AN OBJECT INFRASTRUCTURE FOR HIGH-PERFORMANCE INTERACTIVE APPLICATIONS

A Thesis
Presented to
The Academic Faculty

By

Greg Stephen Eisenhauer

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in Computer Science

Georgia Institute of Technology
May, 1998

AN OBJECT INFRASTRUCTURE FOR HIGH-PERFORMANCE INTERACTIVE APPLICATIONS

Approved:

Karsten Schwan

Mustaq Ahamad

Raja Das

John Stasko

Sudha Yalamanchili
Department of Electrical Engineering

Date Approved by Chairman

Dedication

To Peggy

Who has been a wonderful source of love, support and distraction through most of my graduate career.

To Daniel

Whose arrival gave me new impetus to finish this thesis.

To my parents, Gene and Nancy

Who reared me to have the intellectual curiosity required to complete a doctorate.

To Dorothy

Who brought me lattes and took wonderful care of Daniel while I was busily typing away.

Acknowledgments

A thesis inevitably incurs many debts, and this one is no exception. In addition to those mentioned in the dedication, I'd like to thank my advisor, Karsten Schwan. Karsten's help and guidance have been invaluable in developing a feel for the process of research.

I'd also like to acknowledge the help of Dilma Menezes da Silva. Dilma wrote much of the IDL compiler which supports our distributed object system. She was also a good friend and collaborator throughout my years at Tech. I owe an academic and personal debt to Weiming Gu as well. Many of my thesis contributions are extended from his contributions to program monitoring and steering. He was also an early user of PBIO and a collaborator on early versions of DataExchange.

In addition to those to whom I owe academic debts, I'd like to thank the members of the Systems Lab, past and present, who provided interesting work and social environment. I'll always value the friendship of Jya-Jang Tsai, Christoph Widmer, Thomas Kindler, Bodhi Mukherjee, Hernán Astudillo and all the others with whom I've shared work and beer. I'd also like to thank Lynn Daley. She has been my workout partner and friend since my first days at Tech.

Finally, I'd like to thank Holly. While some people consider a cat in your lap to impede error-free typing, Holly has always made positive contributions and authored most of Chapter 4.

Contents

List of Tables

List of Figures

Summary

Program monitoring and steering systems can provide invaluable insight into the behavior of complex parallel and distributed applications. But the traditional event-stream-based approach to program monitoring does not scale well with increasing complexity. This thesis introduces the Mirror Object Model, a new approach for program monitoring and steering systems. This approach provides a higher-level object-based abstraction that links the producer and the consumer of data and provides a seamless model which integrates monitoring and steering computation. The Mirror Object Steering System (MOSS), an implementation of the Mirror Object Model based on CORBA-style objects, is also described and measured. The thesis demonstrates the advantages of MOSS over traditional event-stream-based monitoring systems in handling complex situations. Additionally, it is shown that the additional functionality of MOSS can be achieved without significant performance penalty.

CHAPTER 1

Introduction

The continuous advances of computing and networking techniques have resulted in the proliferation of high performance parallel and networked computers. This proliferation has been accompanied by the development of increasingly complex high performance and distributed applications that take advantage of the available computational power. Such large-scale modern applications often make simultaneous use of multiple networked parallel machines and workstations as their computational, user interface, and input/output engines. They may support multiple simultaneous end-users, process diverse inputs and generate outputs, all of which may be distributed across physically separated locations and heterogeneous machines.

As applications have grown more complex, the need for their developers to gain insight into their run-time behavior has increased and is now critical for creating, understanding, debugging and tuning those programs. Furthermore, additional productivity gains can be attained by allowing users to control program behavior during execution through program steering.

The development of application-level program monitoring tools[16, 26, 33], and more recently program steering tools[16, 17, 34], has done much to promote the separation of the application and the complex program monitoring and steering interfaces that support extensive user interaction. These tools have made it possible to add interactivity to high-performance applications without extensive application restructuring. They enable the construction of both generic and application-specific clients that attach to, extract information from, and interact with the core application. The set of clients which use monitoring and steering tools to interact with an application is called an *interactivity system*.

Existing monitoring and steering systems have limitations that make them difficult to use in modern programs. Principal among these limitations is that they provide no higher-level model of monitoring data reduction and steering. Most currently existing monitoring and steering systems are oriented towards the processing of "events" that occur when certain pieces of code are executed. The resulting streams of events are the highest abstraction of communication between the application and the event-driven clients which consume the streams. Some event-stream-based systems use techniques such as rate control or on-off switches in sensors in an effort to improve scalability[16, 26], others consider complex functions operating on events[39], but none provide a higher-level abstraction that links the producer and the consumer of data. While sufficient in simple cases, event streams are an inadequate abstraction when the interactivity must scale in terms of event rates and number of end-users. The traditional orientation toward event streams was natural when most monitoring was focused on systems-level libraries in applications written in older procedural languages. However, modern monitoring and steering systems have extended their reach to application-level interactions. At the same time, modern programming practice has recognized the advantages of using object-based abstractions. This is particularly true for distributed systems, where the principles of data encapsulation and well-defined interfaces were embraced long before object-oriented languages became widely available. Yet, these well accepted principles of distributed programming are not directly employed in existing distributed program monitoring and steering systems.

This thesis offers the Mirror Object Model, a higher-level model for programming distributed monitoring and steering systems for high performance applications. The basis of the approach is the idea of *mirror objects* in the interactivity system. Mirror objects are analogues to the objects (or data structures) in the application program and are used to accomplish program monitoring and steering. Application-level object attributes are monitored and the resulting updates are also applied to the state of mirror objects. The methods of application objects are made available to the interactivity system and constitute the mechanism through which steering actions are accomplished. This technique of causing an object in the interactivity system to appear to have the methods and state of an application object is called *mirroring* the object.

Mirror objects themselves may have additional state and methods, separate from those which they receive from the application objects they mirror. These additional methods and state provide a mechanism through which the interactivity system can perform monitoring data reduction or accomplish the computation necessary for algorithmic steering. Application objects may be mirrored at multiple points in the interactivity system, providing support for multiple observers and operators. Mirror objects themselves may also be further reflected at other points in the system. This capability is useful in the progressive refinement and reduction of monitoring data, and also in the creation of meta-level systems which monitor and steer the interactivity system itself. Finally, the interactivity system can also host new objects which do not directly correspond to any application object. These new objects may themselves be mirrored at other points in the interactivity system and further enrich the computational model.

The Mirror Object Model employs object-oriented principles and is particularly suitable for adding interactivity to both traditional and object-based high performance applications. In object-based applications, the approach of making application-level object methods available to the interactivity system means that steering is essentially "free." That is, interactive application steering is made available without requiring the application programmer to write specific code to accomplish it because the existing object methods are employed. Since the programmer of a distributed object-oriented application has likely already spent considerable time in designing his objects' methods and managing the synchronization necessary to maintain information consistency, leveraging this effort for use by the interactivity system is an important consideration.

However, the Mirror Object Model can also be used for applications programmed in more traditional ways in non-object-oriented languages. In this situation the application programmer must bear the burden of describing the program state which is to be made available to the interactivity system and he must provide routines to operate on that state to accomplish steering. This burden is no more than what would be required any time such applications are prepared for steering or adaptation.

Based on the interactivity model described above, this thesis explores new opportunities offered by this consistent model for program monitoring and steering. Because the model presented can be viewed as a set of higher-level abstractions over the functionality of traditional monitoring and steering systems, and because the utility of those systems has been widely explored in other domains, we will not focus on establishing the benefits and possible gains resulting from application-level interactivity. Instead, we use example applications to explore typical structures for interactivity and show that those structures can be represented simply using this interactivity model. Given the representation of those structures, the infrastructure required to support their efficient implementation is presented. We also examine automatic optimizations that are possible given this representation that would be difficult or impossible to perform automatically if the structures were represented in lower-level models. We also show how the Mirror Object Model more effectively supports modes of operation that are difficult with simpler systems, such as steering of an application by multiple users simultaneously and performing data-parallel reduction of monitoring information.

1.1 Thesis Statement

In light of the above discussions, the fundamental theses of this work are stated as follows:

- An object-based model can effectively integrate monitoring and steering functionality and provide a structuring mechanism in an interactivity system.
- Such a model can be implemented in a way that causes no more perturbation than traditional schemes.
- The use of a higher-level model enables optimization and efficient implementation in the interactivity system in circumstances that would tax more traditional models.

1.2 Demonstration of the Thesis

The *Mirror Object Steering System (MOSS)* is built to implement the interactivity model and demonstrate the thesis. MOSS consists of a set of tools and libraries that jointly support the Mirror Object Model, a high-level programming model in the interactivity system.

Because the Mirror Object Model involves a protocol for data transport, event association and object management *above* the normal object layer, the tools and techniques used in MOSS can be used with a wide range of object-based systems and languages both in the application and in the interactivity system. However, in order to demonstrate the thesis we will examine the use of MOSS with a CORBA-compliant object-oriented language designed for use in parallel applications. In particular, we will evaluate the MOSS implementation with respect to the theses enumerated above by:

- Describing how the Mirror Object Model is used in common monitoring situations. This is done in examples throughout Chapters 3 and 5.

- Measuring MOSS performance with respect to basic monitoring and steering functions and comparing it to more traditional monitoring and steering systems. These measurements appear in Sections 5.1 and 5.2.

- Showing how the Mirror Object Model and MOSS more efficiently support monitoring and steering situations which are important in scaling interactivity systems to high event rates and multiple consumers. Section 5.3 describes the advantages of the Mirror Object Model when data-parallel processing is required in the monitoring system. Section 5.4 describes how the Mirror Object Model easily handles update suppression in multi-client situations that are difficult with traditional models. Finally, Section 5.2 describes the potential benefits of *steering action migration*, an optimization enabled by the Mirror Object Model which will be explored further in future work. An implementation approach for this optimization is given in Section 4.2.5.1.

1.3 Contributions

Through its specialized monitoring and steering support, MOSS offers a high-level programming model for constructing an interactivity system, a set of clients that monitor and interact with a running high-performance application. MOSS extends the typical functionality of monitoring and steering systems adding capabilities for data organization, automatic event routing, automatic event suppression, computation specification and optimization. MOSS provides a uniform object-based programming model for monitoring data reduction, data analysis and program steering. It implicitly handles multiplexing of displays and computational clients in the interactivity system to support multiple observers and aid in the exploitation of parallelism in the interactivity system. Additionally, it provides a meta-level interactivity capability in which the interactivity system tools are self-applicable. It provides these functions in a manner which is a natural extension of object-based programming techniques yet can also be applied to applications written in traditional procedural languages.

1.4 Organization of Thesis

This thesis is organized as six Chapters. Chapter 1 is this introduction. Chapter 2 more fully describes the application domain addressed by this work, enumerates some of the problems encountered in that domain and derives goals for the new system. Chapter 3 describes the Mirror Object Model and our implementation, the Mirror Object Steering System (MOSS). The model is also compared to related research. Chapter 4 describes the software infrastructure upon which MOSS is constructed. The characteristics and features of this infrastructure play a major role in the performance of MOSS. In that there is independent research merit in some of the infrastructure components, this section also describes related research and potential future work. Chapter 5 presents an evaluation of the Mirror Object Model and MOSS, including basic measurements and demonstration that it meets the thesis criteria. Finally, Chapter 6 presents the conclusions derived from this work and describes possible future research.

CHAPTER 2

Motivation

2.1 The Application Domain

In order to more fully motivate the development of the Mirror Object Model and MOSS, first consider a specific application domain in which they might be employed. In particular, consider a large scientific simulation running on a set of computational resources, such as a global climate model running on a network of parallel machines[22]. This global model might interact with more detailed local climate, atmospheric or pollution models in order to enhance the accuracy of both the local and global models. These models may run concurrently on a variety of parallel and distributed computing resources, or, ideally, any one of the global or local models should be able to be replaced at runtime with a historical database containing information from previous model runs. In addition, model outputs may be processed using a variety of instruments, including specialized visualization interfaces or computational instruments performing calculations which derive from and expand upon the basic model results. Similarly, model inputs may be provided via other instruments that either utilize live satellite feeds or access stored satellite data on remote machines.

Consider the introduction of observers and an interactivity system into this application scenario. Given the scale of such an application and the speed and complexity of its execution platform, it is easy to imagine that the simulation's progress and results are monitored by multiple scientists in distributed locations. The scientists' interests may vary from wishing to see the "big picture," to investigating in detail subsets of the simulation's output, to collaborating with each other via the computational instruments these models implement. To satisfy such needs, the distributed application has additional components that assemble the information needed to drive various interactive displays, by gathering data from the distributed simulation and performing the analyses and reductions required for these displays. Some of these components may themselves have substantial computational or storage needs and require dedicated, additional resources of their own. They may also require access to additional information, as is the case for the atmospheric model's display with which end users compare observational (satellite) data with model outputs in order to assess model validity or fidelity. Moreover, since end users control the set of computational instruments, input and output components, and the displays, the current set of interacting computational instruments will change dynamically, driven by end users' needs or by the current needs of the running simulation. Figure 1 depicts a sample application configuration.

