Eidolon: Adapting distributed applications to their environment

A thesis submitted to the School of Computer Science and Engineering at The University of New South Wales in fulfilment of the requirements for the degree of Doctor of Philosophy.

Daniel Potts

January 29, 2008

I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

In loving memory of my sons Joshua and Jacob.

CONTENTS

List of Figures

LIST OF TABLES

Abstract

Grids, multi-clusters, NUMA systems, and ad-hoc collections of distributed computing devices all present diverse environments in which distributed computing applications can be run. Due to the diversity of features provided by these environments a distributed application that is to perform well must be specifically designed and optimised for the environment in which it is deployed. Such optimisations generally affect the application's communication structure, its consistency protocols, and its communication protocols.

This thesis explores approaches to improving the ability of distributed applications to share consistent data efficiently and with improved functionality over wide-area and diverse environments.

We identify a fundamental separation of concerns for distributed applications. This is used to propose a new model, called the *view model*, which is a hybrid, cost-conscious approach to remote data sharing. It provides the necessary mechanisms and interconnects to improve the flexibility and functionality of data sharing without defining new programming models or protocols.

We employ the view model to adapt distributed applications to their run-time environment without modifying the application or inventing new consistency or communication protocols. We explore the use of view model properties on several programming models and their consistency protocols. In particular, we focus on programming models used in distributed-shared-memory middleware and applications, as these can benefit significantly from the properties of the view model.

Our evaluation demonstrates the benefits, side effects and potential shortcomings of the view model by comparing our model with traditional models when running distributed applications across several multi-clusters scenarios. In particular, we show that the view model improves the performance of distributed applications while reducing resource usage and communication overheads.

PUBLICATIONS

Portions of this work have been published in the following articles:

Adapting distributed shared memory applications in diverse environments [74]

Daniel Potts and Ihor Kuz

Proceedings of the 6th International Symposium on Cluster Computing and the Grid, Singapore, May, 2006

ACKNOWLEDGEMENTS

There are many people that have influenced my life over the years who have directly and indirectly helped me along the road to completing my thesis. I'd like to begin by thanking my friends, family and work colleagues at Open Kernel Labs for their love, support, encouragement and laughter.

I would like to thank my supervisor Prof. Gernot Heiser for his ongoing support and encouragement. Ihor Kuz, my co-supervisor for this dedication towards improving content, writing skills and attention to detail that contributed to many refinements of this thesis and earlier papers.

I would also like to thank staff, researchers and students at the School of Computer Science and Engineering of the University of New South Wales for their academic, financial and moral support. The Gelato program at the University of New South Wales for their technical support and access to hardware for debugging and running the framework presented in this thesis.

Finally, I'd like to thank my wife Belinda Hannan for her love and support over the years and our little girl, Zara, for providing me with company during the long nights of thesis writeup (even if she was sleeping).

CHAPTER 1

INTRODUCTION

This dissertation considers the challenges of running distributed applications in diverse environments.

We identify a fundamental separation of concerns for distributed applications. This is used to propose a new model, called the *view model*, which is a hybrid, cost-conscious approach to remote data sharing. We employ the view model to adapt distributed applications to their run-time environment without modifying the application or inventing new consistency or communication protocols.

1.1 Overview

Developing parallel programs and running them across many computers in a distributed environment has long been established as an approach to achieve performance and functionality improvements beyond those possible with a single computer.

In order to get the greatest benefit out of this approach, distributed applications, much like their sequential counter-parts, must be optimised for their programming and run-time environments. For distributed applications this often requires careful consideration of network bandwidth and latency, algorithm scalability, and many other characteristics of the application, its middleware, and its run-time environment.

Unfortunately, optimising an application for one environment often results in poor performance or loss of functionality in another. For example, some optimisations, such as taking advantage of specialised communication interconnects for performance improvements, will reduce the portability of an application.

This is a significant problem, as there is a diverse range of distributed computing environments available today. These range from homogeneous clusters, heterogeneous clusters and multi-processors, to Grids and other wide-area computing networks. When developing a distributed application, an applicaton programmer must choose between trade-offs of functionality, portability, performance and time required to optimise for a specific environment.

The programming model used by the application is another consideration that impacts the ability of a program to run across diverse environments, and how easy it is to optimise or adapt to specific environments. The programming model somewhat determines the parallelism approach taken by the programmer. Parallelism is generally provided via explicit message passing, or a shared data model.

Shared data programming models, which implement some form of distributed shared memory (DSM), are often considered to be easier to use as they behave much more closely to that of a convential uni-processor or multi-processor machine. However, DSM applications are considered to be poor performers with poor scalability, due to overheads of providing data sharing, consistency and communication in software. Unlike DSM, most message-passing applications use explicit message transfers, avoiding unnecessary communication and related overheads. As such, message-passing has become a prominent approach to sharing data in distributed applications.

DSM applications, and to a lesser degree, message-passing applications, perform poorly when moved to new environments, across wide-area environments, and in heterogeneous environments. Furthermore, due to a lack of scalability and perhaps more importantly, adaption, DSM applications fail to provide similar levels of functionality to those provided by message-passing environments. However, there has been a recent push to address these issues with approaches such as single-sided MPI [68], and middleware such as InterWeave [23] and JuxMem [5] which are better suited to wide-area distribution.

This thesis identifies a fundamental separation of concerns for distributed applications. The programming model, consistency protocols, communication protocols, sharing interactions and the execution environment are all separate aspects of a distributed application. Traditionally, application and distributed software developers do not differentiate between these aspects of a distributed application, thereby limiting and restricting the application to a specific execution environment. This thesis proposes that by treating these separately, we gain significant flexibility in the design, optimisation and execution of distributed applications.

We present this approach with a focus on shared-data programming models, which are able to benefit most from the flexibility and adaption aspects of this approach. In particular, we are able to apply the techniques developed in this thesis to demonstrate new ways to improve the performance and functionality of DSM applications, thereby allowing DSM to continue to develop as an alternative programming model to message passing.

1.2 Problem areas

Grids, multi-clusters, NUMA systems, and ad-hoc collections of distributed computing devices all present diverse environments in which distributed computing applications can be run. Due to the diversity of features provided by these environments a distributed application that is to perform well must be specifically designed and optimised for the environment in which it is deployed. Such optimisations generally affect the application's communication structure, its consistency protocols, and its communication protocols.

Applications in new environments Since applications are optimised in environment-specific ways, reusing an application in different environments poses some problems. First of all, when an application's execution environment does not match the environment that it was developed for, problems with performance and poor resource utilisation arise.

For example, a DSM application designed to run on a cluster environment using a software-based consistency protocol will not run efficiently on a ccNUMA machine. The application is not able to take advantage of the hardware's built-in data sharing and consistency mechanisms, thereby under utilising the resources available. Likewise, when running a message-passing application in this environment the application will normally communicate over TCP/IP by treating each application instance as if it were running on a separate node, unaware of the underlying hardware interconnect. Furthermore, an application designed to run on a cache-coherent NUMA (ccNUMA) machine may not run efficiently in a cluster environment (with the help of a DSM software layer) due to the application's inability to compensate for added latency penalties resulting from the cluster's slower interconnects [72]. To avoid this, applications must be optimised for every new environment that they run in.

Traditionally this problem can be solved in two ways. Firstly, if a replacement protocol library is available, it can be used. For example, many DSM middleware allow the user to select from a variety of different protocols [12,65]. However, this requires the application to be compatible with the new protocol library. The other choice is to port or modify the application and underlying protocol implementation to match the new environment. Given the considerable differences between environments, most distributed applications need significant and time-consuming modifications if they are to achieve reasonable performance [90].

Applications in non-uniform environments The second problem concerns the execution of distributed applications in non-uniform environments, that is, environments that consist of a collection of smaller environments, each with different characteristics. Take, for example, a two-cluster system where one cluster consists of nodes connected by a high speed interconnect (such as Myrinet) while the other cluster consists of nodes connected by Ethernet. To achieve good performance, an application running in this environment should be aware of the available interconnects and take full advantage of them [11,53]. Thus, when communicating within the Myrinet cluster, a Myrinet-specific protocol rather than TCP/IP over Ethernet should be used. Furthermore, the application should be partitioned in such a way that it takes advantage of the high speed interconnects while avoiding communication over the slower interconnects.

All the above problems relate to the configuration and optimisation of distributed applications in diverse execution environments. The main problem is that applications, and in particular their consistency protocols, require modifications that are highly tailored to the environment in which they run. Any significant change in the environment requires a change in the application.

Application programming models The problems presented above exist regardless of the programming model used for writing our distributed applications. For example, a matrix multiply application may be written for OpenMP as well as for single-sided MPI. The same approach to optimisations must be re-implemented for each programming model.

The trend towards building multi-model applications is another method for optimising an application to suit the execution environment. Multi-model applications utilise hybrid parallelisation paradigms to implement the application using more than one programming model. They utilise programming models that are best suited for particular environments for each part of a program. For example, data distribution is implemented with single-sided MPI while the inner-loop computation is implemented using shared memory. Such an application is well suited to run on a cluster of ccNUMA machines. Examples include the multi-zone NAS parallel benchmarks [52] which experiment with several hybrid implementations and other research into hybrid parallelisation [17, 84].

Multi-model applications require mechanisms in place for programming model interaction. Currently, interaction between programming models is implemented explicitly by the application, restricting the ability to reuse components of a multi-model application and adapt them to different runtime environments.

1.3 Revisiting distributed applications and middleware

All of the problems presented above relate to our ability to develop distributed applications that can share data efficiently in using our diverse computing resources. This includes not only how our applications perform in new environments, but also how they are able to interact with other distributed applications. To solve these problems we require a distributed computing model that provides us with the flexibility to adapt applications to both their runtime environment and their data sharing interactions.

In all of the cases introduced in Section 1.2 the main problem is that applications, and in particular their consistency protocols, require modifications that are highly tailored to the environment in which they are run. Any significant change in the environment requires a change in the application which may be difficult, time consuming and error prone.

There are two shortcomings of traditional distributed application development approaches that this thesis will address:

1. Lack of adaption of distributed applications.

   There are two main issues here. Firstly, applications designed for one environment do not run well in another. Secondly, applications perform poorly when executed in non-uniform environments including diverse and heterogeneous environments. In both cases, traditional models lack the ability to adapt the application to the environments without requiring significant effort from the application developer.

2. Limited data sharing flexibility of applications.

   Separate applications that use different consistency or communication techniques are not able to easily or effectively share data without significant program modification.

In order to address these shortcomings the following requirements will be addressed by the solution provided in this thesis:

Configuration and optimisation of the application so that it better adapts to diverse environments. The requirements in this category can be summarised as follows:

- Application adaption via application-independent protocol selection. This is one approach to optimising an application without making direct changes to the application.
- The ability for applications to handle protocol domains for easy optimisation of locality.
- A method that allows protocols to interact with each other so that applications may select different protocols best suited to particular parts of a runtime environment.

Protocol interoperability expands the requirements related to configuration and optimisation beyond environment adaption to include the data sharing interactions between applications that use different programming models.

As applications can be written in different programming models, they may require the ability to access data or state generated from applications written using other programming models.

Specifically, we require the following:

- The ability to share data between applications that use different programming models and are implemented using different protocols.
- The ability to share data in multi-model applications.

Ease of use of a model that provides a solution to the problems of configuration, optimisation and protocol interoperability. A model that provides these solutions should do so without the complexity and restrictions of traditional solutions.

1.4 Thesis contributions

This thesis explores approaches to improving the ability of distributed applications to share consistent data efficiently and with improved functionality over wide-area and diverse environments.

This thesis proposes the *view model* as a hybrid, cost-conscious approach to remote data sharing. This model has the following properties:

- It separates the application and its programming model from issues of consistency protocol, communication and data flow hierarchies. This provides us with the *flexibility* required to provide configurations and optimisations that are *independent* of application and programming model.
- It allows communication, consistency and synchronisation protocol selection without requiring changes to the application. This enables performance tuning by allowing an execution environment to transparently select the best protocols suited to an application.

- It allows the choice of programming model to be independent of protocol, communication and data flow hierarchies supported by specific implementations of that model.
- It allows an application to use more than one protocol at once for sharing the same state. That is, the model supports interactions between different protocols allowing them to communicate and ensure consistency of data.
- It provides a mechanism for the development of applications that use multiple programming models. That is, it supports the interactions between different programming models and their underlying protocol implementations.

This thesis makes the following contributions:

1. A model based on a separation of concerns of a distributed application.
2. An architecture implementation of our model called *Eidolon* which specifies some constraints on the model suitable for an implementation.
3. An experimental framework and implementation used to evaluate Eidolon and traditional third-party approaches using existing applications.
4. An approach to address lack of adaption of distributed applications to their run-time environment, without modifications to the application, or significant effort from the user.
5. An approach to improve the flexibility of sharing consistent data between distributed applications such as those that use different programming models.

1.5 Thesis road-map

The remainder of this thesis includes the following chapters:

Related Work Chapter 2 surveys existing approaches to running distributed applications in wide-area and diverse environments. Other related work including background material is also discussed.

The View Model Chapter 3 presents the conceptual model. This chapter explores the properties inherent in the model and provides examples of how each property can be used to share data.

Eidolon: A Realisation of the View Model Chapter 4 describes our implementation of the view model as an architecture suitable to use primarily for distributed shared memory and message passing systems.

Eidolon Framework Chapter 5 provides implementation details of the experimental framework. This includes details on how applications are executed, locate data and communicate using our framework.

Programming Views Chapter 6 explores the implementation of several programming models and consistency protocols in Eidolon.

Experimental Verification and Evaluation Chapter 7 presents our experiments and evaluations of Eidolon. We focus on demonstrating the benefits and side-effects of the view model and compare these against traditional approaches.

Using Eidolon Views Chapter 8 discusses approaches to using Eidolon to run distributed applications and as a foundation for distributed systems.

Conclusion Chapter 9 concludes the dissertation by examining how well the challenges of running distributed applications in diverse environments has been addressed and provides a discussion of future work.

CHAPTER 2

RELATED WORK

Distributed computing in diverse environments or across a wide area requires:

1. programming models and protocols for sharing consistent data,
2. services and protocols for locating data based on a unique identifier (e.g., a name or address),
3. heterogeneous shared data access.

These services and protocols should be efficient and resource-aware in order to deliver reasonable performance. In this chapter we explore the related work in these areas.

2.1 Background

This thesis considers two communication paradigms used to share data. These are *shared memory communication* and *message passing communication*. While the approach presented in this thesis is relevant to both communication and programming model paradigms, we focus on shared memory communication, which due to its nature, can benefit significantly from the model presented. We also present related work on message passing systems and their approaches to wide-area and diverse environment distributed computing.

2.1.1 Shared memory communication

Communication occurs by altering the contents of shared memory, usually by using architecture load and store instructions. Synchronisation and mutual exclusion is often performed via barrier and locking primitives, if these primitives are defined to be part of the memory model.

This communication model very closely mimics the execution environment of a computer, providing ease of programming and transparency for the application developer. However, the visibility of updates to shared memory depends on the memory model provided by the shared memory implementation.

SMP and ccNUMA machines implement shared memory communication transparently in hardware. In order to use the same programming and communication paradigms between separate nodes, a software implementation must be provided which simulates a clearly-defined memory model.

Distributed shared memory (DSM) [61] provides the illusion of a cache-coherent multiprocessor to an application.

One popular approach is to use the virtual memory mechanism of modern computers to trap memory operations and keep data coherent via message passing. These are often referred to as page-based DSM systems due to their used of page-fault catching mechanisms of an architecture. The programming environment is the same as that of a shared memory system.

Object-based DSMs maintain consistency at the object level. They generally employ approaches [24, 82] that avoid using an architecture's page-fault catching mechanisms.

Examples of shared-memory-communication programming models include sequential consistency and release consistency. Many traditional hardware architectures provide a sequential consistency model. For multi-processors, it guarantees that all processors observe writes in the same order. Providing sequential consistency across a cluster of nodes requires a software implementation to enforce this memory ordering rule. This is generally achieved through single-writer, multiple-reader protocols.

For many applications, sequential consistency has significant drawbacks for both hardware and software implementations. For hardware implementations, issuing memory operations in order is not always efficient. For software implementations, the granularity of updates can cause problems with false sharing when one or more processors access different data elements that reside within the same update unit. Many early DSM consistency protocols used an architecture's page size as the granularity of updates.

In order to understand why performance of DSM is poor compared to hardware-based systems and to analyse the coherency characteristics of applications running in a DSM environment, we can highlight the following traits of DSM.

Consistency granularity is the smallest unit of memory transfered over the network. This is normally restricted in size by the architecture's supported page size granularity such that a *page* is the unit of consistency granularity. However, some DSM consistency protocols reduce communication by examining the changes to a page and only communicating those changes. In this case, the unit of consistency granularity is much smaller than a page.

False sharing occurs when multiple processors access unrelated variables within the same unit of consistency granularity and at least one is a write access. This causes unnecessary network traffic as consistency state and possibly the data within the contended unit of consistency granularity bounces between nodes unnecessarily.

Fragmentation occurs when an entire unit of consistency granularity is fetched instead of just the word for which the memory operation references. This adds unnecessary traffic on the network. However, it has the benefit of prefetching data.

Protocol overhead includes the latency of communication performed in software over a typical network. Protocol overhead varies with network topology and the method used to maintain consistency state. The cost of servicing the virtual memory requests in software also add to this overhead.

Synchronisation is used to keep data consistent and is normally performed via message-passing primitives. The protocol overhead of message-passing has a dramatic impact on synchronisation. There are several methods for maintaining consistent state used to enforce and provide synchronisation of data such as tagging data with a version or vector timestamp. These vary in communication and computational overhead. Many current synchronisation techniques suffer from unbounded growth in consistency meta state that requires garbage collection to limit growth.

To relieve some of the problems with sequential consistency models, other models have been proposed. Protocols that relax the consistency requirements of shared data differ in the way that they maintain, propogate and apply coherency information at coherence points in the application (eg. *acquire* and *release* locking semantics). In doing so, they achieve performance gains by reducing the amount of communication required over sequential consistency while others focus more on reducing latency.

The protocols discussed below have different levels of consistency. Some are suited to particular applications more than others.

Sequential consistency (SC) [59] is a typical method employed by hardware-based coherent systems such as typical symmetric multiprocessor systems. Ordering rules apply to reads and writes. The first DSM systems implemented this method of consistency.

This method performs poorly due to unnecessary communication when implemented as a software protocol for distributed systems.

Release consistency (RC) [35] weakens consistency model [2] by delaying the propagation and application of coherence information until explicit synchronisation points (distinguished by the use of synchronisation variables). This method significantly improves the performance of many distributed applications.

Release consistency is supported by the Munin system [20] and more recently it is supported by architectures such as the Itanium [44].

Many improved software implementations also exist. Lazy release consistency (LRC) [55] improves the performance of release consistency by propagating changes to data only as demanded by a remote node, instead of propagating changes eagerly.

Home-based lazy release consistency (HLRC) [96] Home-based protocols [45] provide the opportunity to improve scalability and limit the overheads that are typical with homeless distributed protocols. Unlike homeless protocols, they do not allow page changes to be spread across multiple nodes at any one time. HLRC, for example, assigns a node to each page. Any requests for up-to-date data are made to the home node. Changes made to a page are sent back to the home node at a specified synchronisation point.

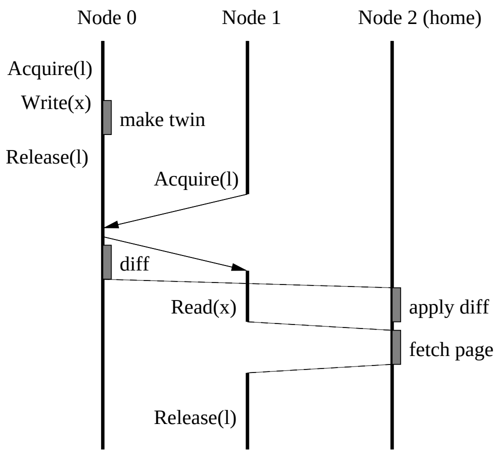


Figure 2.1: Home-based Lazy Release Consistency

Figure 2.1 shows an example of how a home-based protocol works between multiple nodes. Unlike the homeless LRC protocol, where page changes are stored at the writer node(s) and provided on demand to the faulting node, in home-based LRC the page changes (page differences known as *diffs*) containing the new value of `x` are sent to the home node at release or acquire time. Unlike LRC, the page changes made at each node can be discarded as soon as the home node acknowledges reception of the message. This significantly reduces the memory and communication overhead.

Entry consistency (EC) [13] binds data to synchronisation variables and only makes this data consistent at a synchronisation event. This binding enforces not only *when* data is made coherent (such as in release consistency) but also *which* data is made coherent.

EC can be tedious for programmers as they have to explicitly provide the binding.

Scope consistency (ScC) [47] is a refinement and relaxation of EC whereby data accessed within a scope (by acquiring the scope/lock variable) is consistent.

Instead of requiring the programmer to bind data to synchronisation variables, ScC synchronisation variables define scope with which the association of data is achieved dynamically when a write access occurs inside a scope.

ScC reduces some of the burden on the programmer compared to EC. However, extra effort may be required to ensure program correctness, compared to the other consistency protocols mentioned above.

While DSM is generally the ideal model for most users [83], problems with false sharing, write detection and excessive communication has lead to DSM being a less popular paradigm in the distributed computing community. However, we believe this model is better suited for applications that perform irregular data accesses, and because it maps more naturally to the models provided by hardware architectures. This is particularly relevant with the increasing trend of using multi-processor systems for both distributed and general computing.

2.1.2 Message-passing communication

Communication occurs by explicitly exchanging messages, usually via simple send and receive operations. Message-passing communication is implemented in programming models such as MPI [67] and PVM [86].

Arguably message passing is easier for the programmer to reason about as threads of an application must explicitly communicate with their intended source or destination. However, for applications with irregular data accesses, message passing is somewhat troublesome as the source or destination of communication is often unknown.

Furthermore, message passing is typically asynchronous, requiring the message to be copied one or more times when transferring from sender to receiver. Hence message passing does not adapt well to architectures that provide shared memory in hardware, especially for applications that can use shared data without copying.

More recently, one-sided message passing techniques have been introduced as part of the MPI specification [68] to address the difficulty to perform irregular data accesses which benefits some applications. This is also known as remote direct memory access (RDMA), as it uses a uni-directional shared-memory window for shared communication between nodes. One node end-point uses `put` and `get` operations that manipulate data in a shared memory window on a remote end-point. The remote end-point can perform normal memory read and write operations in this memory window.

2.2 Optimising applications via protocol selection

One approach to improving the performance of a distributed application is to improve or replace the underlying middleware software implementation. In this section we discuss these approaches.

2.2.1 Traditional protocols and middleware

The traditional DSM approach to addressing poor performance has typically focused on new or improved protocols that aim to reduce the amount of communication that occurs between nodes such that it more closely resembles the communication in explicit message passing programs. This is generally achieved by detecting and adapting to the data access patterns of an application [3,29,48,69,70].

Carter [19] proposes multi-protocol release consistency allowing each shared variable to use a protocol that best matches the way it is accessed. He demonstrates that DSM applications can come close to the performance of message passing programs by focusing on reducing the amount of DSM-related communication. His work focuses on developing protocols that adapt to the data access patterns of applications and does not directly consider the run-time environment.

Other alternatives have proposed new programming models that in some cases required the application to be modified for correctness. For example, instead of requiring the programmer to bind data to synchronisation variables as is the case for release consistency, scope consistency [47] uses synchronisation variables to define scope. Data is dynamically associated with a scope when a write access occurs inside a scope. The visibility of changes to data is therefore associated with a scope, requiring a worker thread to first obtain a scope before data changes are visible. This approach helps reduce the amount of data associated with each synchronisation variable, thereby reducing excessive communication. It also indirectly assists grouping data with common access requirements together, which is an approach used by entry consistency [13].

Another model, view-oriented parallel programming (VOPP) [42, 43] aims to assist the application programmer to better organise shared data in order to performance optimise the application. It achieves this by requiring the user to group data with similar attributes, such as frequency of access, into objects called *views*. Views become the granularity of data access and are accessed exclusively. This behaviour is provided by the view-consistency protocol. Note that the VOPP model and consistency protocol are orthogonal to the model and architecture proposed in this thesis. Both use the term view, however each model provides it as a unique concept. In both cases, it can be considered a "view" of memory. MultiView [49], a DSM which supports fine-grained sharing, also uses the term "view".

The approaches discussed in this section are only suitable for scenarios where the protocol and or programming model suits the run-time environment and the needs of the application. These protocols may not fully utilise the available resources of some environments. Likewise, adapting an application to a new programming model can require a significant amount of effort from the user.

2.2.2 Specialised protocols and middleware

When the hardware architecture and network topology of the run-time environment is known, as is often the case with a cluster of nodes, the middleware can be optimised to take such characteristics or features into account.

For example, the architecture features of many multiprocessor machines, such as SMP and ccNUMA, provide specialised mechanisms that perform communication and synchronisation at the hardware level between the processors of each node. To take advantage of these mechanisms for data communication and synchronisation, the application or underlying middleware must be modified.

One approach is to modify the application to support multiple threads of execution [79, 87]. Walters *et al.* [91] support multi-threading over a heterogeneous DSM. Their approach supports improved utilisation through load balancing, by allowing threads to migrate across the system. For many sequential applications adding multi-threaded support requires significant re-engineering effort.

Another approach is to modify the middleware to support communication between processors on the same node as a special case, taking advantage of the intra-node hardware cache conherence and synchronisation mechanisms [14, 39, 66, 81, 82]. This approach has the benefit of avoiding changes to the application while making use of the hardware inter-connects.

Besides Ethernet connectivity, specialised interconnects are available which offer remote direct memory access (RDMA) between nodes. For example, MPI-2, which was introduced in Section 2.1.2, supports one-sided message-passing by using a uni-directional shared-memory window for shared communication and is capable of using specialised interconnects that offer RDMA.

While many implementations of one-sided MPI are built using two-sided MPI send/receive operations, Jiang *et al.* [50] provide an efficient, high-performance implementation by utilising the RDMA operations of Infiniband [71]. Other one-sided MPI implementations exist that are optimised for particular machines [8] and interconnects [93]. MPI-2 [68] on the NEC SX-5 [88] supports one-sided MPI operations over a global shared memory, similar to the method used for one-sided MPI presented in this thesis. The one-sided MPI operation is performed directly in shared memory via a memory copy, allowing the operation to utilise the full memory bandwidth of the available system. This technique is also applied successfully to two-sided MPI.

There are also several software libraries that implement optimised methods for common communication operations. The Nexus [31] communication library supports multiple communication mechanisms, allowing a distributed application to utilise the best available method for communicating with other nodes in heterogeneous computing environments. The Madeleine III [9] communication library provides multi-protocol support suitable for multi-cluster environments. Using these communication libraries allows MPI implementations to efficiently adapt communication performed by a distributed application to better suit the environment between communicating nodes.

In multi-cluster environments, applications designed for single clusters experience performance problems due to the higher latency and lower bandwidth of the communication links between clusters. Applications that take into account the underlying network topology have been shown to improve performance and scalability [11]. For example, increasing locality by placing data-sharing nodes together in the same cluster helps avoid communication over the lower performance communication links, thereby improving performance and increasing concurrency. In situations where inter-cluster communication cannot be avoided, communication hierarchies help to hide the latency of using these links [54].

Arantes et al. [6] extends LRC for multi-cluster environments by grouping nodes into clusters whereby intra-cluster communication is preferred. This exploits cluster locality by caching accessed data locally on each cluster. However, this system is targeted at running compute-intensive applications with a small number of nodes. LRC requires all processors to keep all updates in memory in case they are needed by another processor. Furthermore, no attempt is made to reduce the need to use costly vector time-stamps that require information on the progress of each processor in the system. This system works well for small-scale compute-intensive DSM applications, however it is unsuitable for more general use due to the overheads incurred and the unscalable nature of such a system.

2.2.3 Adaptive protocols and middleware

There are several adaptive DSM consistency protocols [3, 22, 28, 70] that attempt to detect and alter their protocol behaviour based on application behaviour. This includes how and when updates are propagated, use of invalidates, granularity of updates and specialised update techniques.

This type of adaption is limited to aspects of the protocol behaviour that can be reasonably changed. For example, release consistency can be implemented by lazy release consistency (LRC) [55] which is a home-less protocol. When using home-less protocols, unlike home-based protocols, data updates to a page do not get propagated to a single node (called a *home*), requiring updates to be requested from each node with recent changes. It is not straightforward to adapt this protocol to a home-based protocol such as home-based LRC (HLRC) [45] which may be a better match for some applications. For these problems, a better solution is to select a more suitable consistency protocol and implementation.

Adaptation by selection of more suitable consistency protocols is provided by Middleware such as DSM-PM2 [4]. DSM-PM2 supports selection of consistency protocol on a per-region basis for data and code in a distributed application. However, while this flexibility provides greater adaptation to an application's consistency requirements it does not support the utilisation of the most efficient consistency protocol for the underlying execution environment.

Another important aspect of adaption is support for heterogeneous environments where nodes may provide different architectures. Early work [62,95] showed that building a heterogeneous DSM was possible. In this work, a modified multi-reader, single-writer protocol was used for consistency between nodes. This thesis does not address heterogeneity of nodes.

MPICH-2 [7] is an open source implementation of the MPI [67] specification. It provides limited support for adaption to run-time environment through selection of communication protocol. For example, MPICH-2 includes support for communicating point-to-point over TCP/IP, shared memory, Infiniband [71] and several other interconnects [36].

