] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[7] J.P. Anderson, J.A. Hoffman, J. Shifman, and R.J. Williams. D825 – a multiple-computer system for command and control. In *Proceedings of AFIPS Fall Joint Computer Conference*, pages 86–96, 1962.

[8] Edwin D. Reilly Anthony Ralston and David Hemmendinger, editors. *Encyclopedia of Computer Science.* Macmillan Publishers Ltd, 4 edition, August 2000.

[9] Russel Atkinson and Edward McCreight. The Dragon processor. In *Proceedings of 2nd ASPLOS 87*, pages 65–69, October 1987.

[10] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Performance evaluation of the orca shared-object system. *ACM Trans. on Computer Systems*, 16(1):1–40, February 1998.

[11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Experience with distributed programming in orca. In *Proc. of the 1990 Int'l Conf. on Computer Languages*, pages 79–89, March 1990.

[12] H. E. Bal, A. S. Tanenbaum, and M. F. Kaashoek. Orca: A language for distributed programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[13] M. Ben-Ari. *Principles of Concurrent and Distributed Programming.* Prentice Hall, 1990.

[14] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University, April 1989.

[15] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, May 1990.

[16] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, March 1990.

[17] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory using multi-protocol release consistency. In A. I. Karshmer and J. Nehmer, editors,

Operating Systems of the 90s and Beyond, number 563 in Lecture Notes in Computer Science, pages 56–60. Springer-Verlag, July 1991.

[18] B. N. Bershad and M. J. Zekauskas. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie-Mellon University, September 1991.

[19] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[20] R. Bisiani and M. Ravishankar. Plus: A distributed shared-memory system. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 115–124, May 1990.

[21] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[22] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, pages 57–72, september 1993.

[23] Grady Booch. *Object-Oriented Analysis and Design with Applications, 2nd Ed.* Benjamin/Cummings Publishing Company, Inc., 1994.

[24] Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. *Programming Under Mach.* Addison-Wesley Publishing Co, 1993.

[25] B. Bryant, S. Sears, D. Black, and A. Langerman. An introduction to mach 3.0's xmm subsystem. Technical report, OSF Research Institute, June 1993.

[26] H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie. Overview of the ksr1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, February 1992.

[27] Jeffrey S. Chase, Miche Baker-Harvey, Henry M. Levy, and Edward D. Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 80–85, April 1992.

[28] David G Cunningham and William G. Lane. Ieee standard 802.3ab – gigabit ethernet networking, 1999.

[29] P. Dechamboux, D. Hagimont, J. Mossière, and X. Rousset de Pina. The Arias Distributed Shared Memory: an Overview. In *Proc. of the 1996 23rd Seminar on Current Trends in Theory and Practice of Informatics*. INRIA, November 1996.

[30] M. Dormanns, W. Sprangers, H. Ertl, and T. Bemmerl. A programming interface for numa shared-memory clusters. In *Proc. of the High Performance Computing and Networking Europe 1997 (HPCN'97)*, pages 698–707, April 1997.

[31] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Annual Int'l Symp. on Computer Architecture (ISCA'86)*, pages 434–442, June 1986.

[32] S. Eggers and R. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of 3rd Architectural Symposium on Programming Languages and Operating Systems*, pages 257–270, April 1989.

[33] S. Eggers and R. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, may 1989.

[34] Susan J. Eggers and Tor E. Jeremiassen. Eliminating false sharing. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, Architecture, pages 377–381, Boca Raton, FL, 1991. CRC Press.

[35] B. D. Fleisch. Using distributed shared memory as a framework for distributed blackboard. In *Proc. of the AAAI-90 Workshop on Blackboard Systems*, July 1990.

[36] B. D. Fleisch and G. J. Popek. Mirage: A coherent distributed shared memory design. In *Proc. of the 12th ACM Symp. on Operating Systems Principles (SOSP-12)*, pages 211–223, December 1989.

[37] A. Forin, J. Barrera, and R. Sanzi. The shared memory server. In *Proc. of the Winter 1989 USENIX Conference*, pages 229–243, January 1989.

[38] A. Forin, J. Barrera, M. Young, and R. Rashid. Design, implementation, and performance evaluation of a distributed shared memory server for mach. In *Proc. of the Winter 1989 USENIX Conference*, January 1989.

[39] M. Gardner. Mathematical Games: The fantastic combinations of john conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[40] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Computer Systems Laboratory, Stanford University, December 1993.

[41] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[42] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherence Interface Working Group, March 1989.

[43] Paulo Guedes, Miguel Castro, and Nuno Neves. The disom distributed shared object memory. In *ACM SIGOPS European Workshop*, pages 138–140, 1994.

[44] S. Gupta. Stanford dash multiprocessor: the hardware and software. In *Proc. of Parallel Architectures and Languages Europe (PARLE'92)*, pages 802–805, June 1992.

[45] D. B. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, February 1992.

[46] E. Hagersten, A. Saulsbury, and A. Landin. Simple coma node implementations. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 522–533, January 1994.

[47] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australiasion Computer Science Conference*, pages 271–280, November 1993.

[48] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, July 1990.

[49] IEEE. *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface.* IEEE, Inc., August 1993.

[50] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[51] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: High-performance all-software distributed shared memory. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 213–228, December 1995.

[52] A. Judge, P. Nixon, B. Tangney, S. Weber, and V. Cahill. *High Performance Cluster Computing*, volume 1, chapter 17, pages 409–438. Prentice Hall, 1999.