That end-users and their associated displays and computational instruments may come and go over the course of a long application run demonstrates the dynamic nature of computational instruments and their connections. Another attribute of instrument interactions is the bi-directional nature of not only instrument-to-instrument communications but also the communications between instruments and end users and between multiple collaborating users. While it is clear that end users frequently interact with displays and with the analysis components used prior to display, it has become more common for users to also interact with the actual simulations while they are running, perhaps to *steer* them toward more appropriate data domains, to enable or disable certain ancillary computations, or to experiment with alternative settings for simulation parameters. The benefits of such dynamic interactions of end users with

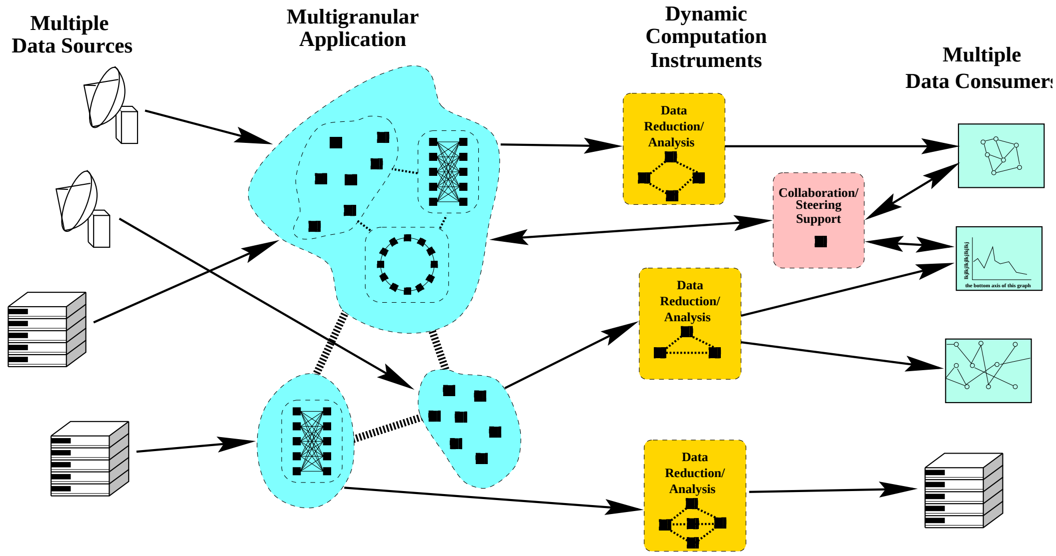

Figure 1: A sample distributed laboratories application configuration.

large-scale simulations are well known for discrete event simulations[25] and have recently been found to offer substantial benefits to scientific and engineering users as well[7, 44]. In addition, it is well known that significant performance improvements may be derived for high performance applications from the runtime configuration of selected application attributes[7], from the online adjustment of their execution environment (*e.g.*, process migration), and from the adaptation of underlying operating system or communication facilities[3].

2.2 The Problem

Given this application domain, the problem is how to control and manage the various components of the interactivity system and the dataflow required to support them. That this can be problematic can be seen by further breaking down the process.

At the core of the interactivity system is online information gathering from the application. Various clients in the interactivity system may be interested in information from various levels of the application, from low-level network statistics to information about application-level constructs. Information is gathered from the application through *instrumentation*, data collection code placed in the application statically or dynamically. The precise manner in which instrumentation is placed in the application is not as important as the realization that the production of data must be managed and controlled at run-time. Control is necessary because the possible quantity of data collected for even a small application would overwhelm any data transport system. However, the dynamic nature of the interactivity system means that compile-time, or even initialization-time, decisions about what data must be extracted from the application are inadequate. As different clients come and go over the course of a long program run, the set of application monitoring data they require will change. The control task is further complicated because of the possibility of multiple independent consumers for any particular type of data. This implies that no particular consumer can make an independent decision about removing or shutting down a data collection point.

After data collection, data processing and reduction is the next task. The mass of available data implies that significant processing may be necessary before data can be presented to end users. Given the proliferation of networked high-performance workstations, the interactivity system may have significant computational resources at its disposal and it is desirable to distribute data processing and reduction across the available resources. However, the dynamic nature of the interactivity system also complicates the data reduction scenario. As shown in Figure 1, many different types of data reduction may be necessary for different clients. As clients come and go, the overall computational demands of the interactivity system will change and it may be necessary to rebalance or rearrange the mapping of data processing responsibilities to the available computational resources. In some cases clients will require the same data reduction and it will be useful for those clients to share data streams. Yet, just as the possibility of multiple consumers complicates initial data collection, it also complicates the process of initiating, placing and shutting down data processing entities as there is no particular client that naturally has knowledge of the data processing requirements of the entire interactivity system.

Another difficulty, routing information within the interactivity system, arises because it is unlikely that each interactivity client will be interested in all monitoring data and routing all data everywhere is likely to overload the network. The event stream model can usually support routing events by type, but this may not be sufficient for managing complex systems. The second problem, matching event data to interactivity client data, is an issue because many monitoring and steering clients maintain information about application state. In particular, interactivity clients tend to maintain information in structures similar to those in the application: per level of the atmosphere in the climate model example, per molecule in a molecular simulation, etc. The creation of these client data structures and the association of event data with them is a problem that escalates with application complexity.

While the efficiency and load balance of the interactivity system is principally affected by the "process/processor/network" structure of the data processing and reduction system, lower-level structuring considerations are equally important. Some data processing may be in the form of simple stateless filters that transform the data from one representation to another, but more complex processing elements may require state to be maintained in the interactivity system. For processing application-level monitoring data, it is reasonable to expect the interactivity system to require state variables for each application-level data structure or object that the state relates to. This implies that the interactivity system is able to create internal data structures that correspond to data structures that exist in the application. It must also be able to route incoming monitoring information and associate the processing of that information with the appropriate state information. This is significant complexity above and beyond simply transporting data to the correct process.

As a concrete example of these problems, consider the atmospheric model described in Section 2.1. This model decomposes the earth's atmosphere into vertical layers and distributes them across processors, with various data encoded as spectral components. Human experts monitoring the progress of the simulation want to see the data in grid form, rather than spectral form, and often want different views of the data, which each require significant further processing. This processing in the interactivity system represents a significant computational load which must also be parallelized via data decomposition. Ideally, the data associated with atmospheric level N would only be sent to the interactivity client handling level N, but an event stream model with only the ability to route by event type cannot divide the event stream in this way. Instead all data must be sent to each client, and the clients must provide code to discard data which they are not interested in and to match relevant incoming data with historical data for the appropriate level. This wasted bandwidth and repetition of client-side event rejection and matching code occurs at every point where data-based decomposition is used in interactivity system and seriously limits the scalability of the system.

The data processing and program interaction requirements of algorithmic program steering pose additional difficulties that distinguish them from the problems of general monitoring data reduction. In algorithmic program steering, computations specified in the interactivity system process data collected from the application for the purpose of steering the application (or components of the application). In algorithmic steering, the overall latency, or time between data collection and steering enactment, is often a critical concern. Obviously, the mapping of computations to computational resources can impact this latency. Overall latency may also be a general concern for the data reduction system. But since most end consumers of data are likely to be user displays on remote workstations, managing the latency of general data processing in the interactivity system is a matter of mapping computation to the appropriate resources outside the application. What differentiates algorithmic steering is that the end consumer of the results of computation is essentially the enactment mechanism in the application. This implies that to meet particularly low latency requirements it would be desirable to have these algorithmic steering computations performed *in the application*. This demonstrates a need for moving computation around in the interactivity system and even migrating it into the application itself.

Just as the monitoring data processing required to reach steering decisions may require calculations to be performed within the application, the process of enacting those decisions almost certainly requires application-level processing. This is the case because steering enactment requires modifying the state of the running application and state modification cannot always be safely performed without some type of synchronization with the application. The extent of the synchronization necessary depends upon the specifics of both the change being enacted and the control-flow within the application. Some steering actions, such as the update of a single atomic data item in the context of an application that is robust to unexpected changes in that item, may be performed with little or no synchronization. However, complicated steering operations may require extensive application-level synchronization and be accomplished only code supplied by the application. This requirement for application code to support the interactivity system also raises the issue of how such application-supplied code is identified and made available to the interactivity system.

As an additional complication, consider that the interactivity system itself is a complex distributed system. The interactivity system can be seen as a tool for gaining visibility and control into the behavior, performance and internal state of the application. Its own behavior and performance are critical to that role. If the interactivity system is useful in developing and managing the complex distributed system that is the application, then the same features should also be useful in managing the complex distributed system that is the interactivity system. This implies a need for the interactivity system to be reflexive. That is, the monitoring and steering tools that are the basis for the interactivity system should also be applicable to the clients that are a part of the interactivity system. This allows the creation of meta-level clients which operate upon other clients in the same manner that the main clients in the interactivity system operate on the application.

Finally, all of these functions must be performed in a heterogeneous environment of networked single and multi-processor machines and they must be performed efficiently in a way that allows the interactivity system to scale to the size of the application and the available resources.

2.3 Summary of Goals

From the discussion in the previous section, a set of requirements or desirable characteristics for an infrastructure to support the interactivity system can be derived. In particular, such an infrastructure must:

- support multiple simultaneous clients and end-users;
- provide for automatic control of monitoring overheads;
- conceptually integrate monitoring data reduction and steering;
- provide a natural model of steering;
- scale gracefully with increasing complexity in both the application and the interactivity system;
- provide semantics that permit important optimizations; and
- permit self-application for meta-level control;

2.4 Limitations of Existing Solutions

A variety of existing systems provide for online monitoring of parallel and distributed applications[16, 26, 33, 27]. Some recent systems also provide for online steering of those applications[32, 34, 43]. However, existing systems have not attempted to address many of the issues and complexities raised in the previous section. A variety of related systems will be reviewed in Section 3.3, but in general existing systems are designed for usage scenarios that are considerably simpler than that outlined in the Section 2.1. Some, such as SCIRun[32] are not designed for a distributed environment at all or work only in offline modes[33]. Of those that are designed for distributed environments, most have concentrated on method and mechanism for extracting data from the application and include sensor-level mechanisms for controlling the flow of data[16, 26].

The model of monitoring provided by these systems views the application simply as a set of sensors of various types and locations. Monitoring clients are consumers of events who may exercise control over event production by changing sensor settings. If steering is supported, as in [34] and [43], there is no semantic link between sensors and the steering mechanism. Furthermore, one common assumption in existing systems is that they are designed to support only a single end-user interacting with an application. This assumption allows simplification in many aspects of the design of a steering and monitoring system, and in particular it allows the use of single point-of-control to manage data collection and distribution. However, the assumption of a single end-user is stretched by the possibility of independent algorithmic steering agents and broken entirely when one considers the possibility of independent human end-users collaborating in the control of a single application. That there is no semantic link between sensors and the steering mechanism limits these systems' ability to perform useful optimizations (such as steering action migration which will be described later).

The next chapters will discuss the Mirror Object Model and the MOSS infrastructure, the ways they improve upon previous work meet the goals set out here.

CHAPTER 3

Mirror Object Model

As discussed in Chapter 1, the structuring abstraction for the interactivity system is the Mirror Object Model. This chapter will present the model in detail and discuss its realization and application in both standard object languages and non-object procedural programs.

3.1 Abstract Model

As the name implies, *mirror objects* are the basis of the approach for structuring the interactivity system. Mirror objects are analogues to the objects (or data structures) in the application program. Generally, their number, structure and relationships correspond to the number, structure and relationship of objects in the application. Application-level object attributes are monitored and their updates are also applied to the state of mirror objects. The methods of application objects are made available to the interactivity system and constitute the mechanism through which steering actions are accomplished. This technique of causing an object in the interactivity system to appear to have the methods and state of an application object is called *mirroring* the object.

Mirror objects themselves may have additional state and methods, separate from those which they receive from the application objects they mirror. These additional methods and state provide a mechanism through which the interactivity system can

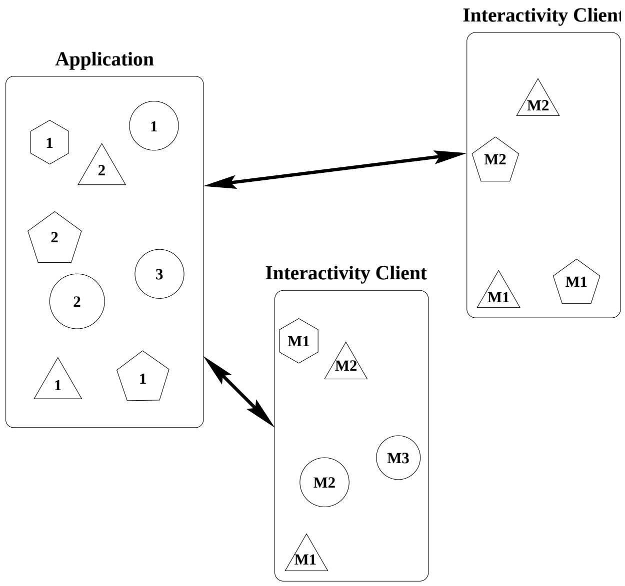

Figure 2: A depiction of application objects and mirror objects in interactivity clients

perform monitoring data reduction or accomplish the computation necessary for algorithmic steering. Application objects may be mirrored at multiple points in the interactivity system, providing support for multiple observers and operators. Mirror objects themselves may be also be further mirrored at other points in the system. This capability is useful in the progressive refinement and reduction of monitoring data, and also in the creation of meta-level systems which monitor and steer the interactivity system itself. Finally, the interactivity system can selectively exclude application objects from mirroring and can also host new objects which do not directly correspond to any application object. This allows the interactivity system to extensively augment application functionality while only selectively mirroring its structure. These new objects may themselves be mirrored at other points in the interactivity system to further enrich the computational model.

3.1.1 Structure

Figure 2 depicts the structural relationship between objects in the application and mirror objects in the interactivity system. A mirroring relationship between objects is indicated by a correspondence between shapes and numbers. The object represented by a triangle containing a two is mirrored by the interactivity object represented by a triangle containing "M2".

The creation of objects in the interactivity system is controlled by a distinguished object in each client that receives notification of the presence of existing and newly created application objects. This object is customized for each client and can create mirror objects based on the characteristics of the application object. Once created, mirror objects are largely independent, performing monitoring and steering as described in Section 3.1.2.