The ability to optimise a distributed application by communication protocol selection does not, however, fully adapt an application to running across more than one different environment at once.

To better support these environments, many MPI implementations including MPICH-2 support the concurrent use of different communication channel implementations. For example, MPICH-2 can send intra-machine messages using a shared memory channel between processors of the same node, while communication between nodes occurs using TCP/IP.

Another existing solution is to use metacomputing frameworks that allow vendor-specific implementations within each machine and communication over TCP between machines. For example, Interoperable MPI (IMPI) [34] allows a distributed application to select not only the best communication protocol but also the best vendor implementation to use. Unfortunately current metacomputing frameworks lack the ability to flexibly support a wider range of interconnects that may exist between clusters or machines. For example, resorting to TCP/IP communication between machines is not suitable when the ccNUMA machines are connected together via Infiniband or other high performance interconnects.

MPICH-Madeleine [10] addresses the inefficiencies of poor heterogeneous point-to-point communication that traditionally required any intra-cluster communication to sub-optimally use TCP. It supports virtualised communication channels that support the communication across the available interconnects between two nodes. This allows two-sided MPI applications to perform well across heterogeneous environments. However, one-sided MPI communication and multi-cast communication will still occur using point-to-point communication, thereby not fully utilising the underlying coherency and communication mechanisms available, for example, in ccNUMA systems.

2.3 Wide-area distributed systems

Maintaining data coherency over a wide-area without impacting on performance remains a challenge for the distributed computing community. Wide-area environments contain a broad range of systems, resources, interconnects and network topologies that create unique data-sharing environments much more complex than that of well known environments such as clusters. To address these challenges, many distributed applications implement their own data management mechanisms or utilise middleware that provides appropriate abstractions for wide-area data access. Distributed data stores attempt to provide a common platform for the efficient distribution of data. There are several challenging problems related to the tradeoffs of scalability, data consistency, efficiency and performance.

InterWeave [23] allows the programmer to map shared segments into programs running on multiple heterogeneous machines. It provides three levels of sharing to deal with different interconnect topologies: hardware-coherent multiprocessors, tightly-coupled clusters using software-based lazy release consistency and more coarsely grained version-based consistency for distributing shared segments. This partly addresses one of the problems tackled in this thesis: taking advantage of local interconnects by using multiple consistency methods. However, it is not a general solution since it is specifically tailored towards particular environments and makes restrictive assumptions about consistency in data sharing.

Khazana [18] is a peer-to-peer data service that provides a common infrastructure for managing distributed shared data, allowing applications the flexibility of trading off consistency for availability and performance. It provides these using aggressive replication and customisable consistency management. Khazana presents applications with a persistent, globally-shared 128-bit address space. Applications access this address space via explicit accesses or by mapping parts of the global address space into their virtual memory space.

Unfortunately, Khazana lacks the ability to allow the use of a protocol designed for optimal execution within a particular environment or to meet specific application requirements. Furthermore, it is not designed to allow multiple environments that each use their own optimal internal protocol to interact consistently. Hence, the ability of Khazana to offer a scalable data store is limited by the choice of consistency protocol selected by the application. It does not take advantage of underlying architectural features that may be present in some environments in which data sharing occurs.

Besides Khazana and InterWeave, there is a wide range of distributed data stores that focus on data access scalability and delivery such as OceanStore [58] and Pangaea [80]. However, these systems are not suitable for maintaining consistent data that is frequently updated during a distributed computation.

One important feature of a DSM model is that shared memory provides location-independent data access. Hardware and software shared-memory implementations are required to keep a directory of where the current data elements are located so that they can be retrieved by any processor (or node). For a wide-area DSM, or a DSM with many nodes, this poses a challenge.

To address this challenge, JuxMem [5] explores the idea of a hybrid DSM and peer-to-peer data sharing service. The DSM component provides location transparency and data consistency, while the peer-to-peer component provides data location and persistence in a Grid environment. JuxMem provides a hierarchical architecture by grouping nodes into clusters which are networked together. A network overlay is provided on top of the physical network and is used to manage and locate shared data. For managing consistency of shared data, JuxMem implements an entry-consistency memory model. This approach makes it suitable to small numbers of nodes accessing shared data concurrently, where the nodes may be part of a large network spread out over a wide area. However, applications are restricted to the limitations of the entry consistency model, which enforces a particular programming paradigm on the programmer.

Legion [38] is an object-based system that aims to provide a single, coherent globally-scalable virtual machine. Legion provides a single, persistent name space for all files and data. Legion relies on object-orientated techniques at the core of the system. While this provides flexibility, it restricts the programming models that can be used in the system. Objects communicate with one another via object method invocations rather than using shared regions of address space or other techniques.

Teamster [21] is a hybrid thread architecture that provides a transparent DSM system. It has been extended to Grid systems in Teamster-G [63,64] by relying on a consistency protocol that implements eager-update page-based consistency. This protocol is suitable for wide-area access where latency is not a critical function of the program. Unlike the model presented in this thesis, it is not able to adapt to the underlying environments utilised by the distributed application.

Wide-area Grid support in Teamster-G is implemented using the Globus [32] meta-computing toolkit for Grid-scale distributed applications. Globus provides a set of low-level mechanisms such as communication, authentication, network information and data access. It is also used to implement higher-level services and programming environments such as MPI. Globus aims to allow applications to configure themselves accordingly to the underlying execution environment and available Grid resources.

MPICH-G2 [53] and PACX-MPI [33] are examples of two MPI implementations designed for metacomputing over wide area and heterogeneous networks. They address issues of wide-area and heterogeneous computing by optimising point-to-point communication by using protocol selection techniques, particularly within environments such as clusters. However, between clusters, communication is generally limited to TCP/IP which may be an inefficient use of the available interconnects between clusters.

2.4 Interoperable middleware

Interoperable middleware provides applications with the ability to use one or more different middleware instances or implementations at the same time.

As introduced earlier, interoperable MPI (IMPI) [34], is a specification for allowing MPI communication between different MPI implementations. This is useful for running a distributed application across heterogeneous environments such as multi-clusters. Vendor-specific MPI implementations that are performance-tuned for a particular environment can be run on each cluster, thereby providing a technique for adaption. Communication between MPI implementations occurs via an IMPI server over TCP/IP. One MPI implementation called LAM [16] implements a MPI personality that supports the IMPI interface.

Likewise, Jimnez et al. [51] formally describe an approach for the interconnection of different distributed shared memory models such that they behave as a single DSM. This approach should allow for interoperable DSMs, however their focus is the formalisation of an interconnection protocol with restrictions and assumptions about the memory models used.

2.5 Other approaches

Tempest [77] is a communication interface that defines a set of mechanisms for implementing shared-memory policies. The tempest mechanisms include low-overhead messaging, bulk data transfer and fine-grained memory management. Using these mechanisms, a program can use various programming models including both message passing and shared memory models.

This thesis realises that restricting protocol implementations to specific abstractions restricts the ability of a model to adapt to a diverse range of environments and their approaches to communicating and synchronising data. Hence, we do not directly define mechanisms for protocol implementation, thereby retaining the advantage that specialised protocol implementations provide to applications running on some environments.

2.6 Multiple model applications

An approach to developing multiple-model applications provides a mechanism for developing a computation that uses the best programming model for each part of an algorithm. This approach is also known as hybrid parallelisation. For example, another optimisation technique, suited to clustered SMPs, requires modifying an algorithm so that it uses both a shared memory and a message-passing programming model [37]. Shared-memory programming is used for data shared between processes on an SMP node, while message passing is used for sharing data between processes on separate SMP nodes. Other approaches [17,40,52,84] explore similar techniques with MPI and OpenMP [26].

This approach can be further extended to support visualisation of program runtime using visualisation toolkits [57] and provide a mechanism for code reuse of common algorithms implemented using particular programming models.

The ability to map from one programming model to another is not always easy or obvious. The model presented in this thesis provides a mechanism for supporting this approach, reducing the need to re-engineer code.

CHAPTER 3

THE VIEW MODEL

In this chapter we present our model that provides an approach to address the problem areas outlined in Section 1.2.

The view model is a hybrid, cost-conscious approach to remote data sharing. It provides the necessary mechanisms and interconnects to improve the flexibility and functionality of data sharing without defining new programming models or protocols.

3.1 Conceptual model

The view model provides a shared data space abstraction based on the concept of *views*. A view represents a region of a shared data space. Examples of such shared data spaces include tuple spaces, shared memory, and name spaces such as those found in Unix file-systems.

Shared data spaces provide mechanisms to store, retrieve, and manipulate data elements. These mechanisms and the structure of a data element are data space dependent. For example, in a tuple space, data exists in the form of tuples which are referenced by a key. In a shared-memory address-space, data is represented as bytes referenced by a unique address.

Besides the shared data space, a view also specifies the data-sharing behaviour of that space. A view's data-sharing behaviour determines how a view interacts with its environment, how it reacts to any external interaction, and how it manages shared data. More specifically, this includes behavioural characteristics such as replication of data, consistency of data, the timeliness of interactions, and the format and structure of interactions.

The goal of the behaviour specification is to provide an outline of what the application expects and what the run-time system is required to provide. For example, the behavioural requirements for a release-consistency programming model could be defined as requiring consistent data for a data sharer using release semantics, eager propagation of data updates and a data-access granularity of a word via a shared data space. This level of specification gives the run-time system some flexibility in how it meets the behavioural requirements of an application.

A view's data-sharing behaviour specification partially determines the data sharing protocols used to implement a view. Data-sharing protocols address the data consistency, communication, and storage aspects of the behaviour specification. Hence, for a view implementation to accurately model a behaviour specification requires, among other traits, the selection of one or more data sharing and communication protocols. For example, a distributed system that specifies a shared-memory model of release consistency would need to provide a suitable release consistency protocol and a communication protocol suitable for communication over a network.

The selection of a data consistency protocol depends on the programming model used by the distributed application, and the type of data sharing the application is likely to use. The selection of a communication protocol generally depends on the runtime environment. For example, for a cluster with an Ethernet interconnect, a TCP/IP communication protocol would be appropriate.

Figure 3.1 illustrates a simplified behaviour specification. As shown, a specification of a release-consistency memory model maps to a view which provides a lazy release-consistency (LRC) protocol, along with TCP/IP communication. This choice is suitable for managing release consistency in software, for a cluster of nodes.

A system may provide several suitable data-sharing and communication protocols that match a behaviour specification. This allows the system to select the protocols required to model a behaviour specification based on environmental parameters and constraints. We call this *protocol selection* and discuss it further in Section 3.2.2.

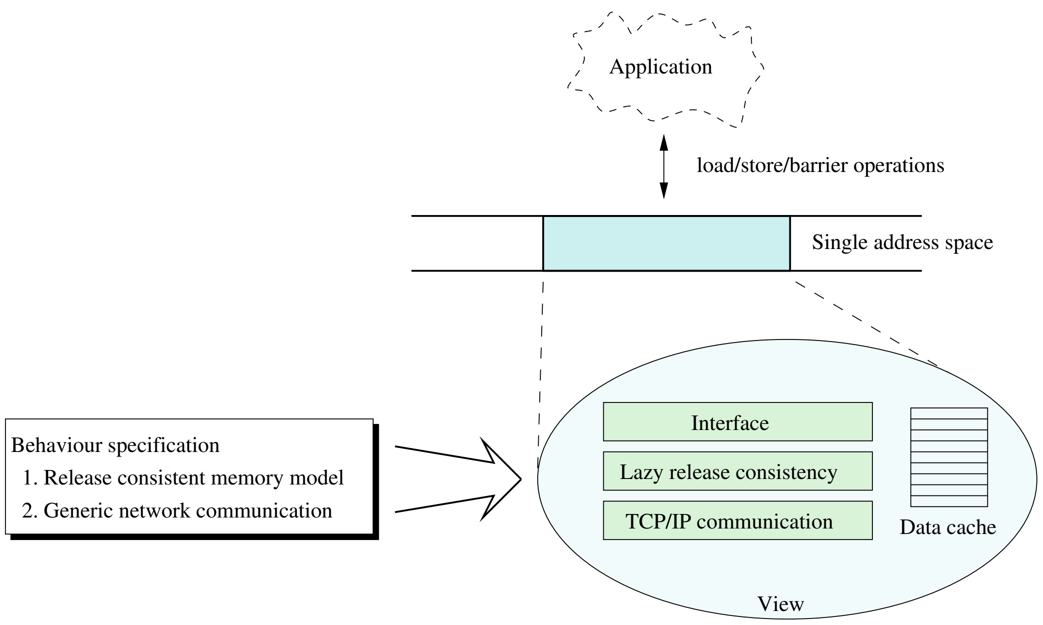


Figure 3.1: A view's behaviour specification, along with requirements of the run-time environment, determines the components used to implement a view.

In our model, a distributed application accesses and modifies shared data through a view. All entities that communicate with a view are known as *view clients*. View clients represent the producers and consumers of data, such as the threads of a distributed application that uses a threaded programming model.

Conceptually each view client is associated with a single view. A thread of a distributed application that utilises several views concurrently is represented by a view client for each view of which it is a member. For example, Figure 3.2 shows a distributed fast-fourier-transform (FFT) application running across two nodes. On each node, the application has access to two separate views. Each view corresponds to data sets with different data sharing semantics. Access to each view is achieved via view clients (VCs). Each VC interfaces to a specific view.

View clients are independent of the view details. A view's behaviour specification does, however, place restrictions on the interactions between the view and its view clients. This means that a view client must always be compatible with a view's behaviour specification. For example, a view that has a specified behaviour of release consistency is not normally suitable for a view client that provides an

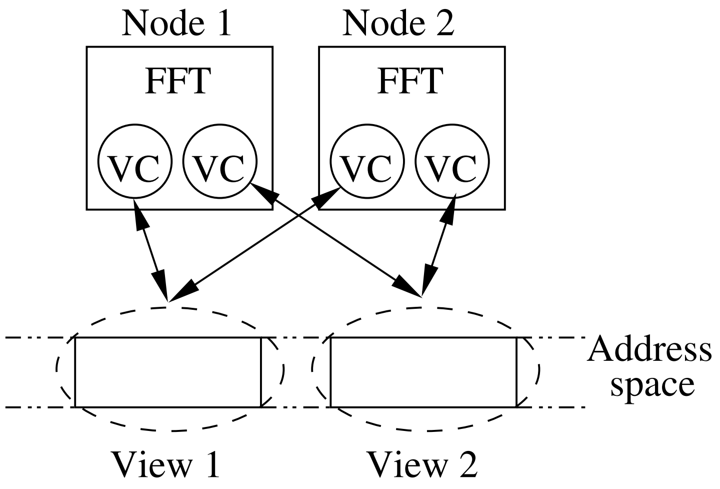


Figure 3.2: The FFT application running across two nodes, accesses View 1 and View 2 via their own view clients (VC).

implementation of two-sided MPI.

As illustrated by Figure 3.3, a single view with several view clients provides a conceptual representation of a traditional middleware. The goal of the view model, however, is to support a variety of relationships between views. Relationships between different views can be formed by sharing regions of data spaces or through mappings from one data space to another. There are three ways of achieving this: non-overlapping views, overlapping views and mapped views.

3.1.1 Non-overlapping views

Non-overlapping views provide a mechanism that allows different regions of shared data space to use different protocols or even programming models for data sharing. One of many uses of non-overlapping views is to allow applications, which may be familiar with their data access patterns, to optimise for their sharing patterns. For example, immutable objects such as input data sets can be shared with a protocol optimised for data delivery and dissemination, while computation occurs using another data region with a more suitable consistency protocol.

Several views may be present within a single data space, with each view representing a separate region of that space. This is illustrated in Figure 3.4 where the

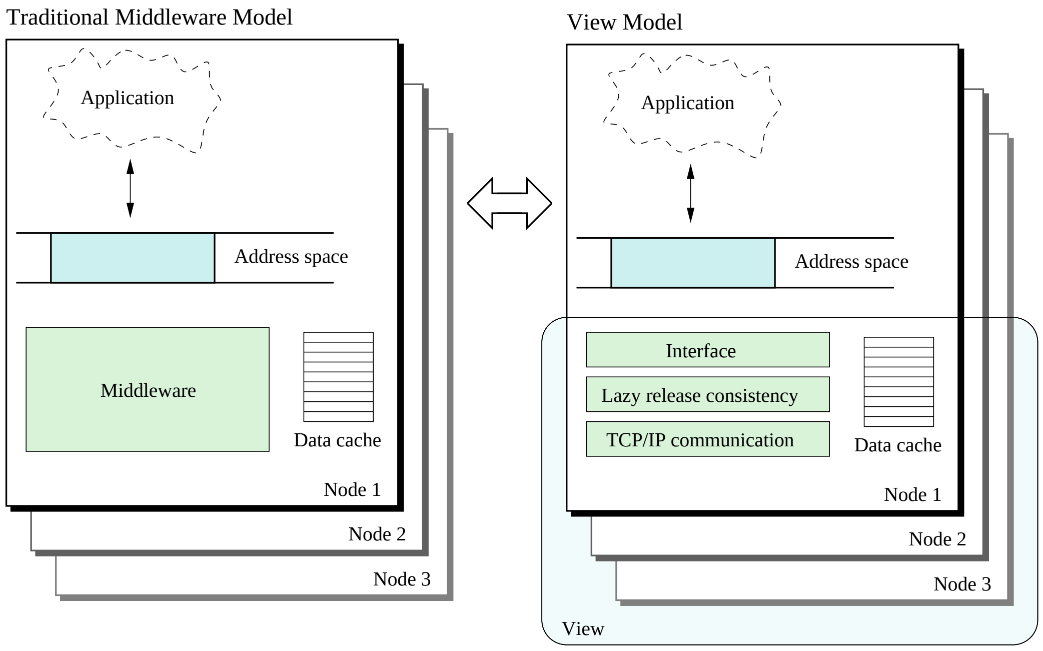


Figure 3.3: An application with a single view for data sharing across all nodes is equivalent to a traditional middleware.

shared data space is a traditional address space. In this example, view 1 and view 2 represent separate regions of the address space, possibly providing different data sharing behaviour within each region. View 1 has two view clients named C1 and C2 which are able to access data in the region covered by view 1. Likewise, view 2's view clients are named C3 and C4 and access data in the region provided by view 2.

Clients within the same view see updates to shared data whenever the view's specified data-sharing protocol makes the updates available. That is, like all data-sharing protocols, the protocol determines when updates are visible to the participants of data sharing. Hence, an update to data made by client C1 will be available to client C2 only when view 1's specified data-sharing protocol makes it visible to its clients.

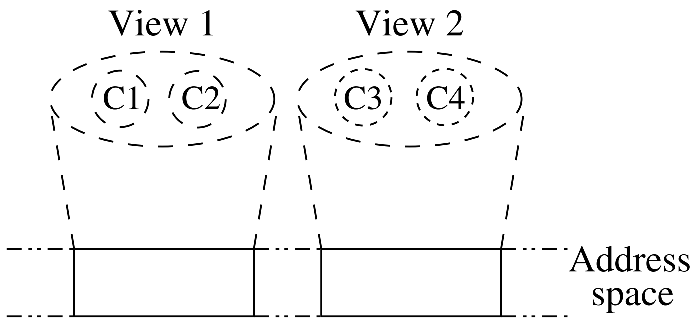


Figure 3.4: Non-overlapping views: two views representing separate regions of a traditional address space. Each view may provide different data-sharing behaviour for the region they represent.

3.1.2 Overlapping views

Overlapping views provides a mechanism whereby two or more views overlap so that they are able to represent the same region of data, while providing different data-sharing semantics to their view clients. Each view may use a different data-sharing protocol or vary some other parameter that makes it better suited to the needs of its view clients.

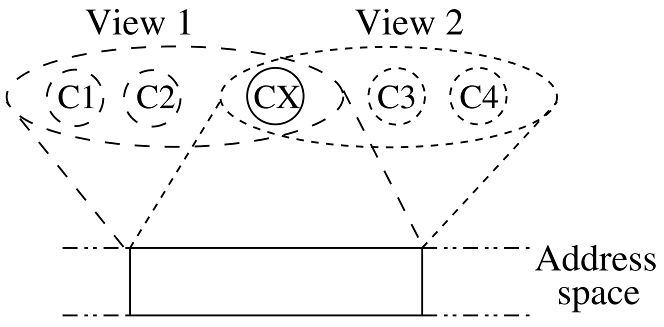


Figure 3.5: Overlapping views: used to represent the same data region, but with different sharing semantics.

Figure 3.5 illustrates the data sharing interactions between two overlapping views. When two views overlap, the effect of a modification in one view on the other view is a result of the interaction between each view's corresponding data-sharing protocols. This interaction is best explained through the introduction of a conceptual view client (client CX, in Figure 3.5). This view client is a client of both views. When client C1 performs a modification through view 1, client CX will be informed of the change according to view 1's data sharing behaviour. Once client CX is informed of a modification from view 1, it performs that modification in view 2. The other clients of view 2 (C3 and C4) will be informed of the change according to view 2's data sharing behaviour.

Each view's data sharing behaviour also determines how and when it is informed of changes. For example, a view that cannot receive notications without polling will have different behavioral semantics.

3.1.3 Mapped views

In Figure 3.5 the overlapping views represent regions of the same data space. It is also possible for views to represent the same data in different data spaces through the use of *mapped views*.

Mapped views are an extension of overlapping views whereby they share data by converting it using a mapping function. The mapping function is implemented by a *view mapping client*. This mapping function may provide translation of data representation from one data space to another, and may interpret interactions between views in a manner that suits the mapped views. The view clients in each view are able to access the same data elements, but with a different data space representation and different view behaviour specifications.

The inherent differences between some data space representations make it difficult and unnatural to provide a generic solution to mapping a data element in one data space to another. Hence, a specialised mapping view client must be implemented to suit the specific sharing requirements of distributed applications that wish to share or interact using different data spaces.

For example, Figure 3.6 shows a conventional data address space and a tuple space that interact with each other via a mapping from one data space to the other. Client CM implements a mapping function, which maps the tuple space representation to a data address space representation. The mapping function uses the data

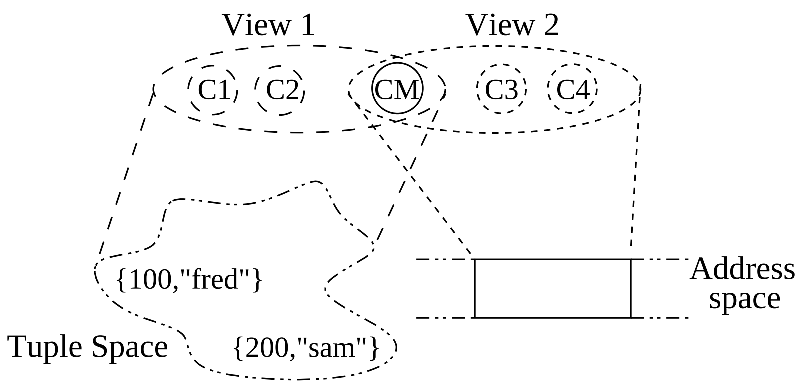


Figure 3.6: Mapped views: a mapping function is used to translate the data and operations from one view into the other.

address as the key to a tuple query. Thus, when client C3, in view 2 requests a data element at address *100* within the conventional data address space, client CM receives this request and translates the request address to a corresponding argument of a tuple query for key *100*.

This key is used by client CM to generate a tuple request into view 1. Once client CM has received a reply to its tuple request, it performs the modification on view 2 by replying to the original request for address *100* from client C3. Client C3 is then able to continue.

From the point-of-view of client C3, the data element appears to have been read directly from the conventional address space. The mapping operation is entirely transparent.

In this example it appears that view 2 is not storing data, and instead acts as a proxy for all data transactions, through client CM. This interaction depends on the view behaviour specification. More complicated views could cache the data locally, and only issue requests via client CM when the views' consistency requirements required it to do so.

3.2 Properties of the model

The conceptual view model presented in Section 3.1 has a number of inherent properties that improve the flexibility and functionality of data sharing. In this section we discuss these properties, but first begin by presenting the layered abstraction from which these properties are derived.

3.2.1 Layered abstraction

Traditionally, distributed applications are written on top of a middleware. The middleware provides a software implementation which manages data communication and consistency of the application running across several nodes of a network. Most middleware acts as a single layer of abstraction which hides the details of the underlying implementation from the application, while providing the application with a clearly defined programming interface.

This separation provides flexibility by allowing a single middleware implementation to be used as a platform for many different distributed applications. Furthermore, this separation allows the middleware to be altered without requiring changes to the application. For example, the communication protocol used within the middleware can be changed so that the application better supports a desired run-time environment. This increases the portability of distributed applications.

The view model, like other middleware, separates applications from the middleware implementation via a programming model API. Furthermore, the view model defines four different layers of abstraction and the interfaces between them. These abstractions define the mechanisms and structure of our conceptual model implementation presented in Section 3.1. For a single application instance, these abstractions of the view model are illustrated by Figure 3.7. At the top level in this figure is an instance of a distributed application. The application is written according to a particular (view independent) programming model API, such as OpenMP [26].

The next level provides an implementation of this API, which corresponds to the system boundary presented by a middleware implementation. This is known as the *programming model client* level because it implements a programming model API as a specialised view client. The functionality provided here is itself implemented in terms of an abstraction layer known as the *view interface*, which provides a programming-model-independent way for the view client to invoke the

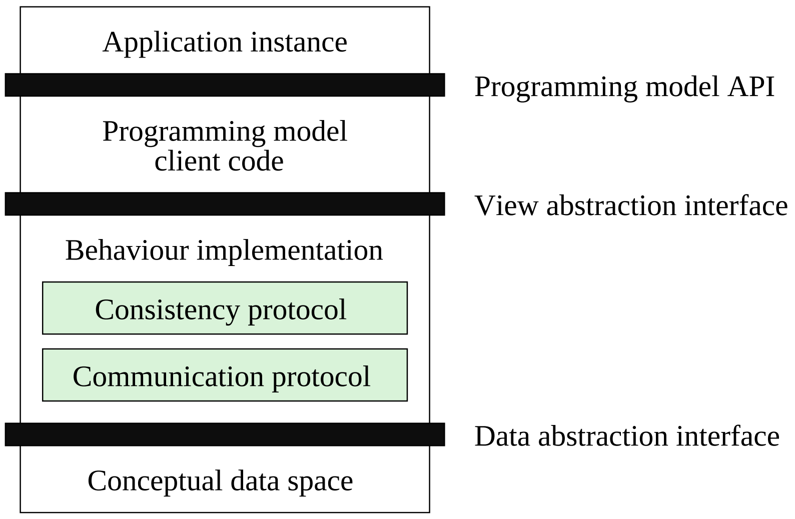


Figure 3.7: Conceptual separation of middleware of the view architecture.

view-specific behaviour implemented at the next level. In essence, it acts as a glue layer and does not normally implement the actual programming model behaviour. Instead, it provides a mapping from the programming model API to the view interface. The view interface will be described in greater detail in Section 4.1.1.

Below the view interface, the behaviour implementation provides the view's data sharing behaviour. This level includes implementations of consistency protocols, communication protocols and other behaviour related functionality discussed in Section 3.1.

The final level of abstraction represents the data space used by the behaviour implementation. The data space has a defined data element structure and an interface for storing, retrieving, and modifying shared data. Conceptually, it represents the data space visible to the application which is manipulated by the abstractions between the application and the data space.

The overhead of these abstractions is implementation dependant. An implementation that glues abstraction layers together via function calls incurs an insignificant amount of overhead. That is, an implementation can construct itself similar to a traditional monolithic implementation where all functional components are linked together into a single binary. The implementation presented in this thesis separates they layering into two parts: the application instance and the view server. This structure incurs the communication overhead of local shared memory and context switch latencies of the run-time system. These overheads are insignificant compared to a page data transfer between nodes.

These layered abstractions and interfaces provide the view model with the flexibility necessary to provide several interesting properties. We present these below.

3.2.2 Protocol selection

Perhaps the most important property of the view model is protocol selection. Protocol selection is the ability to change a part (or all) of a view's behaviour implementation in order to better suit an application's underlying runtime environment or data-sharing characteristics. The selection of a different protocol changes the view behaviour implementation without requiring changes to any interface or other layers. This is akin to replacing the implementation of a traditional middleware system without making changes to the application. However, the view model approach is more precise in that it allows the replacement of a single part of a programming model implementation, such as the communication protocol, without replacing other components. In this case, the view model could be considered an example of reflective middleware [56]. The mechanisms and approach to achieve this using the view model are explained in Chapter 4.

The following example illustrates the benefits of protocol selection. Prior to run-time in a Grid environment, a distributed application does not always know what kinds of network interconnects will be available. The application must rely on protocols that are known to work across the whole execution environment. In this case it is safest to assume a generic TCP/IP communication interface. However, when the run-time environment includes specialised interconnects, such as shared memory or Infiniband, a generic protocol will fail to make full use of these interconnects. In such as case, allowing the application to change protocols can lead to marked performance and efficiency improvements.

Furthermore, on multi-processor machines that support coherent shared memory the application should use the available hardware support rather than relying on conventional software-based protocols. As such, protocol selection mechanisms provide a method for optimising and adapting a distributed application to its runtime environment and resources.

Using protocol selection to change the protocols used for an entire application is an ability provided by many adaptive middleware systems. Combining protocol selection with non-overlapping and overlapping views allows this approach to be used to further adapt the application and its data-sharing characteristics.