[53] Alan Judge. *Supporting application-consistent distributed shared objects.* PhD thesis, Department of Computer Science, Trinity College, October 1996.

[54] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, December 1994.

[55] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proc. of the 16th Int'l Conf. on Distributed Computing Systems (ICDCS-16)*, pages 91–98, May 1996.

[56] P. Keleher. Update protocols and iterative scientific applications. In *Proc. of the First Merged Symp. IPPS/SPDP 1998)*, pages 675–681, 1998.

[57] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[58] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[59] P. Keleher and C-W. Tseng. Improving the compiler/software dsm interface: Preliminary experiences. In *Proc. of the First SUIF Compiler Workshop*, January 1996.

[60] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–10, January 1996.

[61] Gregor Kiczales, Jim de Rivieres, and Daniel Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[62] Povl T. Koch, J. S. Hansen, E. Cecchet, and X. Rousset de Pina. SciOS : An sci-based software distributed shared memory. In *WSDSM'99, 1st Workshop on Software Distributed Shared Memory*, June 1999.

[63] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[64] Tie Lao. Light-weight reliable multicast protocol specification. Internet Draft, oct 1998.

[65] D. E. Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor.* PhD thesis, Stanford University, 1991.

[66] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 148–159, May 1990.

[67] D. E. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. L. Hennessy, M. Horowitz, and M. Lam. Design of the stanford dash multiprocessor. Technical Report CSL-TR-89-403, Computer Systems Laboratory, Stanford University, December 1989.

[68] D. E. Lenoski, J. Ludon, K. Gharachorloo W-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The stanford dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[69] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *Proceedings of ACM SIGOPS 5th Symposium on Operating Systems Principles*, November 1975.

[70] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Department of Computer Science, Yale University, September 1986.

[71] K. Li. Ivy: A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[72] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[73] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, November 1989.

[74] Paul Lu. Aurora: Scoped behaviour for per-context optimized distributed data sharing. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, April 1997.

[75] Paul Lu. Implementing optimized distributed data sharing using scoped behaviour and a class library. In *Proc. of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, June 1997.

[76] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Transactions on Programming Languages and Systems*, 8(1):142–153, January 1986.

[77] M. Mizuno, M. Raynal, G. Singh, and M. L. Neilsen. An efficient implementation of sequentially consistent distributed shared memories. In *Proc. of the IFIP WG10.3 Int'l Conf. on Applications of Parallel and Distributed Computing*, pages 145–154, April 1994.

[78] A. Mohindra and U. Ramachandran. A survey of distributed shared memory in loosely-coupled systems. Technical Report GIT-CC-91/01, College of Computing, Georgia Institute of Technology, January 1991.

[79] L. R. Monnerat and R. Bianchini. Adsm: A hybrid dsm protocol that efficiently adapts to sharing patterns. Technical Report ES-425/97, Federal University of Rio de Janerio, COPPE Systems Engineering Computer Science Department, March 1997.

[80] L. R. Monnerat and R. Bianchini. Efficiently adapting to sharing patterns in software dsms. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, February 1998.

[81] D. Mosberger. Memory consistency models. *ACM Operating Systems Review*, 27(1):18–26, January 1993.

[82] J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proc. of the 28th Hawaii Int'l Conf. on System Sciences (HICSS-28)*, volume I, pages 74–84, January 1995.

[83] M. Raynal and A. Schiper. A suite of formal definitions for consistency criteria in shared memories. In *Proc. of the ISCA Int'l Conf. on Parallel and Distributed Computing Systems (ICPDS'96)*, pages 125–130, September 1996.

[84] A. Saulsbury, T. Wilkinson, J. B. Carter, and A. Landin. An argument for simple coma. In *Proc. of the 1st IEEE Symp. on High-Performance Computer Architecture (HPCA-1)*, pages 276–285, January 1995.

[85] W. Shi, W. Hu, and Z. Tang. An interaction of coherence protocols and memory consistency models in dsm systems. *ACM Operating Systems Review*, 31(4):41–54, October 1997.

[86] J. P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. An empirical comparison of the kendall square research ksr-1 and stanford dash multiprocessors. In *Proc. of Supercomputing'93*, pages 214–225, November 1994.

[87] C. Skousen and D. S. Miller. The sombrero single address space operating system. Technical Report TR-95-012, ASU Computer Science and Engineering Department, September 1995.

[88] SoftWired. ibus programmer's manual, August 1998.

[89] P. Stenstrom, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 80–91, May 1992.

[90] Hamdy A. Taha. *Operations Research: An Introduction.* Prentice Hall, 6 edition, December 1996.

[91] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Using broadcasting to implement distributed shared memory efficiently. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 387–408. IEEE Computer Society Press, 1994.

[92] C. P. Thacker and L. C. Stewart. Firefly: A multiprocessor workstation. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pages 164–172, oct 1987.

[93] K. Thitikamol and P. Keleher. Multi-threading and remote latency in software dsms. In *Proc. of the 17th Int'l Conf. on Distributed Computing Systems (ICDCS-17)*, May 1997.

[94] T. J. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly. Angel: A proposed multiprocessor operating system kernel. Technical Report TCU/CS/1992/10, Dept. of Computer Science, City University, U.K., 1992.

[95] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software write detection for a

distributed shared memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 87–100, November 1994.