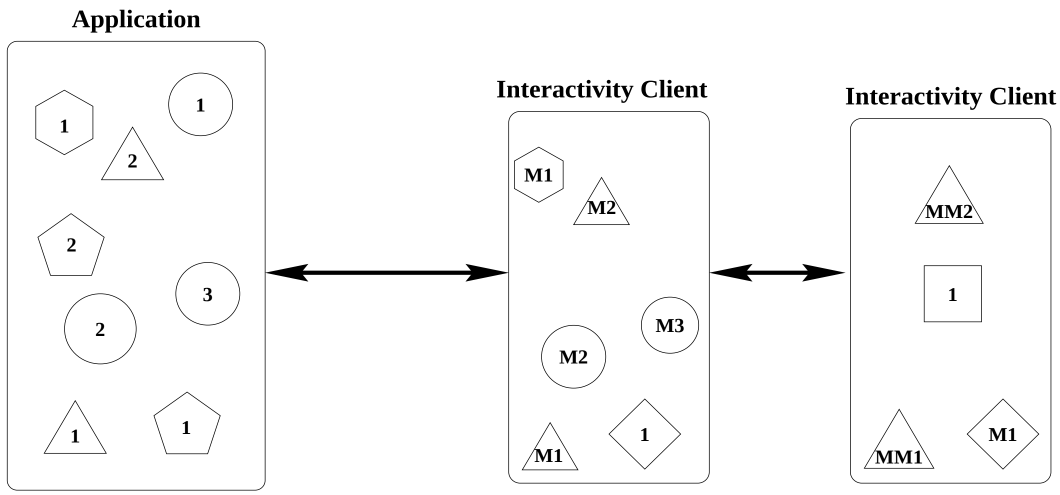

Figure 3: Mirror objects in a meta-level structure

While Figure 2 represents a relatively simple one-level interactivity system structure, structures like the one depicted in Figure 3 are also possible. This figure depicts a two-level interactivity system in which the right-most client mirrors objects in the middle client, which is itself mirroring objects in the application. Mirroring relationships are once again indicated by a correspondence between shapes and numbers with an "M" prefix added to mirror objects. The figure also shows another feature of the Mirror Object Model, the ability to create objects in the interactivity system that *do not* mirror objects in the application. Such independent objects can play a role in monitoring data reduction and steering computations, as well as themselves be the subject of mirroring in meta-level interactivity clients. The diamond-1 object in the middle client and corresponding mirror object in the right-most client have this relationship.

3.1.2 Function

Monitoring and steering in the Mirror Object Model are accomplished through the functional relationships between application objects and mirror objects. In the Mirror Object Model, application monitoring entails the propagation of state variables from object to mirror object. The relationship between the state of an object and a related mirror object in Figure 4.

Upon the arrival of updated state variables, the special method "Update()" is called in the mirror object. This method call can trigger monitoring data reduction, display, or other calculations in the interactivity client. These calculations may update state variables in the interactivity model objects which may cause a similar process to occur in meta-level clients. Invocations to mirror object methods that have been "inherited" from application objects are performed as remote object invocations to the application object from which the method is derived. This is the basic mechanism through which steering is accomplished. A timeline depicting a complete steering interaction is depicted in Figure 5.

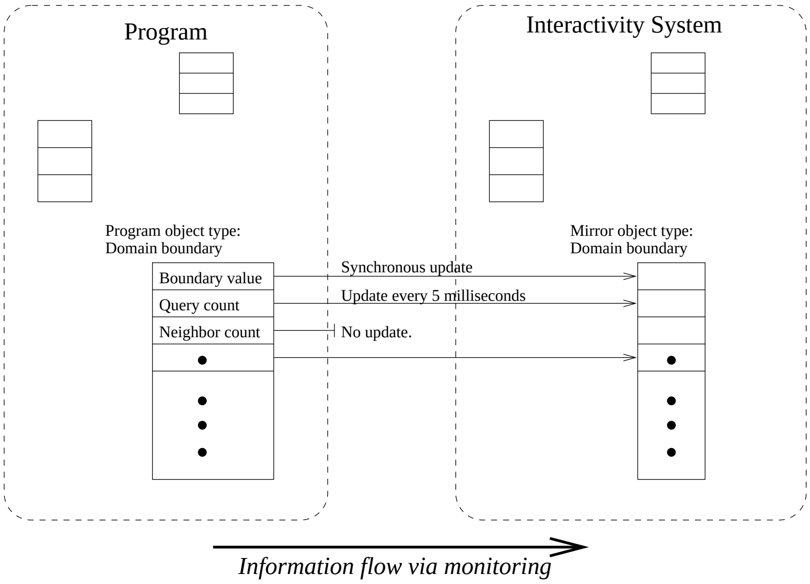

Figure 4: An object in the application and its mirror object in the interactivity system.

3.2 Realization

The abstract Mirror Object Model described in Section 3.1 can be realized in a number of ways depending upon the nature of the object systems, if any, employed in the application and the interactivity system. Because the clients in the interactivity system are more likely to be developed from scratch, that system is discussed separately from the application, where it is more likely to be necessary to deal with an existing program.

3.2.1 In the Application

In order to accomplish its goals of monitoring and steering the application, an interactivity system based on the Mirror Object Model must interface with the application. The manner in which this interface is accomplished depends upon the application and

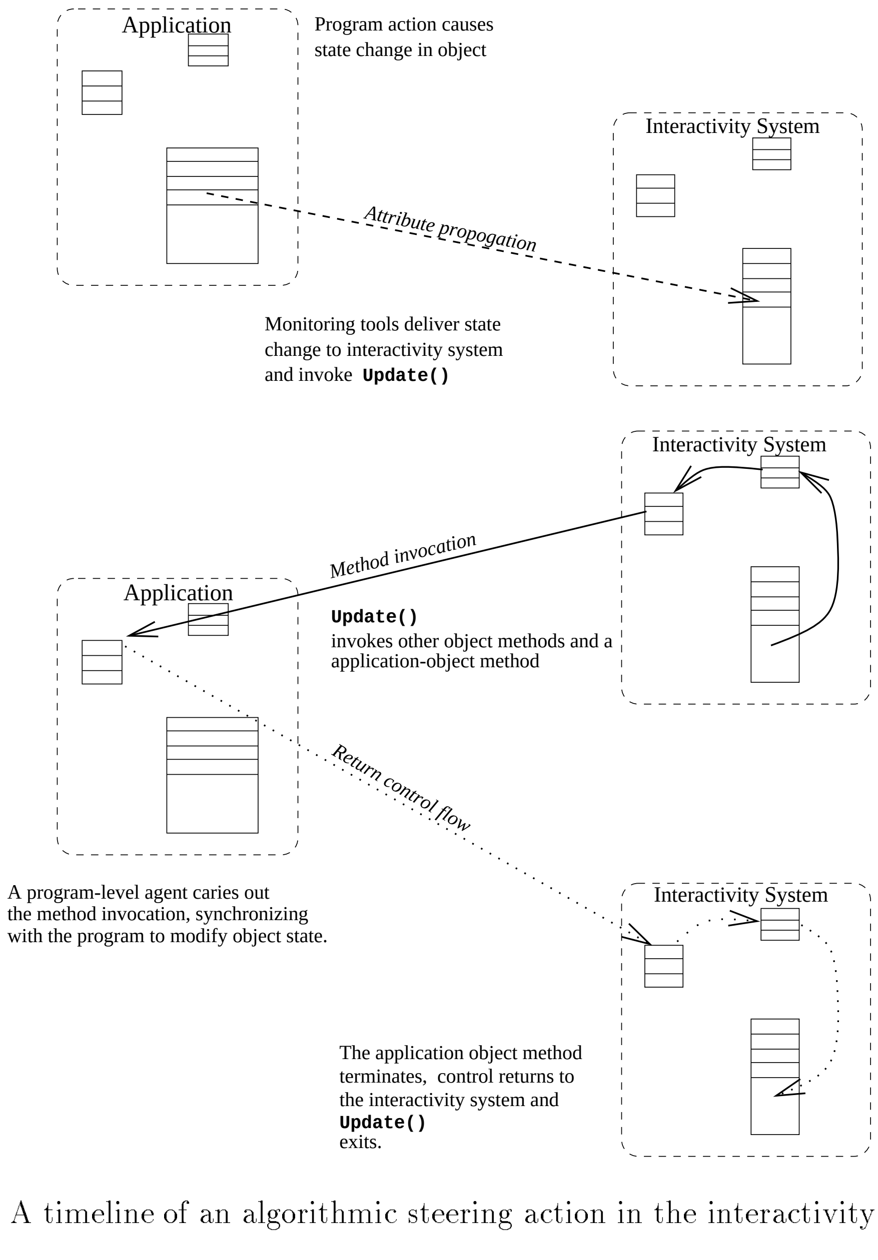

Figure 5: A timeline of an algorithmic steering action in the interactivity system.

its programming environment.

The interactivity system wants to view application components as objects with associated state and methods. This is obviously a semantic match if the application is indeed written in an object-oriented style. But even for applications written in procedural languages, the mirror object approach is still valid. Any simple procedural program will require modification in order to support application-level monitoring and steering. Systems such as Falcon[15] and Progress[43] require the insertion of code snippets into the application at appropriate points. A similar approach which is more agreeable with the object-based nature of the interactivity system is to "objectify" small portions of the application to enable them for steering. The term "objectify" means to identify the components of application state which are of interest for monitoring and steering and "wrap" them with an object-like interface. Given that such an application would almost certainly require modification to allow safe steering anyway, doing those modifications in a way that provides an object-like abstraction does not present significant additional difficulties.

The implementation of object monitoring and remote invocation depends upon the nature of the object system employed in the application. Many object systems provide for remote object invocation. Of those that do not, some will export sufficient information to permit a remote invocation system to be constructed without modifying the program. As a last resort, the program may be modified to identify and register object classes and methods with a custom remote invocation system. The implementation of monitoring may be performed in a similar fashion. The approach taken in the initial implementation is to use a CORBA-style object system with support for remote object invocation and built-in monitoring. However, if such a system was not available, one could enable monitoring by resorting to to program annotation (such as Falcon[16] and Progress[43]), code transformation (as in Pablo[33]) or, if sufficient information was available in the program, dynamic instrumentation techniques

```
/* boundary.idl
**
** An object representation of the particle domain boundaries.
*/

enum dimension { X, Y, Z };

interface boundary {
  attribute float boundary_value;

  attribute dimension dim;
  attribute long index;

  float get_val();
  void put_val (in float value);
};

interface domain {
  attribute long mol_count;
  attribute long index[3];
};
```

Figure 6: An example of an object specification in IDL.

(such as those used in Paradyn[26]).

Our implementation follows the style common to CORBA Object Resource Broker implementations. Object interface declarations, such as the example given in Figure 6, are written in IDL. These IDL declarations are then processed by a stub generator to generate header files and program stub routines that are suitable for use in the target language. These stubs provide support for local and remote method invocation and access to the class variables, and the application programmer then supplies the implementation for the actual methods. For example, the IDL stub generator creates C code and headers and requires the object method implementations to be written in C. These stubs, together with the method implementations provided by the programmer, completely implement the object specified by the IDL interface. These objects can then be used to create new applications, or they can be integrated into existing applications.

3.2.1.1 Instrumentation

Function. Instrumentation, in the context of the Mirror Object Model, is the mechanism through which changes in application object state are detected and transmitted to the mirror objects in the interactivity system. The model described in Section 3.1.2 requires that changes in object state be mirrored in the interactivity system and that those changes cause Update() methods to be called in the mirror objects. This specification leaves room for variations in implementation. The simple approach would be to reflect each change in object state immediately to the mirror objects. This is the approach chosen for the initial implementation, but some useful variations are worth noting. In particular, one might choose to implement the semantic variations that were done with monitoring sensors in [15], that is, variable-rate tracing sensors and sampling sensors. The analogous techniques in the Mirror Object Model would involve arranging instrumentation such that object state were reported less frequently than at every change (such as every Nth change) or that it be reported at regular time intervals regardless of change. Because these techniques were evaluated in detail in [15], they are not considered further here. However, these are very appropriate in the Mirror Object Model and may be employed to further reduce monitoring overheads.

Implementation. In CORBA, access to object attributes occurs only through get/set routines that are generated by the compiler. For the mirror object implementation this implies that instrumentation of the application is accomplished simply by modifying the IDL compiler to generate instrumentation in the attribute set routines. Figure 7 shows the code that is generated for the set procedure for the attribute `index` in the class `boundary` shown in Figure 6. Most of the routine shown implements attribute set functionality required for OTL, the Object Transport Layer infrastructure

```
void impl_boundary__set_index(boundary o, CORBA_long value,
                              CORBA_Environment *ev)
{
   _IDLtoC_boundary_attrib_struct *state;
   static atom_t otl_self_atom = 0;
   static atom_t otl_chan_handle_atom = 0;
   ECSourceHandle event_handle = NULL;
   if (otl_self_atom == 0) {
        otl_self_atom = attr_atom_from_string("OTL:OBJ_SELF");
        otl_chan_handle_atom =
            attr_atom_from_string("OTL:MON_CHANNEL_HANDLE");
   }
   query_attr(o->obj_name, otl_self_atom,
              NULL, (void**)&state);

   query_attr(o->obj_name, otl_chan_handle_atom,
              NULL, (void**)&event_handle);

    state->index = value;
    if (EChas_sinks(event_handle)) {
        boundary_submit_state(event_handle, state);
    }
}
```

Figure 7: The code generated for the set procedure for the attribute `index` in the example class `boundary`.

that provides run-time support for the CORBA system (OTL is described further in Section 4.3.). The final `if` statement and the subroutine call it protects represent the embedded instrumentation.

The Mirror Object Model uses event channels to transport monitoring information to the mirror objects in the interactivity system. Event channels, described further in Section 4.2, are a publish-subscribe communication mechanism similar to the artifact of the same name described in the CORBA Common Object Services specification[4]. The `EChas_sinks()` call in the `if` conditional checks to see if there are any subscribers to this object's event channel that will receive events. If there are none, this check bypasses the remainder of the instrumentation and no events are generated. Because the Mirror Object Model implementation uses one event channel per monitored application object, this allows per-object configurability of monitoring. This is a much finer level of control than has been possible in previous systems and is a valuable mechanism for containing monitoring overhead.