Protocol selection with non-overlapping views

As mentioned in Section 3.1.1, non-overlapping views are useful for representing shared data regions with different sharing semantics. This is achieved by using protocol selection to choose the most appropriate protocols for each shared data region.

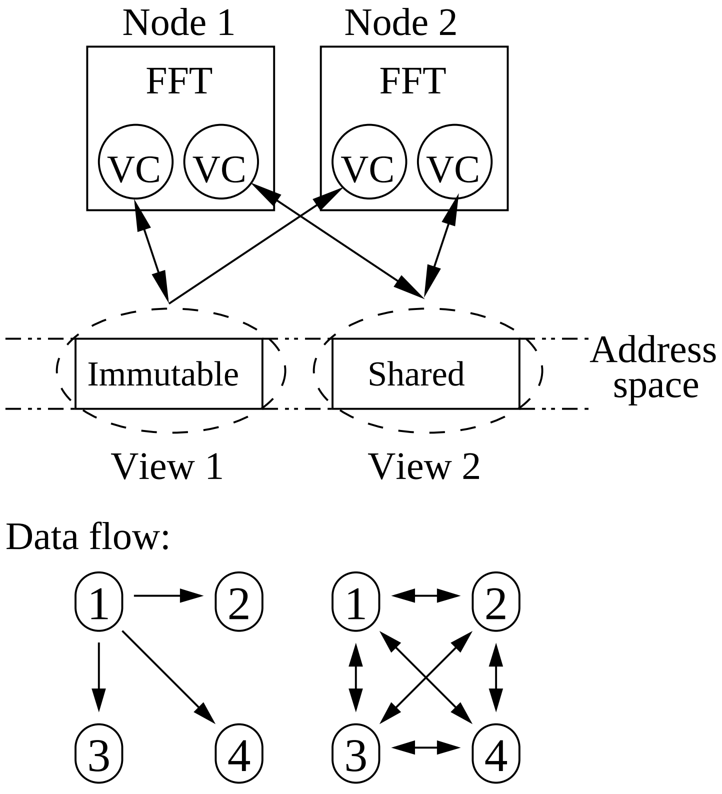


Figure 3.8: An application using non-overlapping views for different data access patterns. At the bottom of this figure we see how data propagates for each view.

Figure 3.8 shows a fast-fourier transform (FFT) distributed application running across two nodes. Like many applications, there are two separate data sharing phases. Firstly, the source data needs to be delivered to the nodes that require it for computation. The data is subsequently used to perform a shared data computation across the participating nodes. In this case, the immutable read-only data can be distributed to nodes by view 1 using a protocol designed for bulk-data transfer while data sharing during computation uses a release consistent protocol in view 2.

For this application, the use of two different data sharing protocols is not necessary in order to run the application. For example, the release consistent protocol used by view 2 could be used to transfer the immutable data sets and for compuation. However, it benefits the application to separate these two run-time phases as it will permit more efficient resource usage. Specifically, the consistency metastate normally used for release consisteny is unnecessary for the immutable data sharing region.

Furthermore, the immutable data region can be distributed by taking advantage of data multicast techniques which use less bandwidth and generally result in higher performance [76].

Protocol selection with overlapping views

In heterogeneous environments such as multi-clusters and Grids, it is important to fully utilise the available underlying resources and communication capabilities. Overlapping views, combined with protocol selection, can be used in these environments to select protocols that best suit each distinguishable locality domains, including each cluster and nodes such as ccNUMA nodes.

To illustrate, consider the scenario shown in Figure 3.9 where several clusters and nodes are spread out over a wide-area network. In this scenario it is desirable to utilise the available underlying resources of each cluster and to favour local area communication over wide-area communication. This can be achieved using the view model by encapsulating each locality domain within a view, with each view using protocols best suited to its environment. For a distributed shared memory application running in this environment, a suitable view structure is shown in

Figure 3.10. In this figure, the Infiniband cluster utilises a view that provides a software-based protocol implementation of home-based lazy release consistency (HLRC) [46]. A view encapsulating the enterprise nodes selects HLRC over TCP/IP, and the ccNUMA view uses a protocol implementation that allows worker threads to communicate directly using hardware-supported shared memory. Access to shared data within the ccNUMA system results in external communication only when the data is determined to be inconsistent by the view behaviour implementation. Any subsequent access to the same data within the ccNUMA system will occur directly over shared memory, without any unnecessary software intervention.

Overlapping views form an interconnected topology. Each view is able to choose different behaviour implementations through protocol selection while being able to interact with each other through the conceptual view client used by overlapping views. This approach allows a distributed application to use different behaviour implementations concurrently.

The topology of interconnected views may be arbitrary allowing the formation of simple or complex interconnected views. However, in this thesis we restrict ourselves to *data-flow hierarchies* where data flows between views in a tree-like interconnect.

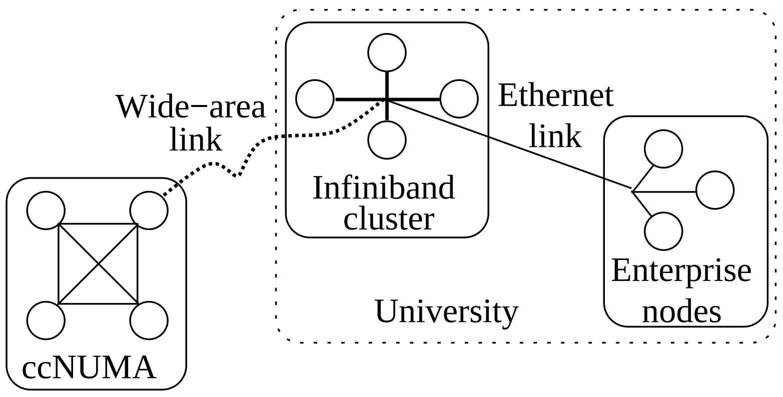


Figure 3.9: An example scenario where several systems and nodes are used for running a release-consistent program in a Grid.

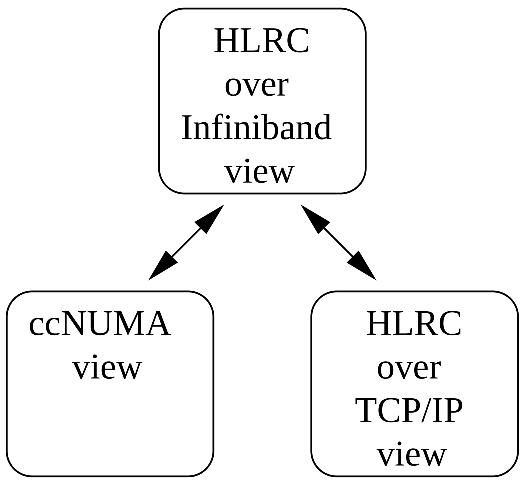


Figure 3.10: View structure and configuration for example scenario where several systems and nodes are used for a running release-consistent program in a Grid.

A topology of interconnected overlapping views provides the following properties:

Optimisation for locality As demonstrated by our previous example, views can be used to encapsulate locality domains and optimise data sharing within that domain via protocol selection.

Data communication amortisation between views Normally, communication between views of an unchanged data element occurs only once, regardless of the number of view clients that request the data element. Once a data element enters a view, it remains local to that view so long as it is considered coherent by the protocol implementation (if the intention is coherent data). Requests for the data element within the view are then resolved internally, without external communication. Likewise when a data element changes states within a view, such as when view clients modify the data element, the changes do not need to propagate outside that view until required to, based on the view's behaviour specification. This property can provide many applications with a significant reduction in communication, particularly when view boundaries are placed across contented network links.

Reduces the impact of the scalability of protocols Many complex protocols, particularly DSM protocols such as LRC, impose significant resource overhead that is proportional to the number of processors (or, in our model, view clients) sharing data. The use of multiple views in this scenario gives the appearance of fewer total processors to the protocol implementation. Using multiple views in this way may allow a protocol to be used over a larger number of real nodes.

Improved flexibility Views transform data and operations on data from one form to another. With this in mind, views can be used for purposes other than data delivery and data sharing. For example, a view designed to store data to a file system could form the basis for data storage of long lived data without the need for an explicit file system interface in the application. Using overlapping views, this approach would ensure long lived data was written back to disk when computation was no longer in progress. Other uses could include transforms on data such as data compression.

Optimised synchronisation token management Synchronisation tokens, discussed further in Section 4.1.1, are used to represent synchronisation primitives such as barriers and locks. Like data elements, synchronisation tokens are also transferred between views.

A view's behaviour specification provides details on what happens when a token enters a view, and how it should be managed. This allows a view's internal implementation of barriers to be provided using multi-cast and other similar approaches [89]. Furthermore, amortisation of barrier messages is possible if a view model implementation defines a barrier token leaving a view to mean all clients within the view have reached the barrier. For other synchronisation primitives, such as locks, other optimisations are possible. For example, a ccNUMA view protocol may choose to use hardware-supported synchronisation primitives in order to minimise synchronisation overheads.

3.2.3 Programming model independence

Another important property of the view model is programming model independence. The view model does not require any particular programming model to be used by view clients. Hence, techniques using protocol selection and overlapping views can be employed and reused regardless of the programming model used by the application.

For example, both of the DSM and one-sided MPI programming model experiments presented in Chapter 7 use the same optimisation techniques, including the use of overlapping views. This also extends to other models, such as traditional message passing. The layered abstraction of the view model also allows the implementation of communication protocols to be reused between different programming models.

Programming models and mapped views provide interoperability

The programming model independence property extends to programming-model interoperability. Programming-model interoperability allows construction and execution of distributed applications that use more than one programming model for data sharing. It also allows interoperability between distributed applications that use different programming models such as visualisation and computational applications.

Support for programming model interoperability can be provided through mapped views. This is achieved by using views with behavioural implementations of different programming models and connecting them as interacting views through a view mapping client.

For example, consider a program that is built up of algorithms written using different programming models. The selection of these algorithms may be at runtime in order to optimise the choice of algorithm dependent on the underlying environments. Sharing data between algorithms using the different programming models requires the use of view mappings. Hence, a fast-fourier transform (FFT) algorithm written to run on traditional shared memory may be used in an inner loop, while MPI is used to distribute the source and results data. Likewise, an external visualisation tool may wish to examine the data generated by the FFT algorithm at run-time by accessing the data through a mapped view.

CHAPTER 4

EIDOLON: A REALISATION OF THE VIEW MODEL

ei · do · lon (ī · dō' l ə n)
n. *pl.* **ei · do · lons** or **ei · do · la** (-l ə)

1. A phantom; an apparition.
2. An image of an ideal.

In the previous chapters we presented our view model and outlined the concepts and properties that are inherent in the model. In this chapter, we define an architecture for the model, which we call *Eidolon*. Eidolon realises the view model by specifying details required for implementation, including a number of constraints and design decisions that are outlined in this chapter. The Eidolon architecture is implemented for our experimental platform and is used for demonstrating and analysing the view model in Chapter 7.

4.1 Views

Interactions between views, their clients, and other views occurs through a view interface. In this section we outline how views are defined, how they communicate and the interfaces they use to communicate.

4.1.1 View interface

Section 3.2.1 introduced the view model's layered abstraction and introduced an important abstraction called the *view interface*. Communication between a view and its view clients occurs through the view interface. Likewise, communication between views can also occur via the view interface. That is, there is no distinction between views sharing data with each other and other clients. This feature allows views to be connected to other views in flexible configurations.

The implementation of the view interface provided by Eidolon defines a set of operations and a data abstraction used by the operations to address data elements.

Data addressing and representation

The view model names and addresses data elements using a unique identifier. When an operation is performed via the view interface, data elements are addressed by this unique identifier. Hence, while the data addressing approach is independent of the view model, it must be specified in order to allow views and their clients to communicate. For example, tuple space addressing could be used by a client and its views to refer to data elements.

At first sight, this seems to restrict views communicating directly with each other to use the same data addressing approach. However, as introduced earlier, the flexibility of mapped views allows for different data space representations through the use of a view mapping client. This mapping client will provide translation of an operation and its data addressing at the view interface.

Some programming models do not have the concept of a data space and do not refer to data elements directly. For example, message passing generally invokes simple data send and receive operations. In this case, data addressing is essentially irrelevant. Changes to data appear as a flow of updates and only the message sender and recipient can infer any meaning about the content of a message. In order to allow communication and interoperability between message passing and addressed-data models, a view mapping client can also be used. This specialised view client takes operations from one view and converts them into a usable representation for the other view.

Eidolon address space

In order to uniquely identify all data elements, Eidolon provides a single-address-space (SAS). Every data element is assigned and identified by an address in this SAS. This includes views and synchronisation tokens which are given a unique address within the SAS.

Eidolon's data addressing approach does not preclude the use of other data addressing approaches such as tuple space, nor does it prevent the use of message-passing models. However, for the purposes of evaluation, the SAS approach gives us the flexibility to explore improvements to DSM applications using the properties of views, while experimenting with the use of other programming models. A detailed examination of other data representations including tuple spaces and two-sided message passing models are beyond the scope of this thesis.

In Eidolon, mapped view clients provide the necessary mechanism to allow interoperability between different programming models and their data space representations. For message passing programming models, while not strictly necessary, the data address is used to refer to a communication end-point known to the application. This approach simplifies the implementation of any view mapping clients and allows us to experiment with a single view interface. For example, a DSM application writing to a specific data address causes an *update propagate* operation to be invoked on a mapped message-passing view. The view mapping client handles this request by treating the update propagate operation as a message sent to a specific worker thread. Other approaches are possible, with a suitable translation provided by a mapped view client.

View interface operations

The Eidolon architecture implements the view interface using a set of operations known as *view interface operations* (VIOs). These operations are sent as messages between view clients and their view pager. The view pager implements the view behaviour and is discussed further in Section 4.1.2. The VIOs are divided into three core categories shown in Table 4.1. First, *data coherence operations* specify actions to be performed on a region of data. Second, *synchronisation operations* control the flow of data and specify possible dependencies between data and nodes that modify the data. Third, high-level *view manipulation operations* allow for views to be created, manipulated and destroyed.

update request	requests updates for a given region
update propagate:	propagate updates for a given region
protection request:	request access to given region
protection propagate:	indication of new region access
acknowledge:	indicates success of an operation
token create:	create a synchronisation token
token request:	request a synchronisation token
token response:	receive a synchronisation token
view create:	create a new view
view select:	select a view for use
view unselect:	release a view

Table 4.1: Summary of view-interface operations.

All data coherence operations specify a single region of data within the SAS along with any other parameters required for the operation. These operations include protection-based read/write/execute permissions. The prototypes of a suitable interface are included below.

Update request indicates the desire of the caller to receive updated data for a specified region.

```
int update_request (view_t view, sas_t base, sas_t ↩
    end, int flags)
```

The specified region is indicated by the `base` and `end` variables. Any permissions set in the `flags` variable indicate the caller's desire for ongoing access without the need for future communication. For example, if the request includes `read` permissions, the caller is asking that it be able to read from the memory region without requiring any further `read` request operations until this access is revoked by the callee.

Update propagate indicates that the caller is sending a data update or reply to an `update request.`

```
int update_propagate (view_t view, sas_t base,  ↩
   sas_t end, int flags, void *data)
```

If permissions are included, they indicate the on-going access rights the recipient has to the data region.

This operation also passes a pointer to a buffer containing the changed memory region. Note, however, that our implementation avoids a buffer copy to the client address space by using direct-mapped zero-copy techniques when possible.

Protection request indicates a request to have specified on-going access to a region.

```
int protection_request (view_t view, sas_t base,  ↩
   sas_t end, int flags)
```

For some programming models and consistency protocols, this operation has no meaning and can be ignored. For example, message passing has no concept of memory protection.

Protection propagate indicates to the recipient that its permissions to a given region have changed.

```
int protection_propagate (view_t view, sas_t base, ↩
    sas_t end, int flags)
```

Acknowledge indicates the success of a previous request for an acknowledge.

The success is encoded within the `flags` variable of the response.

```
int acknowledge (view_t view, sas_t base, sas_t  ↩
   end, int flags)
```

The caller for all operations can request an acknowledgement, which is sent back as an `acknowledge` operation.

Out-of-band data is also supported with all operations. This may be used to convey protocol-specific state between a view and its clients. For example, view pagers that implement the same consistency protocol, such as LRC, may include vector time-stamps as an additional consistency hint with data transfers.

Synchronisation tokens are used by a view to represent barriers, locks and other synchronisation primitives.

Token create will create a new token when invoked on a view. The caller can indicate what the token will be used for by specifying a hint.

```
token_t token_create (view_t view, int type_hint)
```

Token request will request a known token from a view.

```
int token_request (view_t view, token_t token)
```

Token response delivers a token to a client.

```
token_t token_response (view_t view, token_t tokn)
```

All data and synchronisation operations invoked on a view or client are assumed to be processed in order. This allows a view to propagate data changes associated with a lock by first sending the data changes followed by the lock token.

For efficient processing, these operations can be batched together. Change sets used by protocols such as LRC can be supported in this fashion.

The creation and management of views is provided for via view manipulation operations. Views must be established before using any of the data or synchronisation operations, however, this can often be performed transparently to the application through either a library or by the run-time system. [1]

View create creates a view at a specified region in the SAS. The view may be tied to a parent view to establish overlapping or mapped views. This operation returns a view identifier which is used when invoking other VIOs.

[1]Approaches to using views are discussed in Chapter 8.

```
view_t view_create (int type, sas_t base, sas_t  ↩
   end, view_t parent)
```

View select is invoked by all clients requesting access to a view. This allows them to choose the views used to construct their data sharing regions.

```
int view_select (view_t view)
```

View unselect indicates that a view is no longer being used by a client.

```
int view_unselect (view_t view)
```

4.1.2 View clients and pagers

View clients provide the application with its desired programming model interface. The data sharing components of a programming model interface must be abstracted and implemented in terms of operations on one or more views using VIOs. In Chapter 6 we provide implementation details of how this is achieved for several programming models and DSM consistency protocols.

When view clients interact with a view, it is the *view pager* that receives and processes each VIO. The view pager provides an implementation of the view's behaviour specification. This includes most shared memory consistency protocols, however, unlike traditional DSM implementations, the communication protocol used by the consistency protocols is not a part of the view pager. Instead, communication protocols are provided by a separate view client.

To understand how communication occurs between view client and pager, Figure 4.1 shows the interactions between two address space (AS) clients that are running a DSM application across two separate nodes. The AS client behaves as a user-level pager for shared memory regions. It catches page faults within a view and maps data into the address space of the application based on instructions from the view pager.

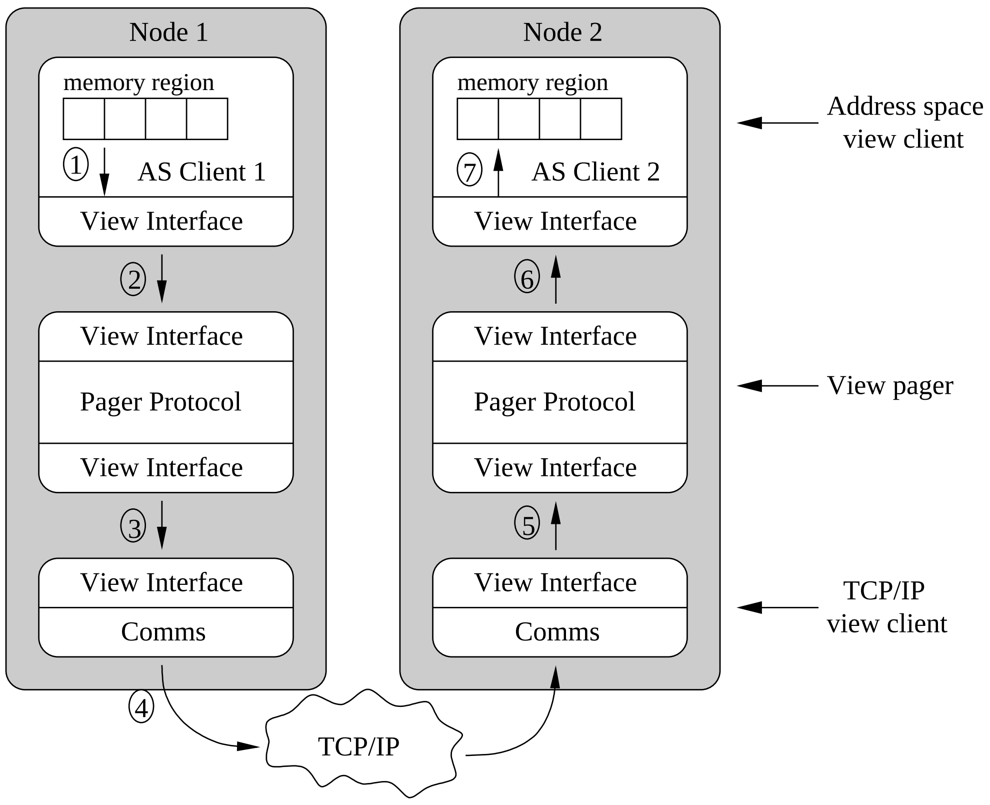


Figure 4.1: Two address-space clients interacting over TCP/IP within a single view.

In this example, a page-fault occurs in *AS client 1* for a data page that is currently only available to *AS client 2*. The first step converts the page-fault into an *update request* view-interface operation.

At step two, the *update request* operation is performed on the view pager leaving the pager to further handle the invocation. The invocation may be a remote procedure call (RPC) or a local function call depending on the configuration of the software.

The view pager then decides what to do based on its behaviour specification (which is implemented as a consistency protocol). In this example, at step three an *update request* operation is invoked on a communication view client. This view client sends the *update request* across the network using TCP/IP. Note that since the view pager is not aware that communication will occur over TCP/IP, it invokes the *update request* operation on the communication view client as it would for any other view client. There is a single view interface for all view clients. Hence at step four, a view client that implements TCP/IP communication converts the *update request* operation and any payload into a message suitable for sending over the network. In this case, there is no data payload other than the arguments required for an *update request* operation. The destination node, *node 2*, receives the message and converts it back into a *update request* operation. At step five, the view pager on *node 2* receives an *update request* operation that appears to be from *AS client 1*. The source of the request is passed as a parameter to each view-interface operation so that the destination knows where to direct any reply operations.

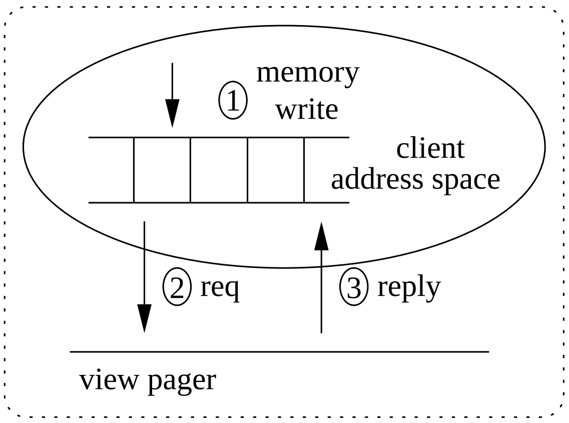
At step six, the view pager determines that it must invoke the operation on the destination client *AS client 2* as this client has the most up-to-date data. At step seven, *AS client 2* processes the operation and returns the data by invoking an *update propagate* operation on its view pager. Note that for local interactions between view pager and client, the data payload relies on zero-copy to avoid unnecessary copying.

When the *update propagate* arrives back at *node 1's* view pager, the pager writes this operation's payload into the view data space. Depending on the current state of the target (*AS client 1*), the view pager may invoke an explicit *update propagate* operation or rely on a cheaper synchronisation operation to notify the target view client. During this process, the original *update request* operation from *AS client 1* has been blocking. Once the view client receives its notification, the operation can continue.

The following example, illustrated in Figure 4.2, clarifies the interactions between a view pager and a client. In this example the view pager implements a distributed shared memory (DSM) programming model. Writing to memory (which, in this case maps directly to the SAS) causes the client to issue an `update_request` including a request for permissions for ongoing write access to that memory area. The view pager replies with an `update_propagate` granting that access. Any future writes to memory at that address occur without generating a new VIO until access is later revoked. The view pager implementation will behave like a typical DSM consistency protocol by mapping each view interface operation to a particular consistency protocol action.

Eidolon's view interface operations map nicely to page fault mechanisms of a typical DSM and thereby provide a clean layer of abstraction for implementing shared-memory address-space clients.

Other programming models may use and interpret VIOs differently. For example, the approach used in the previous example is not adequate for message-passing programming models, as they do not normally represent data using a location-independent unique address. Furthermore, these models do not have the concept of granting access to shared-data regions. Instead, the flow of VIOs between client and pager mimics the communication mechanisms expected by the application. We discuss this further in Chapter 6.



①	store instruction to memory at 0x1000
②	update_request(0x1000,WRITE)
③	update_propagate(0x1000,data)

Figure 4.2: A distributed application client writing to shared memory which triggers communication to a view pager via VIOs.

4.2 Interactions between views

In Section 3.1 we presented our conceptual model and illustrated how an interaction between views occurs via a conceptual view client. For overlapping views

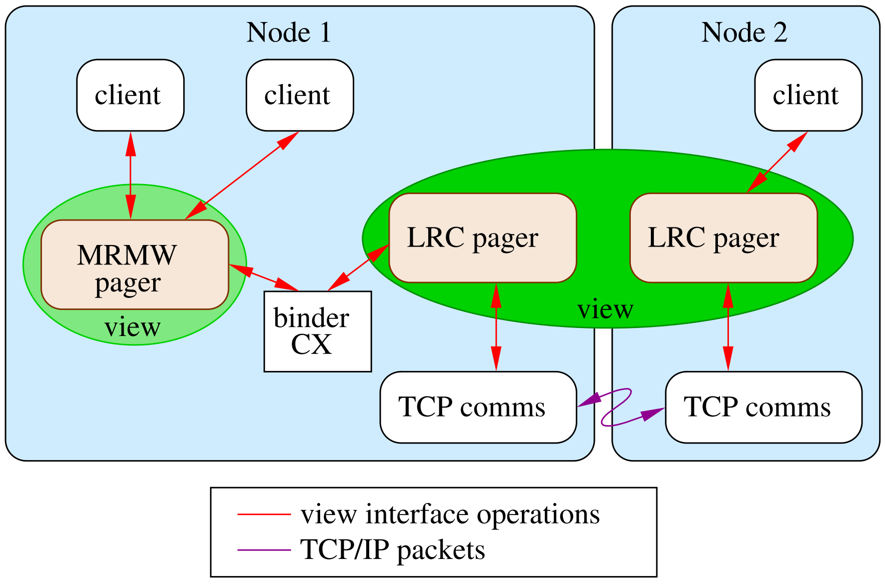


Figure 4.3: Overlapping views are connected via a view binder (CX) client. View interface operations are simply passed-through to the other side.

such a view client takes an incoming VIO from one view and forwards it to the other view. As Eidolon uses SAS addressing for all data operations we avoid having to change the representation of data when communicating between views. Hence, the view client for overlapping views only needs to provide simple proxy functionality and does not need to translate the data addressing method of one data space to that of the other.

Figure 4.3 illustrates a system using two overlapping views to share data. The view client connecting the two overlapping views is called a *view binder*. The view binder is an implementation of our conceptual client CX presented in earlier chapters. Its task is simply to proxy requests. In our implementation, the view binder alters only view identifiers which are associated with each view interface operation. A view binder is created transparently by Eidolon when the user calls `view_create` and provides a parent view as an argument.

Mapped views require a more complicated view client. For example, a distributed application that uses both two-sided MPI communication and shared memory, must translate data space operations via a specialised view mapping client. Both programming models have different methods of referring to data elements.

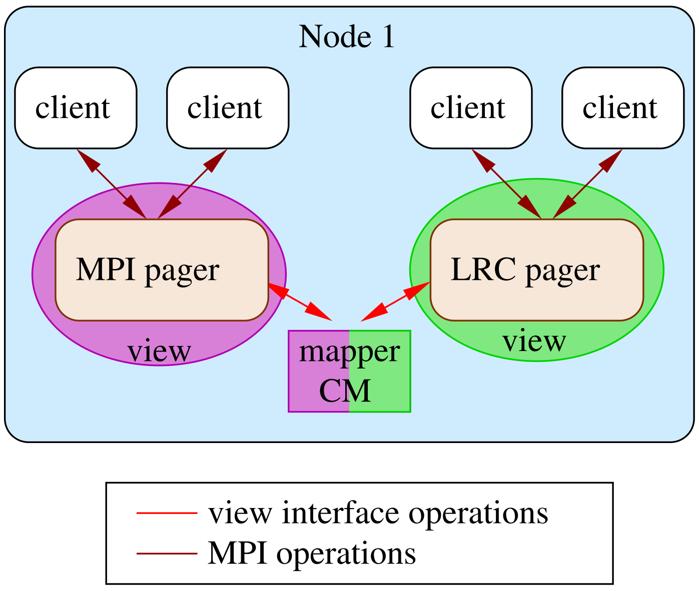


Figure 4.4: Mapped views are connected via a view mapping client (CM). This client performs translations on view interface operations as they go from one view to the other.

In this case, a view client that maps VIOs from one programming model to another needs to interpret how data elements are referred to in each model. In general, this is dependent on the application in question.

Figure 4.4 illustrates a system using a mapped view to translate data from one view into the other. Unlike overlapping views, which use a simple view binder client, mapped views use a more specialised client which translates operations. In this figure, this client is shown as *mapper* and represents our conceptual view mapping client. The views form a data-flow hierarchy with one view as the parent and the other as the child view. It does not normally matter which is the parent view, however in this example, the LRC view should be the parent assuming that data consistency is important. If the MRMW view was the parent view, it would not be required to send consistent data changes to its child view as its view behaviour would not require it.