Mirror objects in the interactivity system subscribe to the event channel of the object they mirror, and the semantics of the event channel ensure that events are routed only to interactivity clients that are interested in receiving them. The channel also naturally takes care of pairing events with the appropriate object when the event arrives at the client, as well as suppressing the transmission of monitoring data when there are no listeners. The suppression of unwanted monitoring events is vital in helping the monitoring system handle the volume of potential monitoring information. Reducing the flow of unnecessary events is critical to the scalability of any monitoring system and is often handled by on-off switches associated with application sensors, a technique that becomes harder to manage when there is more than one event consumer. That event suppression can be done on an object-by-object basis and that it operates automatically, even in the presence of multiple consumers, is a strong advantage for the mirror object system and is demonstrated in an experiment in Chapter 5.

An equally important benefit of the implicit routing provided by event channels is that it solves the scalability problems raised in the example in Section 2.2. With event channels managing data flow between the application objects and the mirror objects in the interactivity system, the decomposition of application and interactivity processing no longer complicates the system. That is, the precise location of each mirror objects is immaterial to the application because the event channel handles information routing. For example, events from atmospheric layer N are routed only to the client which is to process them and are automatically matched with recorded state in the mirror object.

3.2.1.2 Steering

Function. In the Mirror Object Model, program steering is accomplished by remote invocation to the application objects. This is a natural choice if the application is written in an object-oriented style and means that steering is available for "free." That is, interactive application steering is made available without requiring the application programmer to write specific code to accomplish it because the existing object methods are employed. Since the programmer of a distributed object-oriented application has likely already spent considerable time in designing his objects' methods and managing the synchronization necessary to maintain information consistency, leveraging this effort for use by the interactivity system is an important consideration.

However, the Mirror Object Model can also be used for applications programmed in non-object-oriented languages. In this situation, the application programmer must bear the burden of describing the program state which is to be made available to the interactivity system, and he must provide routines to operate on that state to accomplish steering. This burden is no more than what would be required any time such applications are prepared for steering or adaptation.

Implementation. Our initial CORBA based-implementation of the Mirror Object Model, the Mirror Object Steering System or MOSS, is particularly suitable for objectifying certain parts of non-object-based applications to make them available for monitoring and steering, but it could also be used directly in CORBA-based applications or used to develop such applications from scratch. As described above, the IDL compiler generates stub routines for object methods based on the IDL specification. These stubs provide mechanisms through which application programs can invoke object methods, and the provide support for remote invocation. The precise nature of the stub routines is more a function of the supporting object system than of the Mirror Object Model. The role of the stub routines and their relationship with the underlying object support system will be discussed in more detail in Section 4.3. However, the set of routines available to the programmer and those which must be supplied by the object implementor are discussed next as these are of more direct interest to the programmer in adding monitoring and steering to an existing program.

Figure 8 shows the declarations generated by the IDL compiler for the example IDL specification given in Figure 6. The majority of these routines are simple access routines for the object attributes, but `boundary_get_val()` and `boundary_put_val()` are stub routines which give access to the object methods. In addition to those stub routines, the IDL compiler also generates declarations for the method implementations, given below:

```
extern CORBA_void impl_boundary_put_val(boundary object, CORBA_float value,
                                         CORBA_Environment *ev);
extern CORBA_float impl_boundary_get_val(boundary object, CORBA_Environment *ev);
```

The bodies of these routines must be provided by the class implementor. In the case of the example object, the `boundary_put_val()` method is designed to be used

```
/****** Declarations for interface boundary******/
extern CORBA_float
  boundary__get_boundary_value(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_boundary_value(boundary o, CORBA_float v, CORBA_Environment *ev);
extern dimension
  boundary__get_dim(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_dim(boundary o, dimension v, CORBA_Environment *ev);
extern CORBA_long
  boundary__get_index(boundary o, CORBA_Environment *ev);
extern void
  boundary__set_index(boundary o, CORBA_long v, CORBA_Environment *ev);

extern CORBA_float
  boundary_get_val(boundary object, CORBA_Environment *ev);
extern CORBA_void
  boundary_put_val(boundary object, CORBA_float  value, CORBA_Environment *ev);
```

Figure 8: The declarations generated for the IDL specification in Figure 6.

to asynchronously modify the position of the “boundary,” an element of a geometric decomposition in a physical simulation. The position of this boundary can be changed at any time during the simulation, but processors must be guaranteed a consistent view of the configuration. Therefore, the body of the `impl_boundary_put_val()` acquires a global lock before using the `boundary_set_boundary_value()` routine to change the state of the object. This code is shown in Figure 9.

```
CORBA_void
impl_boundary_put_val(boundary o, CORBA_float value, CORBA_Environment *ev)
{
    thr_mutex_lock(configuration_lock);
    boundary_set_boundary_value(value);
    thr_mutex_unlock(configuration_lock);
}
```

Figure 9: The implementation of `impl_boundary_put_val()`.

In general, application-level synchronization is an important, but difficult, issue in program steering. The Mirror Object Model does not impose any particular model for synchronization. This approach allows the model to adapt to various application-level object systems which may have differing styles of synchronization. For example, some object systems, such as Java, provide models of synchronization that are tightly coupled with the object system itself. In particular, Java allows some objects to be designated as monitors, with the result that only one invocation at a time is allowed to be active within them.

3.2.2 In the Interactivity System

The issues involved in creating interactivity system clients are very similar to those described in Section 3.2.1. In particular, since the Mirror Object Model is meant to be applied to itself to enable meta-level monitoring and steering, mirror objects in the interactivity system have many of the same features and responsibilities as objects in the application. Clients in Mirror Object Model could be implemented in any number of OO languages, from SmallTalk to Java, but our initial programming environment is derived from the CORBA-style object system used in the sample applications. To specify the methods and state of the mirror objects, IDL is extended with a "mirrors" keyword that is used in the IDL interface header declaration and operates much like an inheritance specification. Where in normal IDL the specification:

```
interface ibound : boundary {
        ....  attributes and methods ...
};
```

declares that the class `ibound` was derived from the class `boundary`, the mirror specification appears as:

```
interface ibound mirrors boundary {
        ....  attributes and methods ...
};
```

to declare that class `ibound` mirrors objects of class `boundary`. The principal difference in compiler behavior for the latter case is that the compiler generates remote object invocation code for the mirrored methods. Additionally, if one of the methods of the new class is named "update," the compiler generates code to ensure that this method is called when new object state arrives via monitoring. This automatic method call upon update provides a triggering mechanism for interactivity system calculation, which may include monitoring data reduction or algorithmic steering.

Another significant optimization opportunity provided by the Mirror Object Model relates to the potential for relocating interactivity computations. Because the Mirror Object Model conceptualizes those computations as methods associated with the application object, it is possible to analyze those methods and potentially relocate them in order to improve performance, reduce latency or balance loads in the interactivity system. While the general case of relocating a method from an interactivity client to another client or into the application may be prohibitively complex, there are some obviously simple cases where it is possible to reap significant benefits from this ability. For instance, a monitoring client may not operate on every piece of updated data it receives, but instead apply a simple threshold function and operate only if the data exceeds the threshold value. Moving this threshold operation to the source of data eliminates unnecessary data transmission. Similarly, simple algorithmic steering operations may be moved from the interactivity system directly into the application. Because latency can be a significant limitation on the applicability of algorithmic steering, migrating externally-specified steering operations directly into the application can allow new domains for program steering. For example, [28] describes mutex locks that adapt themselves to the most efficient mode of operation based on their usage pattern. Normally this sort of adaptation would be beyond the ability of external steering because of the response time required for adaption, but steering actions within the application should be capable of such rapid response times. This optimization is discussed further in Sections 5.2 and 4.2.5.1.

3.3 Related Research

Program monitoring and steering systems have been under development for many years, providing a body of related work. [17] and [42] surveyed much of this work, including topics such as offline instrumentation and perturbation analysis that have less relevance to this thesis. This section reviews other projects with features relevant to this work, including online monitoring and steering systems and a few other systems with special relevance to the Mirror Object Model. However, because the concentration of this thesis is in the area of monitoring and steering systems, and because the Mirror Object Model could be implemented in many different styles of distributed object system, the breadth of work in distributed object systems is not reviewed.

Among the newer efforts in program monitoring and steering is Progress[43], which also originates at Georgia Tech. Like many monitoring and steering systems, Progress is based on an event-triggered model of monitoring with no obvious higher-level mechanisms for routing data to appropriate clients. As described in Section 2.2, this limits usability in complex interactivity systems. Progress extends the *ad hoc* steering methods of Falcon[16] and implements steering through *actuators* which operate on *steering objects*. Unlike the objects in the Mirror Object Model, Progress' steering objects are not objects in the object-oriented sense of the term, but are distinguished data elements in the application program. Instrumentation points and actuation points are inserted into the application program at compile time. Instrumentation points serve their traditional function of implementing monitoring and actuation points are somewhat like instrumentation sensors in reverse. Furthermore, in contrast to the

Mirror Object Model, steering in Progress is carried out synchronously. When an application thread reaches an actuation point for which a steering action has been set, that action is carried out. Through this mechanism, the intrinsically asynchronous steering action is synchronized with the application by only performing the action at points where the application deems it "safe" by inserting an actuator. In comparison, mirror objects may be used to implement the synchronous style of steering provided by Progress, but in addition they preserve the asynchrony and reliance on the application method to provide any necessary synchronization.

Progress does not seek to address the principal goals of the Mirror Object Model, that of providing a global structuring mechanism for a complex interactivity system, but rather concentrates on lower-level steering techniques.

Autopilot[34] from Illinois is a steering system with features that are somewhat similar to those of the Mirror Object Model. Like Falcon[16], Autopilot sensors have a variety of sensor policies that can help the system scale by reducing overall dataflow. Autopilot also has mechanisms for making local steering decisions, which can aid in scalability. Autopilot is object-based, but not to the same extent as is the Mirror Object Model. Autopilot provides classes of Sensor and Actuator objects which can be instantiated in the application code, but they are not meant to replace or serve as application objects. While they can be subclassed, they do not export application methods for steering purposes. Autopilot does provide for one-to-one matching between client objects and "startpoint" objects in the application and automatically routes events directly, but its steering model is largely limited to specifying times when the data in actuators may or may not be changed. Additionally, as demonstrated in experiments in Chapter 5, Autopilot's current implementation suffers from relative large overheads when transporting large amounts of data. This might limit Autopilot's applicability in complex scenarios.

SCIrun[32] is an environment that supports the construction of programs through the process of graphically connecting computational components in a data-flow-style graph. It also offers a degree of computational steering in that parameters associated with some computational components can be changed dynamically. In many respects, SCIrun is not so much a program monitoring and steering system as it is a program construction environment where some computational elements can accept graphical input. No monitoring system is used in SCIrun and the synchronization of steering with program control is less of an issue since steering must be programmed directly into computational elements.

Viz[20], from the University of Oregon, is a visualization programming system that shares some features with the Mirror Object Model. In particular, Viz is object-based and uses a "data reactive" programming model that allows properties of graphical elements (such as the height of cube) to be tied to attribute values of objects in the program. When the attribute value changes, the corresponding graphical entity changes as well. The implementation of Viz involves "shadowing" STKlos objects (Scheme objects based on the Common Lisp Object System) with Open Inventor objects in such a way that changes in the STklos objects effect corresponding changes in the Open Inventor object. While the resemblance of this approach to the basic functionality of the Mirror Object Model is striking, Viz is not a complete program monitoring and steering system. Furthermore, while the object shadowing technique resembles the mirroring technique, Viz shadow objects exist in the same process and address space as the STklos objects and do not maintain separate state. Also, Viz does not support distribution in any form and there is no mechanism for program steering. A similar approach to visualization by mapping program state data to display characteristics, but without the object abstraction, is employed by Pavane[35]. Because the program data to display mapping is a trivial operation in the Mirror Object Model, this implies that it may also be useful for creating vizualizations in support of the basic monitoring and steering function. This thesis does not explore those potential benefits of the Mirror Object Model, but they may be considered in future research.

TAU[27] from the University of Oregon is a package of program tuning and analysis utilities that include static and dynamic analysis tools with complex graphical interfaces. TAU includes Breezy, an infrastructure for dynamic runtime interaction that is based on breakpoint debugging and targets program interaction for object-based programs written in pC++. Breezy and the Mirror Object Model share similarities in that their approaches to program interaction include the ability to call object methods in the application. However, the Breezy debugger-based approach is difficult to extend to multiple clients. Indeed, the discussion in [18] reflects the difficulty of modifying a single-point-of-control interactivity system for multi-client use. The Mirror Object Model was designed to avoid these difficulties by providing the implicit routing, control and system structuring support necessary to operate in this environment.

Paradyn[26] is a parallel program performance tool. While not a steering system, it is of interest because of the dynamic instrumentation techniques it uses. Paradyn seeks to control monitoring overhead (and thus system scalability) by dynamically inserting instrumentation into a running program. By adding and removing instrumentation points, Paradyn achieves an effect similar to that which Falcon[16] achieved by turning sensors on and off. Paradyn instrumentation points can also include filter/processing code which is dynamically generated (MDL[21]). While Paradyn's centralized monitoring server does not attempt to solve the more global problem of building complex interactivity systems, their dynamic instrumentation techniques are entirely complimentary to the Mirror Object Model's compile-time instrumentation. Performing such dynamic instrumentation and using MDL (or other dynamic code generation technology) for migrating steering actions into the application may be implemented in the Mirror Object Model in the future.