In Eidolon, the approach for interconnecting both overlapping and mapped views is the same. They both use a common interface and operations which are provided by the view interface. These allow us to easily structure more complex view hierarchies in order to match a particular data sharing scenario. Importantly, when constructing these hierarchies, view pagers do not need to worry about the details of whether overlapping or mapped views are used. These details are independent from the views themselves, and they are able to treat all connecting views as regular view clients.

This allows us to create complex data sharing hierarchies. For example, consider a long-running distributed computation that performs computation using any available computing resources and allows the user to visualise progress. The run-time computing resources may be unknown and change over time, requiring changes to the data-sharing structure. Figure 4.5 shows a run-time environment and a conceptual view hierarchy for such a long-running computation.

In this figure, data sharing for computation is performed using release consistency with HLRC used as the primary consistency protocol for managing sharing of consistent data. The view hierarchy mimics that of the underlying interconnected machines. In this structure, an HLRC view is present for each locality domain, i.e., one for each cluster. The ccNUMA cluster establishes further MRMW views on each ccNUMA node. These are configured as child views connected to their parent HLRC view. Note, however, the success of this structure on performance is still very much dependent on the computation-to-communication ratio of the application, and the scalability limits inherent with the run-time environment.

The visualisation component within the visualisation view, receives occasional updates to data without any significant impact on the running computation. That is, it avoids being a direct client of the release-consistent views. Finally, the results are backed via a file backing store. This type of view handles all view data in a coarse grained manner and stores it for later retrieval. It becomes the parent view to which all updates flow, based on the sharing semantics of the views above.

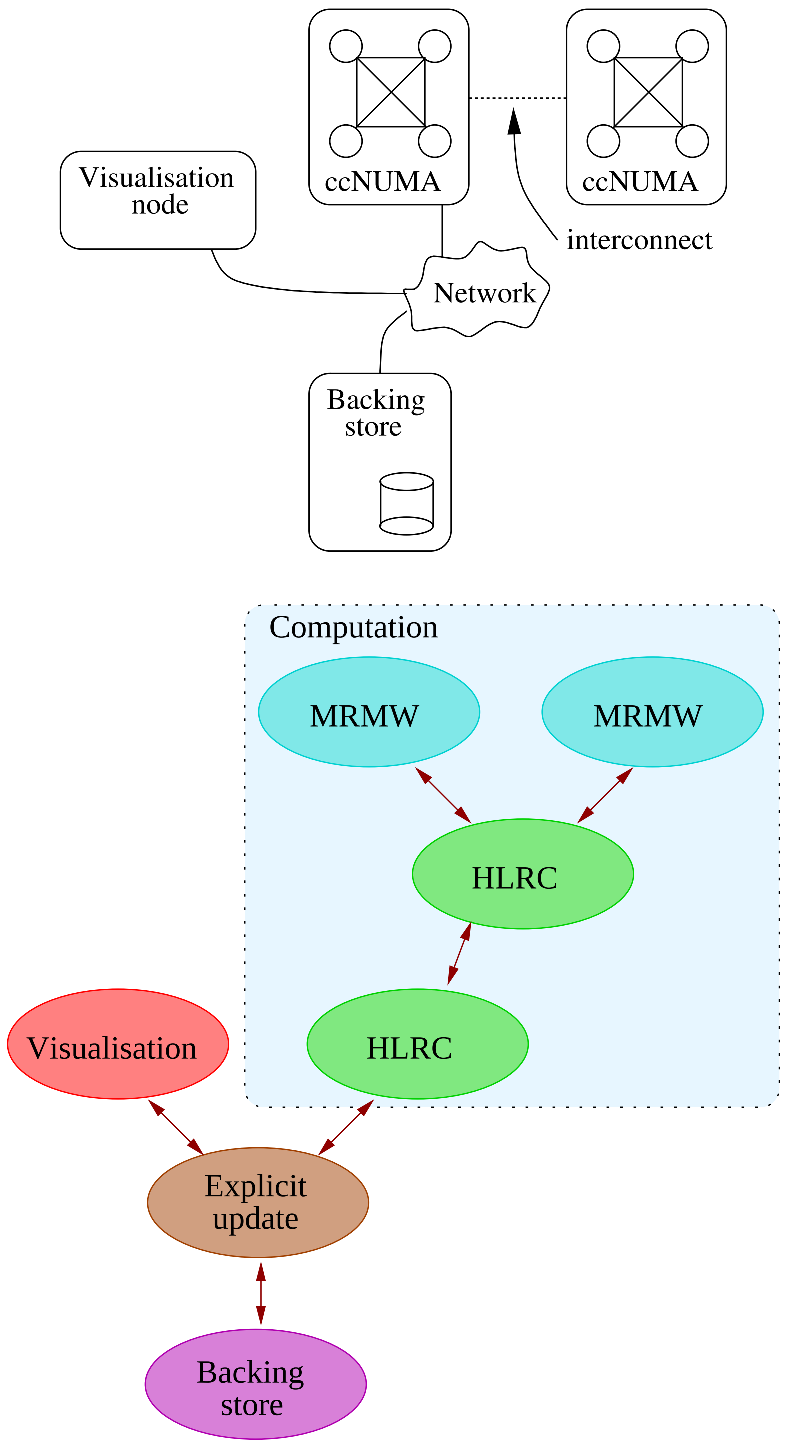


Figure 4.5: A run-time environment (top) and view hierarchy (bottom) for a long-running computation.

CHAPTER 5

EIDOLON FRAMEWORK

This chapter presents the Eidolon framework and the infrastructure it provides for running distributed applications in diverse and wide-area environments.

The Eidolon framework is an implementation of the Eidolon view architecture presented in Chapter 4. The purpose of this implementation is to provide a framework for verifying the view architecture through experimentation and benchmarks using standard distributed applications.

The framework is designed to provide a portable infrastructure for implementing Eidolon. It is structured so that it can be easily ported to run on top of any operating system or possibly within an operating system that provides transparent DSM to applications. The framework also supports several programming model APIs using the layered abstraction of the view model. We configure the framework as a server daemon. Client processes run separately and connect to the server using local Unix sockets and mapped shared memory regions. An alternative configuration of the framework allows for the server and client to be placed together, in a single executable. While this approach means that interactions between view pagers and the client are much faster, there is less flexibility. For example, in this scenario, the server cannot be reused if multiple client applications are running on a single machine.

The Eidolon framework is implemented on top of Kenge [30]. Kenge provides Eidolon with a build system, libraries and device-driver interfaces. It supports device drivers running as Linux kernel modules or as user-level applications, providing us with greater flexibility in choosing how to integrate Eidolon with our host systems and applications. The implementation used for evaluation runs as a user-level application for Linux, and as such, suffers from many of the overheads of multiple memory protection and IPC operations that are required.

5.1 Framework components

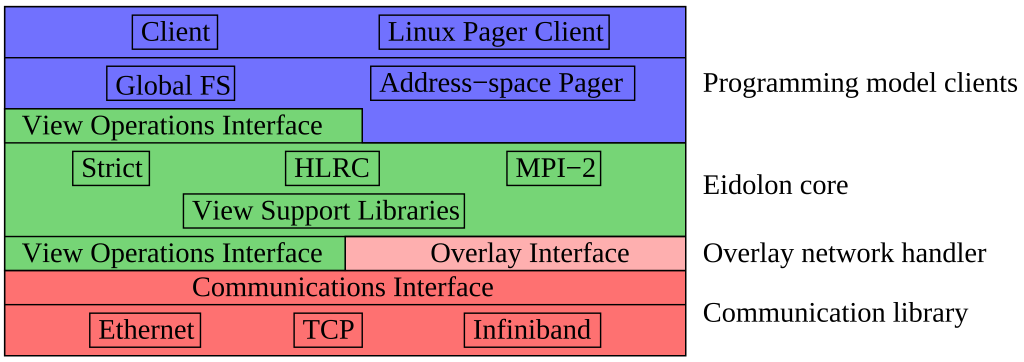


Figure 5.1: Framework for running various distributed applications.

Figure 5.1 shows the framework structure with each of the core components that form an Eidolon application environment. The Eidolon framework implementation provides the following core components:

Programming model clients used by a distributed application. These include DSM address-space pagers, one-sided MPI interfaces, and file system interfaces. These sit on top of the framework and communicate with the rest of the system by using view interface operations as introduced in Section 4.1.1.

Eidolon core provides the key mechanisms for views, and implementations of view consistency protocols such as *Strict*, *HLRC* and *MPI-2* which presented in Chapter 6. These are used to manage data sharing and coherency.

Overlay network handler provides mechanisms for finding a node that is able to answer a query about the location of data in the distributed system. Once the query is answered, the Eidolon core provides data sharing and coherency.

Communication library allows nodes and views to communicate among themselves using a high-performance, low latency interface. The communication library includes support for TCP/IP and Ethernet interconnects. These provide a view client interface. This library is discussed further in Section 5.2.

5.1.1 Client and server structure

As mentioned earlier, a design decision was made to separate the application from the Eidolon middleware implementation, rather than linking or loading the Eidolon middleware into the application. The application includes a small application-to-Eidolon glue layer which implements Eidolon operations and forwards them via Unix sockets to the Eidolon middleware.

This client and server approach allows for more than one client on the same machine to share the same server. The main benefit of this is the ability to use shared memory without additional copying when multiple clients exist on the same node. Between each client and the server is a shared memory region used to host the shared data. However, this approach suffers from the extra latency of communication between client and server over Unix sockets for most operations. For example, instructing the client to change access permissions to a region of memory occurs over this communication channel.

5.1.2 Configuring node view selection

For benchmarking purposes the Eidolon framework supports manual configuration of views on a per client basis. Configuration files are provided to instruct each node to create and or select one or more views. The application then uses these views to perform its data-sharing operations.

Alternative approaches to configuring and using views are discussed further in Chapter 8.

5.2 Communication

Communication is one of the most critical apects affecting performance of distributed computation systems. Achieving high performance and good scalability characteristics relies on being able to communicate a message from one node to another using the most efficient means possible. This often requires the development of communication routines that are optimised for a particular environment and the network hardware used within that environment.

On the other hand to achieve portability, many systems are built using popular communication protocols such as TCP/IP. While providing the desired portability, these protocols incur high communication overheads, which leads to reduced communications performance. At the expense of portability, other systems, such as MPI on Infiniband presented by Jiang *et al.* [50], are designed and implemented using specialised communications hardware in order to achieve high levels of performance.

To investigate the trade-off of achieving portability at the expense of performance, a high performance, multiple transport communication library has been developed. To achieve efficient and optimised communication, this library provides several *transports*, each of which provide an optimised implementation of communication routines for a particular network interconnect or software protocol. These include a raw Ethernet-based protocol, TCP/IP, and specalised shared memory interconnects such as Infiniband [71]. If a specific interconnect is available between communicating nodes, the appropriate transport can be used to communicate messages. When a particular desired transport is not available, an alternative applicable transport is chosen. For communicating nodes where multiple transports are available, nodes may choose the one that offers them the best performance.

This design focuses on providing fast, zero copy, low latency communication where it is available between communicating clients. The complexity of the transport interconnect, message routing and message interpretation are left up to the higher-level routines that use the communication library. This significantly reduces the transport implementation complexity and leaves a lot of room for flexibility.

5.2.1 Network interconnects and protocols

In Chapter 2 we presented several approaches that address communication across specialised interconnects such as Infiniband, and other approaches that support multiple communication mechanisms such as Nexus and Madeleine. Nexus deals with issues of portability by allowing multiple communication methods to be supported transparently. Nexus provides the nodes of an MPI-based system with a single-sided communication mechanism called a remote service request (RSR). The RSR mechanism of Nexus dictates the nature of message delivery and execution.

Message exchanges occur in Nexus by first encoding each field (such a as piece of typed data) in a message one at a time. At the receiving end, the message is decoded field by field until the whole message has been processed. Nexus allows the message fields to be transmitted immediately, delayed or only performed when it is safe to do so by providing this choice through its API. This provides an application with flexibility, however these semantics imply that a data copy to or from message buffers would have to occur in many circumstances. Nexus is overly complicated for the requirements of a high-performance communication library for a framework implementation such as Eidolon, with the added complexity impacting on performance.

Likewise, Madeleine III [9] allows for multiple communication transports at a lower level than Nexus, while supporting the communication abstractions of Nexus. It makes it easier to implement zero copy messaging, especially on interconnects that allow for it.

However, Eidolon requires only a simple zero-copy interface and as such does not require the high-level features and options present in Madeleine. As Madeleine messages are typed, the receiver does not know the size of the message until it decodes the message. Once the message is decoded it can then copy it or ask the sender to transfer the contents.

Due to the nature of the zero copy mechanism in Madeleine, messages larger than MTU require flow control to be implemented. In Eidolon's communications library, handling message fragmentation generated by a communications interconnect is left up to the receiver who is able to make a better decision on the right way to process the message.

The receive delegation phase of our library solves many of the handling issues present in Madeleine. Messages are received by a delegation function in fragments. This allows messages to be processed bit by bit if they are fragmented due to transmission limitations of the interconnect. Furthermore, it is possible for a transport to implement zero copy receive by only transmitting the message fragment information for the receiver to first decode and then receive the message contents using a remote-DMA capability, or by using other techniques such as making an assumption message delivery ordering and destination and then receiving directly into the destination buffer.

Our model provides a simplified message format that provides similar ability to the message pack/unpack operations of Nexus and Madeleine, however we do not deal with the ability to handle typed data. Arguably this makes Eidolon's communication library less flexible than Madeleine but also less complicated, since many of the features are not necessary for efficient message delivery and only serve to restrict performance. For example, our library does not deal with message retransmits that may be required for some interconnects (however, either the transport interconnect, or a higher level may implement this). Hence, this library is designed to offer high performance, common interface for communication without added functionality that is not necessary for the Eidolon framework.

5.2.2 Design

The design goals for this communications library are simple:

- To provide a simple interface that allows point-to-point communication and point to multi-point communication if the transport supplies it.
- To avoid over-generalising such that the efficiency of a transport implementation is restricted. For example, the need for message copying should be avoided.

These goals allow for the development of a high-performance library, that supports several of the modern day features of network devices such as zero-copy and remote DMA while offering an API that is not restricted to a particular network device implementation.

Eidolon allows nodes to communicate using multiple transports with each transport providing an implementation designed to communicate over a particular interconnect. When two nodes wish to communicate, the best available transport is chosen for all on-going communication.

Eidolon also provides a common message format that promotes the use of zero-copy messaging where possible. For example, buffers provided by a view pager will not be copied until it is necessary, such as when communicated over TCP/IP. The message format also supports fragmentation and delivery of partial fragments to view pagers.

Message format

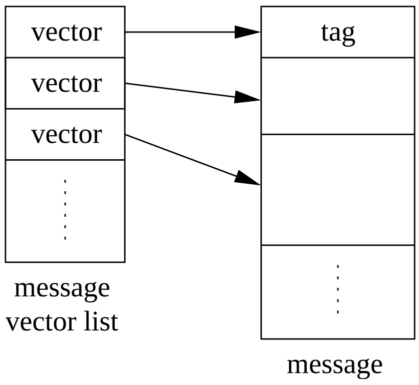


Figure 5.2: Message is represented by a set of vectors

A message is constructed using a list of vectors. Each scatter-gather vector points to a buffer that forms part of a single message. Figure 5.2 shows the message vector list expanded to represent a complete message comprising of the separate buffers. The vector list (shown on the left) is passed to the communications library, representing the complete message (shown on the right) that will be reproduced at the receiver. The use of vectors allows a programmer to quickly build a message consisting of multiple parts without copying the data. Furthermore, it provides the opportunity for the selected transport to use a zero-copy buffer transfer method.

As well as the message buffer vector list, a message also contains a type field, length field and application specific flags. The *type* specifies what type of message is being sent and is used to determine which handler will process the message upon arrival. The handlers are specified by the receiver. The *length* field provides contains the complete message length. The *flags* provide details about the message such as the message's endian format and whether it is fragmented or not. These flags are interpreted and handled by the receiver's handler.

Handling message fragmentation

When transmitting a message the transport may be required to break up a message into multiple fragments, such as packets, for communication. We avoid copying fragments into a complete message by passing each fragment onto the receiver handler. The receiver handler is able to make the best decision about what to do with the fragment. For example, Eidolon views support fragmented data updates, so the communications handlers do not need to re-construct whole messages in this case.

Fragmentation is supported as follows. The first vector of a message represents a *tag* buffer. The tag, which is a unique identifier for each complete message, is communicated with each fragment to assist in processing a message upon reception. A fragmentation header may also be attached to each message by the transport. The header contains details about the fragment, including the fragment number and the byte offset into the message that the fragment contains. This allows for the receiver to process a fragment of a message without having to receive the whole message.

For example, when transfering a piece of data that is larger than the maximum transfer unit of the transport, it gets fragmented. Each fragmented arrives at the receiver which reads the tag header to determine which region of memory to apply the changes to. It then offsets into this region based on the fragment offset information and copies (or DMA transfers) the data directly into the target memory buffer. The target memory buffer is encoded as part of the message transport header and is used to direct the message to the appropriate view pager for further processing.

API

The API is broken down into two categories.

Registration functions

```
register_transport(transport_t trans, send_func_t send)
register_msg_handler(handler_callback_t func)
```

Before being used on a node, a transport must first be registered. This is achieved by calling the `register_transport` function. This registers a message send callback function that will be called by `send_msgv` in order to send a message.

Message handlers process a message based on a `type` field that is communicated with the message.

Communication functions

```
send_msgv (msgv_t *msg, node_t to, msg_type_t type)
delegate_msg (msgv_t *msg, node_t to, msg_type_t type)
```

Each transport implements the `send_msgv` function. This function handles the internal transmission of the message. When the message is received, it is processed by an internal receive function that the transport provides. This function then calls *delegate_msg* which passes a fragment of a message to be processed. This call invokes the appropriate pre-registered message handler.

Note that `node_t` can represent a group of nodes for multi-cast and broadcast domains, however Eidolon does not currently make use of this feature.

5.2.3 Communication transport implementation

In this section we use the Ethernet transport as an example. The implementation of other transports is quite similar except for network interconnect and protocol details. All transports perform with zero-copy semantics until a copy is necessary for communication or protocol interactions.

Nodes that can communicate directly to each other over Ethernet are able to do so without the added overheads of TCP/IP and other network protocols. For a cluster of nodes connected via high speed Ethernet, this results in greater performance and lower protocol overheads [25].

When two nodes first start communicating with each other over Ethernet, they first send node information to each other to establish point to point connections between themselves. Once this occurs, each node caches information about the other node for future communications.

Authentication

In general, transports authenticate end points for security and correctness. This is achieved by first communicating node information to each other via a handshake. Once this occurs, each node caches information about the other node for future communications.

Zero-copy transmit and receive

Figure 5.3 shows how the Ethernet transport transmits a message represented by a *message vector list*. Firstly, for each packet that the transport sends, labelled with *transmitted ethernet fragments* in the diagram, it attaches a *transport* header that contains an Ethernet header and transport specific state. It then appends the *tag* buffer that is represented by the first vector of the message. For each packet of a message this is attached. If multiple packet fragments are sent, a fragment header is attached followed by the payload. The payload contains as much of the message as can fit in the remaining packet space.

The *temporary transport buffers* are extra buffers that the transport requires in order to communicate a message. It contains the transport header, which is used for both message fragments, and each of the fragment headers. In this transport implementation, reference counting is used to keep track of when these buffers can be freed.

The receiving side receives Ethernet fragments. In most cases, each valid fragment is passed to Eidolon for immediate processing. This approach allows Eidolon to handle a message without unnecessary copying. Hence, for normal message data, the only copy performed is from the fragment buffer into place in the view's address space. When a view client in Eidolon is blocked on receiving a complete message, it will not continue until all fragments have been received for correctness.

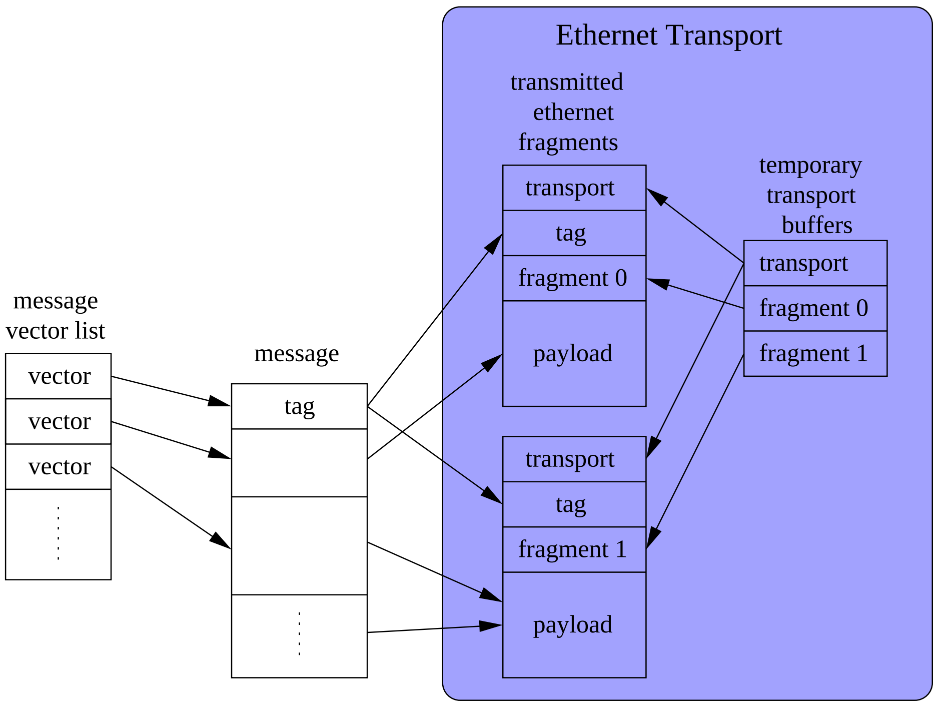


Figure 5.3: Transmitted message over Ethernet using zero-copy.

5.3 Wide-area data communication

In Eidolon, an application's state including data and synchronisation tokens live within views. As explained in Chapter 4, these entities are addressed by a unique, location-independent address inside a single-address-space (SAS). The SAS is provided across all nodes running Eidolon. Using a SAS allows for long-lived data, and multiple applications sharing a single distributed instance of Eidolon.

The distributed threads of an application require mechanisms for obtaining their desired data or shared data regions by locating and then selecting already established views, or by creating views to meet the application's data sharing criteria. Once the views required by an application are known, the application can begin to use and exploit views. This provides our approach to usable wide-area computation.

To achieve this, Eidolon requires mechanisms to resolve a view identifier to its location. For small networks such as clusters, a simple broadcast mechanism can be used to resolve this information. However, for wide-area and more complex distributed networks, ranging from multi-clusters through to Grids, a more efficient approach is needed.

Eidolon can use an overlay network in order to discover the views available to the application with only a small number of messages being sent across the network. The overlay network is also used to resolve the location of a view to a specific node. The application then has access to further information specifically related to that view.

Popular overlay networks implement a distributed hash table (DHT) service. These include Chord [85], Pastry [78], Tapestry [41, 94], OceanStore [58], and Ratnasamy *et al.* [75]. They are capable of resolving a request *key* to its *value* via the DHT while only contacting a small number of nodes. For example, Tapestry can resolve a request in $O(n \log n)$ hops, where $O(n)$ is the number of objects.

Due to the nature of DHTs, one problem is that there is no locality involved when resolving the value of a key. Other approaches such as Plaxton *et al.* [73] implement a simple randomized algorithm which attempts to resolve requests using nearby data.

These overlay networks also make it difficult to resolve a range query. For example, an application in Eidolon may know the address of data it is interested in, however it may not know the SAS address of the view or view identifier. To solve this, there exists overlay networks which support range-queries [1].

Eidolon provides its own overlay networks called *Donut overlays* suitable for mapping a contiguous range of keys to a value rather than the typical distributed hash table key/value pair mapping provided by current overlay networks. Donut overlays are used by Eidolon in wide-area scenarios to resolve the node on which a view lives, to locate and to communicate with nodes prior to establishing direct connections, and to manage view data space allocation. Further discussion on overlays for wide-area data-location are beyond the scope of this thesis.

CHAPTER 6

PROGRAMMING VIEWS

In this chapter we explore several programming models integrated with Eidolon. We examine both the client interface implementation and their view pager implementations.

In Chapter 7, these implementations are used for evaluation of several different view protocols, to allow us to explore the properties of the view model.

Note that while it is possible to take existing protocol implementations and simply wrap them in view interface operations, we found that available protocol implementations lack abstraction layers and are very tightly integrated with their target platform making it infeasible to reuse their implementations.

6.1 Sequential- and release-consistent memory

6.1.1 Client interface

The shared-memory client interface maps shared regions into an address space based on view address. Table 6.1 provides a summary of the client's native action and its corresponding invocation on the view interface. The latter part of the table summarises the view operations invoked by the view pager on the view client.

6.1.2 Library interface

Along with the client interface, we have implemented several programming and thread models:

View client action	**Invocation on view pager**
Page fault	`update_request`
Response to invalidate request	`invalidate_response`

View client action from pager	**Invocation on view client**
`update_propagate`	Map region into address space, copy data if zero-copy is not possible. Update access permission for region.
`invalidate_request`	Update access permissions for region
`update_request`	Reply with data using `update_propagate`, update access permissions.

Table 6.1: DSM client interface.

Custom A simple programming interface that provides a `global_malloc()` for allocating consistent memory from within a single view. Barriers and locks are provided to manage view synchronisation tokens. The view is established by the underlying run-time system and choice of view behaviour depends on the environment.

TreadMarks The TreadMarks API provides a release-consistent memory model to applications and calls for establishing memory regions, barriers and locks.

SPLASH This interface is similar to the custom and TreadMarks interfaces, however it provides an interface for running SPLASH benchmarks.

VODCA A preliminary VODCA [42] interface (a part of VOPP introduced in Section 2.2) has been implemented in order to support VODCA applications. Currently, shared data is backed by consistent data views, however, ideally would be backed by VODCA's native consistency protocol views.

6.1.3 Explicit update consistency

This view type requires the programmer to explicitly request updates or propagate data changes for a specified region of data within the view. This type of view is useful for bulk data transfer to nodes, without any meta-state management overheads that are typical of other protocols.

All view interface operations treat a whole region as a unit of coherency. That is, the client must ensure that single-writer semantics are enforced for each region it specifies in a view-interface operation.

The view home keeps track of the current home of each data region. Data regions migrate to the node of the writer if they exhibit single-writer semantics.

6.1.4 Strict consistency

This protocol, also known as multiple reader, single writer (MRSW) consistency, enforces sequential consistency.

This allows programs without shared-memory annotations (e.g., release-consistency annotations) to execute correctly. A significant drawback of this is false sharing, which leads to excessive communication and poor performance.

The unit of consistency is a fixed page size (8192 bytes on Itanium systems). This page size used is the minimum granularity of all data transfers from the view client. Note, however this does not restrict view clients from using arbitrary region sizes. To reduce the impact of false sharing, hardware that supports smaller page sizes may request page data in units of their page size.

The view home is required to maintain the location of each unit of consistency and enforce the single-writer/multi-reader semantics on each page. Strict consistency states a page as having exclusive (write) access or non-exclusive access. The view home maintains this state for each page the view covers.

Synchronisation tokens such as locks are forwarded to the user. The consistency protocol does not need to maintain any meta-state with the lock. The implementation of this protocol also ignores barriers by forwarding their tokens to view clients. A side-affect of this is that counting of waiters on a barrier is not performed within the view, resulting in more communication. An improvement to this view would count view clients entering a barrier in order to reduce communication externally to the view until all view clients have reached the barrier.

6.1.5 Multiple reader, multiple writer consistency

Multiple reader, multiple writer (MRMW) consistency supports systems that provide shared-memory coherency in hardware.

When an MRMW view receives access to a data region from its parent view, it maps the data region to its clients using the access rights granted to it by the parent view. This allows it to map its maximum access permissions to all clients. Any future shared-memory operations, including writes, occur without any software intervention.

The parent view can revoke this access, which is required for maintaining data consistency outside the child MRMW view. When this occurs, all data sharers within the MRMW view get part or all of their access revoked. This mechanism allows us to place an MRMW view on top of another view, such as a strict consistent view and thereby maintain strict consistency, while allowing hardware-coherent nodes within an MRMW view to perform consistent operations without software management or intervention.

MRMW views are not suitable for communication over TCP/IP or where data sharing and consistency is not managed transparently, unless the intention of the user is to explicitly avoid communication.

This protocol works on whole pages and does not perform sub-page diffing.

6.1.6 Home-based lazy release consistency

Each page of the view has a home in which all updates are propagated, significantly reducing the memory overhead compared to lazy release consistency (LRC) as memory changes can be discarded once committed to the home page.

This protocol uses a vector time-stamp for each page. The vector consists of a time-stamp entry for each node participating in the computation. The time-stamp entry monotonically increases after each modification by a view client.

Consistency intervals are generated in the form of a *write notice*. The write-notice contains a list of pages that have been modified and a timestamp. When changes to pages are committed to their home nodes, the timestamp for the node that made the change is updated for each page's timestamp vector. This is transferred with the data using the out-of-band messaging support of the view interface.