The last system we will review is actually not a program steering or visualization tool, but instead is a program composition system. The work on Subject-oriented Programming[19] at IBM addresses problems in extending and maintaining object-oriented programs. One facet of subject-oriented programming involves extending existing object-based programs by non-intrusively associating new state and methods with the existing objects. This technique can support unplanned extension to programs, division of work during development and separation of independent features. This work is relevant to the Mirror Object Model because of the similarities in carrying out program extensions. The Mirror Object Model extends application objects by adding methods and state that exist in the interactivity system, creating composite subjects as is done in subject oriented programming. Thus, the arguments in favor of using subject oriented programming techniques for creating complex systems also suggest that the Mirror Object Model is an effective way to create the broader "application" made up of the interactivity system associated with a high-performance program. This thesis does not examine the programmability benefits of the Mirror Object Model, but they may be explored in future work.

CHAPTER 4

Infrastructure

The Mirror Object Steering System is constructed on top of several software packages. These packages are not the direct focus of this thesis, but their features and characteristics enable MOSS to meet its functional and performance requirements.

Two of these packages are DataExchange and PBIO. DataExchange is a connection manager and PBIO is a binary data encoding package that handles heterogeneous systems. Together, these packages form a transport-level message passing system. MOSS uses PBIO for parameter and return value marshalling in event transport and remote method invocation. MOSS does not use DataExchange calls directly, but DataExchange provides the basis for the next two packages.

Another important package is Event Channels. Event Channels propagate MOSS object state updates. The characteristics of Event Channels give MOSS automatic event suppression and update routing. Section 4.2 describes the Event Channel implementation in further detail.

Finally, the Object Transport Layer (OTL) provides the basic remote method invocation mechanism through which MOSS accomplishes program steering. OTL is a relatively thin layer on top of DataExchange and is described in Section 4.3.

4.1 DataExchange and PBIO – Transport

As described above, the MOSS infrastructure rests on the facilities of DataExchange and PBIO. Because DataExchange and PBIO have research value in their on right, this section is a relatively free-standing discussion of their research context, goals and contributions. In particular, this section will consider DataExchange in it's role as providing a generic communications infrastructure for the Distributed Laboratories environment. This role includes providing the process/socket level of interconnection, as opposed to the datastructure-to-datastructure relationships established by the Mirror Object Model and MOSS. A final subsection will describe the role of DataExchange in MOSS and describe its contributions to the thesis.

4.1.1 Communication Substrates for High Performance Applications

Communication tools and libraries for high performance distributed computing have evolved substantially during the last few years, from systems enabling distributed parallel computing such as PVM[14], to industrial standards being actively implemented and improved such as MPI[12], to recent proposals for infrastructures that can facilitate the construction of nationwide, networked supercomputers. Our work contributes to such research by addressing several specific issues that arise whenever diverse networked machines are used in research and development settings or even in production environments, by single or multiple end users. The target applications we consider are those in which end users interact with powerful computational tools and with each other across heterogeneous networked machines, termed *distributed laboratories*. The communication demands of such dynamic computational environments share some commonalities with traditional distributed systems as well as with the highly dynamic environment of today's web browsers and CORBA-based distributed object systems, but unfortunately these similar environments do not adequately address the high performance needs of some applications. Moreover, these environments' dynamic communication and computational behaviors are not addressed by standards for distributed high performance computing like MPI.

The needs of dynamic environments like distributed laboratories are the focus of recent research efforts on a variety of fronts. For instance, the LabSpace project at Argonne and at Northeastern University[23] is trying to create a virtual space in which scientists who are distributed across the nation can meet, talk, plan their research, and run their experiments. Support for the dynamic addition of clients is being included in MPI-2[11]. Similarly, the San Diego project KeLP[2] seeks to improve the manner in which accesses to remote data may be made more efficient by investigation of object-based data bases for data storage that enable computations to migrate toward useful data as well as by investigation of alternative, compiler-based approaches to linking interactive applications[41].

Our work complements the efforts described above by assuming that end users should be able to exploit the potentially high performance of vendor-provided implementations of standards like MPI when appropriate for their applications. In addition, when application requirements can be stated statically, users should be able to exploit available tools, such as specialized communication libraries (*e.g.*, fast messages[31]) or the automatic support for parallelization and inter-task communication offered by HPF compilers. Given these assumptions, the 'Distributed Laboratories Project'[36] is developing communication support that addresses the interactive and dynamic nature of high performance applications to enable their use in scientific investigations or in production environments, by single and by multiple end users. Specifically, the *DataExchange* communication library utilized in the implementation of several distributed high performance applications offers the following functionality:

- a publish-subscribe communication model,

- fast heterogeneous data transfer, and
- support for application-level diversity.

In the remainder of this section, we characterize the communication characteristics of high performance applications when they are used in distributed laboratory settings. Next, we identify the requirements these characteristics impose on the underlying communication infrastructure. Using examples from the high performance interactive applications being developed, we also describe the DataExchange communication library. Experimentation with DataExchange across networked machines and with the sample application serves to evaluate the manner in which the identified communication requirements are met.

4.1.2 The Distributed Laboratory Environment

4.1.2.1 Characterizing Distributed Laboratories

In order to illustrate the challenges posed by distributed laboratory environments, we first consider a specific application. In particular, consider a large scientific simulation running on a set of computational resources, such as a global climate model being developed for networked parallel machines[22]. This global model might interact with more detailed local climate, atmospheric or pollution models in order to enhance the accuracy of both the local and global models. These models may run concurrently on a variety of parallel and distributed computing resources, or, ideally, any one of the global or local models should be able to be replaced at runtime with a historical database containing information from previous model runs. In addition, model outputs may be processed using a variety of instruments, including specialized visualization interfaces or computational instruments performing calculations which derive from and expand upon the basic model results. Similarly, model inputs may be provided via other instruments that either utilize live satellite feeds or access stored satellite data on remote machines.

Consider the introduction of observers into this application scenario. Given the scale of such an application and the speed and complexity of its execution platform, it is easy to imagine that the simulation's progress and results are monitored by multiple scientists in distributed locations. The scientists' interests may vary from wishing to see the "big picture," to investigating in detail subsets of the simulation's output, to collaborating with each other via the computational instruments these models implement. The distributed application has additional components that assemble the information needed to drive various interactive displays, by gathering data from the distributed simulation and performing the analyses and reductions required for these displays. Some of these components may themselves have substantial computational or storage needs and require dedicated, additional resources of their own. They may also require access to additional information, as is the case for the atmospheric model's display with which end users compare observational (satellite) data with model outputs in order to assess model validity or fidelity. Moreover, since end users control the set of computational instruments, input and output components, and the displays, the current set of interacting computational instruments will change dynamically, driven by end users' needs or by the current needs of the running simulation. Figure 10 depicts a sample application configuration.

The previous paragraph demonstrates the dynamic nature of computational instruments and their connections. Another attribute of instrument interactions is the bi-directional nature of not only instrument-to-instrument communication but also the communication between instruments and end users and between multiple collaborating users. While it is clear that end users frequently interact with displays and with the analysis components used prior to display, it has become more common for users to also interact with the actual simulations while they are running, perhaps to *steer*

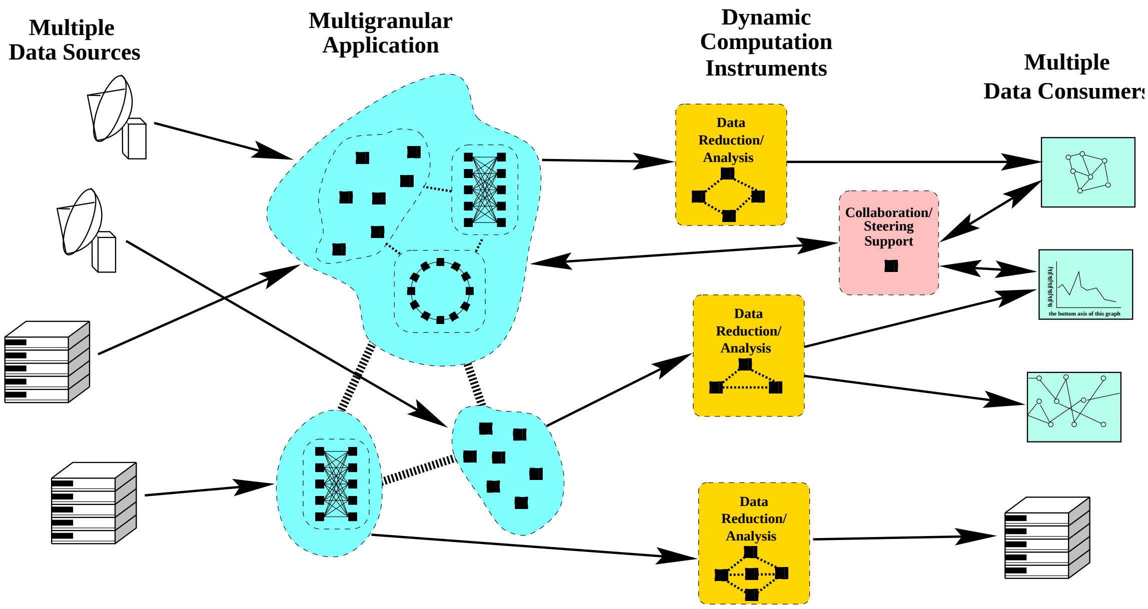

Figure 10: A sample distributed laboratories application configuration.

them toward more appropriate data domains, to enable or disable certain ancillary computations, or to experiment with alternative settings for simulation parameters. The benefits of such dynamic interactions of end users with large-scale simulations are well known for discrete event simulations and have recently been found to offer substantial benefits to scientific and engineering users as well. In addition, it is well known that significant performance improvements may be derived for high performance applications from the runtime configuration of selected application attributes[7], from the online adjustment of their execution environment (*e.g.*, process migration), and from the adaptation of underlying operating system or communication facilities[3].

Taken as a whole, this application scenario has requirements for data management and exchange that go well beyond those of a traditional parallel or distributed application. In particular, it presents a need for efficient high-performance data exchange coupled with data location and analysis in a highly dynamic heterogeneous environment. DataExchange is targeted to meet the process/socket -level communication needs of applications in this environment. These needs are discussed in more detail in the next section.

4.1.2.2 Communication Requirements of Distributed Laboratory Applications

Flexible High-bandwidth Data Transfer. The need for high bandwidth and low latency in communication is common to most high performance applications, and those running in the distributed laboratory are no exception. However, in addition to these commonly known communication demands, distributed laboratory applications have several additional communication demands:

1. *Data-driven, time-constrained computations* – in addition to satisfying the communication demands internal to a laboratory's parallel computational instruments, the communication infrastructure must also provide for suitable latencies of the data transfers from input engines to computational instruments, from instruments to observers' displays, and between multiple, collaborative displays.

2. *Scalable communications* – because any number of different data generators and consumers may co-exist in a large-scale laboratory, it is not sufficient simply to transfer data, but the infrastructure must also permit suitable transformations on the data as it is being transferred, and it must provide mechanisms for efficient data distribution and collection among large numbers of consumers and providers.

As a result of these additional two demands, the distributed laboratory's communication infrastructure cannot rely on inefficient techniques like ASCII transmission of data to handle such issues as heterogeneity in the execution environment. The infrastructure must also facilitate the association of computations with data transfers, as explained in more detail in Section 4.1.2.2.

Operation in a Diverse Environment. The application environment described above is clearly heterogeneous, but it can perhaps be more strongly described as *diverse*. This means that, in addition to operating in an environment where machine-level representations of elementary types may differ, the various components of the application may be developed in a much less tightly coupled manner than a traditional parallel or distributed program. That is, they may not all be produced by a single group or organization and they may include "generic" components designed to operate in many situations.

One implication of this application diversity is that it is not generally practical for all of the components operating in the distributed laboratory to agree at compile-time on the formats of all data records to be exchanged during execution. Even if agreeing on a common and consistent format for the data were possible, systems depending upon such agreement are very fragile and unsuitable for a dynamic environment. This is a problem that has not been addressed in the past by high-performance communication systems. The most obvious approach to operating in an environment this diverse is to transmit data in some easily-interpreted free-form manner, such as encoded in ASCII, and to parse it upon receipt. This approach achieves the required flexibility, but at the cost of seriously compromising the efficiency of the communication system, and so it is not an option for a distributed laboratory.

Dynamic Connection of Clients and Data Flow Management. In the distributed laboratory environment, computational agents may be very long-running, and physical instruments may always have data available. In this environment it is natural to expect displays, their associated data processing agents and other components of the execution environment to come and go as computations proceed. To accomplish this, external clients must be able to locate and connect to data sources dynamically. This functionality tends to be lacking in communication tools created with relatively static topologies in mind (*e.g.*, MPI). Even those which have dynamic client creation may not support external connections or have adequate client/server location facilities (*e.g.*, PVM).

The potential for dynamic connection of clients also means that the flow of data and the processing needs experienced by these flows in the distributed laboratory are themselves very dynamic. In particular, it implies that senders may not have *a priori* knowledge of the identity of all possible receivers. This is a problem for communication systems which require destinations to be explicitly specified by the sender, such as traditional send/receive and RPC-style communication mechanisms. One can, of course, program the application such that it responds to the dynamic connection of clients and automatically routes information flows to interested parties. However, such implementations tend to make inefficient use of network resources[45] and to require the application programmer to create some complex code on top of the basic one-to-one communication mechanisms. Using a generative, or publish-subscribe, communication model instead hides the complexity inside the communication layer and gives that layer more flexibility exploit the capabilities of the underlying network.

Information Filtering and Analysis. As already stated above, a communication infrastructure for distributed laboratories must manage both data flows and the necessary computations controlling those flows and their content. The resulting richness and dynamic nature of the environment and its traffic create additional opportunities for optimization and improvement. For example, a capable communication system may take advantage of external knowledge of the application and reduce overall communication requirements through the use of application-specific filtering and analysis. Because of the application-specific nature of the computation, the main contribution a communication infrastructure can make is in easing the construction of filters and computational agents and providing simple mechanisms through which application-level routines can be activated.