Hence, when communicating with other views that do not use time-stamp vectors, this information can be discarded.

When a new node requests the lock, the request is sent to the previous holder of the lock which sends the required set of write-notices that the requester may not have seen, since last acquiring the lock. The requester can then determine which pages it needs to request from the page's home.

The view home keeps track of each HLRC data home page within the view context. In most cases the view home will normally be the home for each HLRC page, however it has been shown that home-page migration can improve performance, particularly if code is exhibiting migratory data accesses.

The view home also manages all locks that enter the view and ensures a coherent view of data can be obtained for any lock that leaves the view. This also applies for other consistency operations.

A barrier manager keeps tracks of view clients entering a barrier. Once all clients with a barrier are reached, it will signal its view parent, or current authoritative barrier that the barrier may continue. This avoids unnecessary communication for each view client reaching a barrier.

6.2 One-sided message passing

One-sided message passing provides a window of shared memory on a node. One side accesses the shared memory region directly using hardware memory operations, while the other side performs explicit read and write operations into this window. This approach is an extension to traditional two-sided message passing with a simplified shared memory window.

There are several benefits of the one-sided MPI implementation provided by Eidolon over other implementations, such as that provided by MPICH-2. One-sided MPI was designed to take advantage of interconnects that provide Remote Memory Access (RMA) and architectures that provide hardware-coherent shared memory. Eidolon allows a one-sided MPI implementation to take full advantage of these mechanisms for data sharing. The configurable nature of views also allows one-sided MPI applications to perform optimised broadcast and collective communication.

6.2.1 Client interface

Table 6.2 provides a summary of the MPI API and its corresponding implementation via one or more view-interface operations.

View client action	**Invocation on view pager**
`MPI_Win_Create`	`view_create`
`MPI_Put`	Queue transaction for write-batching and begin transfer using non-blocking `update_propagate`.
`MPI_Get`	Queue transaction and begin non-blocking `update_request`.
`MPI_Win_Fence`	Wait until all data transfer queues are empty then signal using `token_request` on fence token.
`MPI_Barrier`	Issue a `token_request` on barrier token.

View client action from pager	**Invocation on view client**
`update_request`	Return data using non-blocking `update_propagate`.
`update_propagate`	Write data into window if not zero-copy, and process corresponding `MPI_Get` request from transaction queue.

Table 6.2: One-sided MPI interface.

Window Creation and Deletion

`MPI_Win_create` is a collective operation performed by all clients in order to allow other clients to access a local memory region.

This operation maps to *view_create* which creates a globally-shared region that overlaps the window specified as arguments to `MPI_Win_create`. At this stage it is possible that Eidolon will create a hierarchy of views in order to adapt to the underlying run-time environment. Once a view is created, all participants must select their appropriate view using *view_select*.

The type of view created depends on the view's behaviour specification and the underlying run-time environment. For example, when running on a ccNUMA system, ideally any communication between nodes should be handled by the cache-coherent memory interconnect provided by the hardware. Hence, for a one-sided MPI application in this environment, we select a view implementation that directly maps the one-sided operations to shared memory. Doing so will significantly reduce overheads by allowing updates to shared data to occur without explicit software intervention.

`MPI_Win_delete` is a collective operation for releasing and freeing all state associated with a window. In the view architecture this maps to a *view_unselect* operation. View state will not be freed until all clients have called this operation or it is forcibly removed.

MPI_Put, MPI_Get, MPI_Accumulate

The design and implementation of these functions depends on the view behaviour specification which the communicating clients must adhere to.

We identify two design options. Firstly, the one-sided MPI data operations map directly to the view-interface operations outlined by Table 4.1 in Section 4.1.1. For example, `MPI_Put` becomes a non-blocking *update propagate*. Once the action is completed, the requester receives an *acknowledge* response. This is arguably the correct approach as the behaviour of such calls will mimic a traditional middleware implementation of one-sided MPI. In particular, as it is non-blocking it can easily pre-fetch data regions, ensuring good overlap of communication and computation.

Alternately, they may sit on top of a view client that catches address-space page faults, such as the view client presented in Section 6.1.1, and not invoke any view operations directly. The MPI operations then become simple memory copies, however they suffer from additional latency that is caused when data needs to be paged in. In this case, the client blocks on a page fault until the data for that page is received from remote nodes.

In our evaluation of Eidolon, we explored both approaches and noted that for the benchmarks performed in our evaluation, we achieved an average 12% improvement in throughput when using explicit requests across nodes communicating via a 100Mbit TCP/IP network.

However, for one-sided communication between clients running on a ccNUMA machine the second approach, where we rely on explicit page faults, performed an order of magnitude better in cases where the shared-memory window is already pre-mapped and accessible by clients running on the ccNUMA node.

We implement `MPI_Put` and `MPI_Get` as non-blocking *update propagate* and *update request* respectively. Each operation is tracked to ensure completion before exiting a `MPI_Win_Fence`, and if necessary, is copied to its destination. `MPI_Accumulate` iterates over the window performing the accumulate operation.

Synchronisation

MPI synchronisation primitives are implemented by using the view token operations. In our implementation, when each client invokes `MPI_Barrier`, it issues a *token request* operation to the view pager managing the token. Once all clients have issued a token request, the view pager sends a *token response* to each client. The barrier call then exits and allows program execution to continue.

The function `MPI_Win_fence` is a collective synchronisation operation issued by all clients sharing a window object. It ensures that all previous requests within the window have completed and also begins a new epoch for future requests. To ensure all clients wait for all requests to complete, tokens are used to implement a barrier call. In this case the view pager also ensures that any previously outstanding operations are completed.

The operations `MPI_Win_start`, `MPI_Win_post`, `MPI_Win_wait` and `MPI_Win_complete` provide window synchronisation that can be restricted to include only the clients actually exchanging information. Currently these MPI operations are implemented sub-optimally using barriers.

6.2.2 Protocol implementation

A view protocol for one-sided MPI operations is not necessary, as other shared memory protocols such as strict consistency are sufficient. However, the main benefit of a view pager implementation is to provide more efficient processing of the communication generated from `MPI_Get` and `MPI_Put` events. This is achieved by queuing `update_request` and `update_propagate` operations within the view pager, similar to the method discussed above for the client interface. The view pager can then perform write-combining and amortisation of these operations by delaying any communication until an `MPI_Win_Fence` or a communication threshold is reached.

6.3 Two-sided message passing

Two-sided message passing relies on explicit communication between two worker threads (or nodes). This includes point-to-point and bulk forms of communication.

In this thesis we do not explore this model further. However, we speculate that the view model can benefit two-sided message passing systems such as MPI in a number of ways. Protocol selection ensures communication between two end-points uses the best available communication protocols including multi-cast and other forms of bulk communication. Through overlapping views, an efficient communication path between nodes can be represented.

The complexity of developing a complete two-sided message-passing interface is comparable to that of a one-sided implementation as traditional send and receive map well to the view API. However, care is required to ensure that performance meets that of a traditional two-sided implementation, particularly when coupled with other views.

CHAPTER 7

EXPERIMENTAL VERIFICATION AND EVALUATION

The evaluation in this chapter evaluates the view model using Eidolon. We evaluate the properties of the view model presented in Section 3.2 using techniques including protocol selection, locality domains and view hierachies to adapt applications in environments such as multi-clusters.

The benchmarks used to evaluate the view model are derived from benchmarking suites including SPLASH-2 and TreadMarks [65] while others are handcrafted implementations of well known algorithms. These benchmarks are executed across a variety of small non-uniform cluster scenarios.

One-sided MPI micro-benchmarks are also evaluated to illustrate the use of views for alternative programming models. Finally, we evaluate the ease of use of Eidolon for running unmodified applications.

We argue that the evaluation of Eidolon on multi-clusters extends to Grid environments. Many applications, particularly those that require significant low-latency communication, use a small number of nodes when run on Eidolon (or any distributed system). In a Grid environment the nodes used by the application may be selected from a larger pool of nodes on a Grid. Alternately, applications that have low communication resource requirements are run using a large number of nodes. These applications are less interesting cases for demonstrating and experimenting with Eidolon.

Grid environments have many characteristics in common with multi-clusters.

Both can consist of a variety of heterogeneous nodes of varying performance and architecture, networks with specialised interconnects and varying latency and unique topologies. In essence, Grid environments for distributed computing are an extension of multi-clusters to include more nodes over wide areas.

The experiments performed in this chapter run parallel application benchmarks that exhibit a variety of different data-sharing characteristics. We examine the benefits that Eidolon views provide these applications without directly modifiying or re-configuring the application to be better suited to their run-time environments. In order to provide our evaluation, we consider the following application characteristics:

Speed up To most users, and for obvious reasons, the overall execution time of a distributed application is the most important measurement to consider when analysing the performance of a distributed system or software. We begin by presenting the execution time and relative speed ups for a number of different distributed applications.

Speed up provides an indication of how well a distributed application performs in different environments. There are, however, a number of factors that influence the execution time and potential speed up of a distributed application. This includes algorithms and data sets used by the application, underlying architecture scalability limits and software consistency protocol scalability limits. For many distributed applications, these factors play a major role in determining the limitation of performance improvements that are possible.

To begin to understand these limitations, we must first consider the *concurrency and load balancing characteristics* of a program [92]. The concurrency and load balancing characteristics of a distributed application indicate how well the program will utilise the computing resources available to it. An understanding of these characteristics allows an application programmer to modify an application to better utilise resources.

The view model does not directly alter the application based on these characteristics. Hence, in order to more clearly examine the properties of the view model, we examine the characteristics discussed below.

Data access patterns Graphing data access over time illustrates the behaviour of an application as seen from the underlying middleware system. We present data access patterns in order to examine the temporal and spatial locality properties of different applications and view configurations.

Data access locality Client data accesses often result in external communication from one node to another before a data request can be resolved. Minimising the time it takes to resolve these requests can be achieved through data-caching hierarchies, such as those provided by view hierarchies. We examine the distance a data request has to travel to be resolved by presenting the ratio of resolving requests within a client and views.

Network characteristics (bandwidth and latency) The communication-to-computation ratio of many distributed applications directly affects its ability to scale performance with additional nodes. By examining an application's network characteristics we gain information about potential communication bottlenecks.

7.1 Run-time environment and execution scenarios

The execution environment consists of a collection of multi-processor Itanium nodes connected via Ethernet. For most experiments, these are configured into two different environments.

The first environment, illustrated by Figure 7.1, includes two four-way cc-NUMA nodes connected via 100Mbit/s Ethernet. One node has 900MHz Itanium processors, while the other node has 1.5GHz Itanium-II processors. In this environment communication between processors of each node has more bandwidth with lower latency compared to communication between processors of different nodes.

The second environment, illustrated by Figure 7.2, is a multi-cluster. Cluster 1 contains several SMP nodes along with a ccNUMA node. The nodes of cluster 1 are connected to each other via 100Mbit/s switched Ethernet and each processor is 900MHz. Cluster 2 contains an identical configuration of nodes to

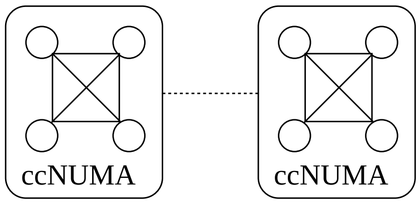


Figure 7.1: Eight processor dual ccNUMA environment.

Cluster 1, however, these nodes are connected via 1000Mbit/s switched Ethernet and each processor is 1.5GHz. The two clusters are connected via 100Mbit/s Ethernet. In this environment there is a communication hierarchy which reflects the environment's network topology. Communication between processors of the same node should be favoured over communication between nodes of the same cluster. Likewise, the cost of communicating between clusters is less desirable than communication between nodes of a single cluster.

The experiment results presented in this chapter number each processor starting with the processors of the ccNUMA in Cluster 1.

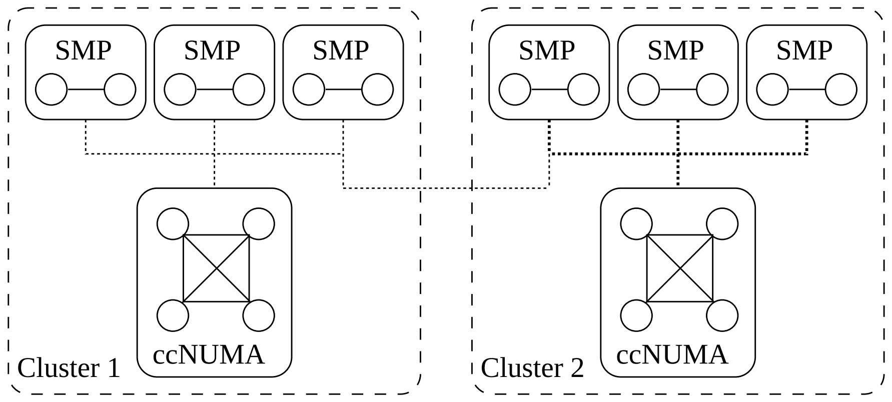


Figure 7.2: Twenty processor multi-cluster environment. Cluster 1 nodes are connected via 100Mbit/s Ethernet. Cluster 2 nodes are connected via 1000Mbit/s Ethernet. The link between clusters is 100Mbit/s Ethernet.

7.1.1 View configurations

There are a number of possible view configurations suitable for the run-time environments used in these experiments. Firstly, we consider a single view configuration that uses a single consistency protocol instance across all nodes as illustrated by Figure 7.3. Protocols are chosen between single-reader/multi-writer (SRMW) strict consistency, multi-reader/multi-writer (MRMW) consistency, and home-based lazy release consistency (HLRC). A single-sided MPI protocol is used for MPI experiments. This configuration mimics that of a traditional middleware system and provides a base-line for all experiments. This configuration can be varied slightly via protocol selection.

Secondly, we examine Eidolon in scenarios where locality of access and adaption to network topology are important. For multi-clusters we configure our benchmarks to use a view for each cluster as illustrated by Figure 7.4. For clusters of multi-clusters or SMP nodes, we build a hierarchy of views whereby each view encompasses a locality domain as illustrated by Figure 7.5.

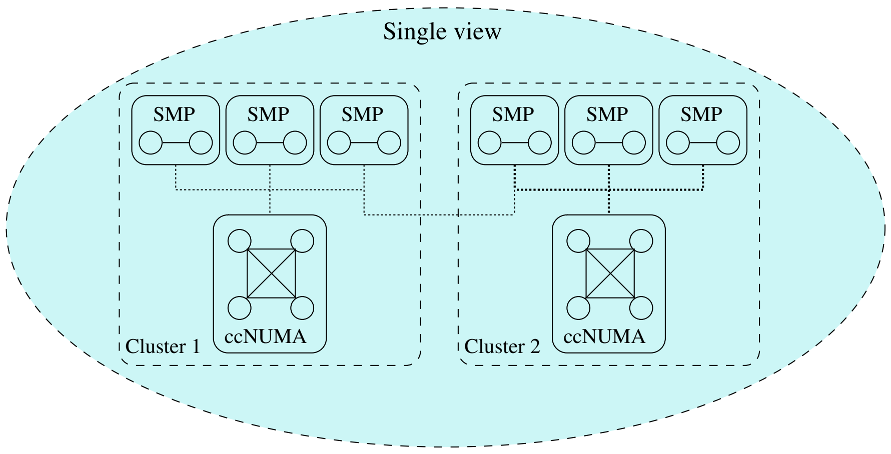


Figure 7.3: Twenty processor multi-cluster environment with a single view encapsulating all processors.

Finally, we combine multi-cluster view configurations with protocol-selection techniques to adapt data sharing to use the most efficient means available. For communication within ccNUMA and SMP nodes, the MRMW consistency protocol is used as it supports the direct use of each node's hardware data consistency

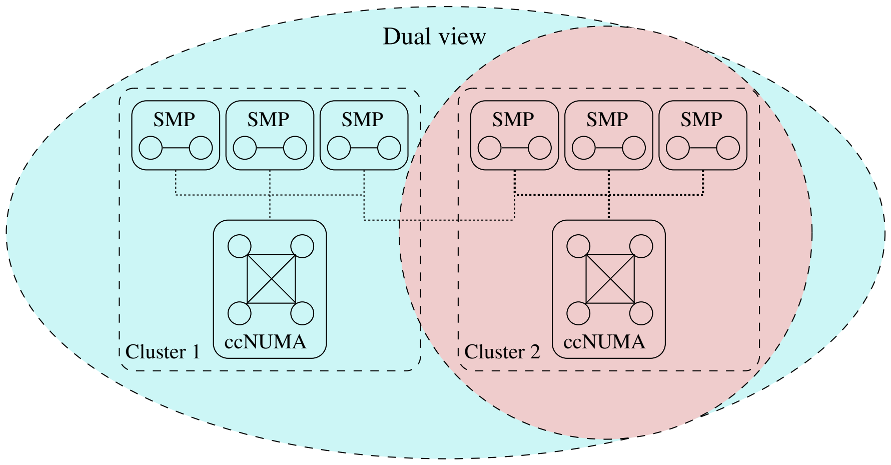


Figure 7.4: Twenty processor multi-cluster environment with a second view encapsulating the processors of cluster two.

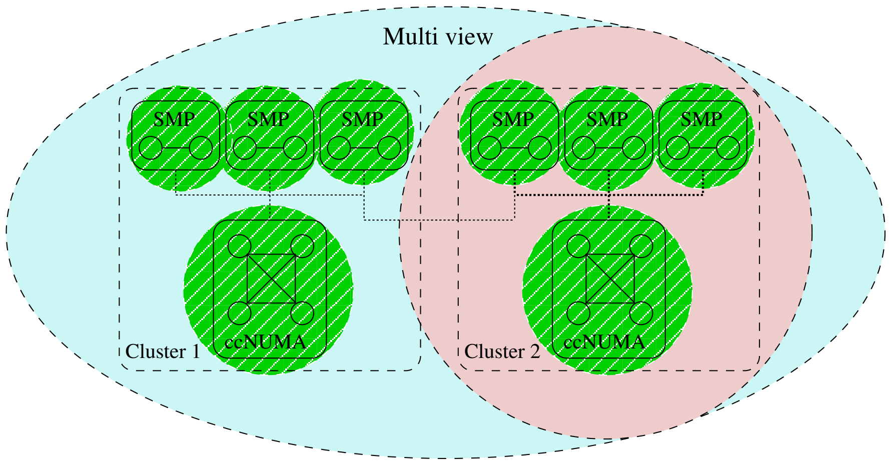


Figure 7.5: Twenty processor multi-cluster environment with views encapsulating each cluster and the processors of each machine.

mechanism without the overheads of software intervention.

7.2 Optimisation and configuration experiments

In this section we examine several distributed application benchmarks using Eidolon with different view configurations. Benchmarks were chosen that offer a variety of different data-sharing characteristics.

7.2.1 Matrix multiply

Matrix multiply computes the matrix product $C = AB$ where A is a m-by-n matrix and B is a n-by-p matrix. Each entry of new matrix C is calculated by computing the inner product of every row of matrix A with every column of matrix B as follows:

$(AB)_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj} = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in}b_{nj}$ where $1 \leq i \leq m$ and $1 \leq j \leq p$.

The implementation used for benchmarking is a somewhat naive $O(n^3)$ number of operations, however it performs well by computing along rows of matrix C for improved page-access locality. This minimises write false-sharing, which can occur when using protocols such as strict consistency when the elements of C are updated by different processors. The matrix sizes for benchmarking is 1200x1200 extended floating-point integers.

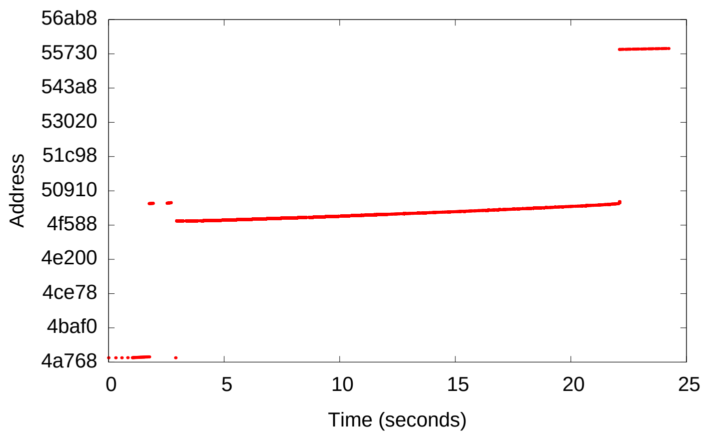


Figure 7.6: Data access pattern for a single client when running matrix multiply across 16 processors.

Figure 7.6 shows the data access pattern of a single distributed worker thread. For this application, there are three distinct address ranges of data access that correspond to matrix A, B and C (top). Data access is regular with good temporal and spatial locality.

Optimising matrix multiply for multi-processor environments

Multi-processor environments are becoming increasingly more common. While many applications that were traditionally designed for cluster environments are capable of running on multi-processor systems, they communicate using transports such as TCP/IP rather than directly over shared memory. In particular, cluster applications are able to run naively in multi-processor environments by treating each processor as a separate cluster node. Software consistency protocols are used to ensure shared data is kept consistent. On the other hand, multi-threaded applications are able to run on multi-processors and as they make use of shared memory, are more resource efficient than their cluster-based counterparts. However, as they rely exclusively on shared memory communication provided by the multi-processor, they are not able to run in cluster environments. Hence, there is often a trade off between portability to new environments and performance for distributed applications.

Figure 7.7 compares two traditional implementations of matrix multiply on a 4-way ccNUMA machine. As expected, the multi-threaded approach is better suited to shared-memory multi-processors. The worker threads of the application share an address space whereby the consistency of memory operations is maintained by the hardware. The cluster approach, which implements strict consistency protocol, incurs a performance impact from processing 6171 page faults that occur so that the implementation can enforce shared data consistency during the run-time of this application.

This benchmark illustrates that better resource utilisation, achieved through approaches such as direct communication over hardware shared memory, is one method of optimising applications for multi-processor environments.

Speedup of matrix multiply for dual cluster

Figure 7.8 shows the speedup of matrix multiply executed across two four-way ccNUMA machines for several different view configurations summarised in Section 7.1.1.

The first run, *single strict*, shows a traditional scenario with a single strict consistency view. That is, the strict consistency protocol is used on across all

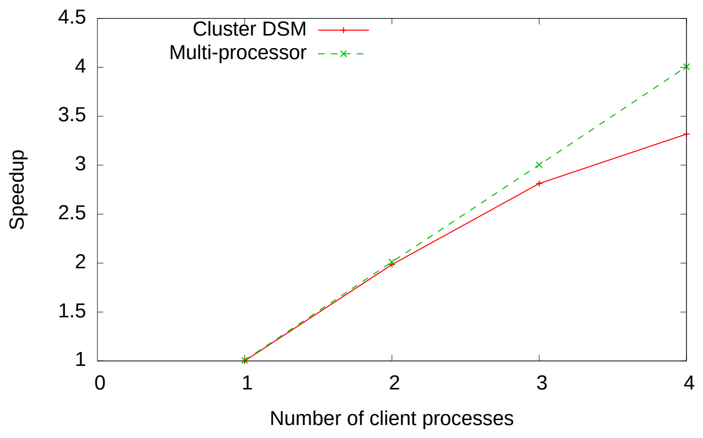


Figure 7.7: Two implementations of matrix multiply running on a multi-processor system. The first is a cluster-based implementation capable of running on clusters and multi-processor systems. The second is a multi-threaded implementation, that runs faster through transparent utilisation of hardware shared memory.

processors, treating each processor as a separate node of a cluster. This is equivalent to taking an existing traditional cluster implementation of matrix multiply and running it in this environment. In this scenario, we see a maximum speedup improvement at seven processors. Adding more processors degrades performance.

The poor performance and degradation in speed compared to the other view configurations can be attributed to communication bottlenecks. The top graph of Figure 7.9 presents the bandwidth utilisation across the 100Mbit/s Ethernet link between each ccNUMA node when configured to use a single strict consistency view. For this benchmark the outgoing data saturates the connection, and is caused by false sharing. The incoming data uses approximately half the available bandwidth.

The next benchmark run, using the *dual-strict* view scenario, places the processors of each ccNUMA machine into their own view, creating a dual-view scenario. This establishes domains of locality whereby local client accesses can be resolved without external communication. For example, if a read-only data page is already present within a view, other view clients can access it directly. In terms of view hierarchy, the processors of the second machine are part of a child view. Within the child view, any request that cannot be handled internally, is sent as a

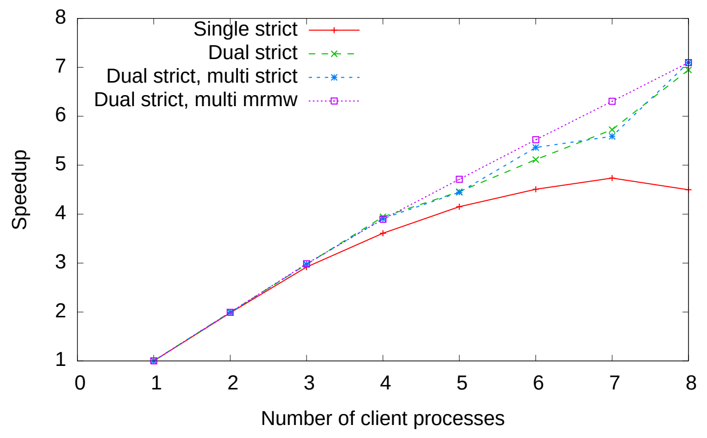


Figure 7.8: Execution time speedup of matrix multiply when run in different view configurations across the processors of two 4-way ccNUMA machines.

request to the parent view. The parent view contains the processors of the first machine, however only one server (running on one processor) handles requests from child views.

The *dual-strict multi-strict* view scenario creates an extra level of views for each ccNUMA machine. These new views are child views of each Strict view. In this environment, this view configuration does not benefit data sharing as locality domains have already been established with a strict view per ccNUMA machine. It is normally used for larger clusters to establish hierarchical locality domains. However, this result illustrates that in this case, the overhead of extra views is minimal.

The final run, *dual-strict multi-mrmw*, encapsulates the processors of each cc-NUMA machine in a MRMW view. These views are child views of the Strict view. This ensures that all data sharing between processors of each machine occurs over shared memory whereas in the previous view configuration *dual-strict multi-strict* used a software-based consistency protocol to maintain consistency between processors. As a higher proportion of data sharing is handled by shared memory, a further speedup improvement is evident.

The speed up improvements over a single view are clearly evident, due to the lower communication overheads of the dual-strict multi-strict configuration. In

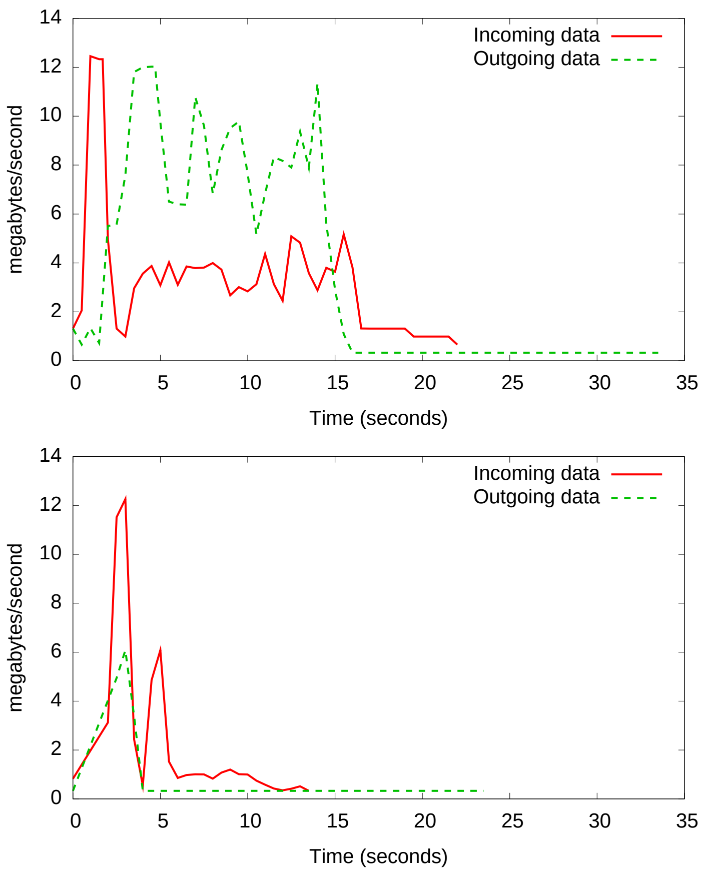


Figure 7.9: Bandwidth utilisation between ccNUMA machines for a single strict view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running matrix multiply with 8 processes.

Figure 7.9 we see this configuration has much lower communication bandwidth utilisation than the single strict view configuration. This is due to the dual-strict view configuration which ensures read-only matrix data is sent only once over the Ethernet link, and also allows read or write requests for data already present within each locality domain to be resolved without external communication.

Speedup for cluster of non-uniform multi-processors

The twenty processor, multi-cluster configuration outlined in Section 7.1 provides another environment to analyse the properties of views. Figure 7.10 shows the speed up of matrix multiply across the processors of this system. In this environment, we see steady speedup for all view configurations until external communication is required at processor five and onwards.