4.1.3 Contributions of DataExchange

The DataExchange library addresses the requirements described in Section 4.1.2. First, DataExchange is based on PBIO[6], a binary I/O package which provides a variety of mechanisms for handling operations in a diverse environment. Second, DataExchange provides facilities for naming and locating data sources, for dynamically connecting to those data sources and for automatic and configurable redirection of data flow. Third, to support flexible data processing, analysis and reduction, DataExchange also offers an active-message-style processing message processing facility. This processing facility can also serve to further refine and extend DataExchanges data flow management functionality by making content-based message forwarding decisions. The remainder of this section will briefly examine these features of DataExchange and, where applicable, compare them with the features of similar packages.

4.1.3.1 Binary I/O Support

Most modern communication libraries provide at least some support for transparently transferring information between machines with differing data representations, but the extent of support varies. PVM[14] allows per-message construction and transfer of information through packing and unpacking of elementary data elements in communication buffers. The application is responsible for ensuring that unpack operations occur in the same order as pack operations. MPI allows the specification and use of user-defined data types in communication. User-defined types in MPI can be defined recursively to create structures which include elementary and user-defined data types and arrays of those types. These user-defined types may be used directly in send and receive operations and the MPI library will perform any necessary packing and unpacking operations. In MPI, type matching between sender and receiver depends upon the type's signature, which is the sequence of elementary data types in the type definition, and the application is responsible for ensuring that the types used by the sender and receiver match. Matching types must have the same sequence of field types, though they can be at different offsets in the record.

While the facilities for user-defined data transport provided by PVM and MPI suffice for the relatively tightly-coupled domain of parallel programming, they are less adequate for many cooperating programs that would make up a distributed laboratory. In particular, consider the problem of adding a new field to a widely used message format. Ideally, adding a new field to an existing message format should only affect those programs which must use or operate on that field. Programs which are not interested in the new field or which should merely forward its data without interpretation should not require modification. However, because both PVM and MPI require senders and receivers to agree on type signatures, adding a new field requires a simultaneous upgrade to all the programs in the lab that might operate on messages of the original format. This is obviously impractical in widespread and diversely managed environments like distributed laboratories.

Similar problems arise in trying to create reusable distributed laboratory tools, the equivalent of instruments in a physical laboratory. One would like to have displays and filter programs that could be applied to a variety of programs and situations in the distributed lab without requiring *a priori* knowledge of the data formats involved.

However, message passing systems which rely on implicit receiver/sender knowledge of the data formats being exchanged do not generally transmit sufficient information for an external party to meaningfully interpret a message in an unknown format.

To address these types of issues, DataExchange relies on PBIO to support its data transfer operations. In addition to other features, such as named data types and the ability perform type conversion between matching fields of different basic types, PBIO supports more elaborate and flexible type matching than either PVM or MPI. In particular, PBIO supports type matching based on *field names* rather than just elementary types. PBIO's type matching is also more flexible in that receivers are not required to specify every field in the types to be exchanged, but need only specify the names and types of the fields that they are interested in. Receivers will automatically extract the named fields of interest and ignore unspecified fields.

Name-based type matching while preserving efficient transmission is possible because PBIO transmits meta-level **format descriptions** before transferring any data records. A format description consists of the name of the record (message) format, and the names, sizes, offsets and data types of each of the fields in the record. A previously registered format may also serve as the basic type of a field, allowing the creation of complex nested record types. PBIO also supports fields which are statically sized one or two dimensional arrays or dynamically sized one dimensional arrays of simple or derived datatypes. Record meta-information is transmitted only once. Thereafter, transmission occurs in the writer's native format, and the PBIO library on the receiver transparently handles discrepancies between the writer's format and the format required by the reader. Transmitting in the writer's native format allows complex messages to be sent over the wire without any copy or "buffer building" stage. In the case of transfers between homogeneous machines, the only additional overhead imposed by PBIO is the transmission of a 4-byte format ID.

Because PBIO record meta-information is transmitted on the wire rather than relying on implicit knowledge in the sender and receiver, the task of creating generic tools and displays that can be "plugged in" to applications in the distributed laboratory is eased. Besides the ability to translate between wire and native record formats described in the previous paragraph, PBIO provides a variety of facilities for manipulating format meta-information and extracting individual fields. These functions support the special needs of programs which are required to process a portion of the data in a record and forward the remainder untouched.

4.1.3.2 Location, Connection and Flow Control

Section 4.1.2.2 describes the dynamic nature of the distributed laboratory environment. This nature creates the requirement that a new display be able to locate and connect to information sources and that communication topology be dynamic at run-time.

Dynamic location of communicators is seldom directly addressed by communication packages. Basic socket implementations require the use of a port number that is known to all parties and there are no common mechanisms for publishing a dynamically created name/port number association. PVM supports communication with a dynamically forked remote process, but lacks the ability to initiate communication between two unrelated processes. MPI contains no support for initiating communication between unrelated processes, though MPI-2 will provide some primitive communicator location facilities. CORBA[37] recognizes the problem of dynamic location and includes a specification for the CORBA Name Service (which is getting widespread implementation). The Spring Operating System[40] also has an extensive name service. However, Spring is not widely used, and the CORBA name service is not useful outside of the CORBA domain. While a naming service to support communication in the distributed laboratory need not be overly complex, it must be available.

The ability for a client to selectively receive only a portion of the data available from a particular source is also a capability that is seldom supported by communication libraries. However, the distributed laboratory environment has the potential for large flows of scientific and monitoring data. While clients always have the option of receiving everything and discarding that data which is not of interest to them, adding the ability for them to specify what is interesting so that the uninteresting is not transmitted (and so does not take bandwidth) may be key to reducing overall network resource demands.

DataExchange contains facilities to support the location, connection and dataflow control needs of a dynamic communication environment. Its simple name server helps clients locate and connect to data sources in the distributed laboratory. DataExchange also allows clients and data sources to manage the redistribution and transmission of data on a per-message-type basis. This allows DataExchange to support highly dynamic applications while limiting unnecessary network overhead. These facilities are explained in greater detail in Section 4.1.4.

4.1.3.3 Information Processing and Filtering

In an environment like the distributed laboratory, it is useful to be able to create generic filter and display programs that can be assembled in a plug-and-play manner. Both filter and display-type programs are often constructed in an event-driven programming style. Like a variety of other packages, DataExchange provides the ability to bind application level functions to communication system events. DataExchange events include the connection of new clients, shutdown of client connections, and message arrivals. DataExchange also allows different functions to be bound to each type of message. With this level of support, simple filter programs can often be reduced to a very few lines of code.

This basic support for binding application functions to message arrivals also provides for the creation of data-dependent routing and forwarding of messages. Data-dependent routing could be used to supplement the record-type-based forwarding that is built in to DataExchange.

4.1.4 The DataExchange Communication Library

DataExchange facilitates communication between processes in a distributed laboratories environment through a set of services. For example, a visualization client receiving large amounts of data from a global atmospheric model may at some point no longer need messages containing updated species concentration values. Instead of being forced to discard the species concentration messages upon arrival, DataExchange provides a means for the visualization client to issue a command to the server to stop sending messages of that kind, thus freeing bandwidth that would otherwise be wasted. DataExchange currently executes on Sun Solaris, SunOS, SGI, Linux, RS6000, and Windows NT platforms. A complete description of the library interface can be found in [9].

4.1.4.1 Concepts

Before discussing the services provided by DataExchange, we introduce several general concepts, an understanding of which is useful for understanding those services. One enables DataExchange communication by creating a *DataExchange context*. The context, specific to a single process, supports a group of related connections. The separate communication context insulates communication internal to the library from external communication and provides a means of associating related communication channels. A DataExchange context is created with a call to `DExchange_create()`, which creates the context by initializing a `DExchange` data structure. The returned pointer to the data structure is attached to subsequent calls to the library.

An individual connection has associated with it a *DEPort structure*. The DEPort structure maintains information about the connection, such as its hostname and port id. A DEPort structure is created automatically when a connection is established.

As described in Section 4.1.3.1, efficient transmission is possible because PBIO uses meta-level *format descriptions*. An application registers its format descriptions within its local DataExchange context. The DataExchange then forwards the format registration to all connections within its context. This forwarding step is necessary to ensure that data of that type can be handled by all processes seeing such data.

Because format descriptions may differ slightly due to byte ordering and word size differences between platforms, DataExchange has mechanisms to resolve differences in format descriptions when encountered. This resolution takes place only once within the context of the DataExchange with which the format was registered. That is, a format description received within one DataExchange context will not invoke format resolution against a format description received within another DataExchange context. Formats are registered using `DExchange_register_format()`. Each format is assigned a format ID which is used to identify the data type on subsequent operations.

4.1.4.2 DataExchange Services

Dynamically Locating and Connecting to Data Sources In a distributed laboratories environment it is natural to expect displays, their associated data processing agents, and other components of the execution environment to come and go over the course of time. A library that supports dynamic locating and connecting to data sources must provide several services: dynamic location of communicators, dynamic connection to the communicator, and the ability to direct events to the new client without disrupting the data source.

Dynamic location of communicators in an environment characterized by a non-static topology is seldom addressed by communication packages. DataExchange provides a means for locating communicators in such an environment through a *group server*. The group server is a simple name service that relieves an application of the responsibility of knowing the specific location <hostname, port> of a connection point. The application need only know an agreed upon name to locate and connect to a connection point. The group server implements its service as a repository of name, connection pairs. The name, a <user name, application name, group type> triple, uniquely identifies a connection point. *User name* is usually the userid associated with the process establishing the group. If not given, it is derived from the execution environment. The *application name* is an agreed upon name identifying the set of communicating processes. *Group type* is provided to further delimit the application name. Group type is used in cases where an application supports several connection points for different purposes. For example, one of the processes participating in the global atmospheric simulation may establish a connection point for all distributed components of the simulation. That same process may need to establish a communication point to enable communication between the model and the filters and visualization/steering clients. The two communication 'groups' would be established with the same application name but differentiated by group type.

The group server accepts queries based on any field of the <user name, application name, group type> triple, and supports both exact matching and regular expressions. In response to a query, the group server returns a list of connection points from which the process originating the query can select.

Dynamic connection to a data source requires the ability of an event-based application to respond to arriving events, including connection request events. Event-based processing is supported by the `DExchange_poll_and_handle(DExchange de, int block)` routine, the basic event handling call in DataExchange. The poll and handle function is analogous to the `XtAppNextEvent()/XtAppDispatchEvent()` pair in X windows toolkit programming where a DataExchange event is a basic communication occurrence, such as a connection request or message arrival. The function queries all communication channels associated with a DExchange context and processes events from each connection having input ready. The `block` parameter specifies whether or not DataExchange should block (suspend program execution) waiting on an event if none is ready.

In addition to services for dynamically locating data sources and accepting connection requests dynamically, a communication library meeting the dynamic needs of distributed laboratories must also provide a means for a newly connected client to begin receiving data for which it has registered an interest. DataExchange provides such a service through default forwarding behavior. A DataExchange server will automatically forward all messages it receives to all connections that have registered an interest in the messages. This default behavior can be modified through control functions described in Section 4.1.4.2 that allow the selective forwarding and blocking of messages based on format description.

Analyzing and Filtering Data Many event-based applications must handle arriving messages of multiple data formats, where the action to be taken is often specific to the data format. DataExchange provides a means for associating an action to a data format through **action routines**. Action routines are associated with a particular data format and invoked when a message of that format arrives. Similarly, applications may register routines to be called when a connection is established or shutdown. Action routines may be registered and unregistered with DataExchange at any time.

DataExchange provides two types of action routines for handling incoming data, functions and filters. **Functions** are called *after* any implicit data forwarding whereas

filters are called *before* data forwarding and can return a value which preempts further processing. Functions are useful for gathering statistical or debugging information on the event stream. Filters, on the other hand, are useful for filtering events, reordering an event stream, or consuming events and generating derived events. Functions and filters are associated with specific data formats though the same routine can be registered as a handler for multiple data formats.

Controlling Flow of Data As mentioned previously, the default DataExchange behavior is to resend all data received from one connection to all of its other connections (except the sender). This mechanism is useful for constructing interesting servers and managing data flow in networks of communicating DataExchange programs. The DataExchange library allows control of this resending in ways which can be loosely divided into *record forwarding* and *record blocking* controls. Record forwarding and blocking controls both affect the implicit resending of received data, but in different ways. Forwarding controls are available to the DataExchange program performing the resending operation and allow that resending to be controlled on a per-format basis. Forwarding controls are therefore local controls in the sense that a DataExchange program calling those routines affects its own resending. The complementary record blocking controls are instead used by the DataExchange programs *to whom data is being resent* to affect the type of data that is resent to them. In particular, by using the record blocking controls a DataExchange program can tell another communicating program *not* to send it records of a particular format. Record blocking controls thus affect resending on a per-connection basis, but they differ from the forwarding controls in that blocking routines invoked by one DataExchange program affects resending that occurs *in other communicating programs*. In this sense, record blocking is a remote operation and is useful to any DataExchange program, be it client or server, that wants to influence what data is sent to it by others to whom it is connected. On the other hand, record forwarding controls affect resending in the DataExchange program invoking the controls and are thus a local operation. In both record forwarding and blocking, DataExchange's decision to send or not send a particular record to a particular client is based solely on the type or format of the record.

4.1.4.3 Sample Application

To understand the utility of DataExchange's features, it is useful to examine a sample application. In this section we examine the global atmospheric model previously mentioned. This application is a collaborative effort between computer scientists and atmospheric scientists at Georgia Tech. Models like these are important tools for answering scientific questions concerning the stratospheric-tropospheric exchange mechanism or the distribution of species such as chlorofluorocarbons (CFCs), hydrochlorofluorocarbon (HCFCs) and ozone. Details of the model's solution approach, parallelization, and performance results are described in [22].

A common scenario in a distributed laboratories environment is to connect one or more visualization/steering/filtering/analysis clients to such a scientific model. Our simple example consists of three applications: a supplier, a filter/forwarder, and a consumer. The *supplier* (i.e., the atmospheric model) generates ozone concentration data that is sent to the clients. To keep our example simple and focused on DataExchange communication mechanisms, we hide the complexity of data generation in a simple dummy procedure in the example below. The second application is a *filter/forwarder* that accepts connections from the supplier and visualization clients, then filters records it receives based on a user supplied range of levels. The final component, the *consumer,* emulates the communication behavior of the visualization/steering client. It accepts filtered records for processing but as was done for the supplier, any processing or visualization is left unspecified. Processing model data

```
typedef struct _Limits {
    int lower_bound;
    int upper_bound;
} Limits;

typedef struct _XMix {
    int numSteps;                 /* logical time step           */
    int level;                    /* atmospheric level:  0..36   */
    int count;
    double * mix_ratio; /* species concentration in spectral form  */
} XMix, *XMix_ptr;

static IOField x_mixing_list[] = {
    {"numSteps", "integer", sizeof(int), IOOffset(XMix_ptr, numSteps)},
    {"level", "integer", sizeof(int), IOOffset(XMix_ptr, level)},
    {"count", "integer", sizeof(int), IOOffset(XMix_ptr, count)},
    {"mix_ratio", "float[count]", sizeof(double), IOOffset(XMix_ptr, mix_ratio)},
    {NULL, NULL, 0, 0}
};
```

Figure 11: Types and format descriptions used in example

would include transforming the data from its spectral form used in the model to a grid form needed to make the data useful for display using a 3D graphics library such as SGI Open Inventor.

Figure 11 contains type and format definitions used throughout the example. The first type definition defines a client data structure used by the filter/forwarder application. The second type definition defines a species concentration record. Its format description for purposes of binary encoding/decoding is given as the static array `x_mixing_list`. The `XMix` type and `x_mixing_list` format description are used in the supplier, forwarder, and consumer processes.

The supplier, whose code is shown in Figure 12, creates a DataExchange context with a call to `DExchange_create()` and registers a data format, *x mixing ratio*. Registering the format has the effect of telling DataExchange how to interpret the buffer it receives when subsequent calls to `DEport_write_data()` are made.

```
void main()
{
    XMix * xmix;
    DEPort dep;
    comm_group_return group_return;
    int format_id;
    DExchange de;
    de = DExchange_create();
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    group_return = matching_comm_groups("atmosphericModel","model" );
    if (group_return->count > 0)
        dep = DExchange_initiate_first(de, group_return, FALSE);
    format_id = DEget_format_id (de, "x mixing ratio");
    while (1) {
        xmix = gather_x_mixing_data();
        DEport_write_data (dep, format_id, (void *) xmix);
    }
}
```

Figure 12: Supplier of Species Concentration Data

The supplier then queries the group server with the application name, group type pair <"atmosphericModel", "model"> which returns a list of connection point(s) matching the query. Connection establishment is accomplished with the subroutine `DExchange_initiate_first()`, where the first parameter is the DataExchange context to which the connection will belong. The second parameter, `group_return`, is the structure returned by the group server. It contains a list of connections satisfying the query. `DExchange_initiate_first()` will traverse the list, attempting to make a connection with each connection point; it will return upon making the first connection. The final parameter is the block_by_default parameter. The value of FALSE tells the remote server, the forwarder in this case, that it should not block delivery of data to this client. The return value is a DEPort value that is used in subsequent reads/writes. The model then obtains the ID for the format it previously registered, subsequently dropping into a loop to gather and send data.

```
static int xMixFilter(DExchange de, DEPort dep, int format_id,
                      void *data, int data_length, void * cli_data_arg)
{
    XMix *rec = (XMix *) data;
    Limits *lim = (Limits *) cli_data_arg;
        /* return value controls forwarding */
    return !(rec->level < lim->lower_bound || rec->level > lim->upper_bound);
}

void main(int argc, char *argv[])
{
    char *group_id = NULL;
    Limits *cli_dat = malloc(sizeof(Limits));
    DExchange de = DExchange_create();
    cli_dat.lower_bound = atoi(argv[1]); /* lower and upper bounds for */
    cli_dat.upper_bound = atoi(argv[2]); /* filtering of atmospheric levels */
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    DExchange_register_filter(de, "x mixing ratio", xMixFilter, (void*)cli_dat);
    if (DExchange_listen(de, 0) == -1) exit(-1);  /* open socket, select port */
    group_id = setup_comm_group(NULL, "atmosphericModel", "model", 300,
                               DExchange_host_name(de), DExchange_inet_port(de));
    while (1) {
        DExchange_poll_and_handle(de, TRUE);
    }
}
```

Figure 13: Forwarder that filters species concentrations depending upon a user specified range of levels.

The forwarder, as shown in Figure 13, creates a DataExchange context, registers the format *x mixing ratio* and the action routine `xMixFilter`. `xMixFilter` is a handler routine invoked upon receipt of messages of type *x mixing ratio*. Once the action routine is registered, the forwarder establishes a group with the group server with a call to `setup_comm_group()`. The group is established with the name <"atmosphericModel", "model">. The final parameter, 300, specifies the length of time in seconds for which the group will remain active. Once the group is established, the forwarder waits on events, invoking a handler which filters species concentration records for levels outside the range specified by the user supplied data, then forwards qualifying records to all connections except the one on which it received the message.

The final application component, shown in Figure 14, is the consumer. The consumer registers the format *x mixing ratio* and the action routine `xMixHandler`. Though the format registered is the same as is used by the supplier and filter/forwarder, it need not be. DataExchange handles differences in byte order, field ordering and field count as well. `xMixHandler` handles species concentration messages (of data type *x mixing ratio*). The client queries the group server to initiate communication with the filter/forwarder in the same manner as was done by the sender. The client then drops into a while loop, servicing events. An arriving message automatically invokes the handler to process the message. To illustrate blocking, we have included the *if* statement following `DExchange_poll_and_handle`. When logical timestep 5000 is reached, `DEport_set_format_block()` is called with the format *x mixing ratio*. The effect of the call is to block further transmission of species concentration messages at the filter/forwarder.

4.1.5 Performance

To a great extent the performance of DataExchange depends more strongly upon the performance of the underlying network transport layer than on any characteristic of

```
int numSteps;                           /* current logical time step */
static void xMixHandler(DExchange de, DEPort dep, int format_id,
                        void * data, int data_length, void * cli_data_arg)
{
    XMix * rec = (XMix *) data;
    numSteps = rec->numSteps;
    process_species_concentration(rec);    /* do some work */
}

void main()
{
    DEPort dep;
    comm_group_return group_return;
    DExchange de = DExchange_create();
    DExchange_register_format(de, "x mixing ratio", x_mixing_list);
    DExchange_register_function(de, "x mixing ratio", xMixHandler, NULL);
    group_return = matching_comm_groups("atmosphericModel", "model");
    if (group_return->count > 0)
        dep = DExchange_initiate_first(de, group_return, FALSE);
    while (1) {
        DExchange_poll_and_handle (de, TRUE);
        if (numSteps == 5000)
            DEport_set_format_block(dep, "x mixing ratio", TRUE);
    }
}
```

Figure 14: Consumer accepts filtered species concentration messages until logical time reaches 5000.

DataExchange itself. Some DataExchange features, such as the ability to block the transmission of unnecessary data, may yield performance gains over approaches that do not have such support. However, the extent of these gains depends strongly upon the application and the amount of unnecessary data that DataExchange prevents from being transmitted. Since a particularly well-crafted application would transmit no unnecessary data, the strength of DataExchange is that it facilitates the construction of plug-and-play applications that are well-crafted in that respect.

However, a unique aspect of DataExchange that does affect performance is its support for application diversity. That is, DataExchange is able to support the rapid exchange of binary data between application components without *a priori* agreement on the fields being exchanged, much less the types, sizes and layout of those fields. Because PBIO always transmits information using the native data layout of the sender,[1] this implies that a *format conversion* must be performed on the receiving end before the message can be processed as native data. When identical structures are exchanged between homogeneous machines, no conversion is necessary. But some conversion processing is required when data is exchanged between application components whose data representations vary in the number, types or layout of fields in the data, or if the communicating machines have differing native data representations, such as using different byte orders. Furthermore, because DataExchange has no *a priori* knowledge of the native formats used by the program components with which it might communicate, the precise nature of this format conversion cannot be determined until run-time. This conversion is another form of the "marshaling problem" that occurs widely in network communication. That marshaling can be a significant overhead is well known[5], and tools such as USC[29] attempt to optimize marshaling with compile-time solutions. Unfortunately, the dynamic form of the marshaling problem in DataExchange rules out such static solutions. As described above, the

[1] As described in Section 4.1.3.1.

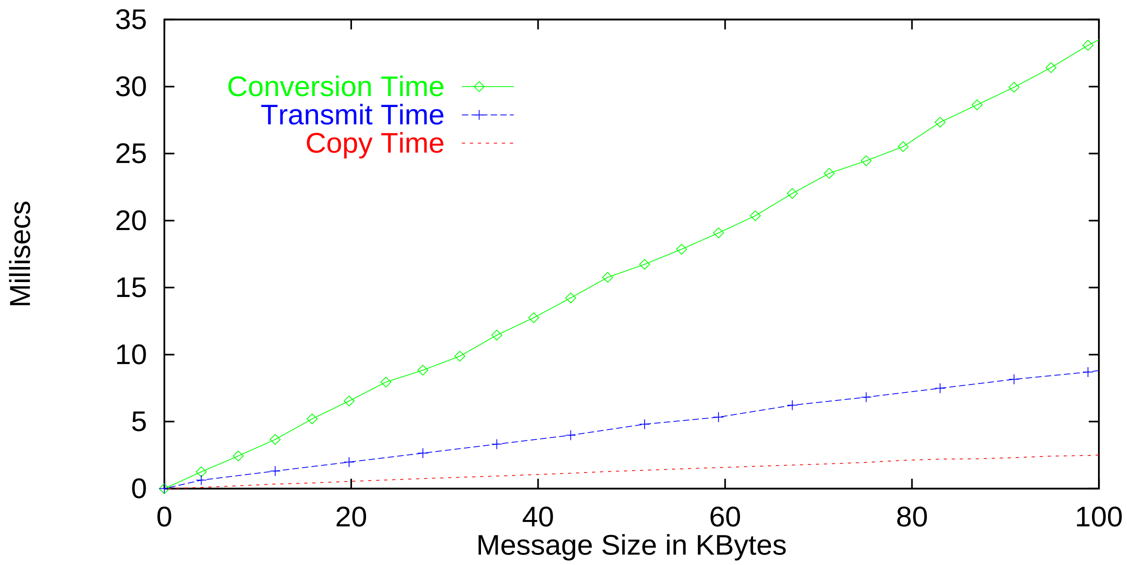

Figure 15: Comparison of convert, copy and transmit times.

conversion overhead is nil for some data exchanges, but for others, it can be significant. To put the overhead in context, Figure 15 shows a comparison of the times required to transmit, copy and convert a message.

The particular values given in Figure 15 depend upon a number of factors. The message transmit time is based on a 15Mb/sec transmission rate. The copy time is the time required to perform a bcopy on an UltraSparc 140. While both of the previous values vary strictly as a function of message size, the amount of time required to convert the message from the "wire" format to the receiver's native format is highly variable, depending upon the precise differences between the two formats. In this case, the values are derived from a situation where both byte-swapping and a size conversion for each data element were required to convert from wire to native format. While this represents a near worst case for data conversion times, the comparison is far from satisfactory. In this worst case, data conversion is taking three times as long as the time to transmit the message over the network and an order of magnitude longer than a simple move operation. In many situations, this might be an unacceptably high price to pay for supporting application diversity.

While a portion of the costs involved in doing the message conversion of Figure 15 are the simple consequence of the raw number of operations involved in performing the data conversion, a significant fraction of the overhead is due to the fact that the conversion is essentially being performed by an interpreter. This implementation choice and resulting excessive overhead is a direct consequence of DataExchange's lack of compile-time information about the precise nature of the conversion required. One would hope that a more efficient implementation would be able to bring the cost of conversion to approximately the time required for the bcopy operation.

Since DataExchange is targeted towards high performance applications which can ill afford the increased communication costs associated with interpreted conversion, we have recently begun experimenting with dynamic code generation in an attempt to reduce these costs. The results, shown in Figure 16, have been highly encouraging.[2] For the type of conversion used in Figure 15, the dynamically generated conversion routine operates between three and four times as fast as the interpreted version. This improvement removes conversion as the dominant cost in communication and brings conversion costs to near the level of a copy operation.

However, employing dynamic code generation for conversions means that DataExchange must bear the cost of generating the code as well as executing it. Because the format information in PBIO is transmitted only once on each connection and data tends to be transmitted many times, conversion generation is not normally a significant part of DataExchange overhead. Yet that overhead must still be considered to determine whether or not the use of dynamic code generation results in performance gain.

[2]The dynamic code generation described in this section was implemented using the Vcode package[10] developed at MIT by Dawson Engler.