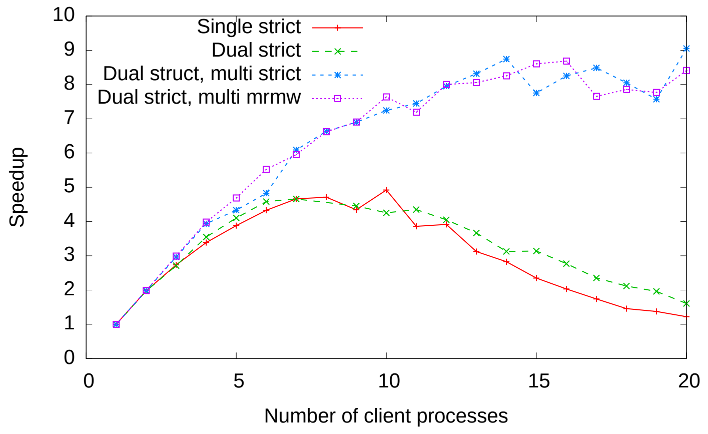
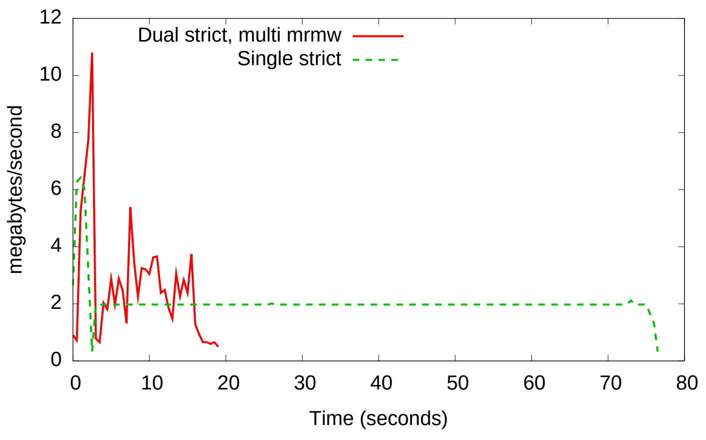


Figure 7.10: Execution time speedup of matrix multiply benchmark when run in different view configurations across the processors of a multi-cluster.

The speedup results for each view configuration appear to be split into two groups. The first containing the single-strict and dual-strict view configurations, improve in speedup until around 10 processors before dropping off. The second group performs much better and maintains a higher level of speedup for increasing number of processors. One reason for the difference in performance is due to scalability limitations of the software protocol implementation. In scenarios that use multiple views with view clients (representing worker threads) spread between views, each view has fewer view clients and therefore reduces the client management overheads, and effects of latency between clients for a consistency protocol [27]. In general, multiple views distribute the management overhead when large numbers of view clients are present.

Furthermore, network bandwidth utilisation, particularly on contented network links can have a direct impact on performance. Figure 7.11 compares the incoming and outgoing bandwidth utilisation between clusters for single-strict view and dual-strict multi-mrmw view configurations when running matrix multiply across 16 processors. While the bandwidth utilisation is much lower than



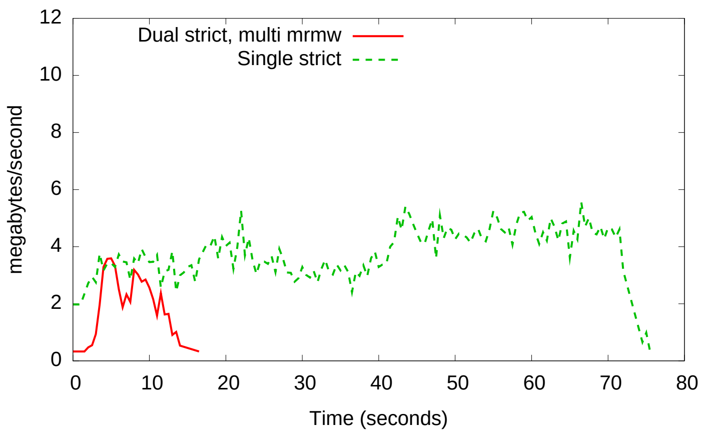
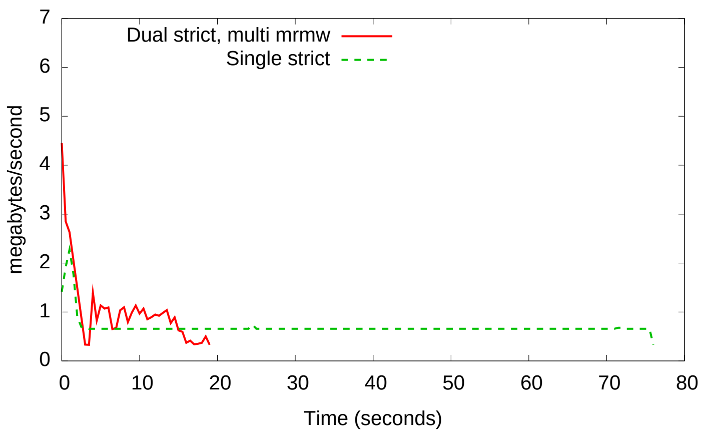


Figure 7.11: Network bandwidth utilisation into (top) and out of (bottom) cluster 2 for matrix multiply across 16 processors.

the dual-machine case in Section 7.2.1, we see significant communication for the single-strict view configuration due to false sharing. The effects of false sharing are reduced for the dual-strict dual-multi view configuration as most requests can be handled in each locality domain.



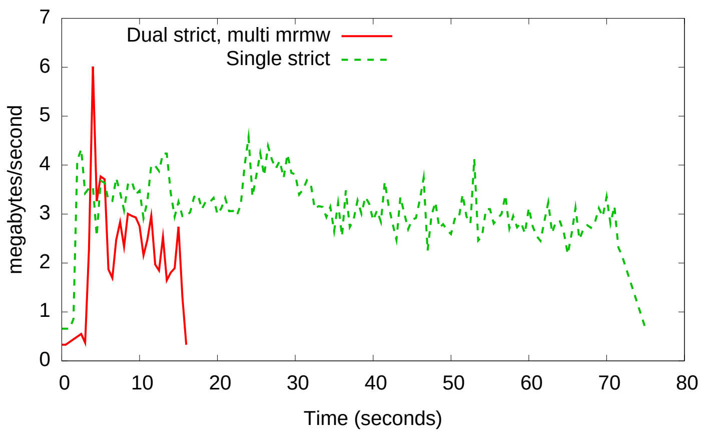


Figure 7.12: Network bandwidth utilisation into (top) and out of (bottom) a dual SMP processor node for matrix multiply across 16 processors.

The bandwidth utilisation for a single node in cluster 1 is illustrated in Figure 7.12. Interestingly, the bandwidth utilisation for this node is comparable to the communication between clusters.

The smaller improvements to speedup for the dual/multi view configurations as we increase the number of nodes is largely due to a lack of load balancing in the application, which divides work to be done evenly. While increasing the number of processors to execute the application reduces the amount of work do be done by each processor, the faster nodes of cluster 2 complete sooner and therefore contribute less to a reduction in run-time.

Differences in speedup between each view configuration can be best explained by examining the number of data operations that occur for each view configuration. Strict consistency relies on a centralised view pager, called the *primary view pager*, to resolve requests that other nodes propagate requests to when necessary. Hence, we examine the number of request operations that are processed by this view pager in Figure 7.13.

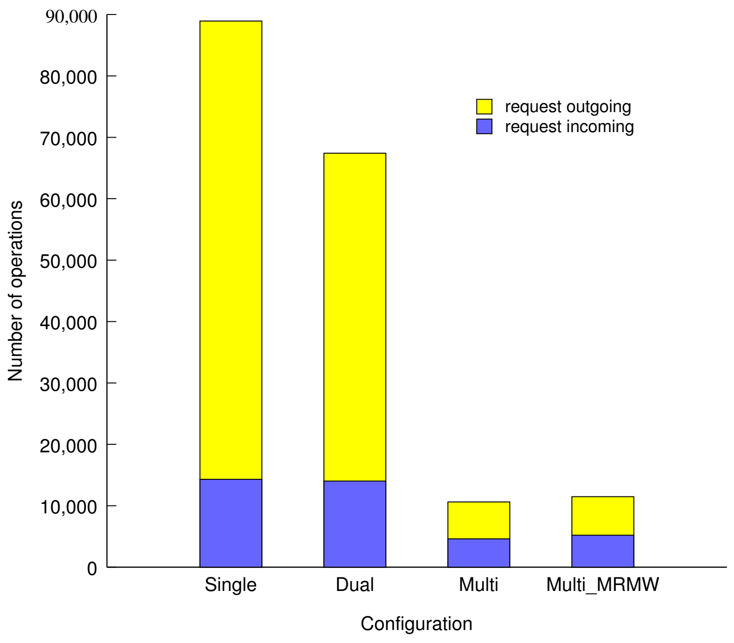


Figure 7.13: Number of requests in and out of the primary node for different view configuration scenarios. A reduction in operations typically corresponds to improved performance.

In the single view configuration, the primary view pager is required to manage the data operations for all worker threads in accordance with the strict consistency protocol. This results in a high number of incoming and outgoing requests. In particular, the large number of outgoing requests is the result of read-only data revocations from multiple readers along with some contention from false sharing. This occured mostly for matrix B, and is evident in the data access plot in Figure 7.6.

The introduction of dual views for locality reduces the number of outgoing requests, however there is only a small reduction in incoming requests for data. The reduction comes from cluster 2 which now handles many requests internally.

The reduction in number of operations for each successive view configuration corresponds well to their measured speedup shown in Figure 7.10. In particular, dual-strict multi-strict significantly reduces the number of requests.

The speedup of dual-strict multi-strict and dual-strict multi-mrmw are similar with a small increase in operations for the latter. These are due to differences at run-time and vary with each run. In this case, the lower overheads when using MRMW allow for slightly better speedup with the added operations attributed to a small increase in false sharing.

Configuration	% handled locally	# outgoing	# internal comms
Single strict	0%	-	-
Dual strict	24.5%	4629	5338
Dual-strict multi-strict	59.3%	1854	1368
Dual-strict multi-mrmw	50.0%	2031	904

Table 7.1: Number of requests handled locally for different view configurations when running a matrix multiply.

Analysis of the operations on the primary view pager only provides part of the picture on the benefit of views. For our multi-cluster scenarios, we are especially interested in the benefit of views that act as locality domains whereby they handle requests locally without external communication. Table 7.1 presents the locality effects of views. The first column shows the percentage of requests from within cluster 2 that are handled internally. The remainder are sent over the link to cluster 1 to be handled. The more requests handled locally means lower average latency and lower bandwidth utilisation across the link between clusters. Hence, for a single strict view spanning all processors of both clusters, no requests are handled locally with cluster 2 as the primary pager lives on cluster 1.

The second column shows the number of requests sent out of cluster 2. The *multi* view configurations that establish a view per node generate less than half the outgoing cluster 2 operations. The third column shows the number of communication requests sent within cluster 2.

7.2.2 Red-black successive over-relaxation

Red-Black Successive Over-Relaxation (SOR) is a method for solving a linear system of equations which divides a red and black array into equal sets of rows, each of which are assigned to a different worker thread. The arrays are allocated in shared memory and access synchronised using barriers.

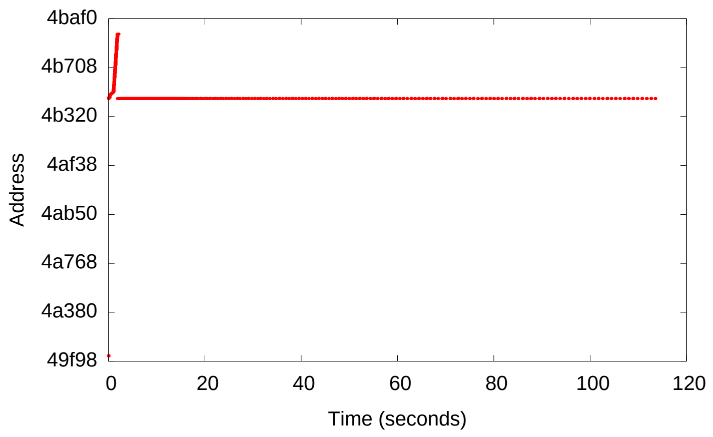


Figure 7.14: Data access pattern for a single client when running SOR across 8 processors.

This benchmark is run on a 4000x2000 matrix of floating-point numbers for 100 iterations. As the row size is not a multiple of hardware page size, false sharing between processors will occur at adjacent rows as these rows will share a page.

The work is divided up equally among worker threads, which on some systems leads to load imbalance. For example, machines with faster processors are likely to complete their work sooner than slower machines and will be idle for the remaining time.

The data access pattern for a single worker thread running SOR is illustrated by Figure 7.14. It is dominated by access to a small region of matrix data for the majority of a benchmark run. Communication normally occurs when the algorithm crosses row boundaries and when any false sharing is present. SOR has a high computation-to-communication ratio.

Figure 7.15 shows good speedup improvements for increasing number of pro-

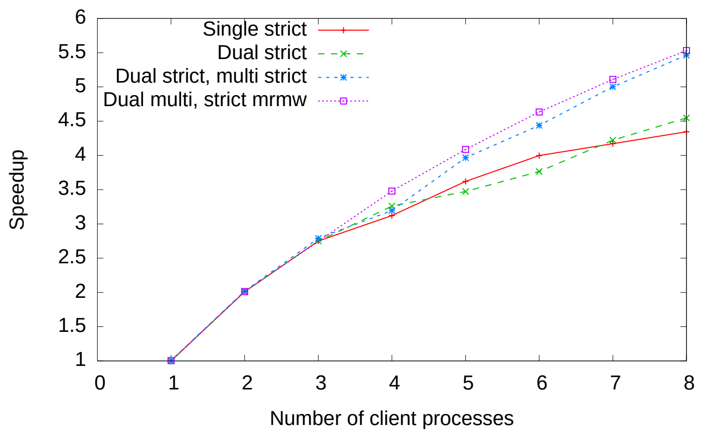


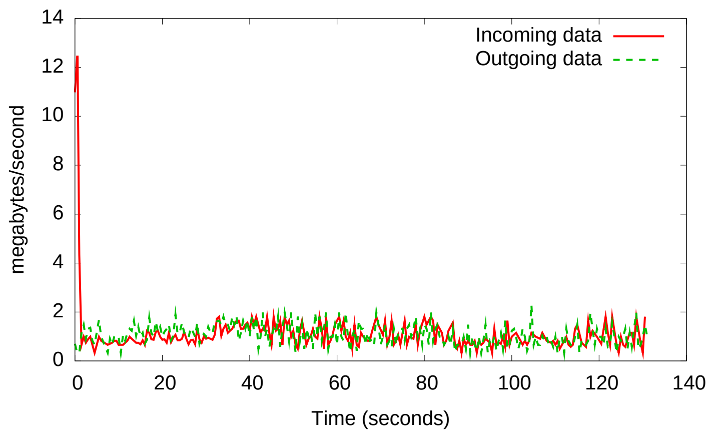
Figure 7.15: Execution time speedup of SOR benchmark when run in different view configurations across the processors of two 4-way ccNUMA machines.

cessors across all view configurations. SOR benefits most from view configurations that provide locality domains which group sets of processors of each cc-NUMA node together.

View configuration *dual-strict multi-strict* outperforms *dual-strict* by grouping processors together as it benefits from lower overheads of communication via a single view pager for each multi view.

Locality domains reduce the bandwidth utilisation between nodes, as illustrated by Figure 7.16. This figure shows the bandwidth utilisation between cc-NUMA nodes for the single view and the dual-strict multi-mrmw view configuration. In both cases bandwidth utilisation remains low for the duration of the benchmark run leading to the high computation-to-communication ratio and good speedup achieved by SOR. As expected, dual-strict multi-mrmw has lower bandwidth utilisation due to the amortisation of data requests and replies between nodes.

The introduction of MRMW for the dual-strict multi-mrmw view configuration also slightly improves speedup beyond that of dual-strict multi-strict. This is due to a reduction in overheads for handling page faults for the processors of each ccNUMA node that is normally required for software-based protocols.



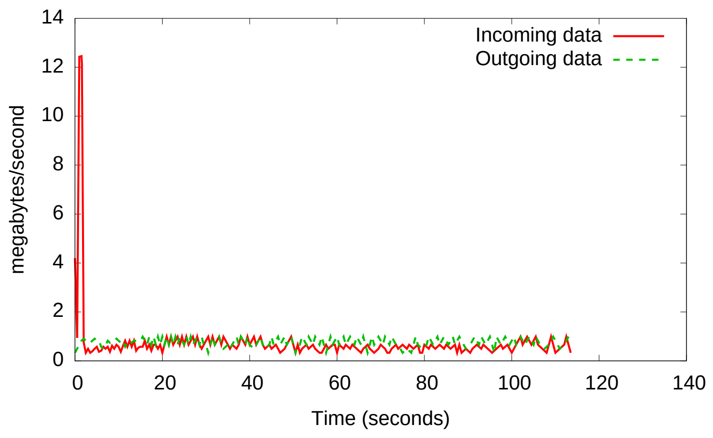


Figure 7.16: Bandwidth utilisation between ccNUMA machines for a single view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running SOR.

7.2.3 3D-FFT

The 3-D Fast-fourier transform (FFT) benchmark from TreadMarks solves a partial differential equation using three dimensional forward and inverse FFTs. To perform the 3-D FFT, this benchmark performs a number of 1-D FFT and array transpositions. Between iterations, a global memory barrier is used to synchronise processors.

This benchmark was configured with parameter $M = 7$ in order to perform the 3-D FFT over a 128x128x128 array of double precision complex numbers for four iterations. A 3-D FFT for a data-set of this size executes in 41 seconds on a

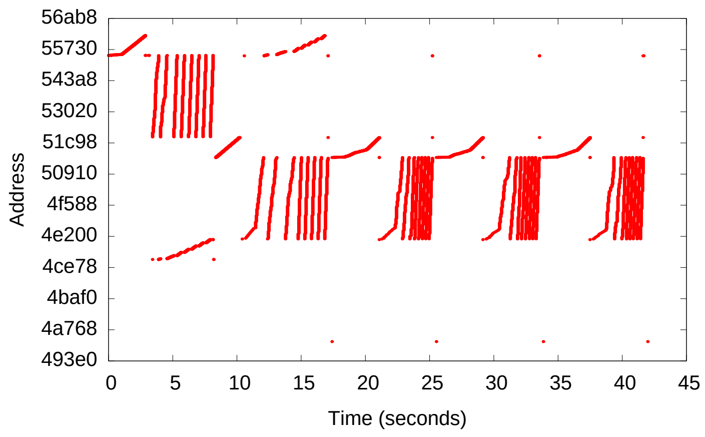


Figure 7.17: Data access pattern for processor five running FFT.

single Itanium processor.

The data access pattern of a worker thread running FFT on processor five is shown in Figure 7.17. FFT worker threads walk iteratively over the arrays present in shared memory accessing a large amount of shared memory quickly, creating the vertical stripes of the data access pattern. This data access pattern places a significant burden on memory management, impacting the overall scalability of this application.

The speedups for different view configurations are shown in Figure 7.18. For optimised view scenarios including *dual-strict multi-strict* and *dual-strict multi-mrmw*, there is good speedup while the worker threads run exclusively on a single ccNUMA machine (up to 4 worker threads). From 5 worker threads, communication over TCP/IP impedes performance reducing any speedup benefits.

A comparison of bandwidth utilisation for single and *dual-strict multi-mrmw* view configurations between machines is shown by Figure 7.19. Interestingly, the bandwidth utilisation for single view is lower than dual-strict multi-mrmw indicating that the communication bottleneck between machines is not the main limiting performance factor for single view configuration. The speedup of the single view configuration for worker threads on the same machine (less than 5 worker threads) also indicates that communication between machines is not the primary bottleneck limiting performance improvements. For dual-strict multi-mrmw we

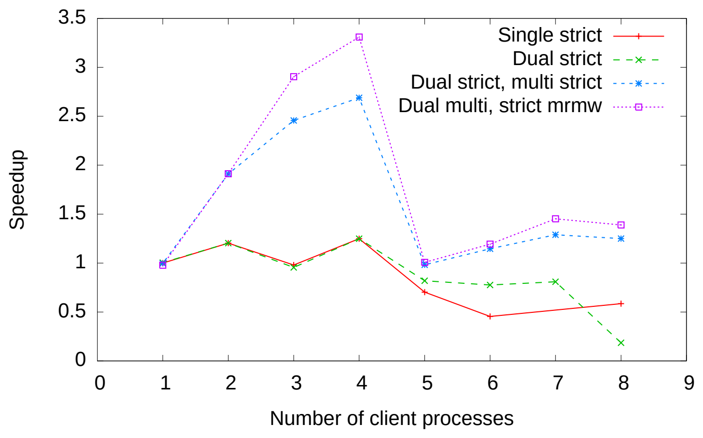


Figure 7.18: Execution time speedup of FFT when run in different view configurations across the processors of two 4-way ccNUMA machines.

see the communication link is almost always saturated with bursty traffic patterns.

The single and dual view configurations perform poorly, even on a single ccNUMA machine. These results show that the communication overheads are significant for this benchmark and are the limiting factor for scalability of this benchmark.

Figure 7.20 shows the speedup for our twenty processor multi-cluster environment. As expected from the previous dual machine results, there is no speedup achieved beyond 4 worker threads due to communication overheads.

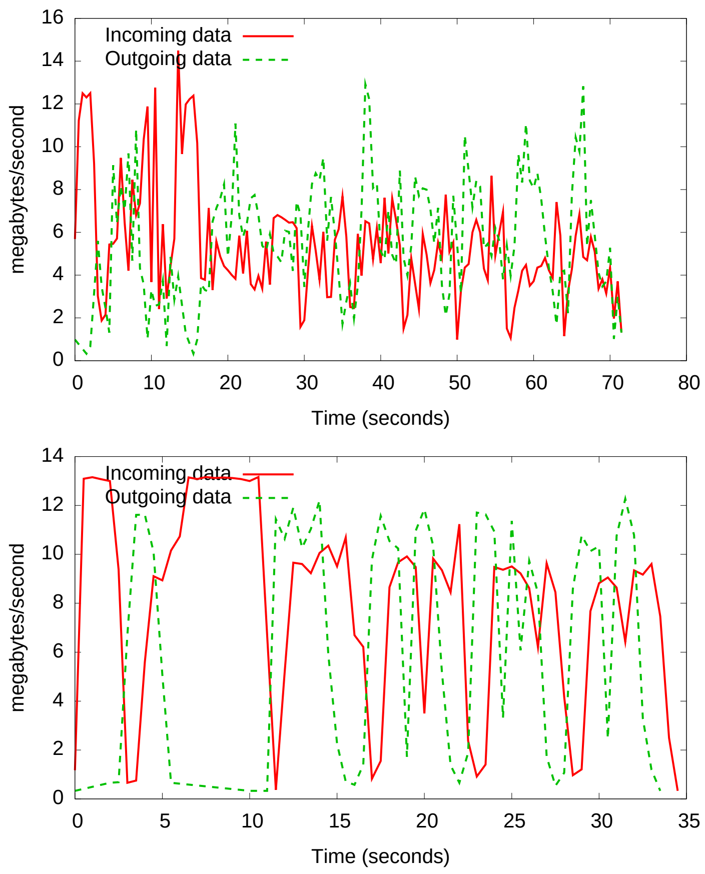


Figure 7.19: Bandwidth utilisation between ccNUMA machines for a single view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running FFT.

7.2.4 Parallel scan

Parallel prefix algorithms allow us to parallelise a serial algorithm at the cost of doing more work overall. The problem is distributed across several nodes using a divide and conquer approach, leading to a reduction in overall execution time.

Some serial algorithms that are difficult to parallelise are often parallel prefix algorthims. Consider a book where we know the size of each chapter. If we want to calculate the offset from the start of the book for each chapter, we have a parallel prefix problem. This problem is solved serially by summing the size of each chapter to calculate the offsets.

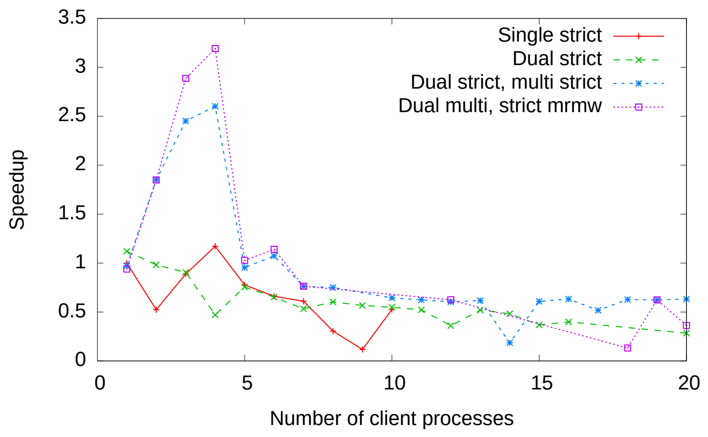


Figure 7.20: Execution time speedup of FFT when run in different view configurations across the processors of a multi-cluster.

The parallel prefix algorithm [15] calculates an associative function, f, on all prefixes of an n-element array. That is, $s[0]$, $f(s[0], s[1])$, $f(s[0], f(s[1], s[2]))$, ..., $f(s[0], ...f(s[n-2], s[n-1])...)$, which may be executed using $\Theta(n)$ processors in $\Theta(\log n)$ time.

```
for j = 0 to log2(n-1) do
    for i = 2j to n-1 parallel-do
        s[i] = f(s[i-2j], s[i])
```

The inner loop is executed across all nodes in parallel. The n-element array is divided equally between all nodes. For each inner loop iteration, each node updates the array for elements that fall within each division.

Since this algorithm above overwrites the elements of the matrix on each iteration, each node makes a local copy of the array elements it requires before each inner loop cycle.

For benchmarking, we use a hand-crafted implementation of the parallel prefix algorithm described above called *pscan* (short for parallel scan). The data access pattern for a client running pscan is shown in Figure 7.21. Compared to 3D-FFT data access is less demanding for pscan.

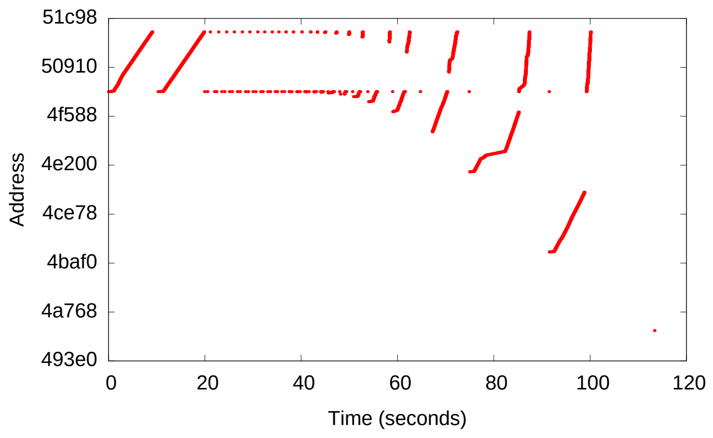


Figure 7.21: Data access pattern for a client running parallel-scan.

The speedup results for each view configuration in a dual ccNUMA environment are shown in Figure 7.22. Overall, there are speedup improvements as the number of worker threads on the system increases.

For this benchmark there is a degradation impact to speedup once communication is required across the network link between machines. This is the case for all view configurations, however the speedup numbers recover slightly for all configurations except the single view configuration.

A comparison of bandwidth utilisation between nodes for single view and dual-strict multi-mrmw view configurations in Figure 7.23 indicates that at times the benchmark is limited by bandwidth. However, the network is not saturated.

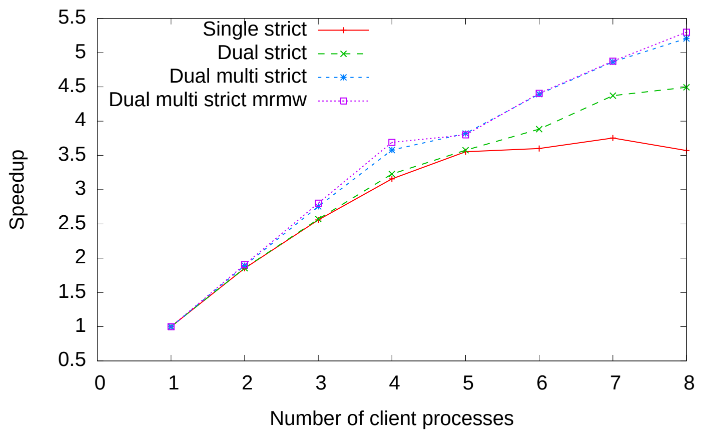


Figure 7.22: Execution time speedup of parallel-scan when run in different view configurations across the processors of two 4-way ccNUMA machines.

7.2.5 TSP

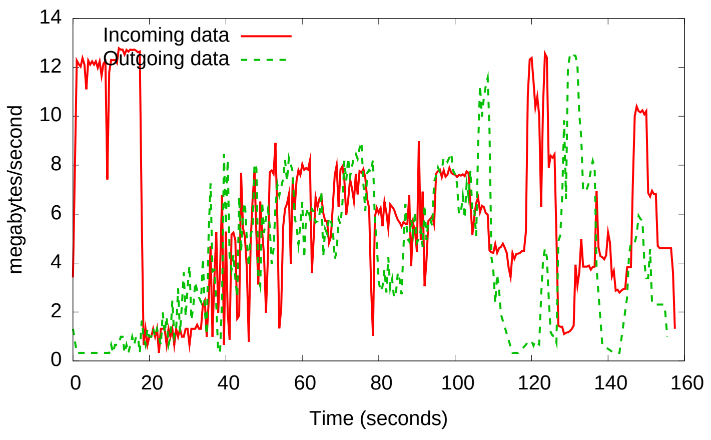
The traveling salesman problem (TSP) [60] involves discovering the shortest route which visits all cities and returns to the starting point. Each possible intermediate solution is called a *tour*. The discovery of the shortest tour is classified as an NP-Hard problem.

This benchmark implements TSP using a priority queue that contains partially evaluated tours. A lock protects access to the queue. A worker thread runs on each processor and attempts to lock the queue to acquire a partially evaluated tour to be solved.

Data sharing is required for most data structures used by TSP, including the priority queue and an array of structures for maintaining tour state. The layout of these structures provide the potential for false sharing.

We solve a 19 city problem using the input data set `19b` provided with the TreadMarks implementation of TSP. Figure 7.24 shows the irregular data pattern for a client running TSP. The same data pages are accessed several times, indicating potential false sharing issues of the application. However, with the use of HLRC consistency protocol, false sharing during each iteration is avoided.

The speedup results for TSP are shown in Figure 7.25. The first view configuration shows a single HLRC view running across all processors of two ccNUMA



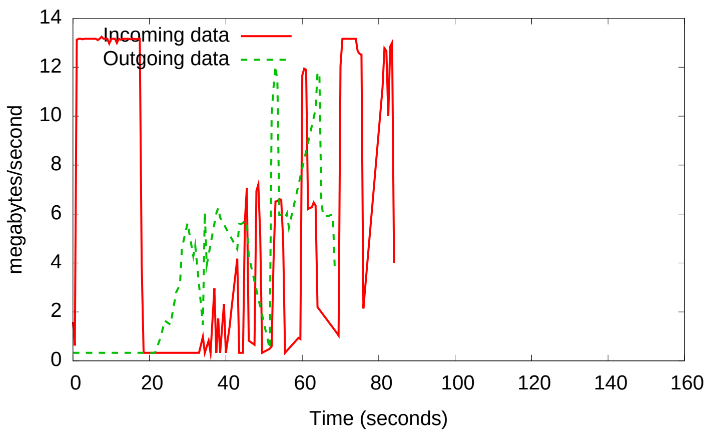


Figure 7.23: Bandwidth utilisation between ccNUMA machines for a single view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running parallel scan.

machines. There is very little speedup increase (or decrease) in this view configuration which was found to be due to lock contention. That is, most worker threads spent their time waiting to access the work queue, which is protected by a lock. The path discovery algorithm runs quick enough such that the benefits of parallelism for this data set are minimal. It is likely that a higher computation requirement of this benchmark by using more complex route data set would produce a greater speedup by offsetting locking costs.

The next view configuration shows a dual-strict view scenario. The speedup does not improve as the number of processors used by the benchmark increases, even on a single ccNUMA node with fast interconnects and large communication

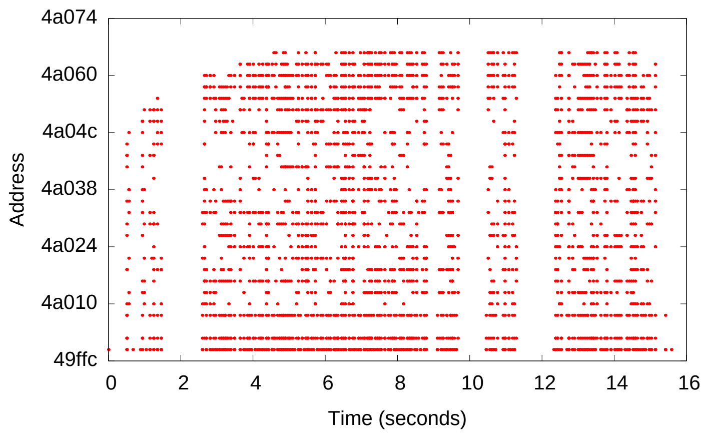


Figure 7.24: Data access pattern for a single client when running TSP across 8 processors.

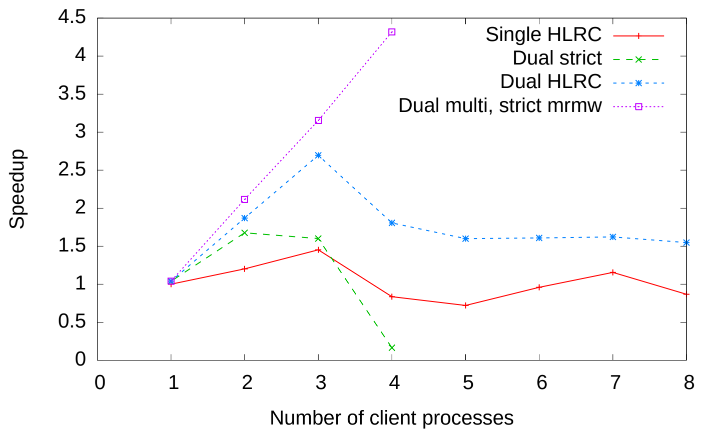


Figure 7.25: Execution time speedup of TSP when run in different view configurations across the processors of two 4-way ccNUMA machines.

bandwidth. This is caused by high contention due to false sharing of data pages. Hence, while HLRC does not perform well for this benchmark, it does alleviate some problems prevalent with strict consistency that would be beneficial to other applications.

The next view configuration uses dual HLRC views and shows performance improvements over a single HLRC view. Initally, speedup is almost linear as the number of worker threads increases, however lock contention impacts this benchmark beyond 3 worker threads. Figure 7.26 compares the bandwidth usage for the single HLRC and dual HLRC view configurations. There is significantly lower bandwidth utilisation in the dual HLRC case, which contributes to the overall speedup improvement.

Finally, the last view configuration uses MRMW on a single ccNUMA node. There is linear speedup as all communication occurs without software intervention. This configuration was not run beyond a single node, as MRMW does not perform correctly beyond a single node without being coupled with another view (and would therefore have a software overhead).

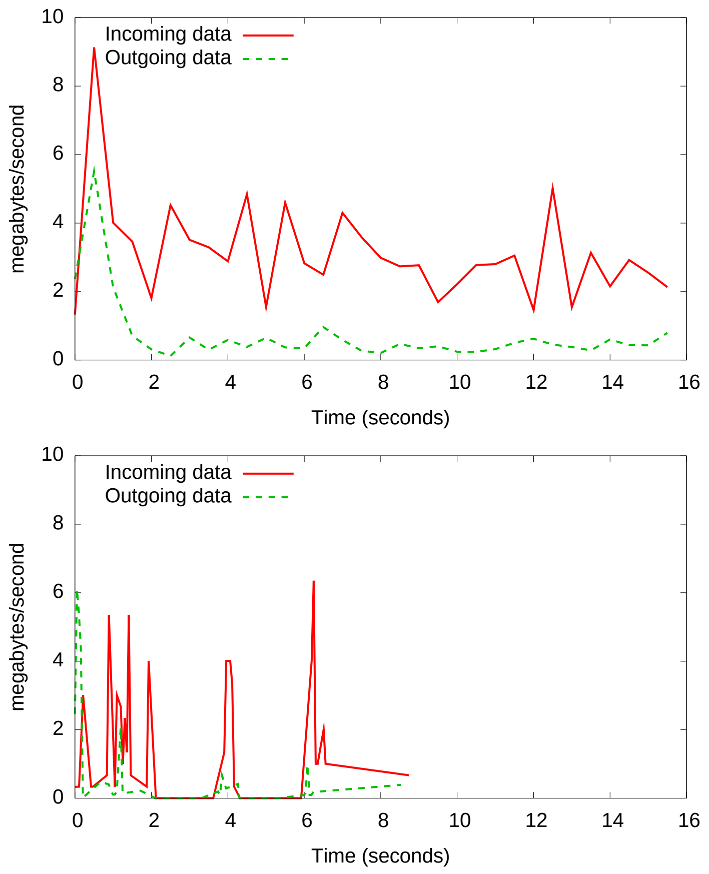


Figure 7.26: Bandwidth utilisation between ccNUMA machines for a single HLRC view configuration (top) and dual-HLRC view configuration (bottom) when running TSP.

7.2.6 Quick sort

Quick sort (QS) employs a divide and conquer approach to recursively sort an unsorted input list using on average $\Theta(n \log n)$ comparisons to sort n items. This is achieved by dividing the unsorted input list into sub-lists, which are then solved recursively and in parallel by worker threads.

The implementation of this benchmark stores sub-lists on a task stack. The task stack is provided in shared memory and accessed by worker threads running on each node. Once a sub-list is processed, it is returned back to the task stack. Access to the task stack is protected by a lock.

For this benchmark, we sort 10 million integers with a default bubble threshold of 1024. The bubble threshold indicates the point from which work is distributed to other processors. This threshold limit is small for today's systems and likely to significantly degrade benchmark performance.

The data access pattern for a client is shown by Figure 7.27. This access pattern illustrates low demand for data, with some false sharing across time.

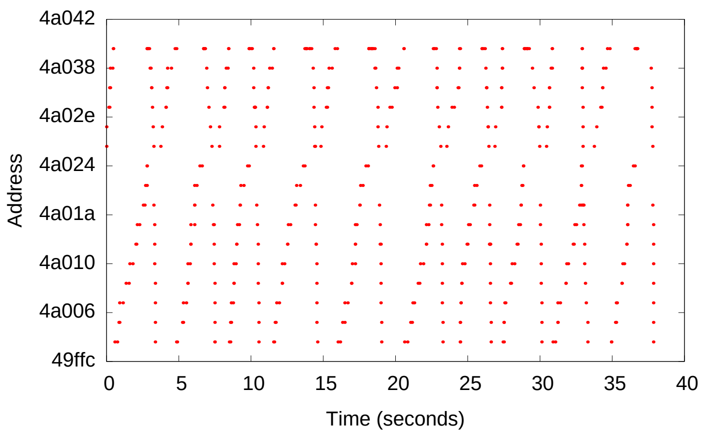


Figure 7.27: Data access pattern for a single client when running QS across 8 processors.

Figure 7.28 shows the speedup for each view configuration. The speedup results are similar to other benchmarks where we see that as the number of clients increase performance degrades. In the case of this benchmark, we see performance benefits on a single node when using the *dual-strict multi-strict/mrmw* view con-

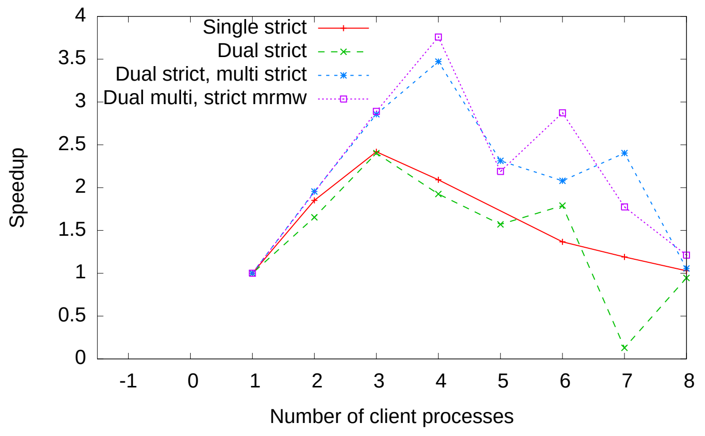


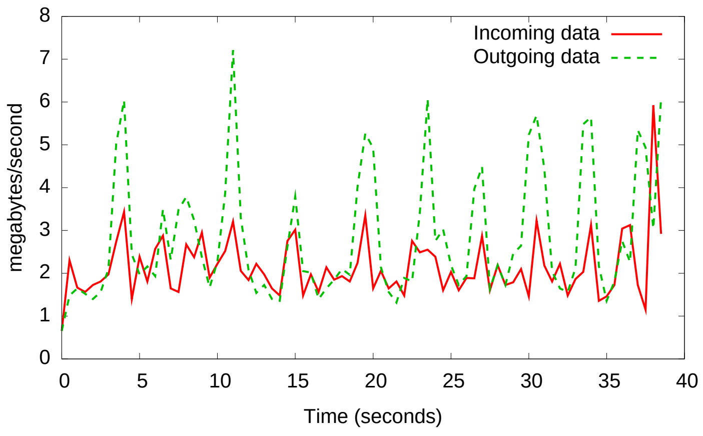
Figure 7.28: Execution time speedup of quick sort when run in different view configurations across the processors of two 4-way ccNUMA machines.

figurations however, once communication across the network is required, speedup drops. Locking for update to the shared quick sort list is the main contributing factor to latency and poor speed up in each view configuration, however the point of degradation varies based on the level of adaption of each view configuration. Poor performance of this application when network communication is required can be alleviated by reconfiguring the application to perform a larger amount of local processing on each node by altering the bubble threshold. This is performed for the integer sort application in Section 7.2.7, which exhibits similar performance when not reconfigured.

The communication bandwidth utilisation illustrated by Figure 7.29 shows bursts of traffic that correspond to the client doing work requiring communication, then waiting for the lock to update its results. As expected, the single view configuration has a higher bandwidth utilisation, however the communication link is not saturated.

7.2.7 Integer sort

Integer Sort (IS) ranks an unsorted list of keys using a bucket sort. In this implementation, each worker thread has a private array of buckets and shares access to a shared array of buckets. Each worker thread performs the sort on its own private



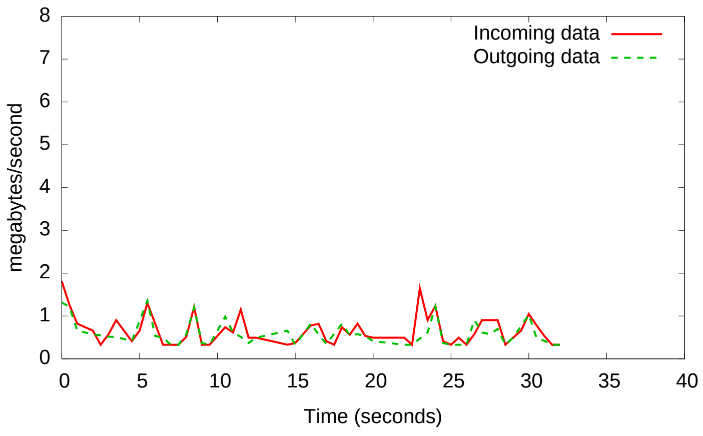


Figure 7.29: Bandwidth utilisation between ccNUMA machines for a single view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running QS.

array, before adding its values to the shared array. Synchronisation of the shared array is protected by a lock. Barriers are also used for each iteration performed by worker threads.

This benchmark sorted 2^{26} keys ranging from 0 to 2^{16} on each worker thread for 10 iterations of the algorithm.

The data access pattern for IS is shown in Figure 7.30. This access pattern appears almost identical to the QS benchmark and has similar performance results when the IS application uses a smaller worker-thread key assignment. The performance results for IS is shown by Figure 7.31. There are good speedup improvements when using multiple views. In particular, dual-strict multi-strict

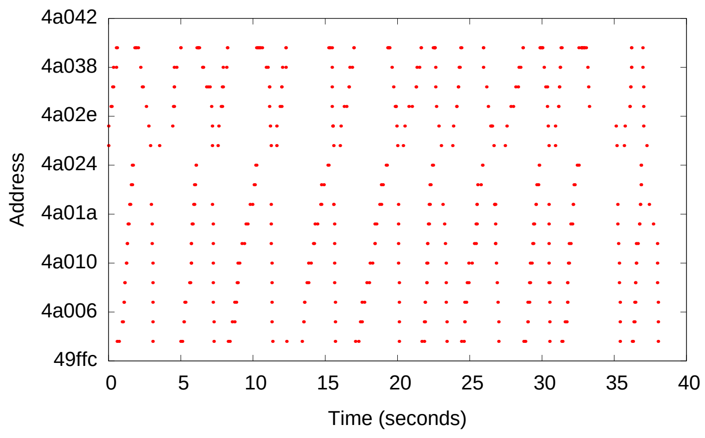


Figure 7.30: Data access pattern for a single client when running IS across 8 processors.

outperforms the other view configurations.

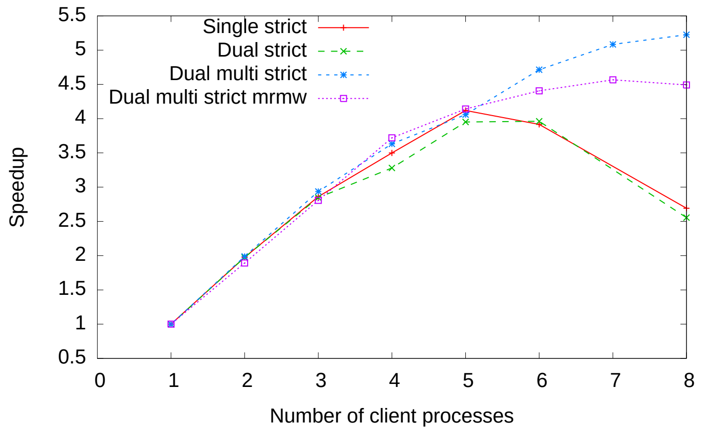


Figure 7.31: Execution time speedup of integer sort when run in different view configurations across the processors of two 4-way ccNUMA machines.

Network bandwidth utilisation between clusters (ccNUMA nodes) when running with 8 client processes is shown in Figure 7.32. On each interation of the benchmark, shared data is copied locally to each node. This corresponds to the peaks in the bandwidth utilisation graphs and is many-to-many communication.

In wide-area and grid environments this benchmark is best configured by increasing the number of keys processed by each node on each iteration in order to amortise the latency of barrier synchronisation required on each iteration. Our experimentation with these parameters increased the key range processed on each worker thread from 2^{15} to 2^{16} to accommodate this for our twenty processor multi-cluster environment with processors capable of processing that key range quickly.

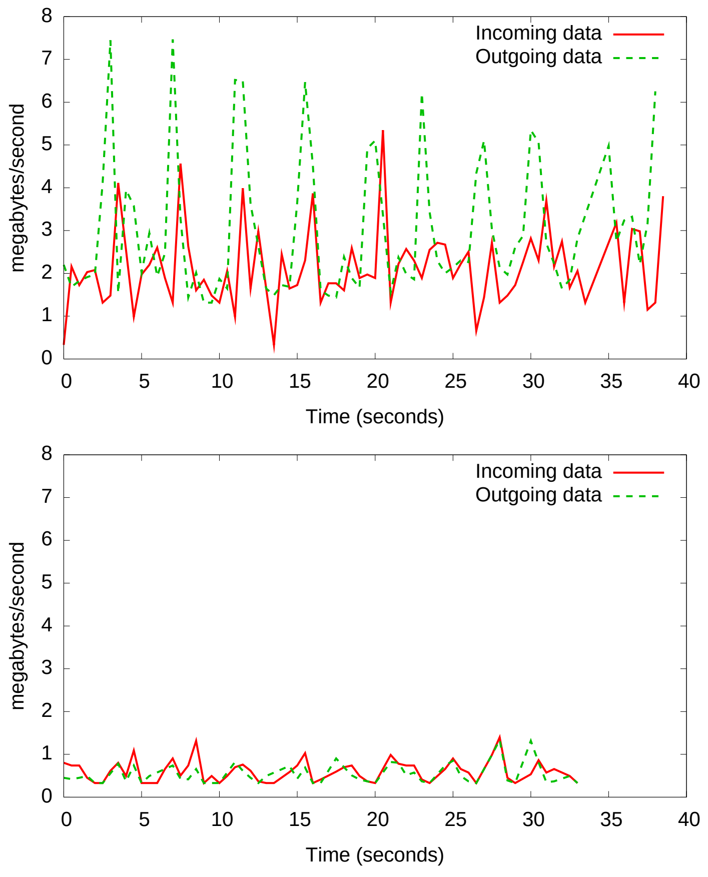


Figure 7.32: Bandwidth utilisation between ccNUMA machines for a single view configuration (top) and dual-strict multi-mrmw view configuration (bottom) when running IS.

7.2.8 Single-sided MPI micro-benchmarks

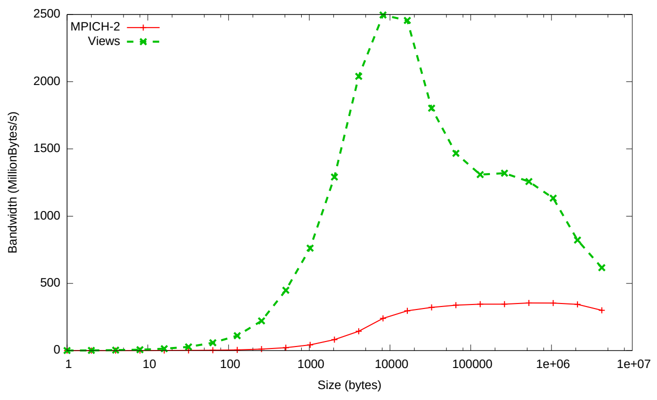
The purpose of the micro-benchmarks in this section is two-fold. Firstly, to demonstrate the use of views in an alternative programming model. Secondly, to clearly illustrate the impact of views for simple data operations compared to that of MPICH-2, a popular MPI environment that implements one-sided operations. Furthermore, using this benchmark we can evaluate one-to-many operations for both view and MPICH2 configurations.

To examine throughput, latency and a variety of view configuration scenarios, these micro-benchmarks are based on the following `MPI_Get()` bandwidth benchmark which is a modified OSU MPI bandwidth benchmark [50]:

```
win = MPI_Win_Create(window_buf, window_size)
for (s=1; s <= max; s *= 2) do
    if my_id == 0 then
         write_random_data_to_buf(window_buf)
    MPI_Win_fence(win)
    if my_id != 0 then
       MPI_Get(local_buf, s, win)
    MPI_Win_fence(win)
```

This benchmark simulates distributing data from a data producer to one or more data consumers using MPI single-sided operations. Firstly, it creates a shared memory window using `MPI_Win_Create`. It then iterates using varying data message size. On each iteration, one client writes random data into this window, which is then read by the other clients. The benchmarks utilise `MPI_Win_fence` for synchronisation of all participating worker threads.

MPICH2 has an optimised point-to-point TCP communication library. For single-sided MPI operations, there is no support for multi-cast communication. Likewise, Eidolon uses only point-to-point communication for communication over TCP/IP. However, view configuration scenarios that use multiple views may alleviate this drawback through amortization of communication across criticial network links on view boundaries. The Eidolon implementation is described in Section 6.2.



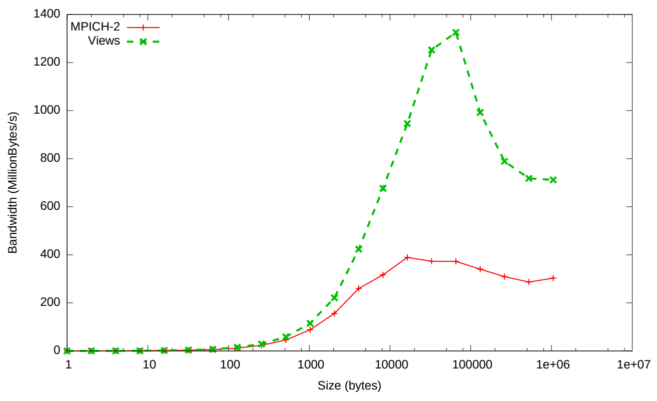


Figure 7.33: Single-sided MPI_Get (top) and MPI_Put (bottom) bandwidth

The first set of experiments compares the bandwidth of `MPI_Get` between two clients running on MPICH-2 and on views. Included in this set of experiments is `MPI_Put` which performs data transfer in the reverse direction. These experiments are executed on a single Itanium ccNUMA machine using the shared memory communication mechanisms available to each middleware implementation. After an initial warm up, the `MPI_Get` benchmark performs several iterations of `MPI_Get` specifying new regions of the window on each call. This ensures that new data is fetched, guaranteeing that communication occurs in the view implementation which may otherwise fetch a local, but consistent copy of the data. The results for this experiment using both `MPI_Get` and `MPI_Put` are shown in Figure 7.33.

The bandwidth available for the benchmark running on MPICH-2 is limited due to the overheads of the implementation. While MPICH-2 supports communication over shared memory, it does so by performing two-sided message passing of the message in shared memory. The bandwidth available when using Eidolon views is only limited by the system memory and bus bandwidth between processors.

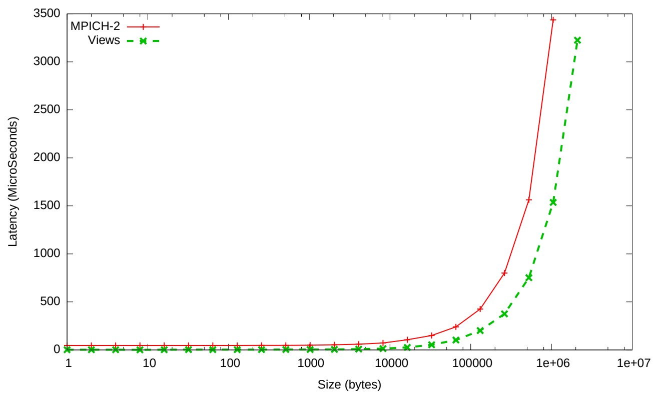


Figure 7.34: Single-sided MPI_Get latency.

The next benchmark measures latency of the `MPI_Get` operation, shown in Figure 7.34. The view performance results for this benchmark show that the view pager implementation for shared memory systems has very low latency. For small messages the measured latency is around 3 microseconds as the only overhead is the cost of transferring the data from memory or the remote processor's cache. The MPICH-2 results show the performance of the optimised two-sided communication implementation. For small message sizes the latency is 46 microseconds which is quite low when considering the synchronisation requirements of the implementation.

The next benchmark simulates a scenario where a one-sided MPI operation is used to transfer the same data set to multiple client processes. Figure 7.35 shows

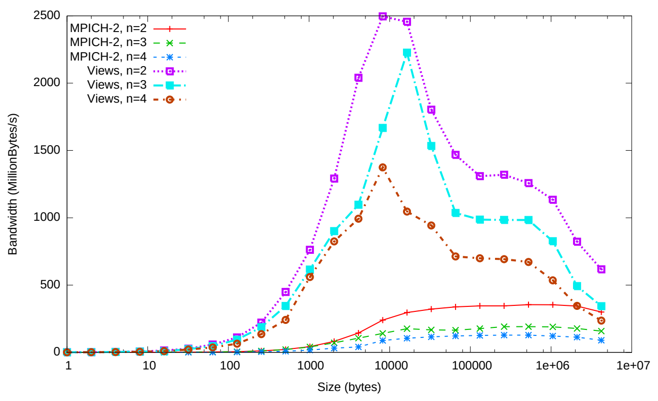


Figure 7.35: Single-sided MPI get multi-point on a 4-way ccNUMA machine.

the achieved throughput for client processes when there are multiple processes performing a `MPI_Get` on a single 4-way SMP system.

The MPICH-2 implementation sends each data message to each client via point-to-point connections. Hence, the bandwidth available to each client is reduced as the number of clients on the system increases. The view implementation performs much better, however as the number of clients increase, the available system bandwidth for each client is proportionally lower. Peak throughput occurs when the benchmark data fits within the shared system cache. Beyond this point, each processor gets a proportional share of the system's memory bandwidth. This degrades as the working set grows beyond cache sizes.

The final benchmark examines a dual four-way ccNUMA machine environment outlined in Section 7.1. This environment forms a small heterogeneous multi-cluster where communication between clusters (each ccNUMA machine) occurs via TCP, and ideally, internal system communication occurs via shared memory. In this benchmark the number of clients used for each run is split evenly across both systems.

The results in Figure 7.36 show that bandwidth scales better with number of processes for the view implementation than the MPICH-2 implementation. This demonstrates the ability of views to encapsulate groups of clients into specific sub-environments and to optimise communication between each sub-environment.

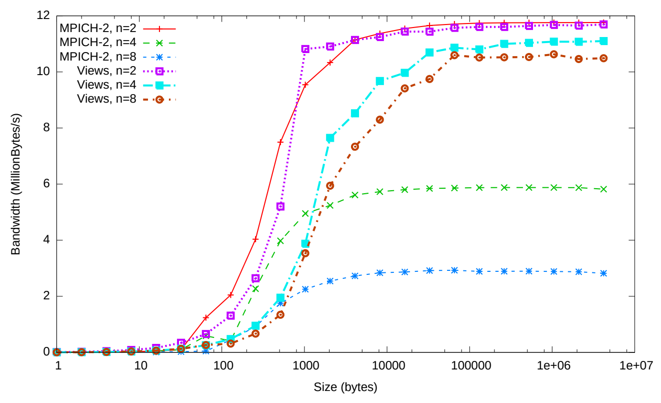


Figure 7.36: Single-sided MPI get multi-point on two 2-way ccNUMA machines.

The current view implementation for TCP suffers from additional latency as each request for data currently blocks. For small messages we also see additional overhead as the number of processes increases due to a poor barrier implementation that does not scale well. Both these problems are implementation issues that are straightforward to improve with a more complete implementation. Note that the clients running on the same system as the target of the `MPI_Get` calls will complete much sooner than the remote nodes as they have the local bandwidth availability shown in Figure 7.35.

7.3 Ease of use

In Section 7.2 we demonstrated the performance and resource usage benefits that the view model can provide to applications running in diverse environments. Other approaches to improving performance and resource usage of the application include direct modification of the application or underlying middleware implementation using approaches discussed in Section 2.2. These modifications are time consuming and may be infeasible on dynamic systems where it is difficult to know what modifications are required prior to run-time. However, these alternative approaches are somewhat orthogonal to the approaches available when using the view model. That is, the user is free to devote resources to improving the application using other available techniques regardless of their use of the view model. In this section we argue that the approaches available when using the model require less effort by the user than the effort required to adapt applications through other approaches.

We believe that most, if not all distributed applications can be adapted to the view model through an implementation such as Eidolon by implementing the application's API using view interface calls that are described in Section 4.1.1.

An application may invoke view interface calls explicitly, however this requires it to have knowledge about its run-time system in order for it to define a suitable view hierarchy. Alternately, Eidolon provides a client support library that transparently handles non-overlapping, overlapping and mapped view calls for any client application, or a programming model API implementation. These approaches are discussed further in Chapter 8.

Table 7.2 outlines the effort to port a variety of different distributed applications and middleware components to Eidolon. These are discussed separately below.

7.3.1 Simple process-based applications

The matrix multiply application presented in Section 7.2.1 was developed into two implementations. The first is a traditional multi-threaded application suitable for running on an shared-memory multi-processor. The second implementation provides a cluster-based, distributed shared memory. In both cases the application relies on a global shared memory for matrix data only and uses barriers for synchronisation between worker threads.

The number of modifications to directly adapt matrix multiply, which does not use a pre-defined API, was 5 out of 400 lines of code which consisted of calls to initialise the application and changes to memory allocation and barriers.

Matrix multiply in Eidolon makes use of Eidolon's client support library. Global memory allocation using *malloc* is replaced by *client_obtain_view* calls. This call invokes *view_create* and *view_select* view interface calls on behalf of the client, and manages view hierarchies based on rules specified by the underlying run-time environment. A global barrier is used for synchronisation which is implemented in terms of *token_request* and *token_reply*.

7.3.2 API libraries

Other applications that are written on top of well-known APIs including the benchmarks from TreadMarks, Splash-2 and OSU's MPI-2 benchmarks do not require any modifications. These applications can be run immediately without any changes or effort from the user other than compiling and linking the application against Eidolon's TreadMarks client library.

Implementing a programming model API such as TreadMarks in terms of view interface operations is straightforward. The TreadMarks implementation for Eidolon was 235 lines of code. This implementation makes direct view interface calls using sequential consistent and release consistent view pagers.

7.3.3 New protocols

Providing an implementation of a new protocol view pager in Eidolon requires implementing the protocol in terms of view interface operations. The effort required depends on the complexity of the protocol. Compared to other middleware implementations, implementing view pagers in Eidolon means that the protocol implementation does not need to provide additional code for communication between nodes or client interface implementations as these are usually already provided and abstracted by the view model. This reduces code complexity and improves code reuse, which we argue reduces the effort required by the programmer.

7.3.4 New programming models

While it is not expected that a programming model needs to be developed that support views, we nevertheless examine the effort required to develop a new programming model in Eidolon.

New programming models require careful thought on how they map to view interface operations, and their expected behaviour. They require the implementation of view pagers, and in many cases the implementation of client interfaces and APIs. An implementation of one-sided MPI client interface is around 500 lines of

Porting application/protocol	Modifications	Time
Multi-threaded application: matrix multiply	5 of 400 lines	2 hours
TreadMarks applications: 3D-FFT, SOR, etc	0 lines	0 hours
TreadMarks API with shared memory, locks, barriers, etc	235 lines	8 hours
Simplified VODCA API with shared memory, locks, barriers, etc	100 lines	8 hours
New paging protocol: HLRC	2000 lines	5 days
New programming model: one-sided MPI	1500 lines	7 days

Table 7.2: Comparison of effort required to port different distributed application components to Eidolon.

code. A view pager implementation optimised for cluster environments is around 1000 lines of code. Note however, that our implementation supported the use of other shared memory view pager implementations such as strict and MRMW.

The simplified VODCA API was implemented using an application client-level library and did not require the implementation of new view pagers. As such, it was quickly implemented in only 100 lines of code. It may be possible to improve performance with an implementation of specialised view pagers that better map view operations to the behaviour of VODCA, however this was not performed as part of this thesis.

CHAPTER 8

USING EIDOLON VIEWS

The evaluation in Chapter 7 showed that the use of views results in better performance and resource usage (such as bandwidth utilisation) in run-time environments where we do not know details of the underlying environment, and in diverse heterogeneous environments that have a non-uniform or hierarchical network topology.

Use of views in these environments requires the construction of view hierarchies of overlapping views. Typically specification of views begins at the distributed application (or client library) which specifies what shared memory regions are to be created using non-overlapping views. Each of these views may then be abstracted by overlapping or mapped views.

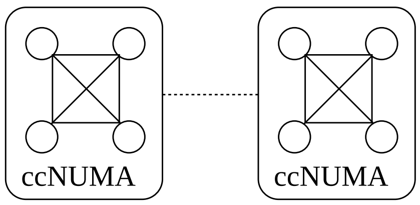


Figure 8.1: Eight processor dual ccNUMA environment.

Consider the dual ccNUMA environment illustrated in Figure 8.1 that was also used as a benchmarking run-time environment in Section 7.1. Applications such as matrix multiply create several shared memory regions, each with a special purpose. In this case, one region for read-only source matrix data, and another for the computed result matrix. The creation of views in this scenario is dependant on the approach taken.

8.1 Explicit specification of views in application

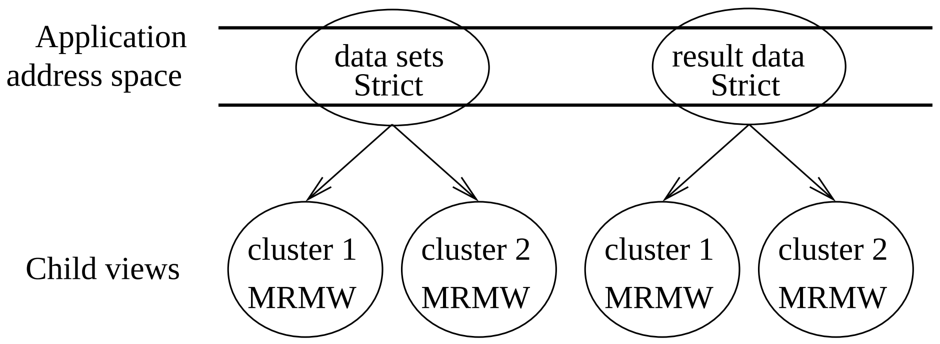


Figure 8.2: Matrix multiply application with two shared memory regions and its view configuration

One approach to allocating views and creating view hierarchies is to explicitly define them in the application by using a combination of `view_create` calls. Firstly the allocation of these regions would invoke the creation of (for argument's sake) two non-overlapping views illustrated in Figure 8.2, one for the read-only data sets and one for the computation and result data. Any subsequent overlapping views would be dependent on the run-time environment. Explicit specification of these would require knowledge about the environment prior to run-time. Hence, in this case for each region we create a child view. Each child view encompasses data sharing on each ccNUMA machine. The explicit creation of the source data views in this scenario is shown below:

```
/* create view for source data matrices,
 * which are allocated together.
 */

if (my_node_id() == 0) {
    v_src = view_create(matrix_src_address,
                            sizeof (matrix_src_data),
                            NO_PARENT, STRICT_CONSISTENCY);
    v_src_cluster_1 = view_create(matrix_src_address,
                               sizeof (matrix_src_data),
                               v_src, MRMW_CONSISTENCY);
    v_src_cluster_2 = view_create(matrix_src_address,
                               sizeof (matrix_src_data),
                               v_src, MRMW_CONSISTENCY);
}

/* select appropriate view */

if (in_cluster_1(my_node_id()) == True)
    view_select(v_src_cluster_1)
if (in_cluster_2(my_node_id()) == True)
    view_select(v_src_cluster_2)
```

8.2 Specification of views in a Grid environment

In terms of data sharing and distributed computation, we classify a *grid environment* as an environment that contains one or more of the following properties:

- one or more clusters of machines,
- a collection of heterogeneous machines of varying architecture, number of processors, and available computational resources,
- diverse network topology, interconnects and latency characteristics,
- the type and number of machines used for computation are not known prior to run-time or when writing the application,
- machines exist in administrative domains with restrictive security and computation policies,

- machines, interconnects and other computing resources may come and go.

Grid environments require the ability to dynamically configure a view hierarchy for an application at run-time. The topology of the view hierarchy should depend on the sharing requirements of the application, any locality domains and local knowledge about the most appropriate view pager implementations available that meet the view behaviour specification of the application.

Hence, in the matrix multiply example presented in Section 8.1, the overlapping views used by the application are determined by the underlying environment. Firstly, the application creates the views it needs to share data by providing a view behaviour specification. In this case, it indicates that it needs release consistency. When each node attempts to access a view, the node determines if it needs to create child views to enhance data sharing on each ccNUMA node leading to the view hierarchy illustrated by Figure 8.2. Once the views are created, each node selects the appropriate view that best represents its sharing requirements and characteristics.

There are several approaches machines could use to determine when to create new views and what is the best type of view to use. One approach would be to have a set of defined rules that specify what action the framework implementation (e.g. Eidolon) should take when it needs to create a new view. For example, the administrator of a cluster could set rules that state when it detects a new application view (as the view is unknown to the cluster), it will create a child view of the same type. This gives it benefits of locality. Secondly, it may evaluate if the cluster is capable of using an optimised view. This can be achieved by knowing which optimised views can be used for each type of view. For example, our experiments used MRMW views in-place of strict views.

8.3 Applications

The evaluation presented in Chapter 7 showed that a view specification that considers the resources and topology of an environment can lead to performance increases. It also showed that some applications are not suitable for distribution across multi-cluster environments which is largely due to the computation-to-communication ratio of today's systems. However, this is not a problem for many Grid applications which are specifically designed for the high latency and non-uniform aspects of Grids.

CHAPTER 9

CONCLUSION

In this chapter we conclude this thesis with a review of the thesis contributions and discussion of future work.

9.1 Summary

Chapter 2 surveyed existing approaches to running distributed applications in wide-area and diverse environments. In particular, this chapter examined many of the approaches used to improve the performance of distributed applications when moved to new, wide-area and other diverse environments. Other related work including background material was also discussed.

This chapter concluded that there is no suitable solution for adapting distributed applications in diverse environments that provides a high level of functionality, performance, resource usage and ease of use. Furthermore, DSM applications are far behind in terms of their ability to adapt to diverse environments, which is likely to be a contributing factor to their declined use over the past decade.

Chapter 3 presented the conceptual model named the *view model* along with its properties to address data sharing limitations of existing distributed application middleware. This chapter explores the properties inherent in the model and provides examples of how each property can be used to share data.

Chapter 4 described our implementation of the view model as an architecture.

This architecture, known as Eidolon, used the properties of the view model in a single-address-space for use primarily with distributed shared memory (DSM) applications.

Chapter 5 provided implementation details of the experimental framework used to evaluate the properties of the view model.

Chapter 6 explored the implementation of several programming models and consistency protocols in Eidolon. The view client and view pager implementations are discussed, along with how they interact using view interface operations.

Chapter 7 presented our experiments and evaluations of Eidolon. We demonstrated that using views can lower the resource usage and improve performance for a variety of application benchmarks while maintaining ease of use. These improvements allowed us to adapt unmodified applications to their environment which is essential for running distributed applications in wide-area environments.

Chapter 8 discussed approaches to using Eidolon to run distributed applications and as a foundation for distributed systems.

9.2 Thesis contributions

This thesis made the following contributions:

1. A conceptual model known as the *view model* which is based on a separation of concerns of a distributed application. This separation of concerns allowed us to define a set of mechanisms including non-overlapping, overlapping, and mapped views. These mechanisms allow applications to easily use approaches including protocol selection, locality domains and protocol interoperability to improve their data sharing capabilities.

2. An architecture implementation of our conceptual model called *Eidolon* which was also used for an experimental framework used to evaluate the view model.

3. An experimental evaluation of the view model for improved data-sharing functionality and performance of distributed applications. Applications demonstrated performance, resource usage and scalability improvements when configured to use a view configuration appropriate for the underlying run-time environment. Independently from views, it showed many applications without modification are not likely to scale to large numbers of worker threads or the unbalanced latencies of many diverse environments.

4. The view model provides an approach to address lack of adaption of distributed applications to their run-time environment, without modifications to the application, or significant effort from the user.

5. An approach to improve the flexibility of sharing consistent data between distributed applications such as those that use different programming models.

9.3 Further work

The current Eidolon implementation and evaluation provides a proof-of-concept that the properties of views can benefit development and deployment of distributed applications in diverse environments. In the evaluation it was demonstrated that the views provide some resilience against the affects of bandwidth limitations and latency that is present in multi-clusters. We believe the approach taken expands to Grids and other wide-area systems. The next step is to employ the techniques used with this model into platforms designed for Grid and wide-area systems along with their applications.

Other programming models such as two-sided message passing should also be explored in order to understand the full benefits a view model would provide these models. We anticipate that the same benefits of protocol adaption, and affects of locality domains such as multi-cast message communication amortisation would transfer to these models. However, it is unclear if these benefits would be significant.

9.4 Closing remarks

As there is an increasing desire to run distributed applications in diverse environments, there needs to be changes not only to the applications and their algorithms but also to the underlying approach we use to develop our middleware and share consistent data.

The model proposed in this thesis helps address data-sharing functionality and flexibility necessary for diverse environments. We hope this model and the approaches presented in this thesis simplify and promote improved data sharing capabilities across several distributed computing paradigms.

APPENDIX A

EIDOLON INTERNALS

The framework is implemented in C, using the Kenge [30] build environment and makes use of many of Kenge's standard libraries. The framework code is abstracted and separated into multiple libraries that allow it to be used in other projects. The structure is very similar to that presented in Chapter 5.

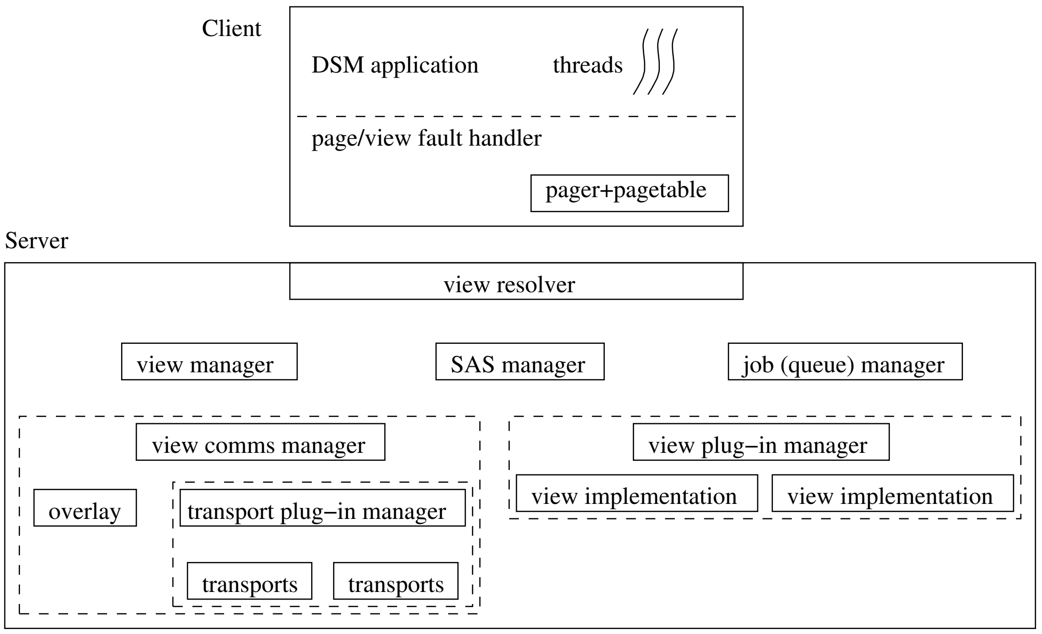


Figure A.1: Common framework component libraries for an Eidolon instance.

Figure A.1 shows the component library structure for an Eidolon application instance. Each component is built as a library. Eidolon client and servers are built separately. The components within each are linked together and typically utilise a function call interface between components. It is possible to build both client and server together for application scenarios where this is appropriate, however we do not illustrate this here. These components can be summarised as follows:

- Page and view fault handlers

  Catches page faults and view interface calls. Different client implementations may have different ways of catching or annotating data access and view coherence operations that required handling by Eidolon.

- Address space and Protection Domain Pager

  This pager stores mappings for clients within the same address space, on the same node. This is useful for clients managing page-based systems, as it caches access rights. However, it is not necessary.

- View resolver: local node view handler and index library

  On each node, this library keeps track of views used actively on that node and maintains a lookup table for views.

- SAS manager: Shared address space arbitrator

  The shared address space used by applications needs to be allocated and delegated globally. A simple arbitrator is written which allows nodes to obtain a region of the shared address space in which to allocate memory and create root views within.

- Job (queue) manager: Pending job library

  This library keeps track of various pending jobs in the system. For example, message retransmits are handled via a timeout mechanism provided by this library. Pending requests to remote nodes are also indexed by this library to allow any pending requests that are waiting on a reply to be restarted.

  Threads waiting on a request can be put to sleep until a timeout or the pending request is resolved.

- View communications manager

  This library manages the transfer of view data and state between nodes.

- Active sharer set tracking

  This library keeps track of dependencies such as remote nodes and local clients that are using a view, in a view pager implementation. It allows view pagers to support an unbounded number of nodes without excessive resource and performance penalties. Active sets are compressed into bit fields that make dependency lookups fast.

- Plug-in view pager support

  Allows new view pagers to be plugged into the system at run time.

- Overlay

  Keeps tracks of views, and provides mechanisms for resolving views.

- Plug-in communications transport support

  Allows new transport interconnects to be plugged into the system at run time.

BIBLIOGRAPHY

[1] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *Sixth International Conference on Cooperative Information Systems (CoopIS 2001)*, 2001.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering—A new definition. In *Proceedings of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[3] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *Proceedings of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, 1997.

[4] Gabriel Antoniu, Luc Bougé, and Raymond Namyst. Generic distributed shared memory: the DSM–PM2 approach. Technical Report RR2000-19, LIP, ENS Lyon, Lyon, France, May 2000.

[5] Gabriel Antoniu, Luc Boug, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the Grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[6] Luciana Bezerra Arantes, Pierre Sens, and Bertil Folliot. An effective logical cache for a clustered LRC-based DSM system. *Journal of Cluster Computing*, 5(1):19–31, 2002.

[7] Argonne National Laboratory. MPICH2. http://www.unix.mcs.anl.gov/mpi/mpich2/.

[8] Noboru Asai, Thomas Kentemich, and Pierre Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 18, New York, NY, USA, 1999. ACM Press.

[9] Olivier Aumage. Heterogeneous multi-cluster networking with the Madeleine III communication library. In *Proceedings of the 11th Heterogeneous Computing Workshop (HCW 2002)*, page 172, Fort Lauderdale, April 2002. Held in conjunction with IPDPS 2002, IEEE Computer Society. 12 pages. Extended proceedings in electronic form only.

[10] Olivier Aumage and Guillaume Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, Tokyo, May 2003. IEEE.

[11] H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 784–790. IEEE Computer Society, 1998.

[12] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Conference on Principles and Practice of Parallel Programming*, pages 168–176. ACM, 1990.

[13] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, September 1991.

[14] Angelos Bilas, Liviu Iftode, David Martin, and Jaswinder Pal Singh. Shared virtual memory across SMP nodes using automatic update: Protocols and performance. Technical Report TR-517-96, Department of Computer Science, Princeton University, Princeton, New Jersey, USA, October 1996.

[15] Paul E. Black. Parallel prefix computation. in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology, December 2004. (accessed 14 January 2007) Available from: http://www.nist.gov/dads/HTML/parallprefix.html.

[16] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[17] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE*

conference on Supercomputing (CDROM), page 12, Washington, DC, USA, 2000. IEEE Computer Society.

[18] John Carter, Anand Ranganathan, and Sai Susarla. Khazana: An Infrastructure for Building Distributed Services. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pages 562–571, Amsterdam, The Netherlands, May 1998.

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

[20] John B. Carter, John K. Bennett, and Will Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on OS Principles*, pages 152–64, 1991.

[21] J. B. Chang and Ce Kuen Shieh. Teamster: a transparent distributed shared memory for cluster symmetric multiprocessors. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid*, pages 508–513, 2001.

[22] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 383–386, Anaheim, CA, USA, April 2005.

[23] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. InterWeave: A middleware system for distributed shared state. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.

[24] Benny Wang-Leung Cheung, Cho-Li Wang, and Francis Chi-Moon Lau. LOTS: a software DSM supporting large object space. *cluster*, 0:225–234, 2004.

[25] Giuseppe Ciaccio. Optimal communication performance on fast ethernet with GAMMA. In *Proceedings of the 12th Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing Workshops*, pages 534–548, Orlando, Florida, USA, April 1998.

[26] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[27] Eyal de Lara. The effect of contention on the scalability of page-based software shared memory systems. Master's thesis, Rice University, Houston, Texas, United States, January 1999.

[28] Christopher Diaz. Demand-Update: Scalable consistency for distributed systems in wide-area networks. In *Proceedings of the International Conference on Emerging Technologies*, August 2003.

[29] Christopher S. Diaz and Jim Griffioen. Measuring consistency costs for distributed shared data. In *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 170–181, London, UK, May 2000. Springer-Verlag.

[30] University of New South Wales DiSy Group. Kenge. http://www.l4hq.org/projects/env/kenge/, 2004.

[31] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.

[32] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[33] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed computing in a heterogeneous computing environment. In *PVM/MPI*, pages 180–187, 1998.

[34] William L. George, John G. Hagedorn, and Judith E. Devaney. IMPI: Making MPI interoperable. *Journal of Research of the National Institute of Standards and Technology*, 105(3):343–428, 2000.

[35] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessey. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Symposium on Computer Architecture*, pages 15–26. IEEE/ACM, 1990.

[36] Richard B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, 1996.

[37] L. Giraud. Combining shared and distributed memory programming models on clusters of symmetric multiprocessors: Some basic promising experiments. Technical Report WN/PA/01/19, CERFACS, Toulouse Cedex, France, September 2001.

[38] Andrew S. Grimshaw and William A. Wulf. Legion-a view from 50,000 feet. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 89. IEEE Computer Society, 1996.

[39] Attila Gursoy and Ilker Cengiz. Mechanism for programming SMP clusters. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume IV, pages 1723–1729, Las Vegas, June 1999.

[40] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 10, Washington, DC, USA, 2000. IEEE Computer Society.

[41] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed data location in a dynamic network. Technical Report UCB/CSD-02-1178, April 2002. Updated version to appear in SPAA 2002.

[42] Zhiyi Huang, Wenguang Chen, Martin K. Purvis, and Weimin Zheng. VODCA: View-Oriented, Distributed, Cluster-based Approach to parallel computing. In *(CDROM) Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2006)*, page 15, Singpore, May 2006. IEEE Computer Society.

[43] Zhiyi Huang, Martin K. Purvis, and Paul Werstein. Performance evaluation of view-oriented parallel programming. In *Proceedings of the 34th International Conference on Parallel Processing*, pages 251–258, 2005.

[44] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the IA-64 architecture. *IEEE Micro*, 20(5):12–23, 2000.

[45] Liviu Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Princeton University, Dept of Computer Science, 1998.

[46] Liviu Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Dept. of Computer Science, Princeton University, June 1998.

[47] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96)*, pages 277–287, 1996.

[48] Ayal Itzkovitz, Nitzan Niv, and Assaf Schuster. Dynamic adaptation of sharing granularity in DSM systems. *The Journal of Systems and Software*, 55(1):19–32, 2000.

[49] Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage – fine-grain sharing in page-based DSMs. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 215–228, Berkeley, CA, USA, 1999. USENIX Association.

[50] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of the 4th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid*, April 2004.

[51] Ernesto Jimnez, Antonio Fernndez, and Vicente Cholvi. Decoupled interconnection of distributed memory models. In *Proceedings of the 7th International Conference on Principles of Distributed Systems*, volume 3144 of *Lecture Notes in Computer Science*, pages 235–246. Springer, November 2003.

[52] Haoqiang Jin and Rob F. Van der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 01:6b, 2004.

[53] Nicholas T. Karohis, Brian Toonen, and Ian Foster. MPICH-G2: A Grid-enabled implementation of the message passing interface. *The Journal of Supercomputing*, 63(5):551–563, 2003.

[54] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the 14th International Parallel Distributed Processing Symposium*, pages 377–84, Cancun, Mexico, May 2000.

[55] Peter Keleher, Alan L. Cox, and Willie Zwanepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 13–21. ACM/IEEE, 1992.

[56] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[57] Dieter Kranzlmüller, Paul Heinzlreiter, Herbert Rosmanith, and Jens Volkert. Grid-enabled visualization with GVK. In *European Across Grids Conference*, volume 2970 of *Lecture Notes in Computer Science*, pages 139–146, Santiago, February 2003. Springer.

[58] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[59] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28:690–1, 1979.

[60] Eugene L. Lawler, Jan Karel Lenstra, A. H. G. Rinnooy Khan, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley and Sons, 1985.

[61] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Phd thesis, Yale Univ., Dept. of Computer Science, 1986. RR-492.

[62] Kai Li, Michael Stumm, David Wortmann, and Songnian Zhou. Shared virtual memory accomodating heterogeneity. technical report CSRI-220, Computer Systems Research Institute, University of Toronto, Canada, 1988.

[63] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, and Ce-Kuen Shieh. Teamster-G: A Grid-enabled software DSM system. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid*, volume 2, pages 905–912, May 2005.

[64] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, Ce-Kuen Shieh, and Pei-Hsin Fan. Enabling software DSM system for Grid computing. In *Proceedings of the 8th IEEE International Symposium on Parallel Architectures, Algorithms and Networks*, volume 2, page 6, December 2005.

[65] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, 1997.

[66] Steven S. Lumetta, Alan Mainwaring, and David E. Culler. Multi-protocol active messages on a cluster of SMPs. In *High Performance Networking and Computing: Proceedings of the ACM/1EEE SuperComputing Conference*, San Jose, California, USA, November 1997.

[67] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[68] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TX, 1996.

[69] Luiz Rodolpho Monnerat and Ricardo Bianchini. Efficiently adapting to sharing patterns in software DSMs. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 289–299, Washington, DC, USA, February 1998. IEEE Computer Society.

[70] M. C. Ng and Weng Fai Wong. ORION: An adaptive home-based software distributed shared memory system. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)*, pages 187–194, July 2000.

[71] Gregory F. Pfister. An introduction to the InfiniBand architecture. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 42, pages 617–632. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[72] Aske Plaat, Henri E. Bal, and Rutger F. H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems*, 17(6):769–782, 2001.

[73] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[74] Daniel Potts and Ihor Kuz. Adapting distributed shared memory applications in diverse environments. In *Proceedings of the 6th International Symposium on Cluster Computing and the Grid*, Singapore, May 2006.

[75] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[76] Felix Rauch. *Distribution and Storage of Data on Local and Remote Disks in Multi-Use Clusters of PCs*. PhD thesis, Dept. of Computer Science, Swiss Federal Institute of Technology (ETH Zurich), Zurich, Switzerland, 2003. ISBN 3-89649-893-2.

[77] Steven K. Reinhardt, Lames R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–336. IEEE, 1994.

[78] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[79] Sumit Roy and Vipin Chaudhary. Strings: A high-performance distributed shared memory for symmetrical multiprocessor clusters. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 90–97, 1998.

[80] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–30, December 2002.

[81] Rudrajit Samanta, Angelos Bilas, Liviu Iftode, and Jaswinder Pal Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proceedings of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, 1998.

[82] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. In *Proceedings of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, page 125. IEEE Computer Society, 1998.

[83] Angela C. Sodan. Message-passing and shared-data programming models — wish vs. reality. In *Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications*, pages 131–139, Washington, DC, USA, 2005. IEEE Computer Society.

[84] Alexander Spiegel and Dieter an Mey. Hybrid parallelization with dynamic thread balancing on a ccNUMA system. In *Proccedings of the Sixth European Workshop*

on OpenMP, pages 77–81, KTH Royal Institute of Technology, Stickholm, Sweden, October 2004.

[85] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM Conference on Communications*, pages 149–160, San Diego, California, USA, 2001.

[86] V S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[87] Kritchalach Thitikamol and Pete Keleher. Multi-threading and remote latency in software DSMs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, May 1997.

[88] Jesper Larsson Traff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 1, Washington, DC, USA, 2000. IEEE Computer Society.

[89] Nian-Feng Tzeng and Angkul Kongmunvattana. Distributed shared memory systems with improved barrier synchronization and data transfer. In *Proceedings of the 11th International Conference on Supercomputing*, pages 148–155, New York, NY, USA, 1997. ACM Press.

[90] Abdul Waheed and Jerry Yan. Performance modeling and measurement of parallelized code for distributed shared memory multiprocessors. In *Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page 161, Washington, DC, USA, 1998. IEEE Computer Society.

[91] John Paul Walters, Hai Jiang, and Vipin Chaudhary. An adaptive heterogeneous software DSM. In *Proceedings of the 2006 International Conference Workshops on Parallel Processing*, pages 266–272, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[92] S. C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[93] Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *IPDPS 2002, Workshop for Communication Architecture in Clusters (CAC '02)*, Fort Lauderdale, Florida, April 2002. IEEE Computer Society.

[94] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[95] Songnian Zhou, Michael Stumm, Kai Li, and David Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3:540–54, 1992.

[96] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proceedings of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.