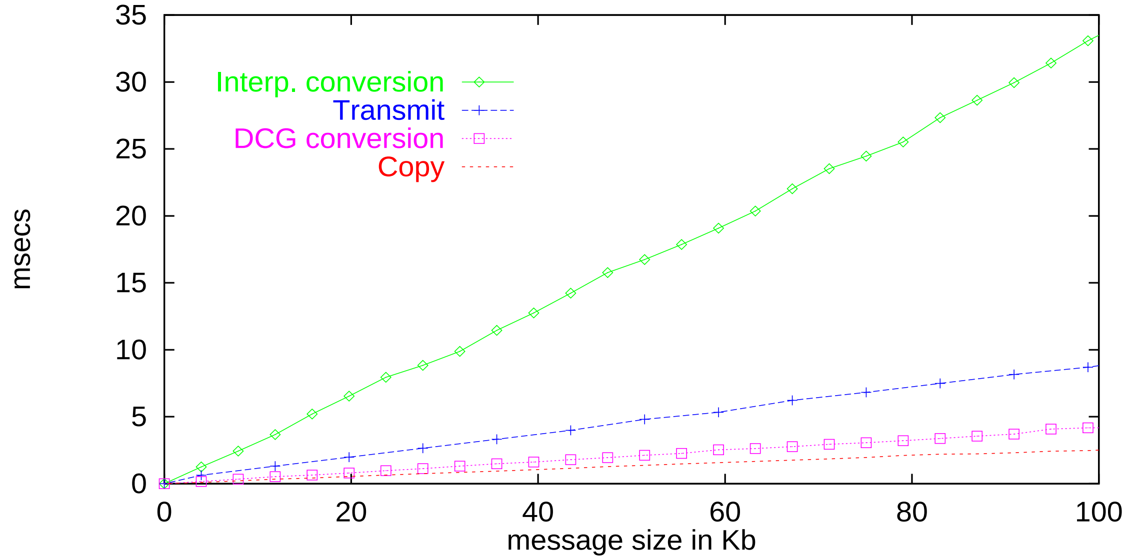

Figure 16: Comparison of interpreted conversion, dynamically code generated conversion, copy and transmit times.

The proportional overhead imposed by doing dynamic code generation varies dramatically depending upon the internal structure of the record. This differs from the situation in Figure 16, where the worst-case conversion run-time is more dependent upon the size of the message than its structure. To understand this variation, consider that the conversion code for a record which contains large internal arrays consists of a few `for` loops that process large amounts of data. In comparison, a record of similar size consisting solely of independent fields of atomic data types requires custom code for each field. The result is that for records which consist solely of arrays, dynamic code generation almost always benefits performance. For array-based records of around 200 bytes the time to generate and execute dynamic conversion code is less than the time to perform an interpreted conversion. At that point, dynamic code generation is a performance improvement even for formats that are only used once.

The situation is less clear for record formats consisting mostly of individual atomic

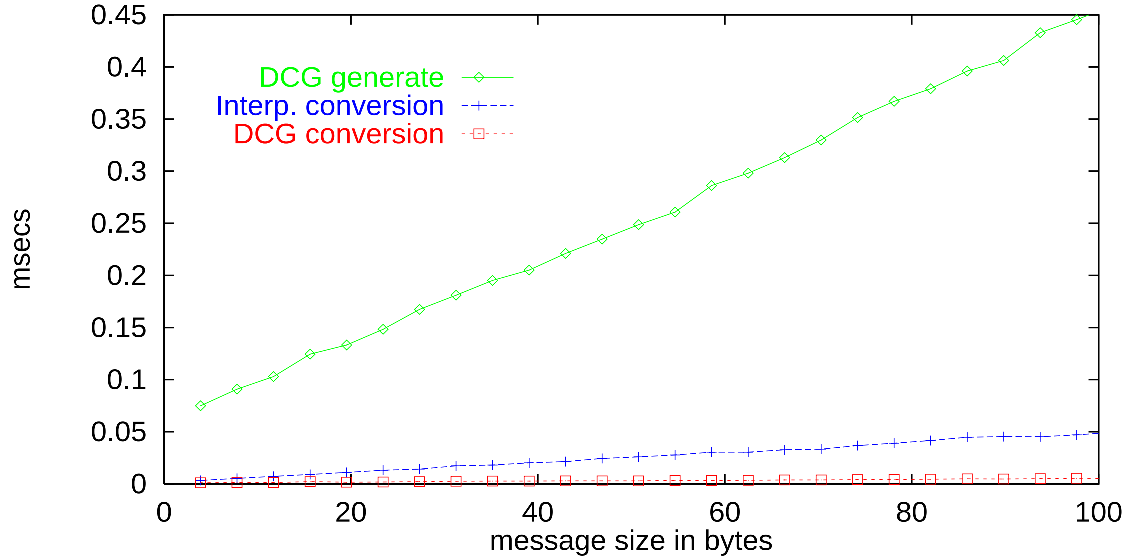

Figure 17: Comparison of interpreted conversion, dynamically code generated conversion, DCG conversion generation times.

fields. Figure 17 compares the times for such records. This figure shows that for this type of record dynamically generated conversions run nearly an order of magnitude faster than interpreted conversions, but the one-time cost of doing the code generation is relatively high. Obviously, if many records are exchanged, the costs will be amortized over the improved conversion times. But for one-time exchanges dynamic code generation for conversions may be more expensive than simple interpreted conversions.

4.1.6 Conclusions

The current trends in high-performance computing applications are towards large loosely-coupled distributed computation and more dynamic models for both the computation itself and for run-time interaction with the computation. Unfortunately, these trends are not well supported by traditional message passing environments.

DataExchange has proven to be a useful and flexible tool in creating complex plug-and-play interactive applications, including a global climate model, abstracted here as a sample application. We also offer results on a novel use of dynamic code generation within the communication library. These results impact DataExchange performance in its support for application diversity, a less restrictive approach to programming cooperative programs that removes the requirement for *a priori* agreement on the contents and formats of information to be exchanged within the application. We show that application diversity can be supported without great impact on communication costs.

4.1.7 Contributions to Thesis

DataExchange was principally designed to address the scenario described in Section 4.1.2. That is, it was designed to meet the stream-style communication needs for interactive applications. As such, it provides an important communication infrastructure and the functions and features described above contribute substantially to the claims of this thesis. DataExchange allows the MOSS infrastructure to easily support plug-and-play clients and efficiently exchange binary information across heterogenous architectures. The efficiency of DataExchange and PBIO are largely responsible for the performance results described in Section 5.1

However, more support was necessary in order to provide all of MOSS functionality. Essentially, raw DataExchange provides the event-stream-style of update routine whose problems are described in Section 2.2. In particular, the DataExchange mechanisms for controlling the flow of records are record-type-based and not suffficient to support efficient and direct routing of monitoring updates between application objects and their mirror objects. Additionally, the function and filter mechanisms in DataExchange, while powerful, do not provide the complete functionality of a distributed object system. In order to fulfill these further needs, the Event Channel and Object Transport Layer packages were developed. The remainder of this chapter describes those packages.

4.2 Event Channels – Control and Dataflow

In order to fulfill it's goal of directly routing monitoring information from its source (such as an application object) to destination (a mirror object), MOSS uses Event Channels, additional communications functionality built on top of DataExchange. These Event Channels are similar in semantics to the Event Services specification in CORBA. CORBA Event Services were designed to be implementable on top of basic object invocations. This section first presents possible object-level implementation of event services as a justification for *not* implementing them on top of object invocations. It then presents the DataExchange-based Event Channel implementation and describes ongoing work. Finally, we describe the manner in which Event Channels make contributions to the thesis.

4.2.1 Event Services Introduction

The CORBA model defined by the Object Management Group (OMG) provides a means by which objects may make and receive requests and responses in a distributed environment. The OMG Common Object Services (COS) specification defines a set of event services to support asynchronous communication between these potentially distributed CORBA objects. Event suppliers produce data that is eventually to be consumed by event consumers. The base model for event communication matches consumers and suppliers in a one-to-one way. An *event channel* is an intervening CORBA object that allows asynchronous event communication between multiple consumers and suppliers. Rather than directly communicating with the ultimate event source or consumer, event channel clients submit and receive events to and from *proxy objects* operating on behalf of the event channel.

While the nature of the interaction between clients, event channel objects and event channel proxies is fairly well specified, the COS specification leaves open a variety of implementation and optimization options. Once events have entered the domain of the event channel via a proxy, the nature of event propagation and delivery is unspecified. Because high bandwidth and minimum latency of event services will be key factors affecting performance of distributed event-based applications, this paper is concerned with possible implementation options for COS event channels.

To provide context and requirements for discussing event channel implementations we first examine the manner in which event channels might be used in a sample application, in this case an online distributed monitoring and steering system. The next section examines possible event channel implementation approaches based on CORBA facilities and reaches conclusions on the limits of their efficiency. Section 4.2.4 presents a set of non-CORBA facilities for providing event services and examines the manner in which they could be used to implement COS event channels.

4.2.2 Event Channel Use

In order to propose and evaluate possible implementation options for CORBA event channels, it is useful to understand how event channels might be used in an application. In this section we describe an online distributed monitoring and steering system and discuss how it might be implemented as a set of CORBA objects and how event channels might be used in its construction.

4.2.2.1 Online Monitoring as a Distributed Application

The relationship between a distributed application, a monitoring system, and the displays and interaction mechanisms it drives is shown abstractly in Figure 18. The high-level requirements for an online monitoring system are that it interact with the application, collect, reduce and analyze data and distribute the results to the appropriate displays. Because monitoring systems are typically employed in situations

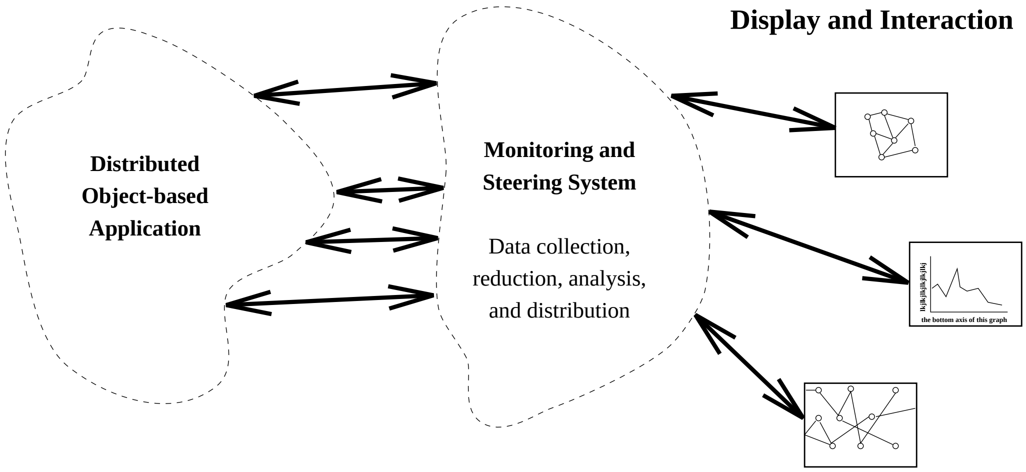

Figure 18: The relationship between an application, monitoring systema and displays and interaction mechanisms

where application perturbation must be kept to a minimum, the monitoring system's interactions with the application are kept as non-intrusive as possible, and its processing demands are met by resources outside domain of the application. While it is possible to implement the monitoring system in a centralized way with all raw monitoring data being sent to a central location for processing and distribution to displays, other approaches are more efficient and therefore could handle higher event rates before saturating. For example, it is desirable to perform monitoring data reduction and analysis as close to the point of data production as possible to reduce transmission bandwidth requirements. However, because multiple independent data collection processes may be necessary, all of the data for a particular calculation may not be available at single processing site. In this circumstance, more levels of data exchange may be necessary before the particular calculation can be performed. Generally, the data reduction and analysis can be viewed as a dataflow graph as shown in Figure 19, with raw monitoring data entering on one side of the graph and processed data driving displays on the other side. If application steering is to be performed, it may be

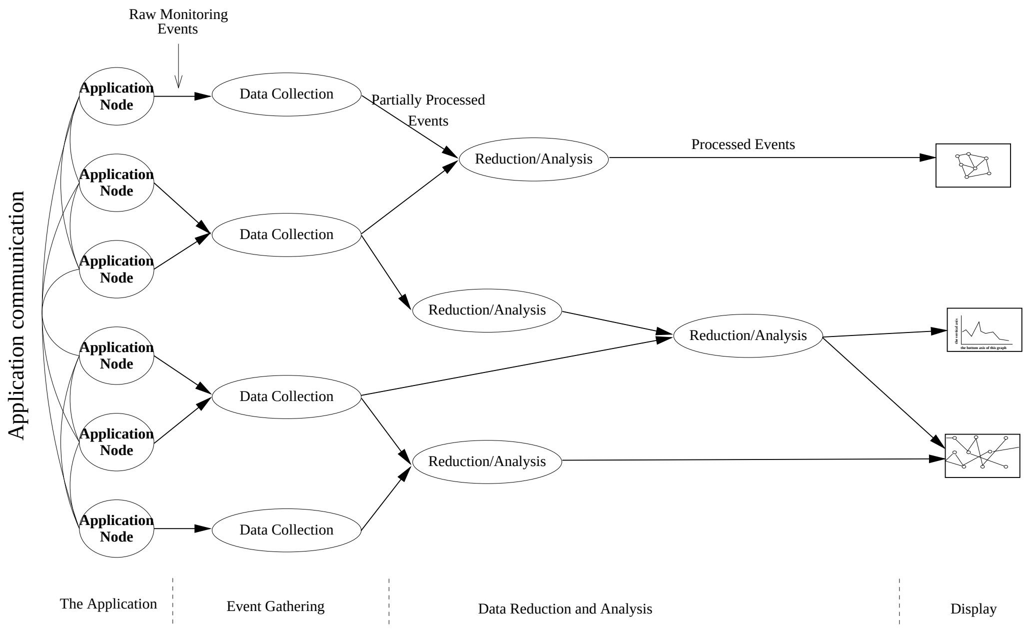

Figure 19: A possible monitoring processing graph showing dataflow.

initiated by human users interacting with displays or algorithmically, through calculations and application changes initiated at some point in the data reduction. However, steering interactions are generally much less frequent than monitoring events, so this discussion will concentrate largely on monitoring event flow.

4.2.2.2 The CORBA/COS Implementation

To achieve the type of application dataflow show in Figure 19, the processing associated with nodes in the graph must be distributed among the available processing resources. In the CORBA model, processing is distributed by assigning CORBA objects, and therefore the calculations associated with their methods, to the available ORB servers. Thus the nodes in the graph of Figure 19 become CORBA object servers (ORBs) containing objects that perform the data gathering, analysis and reduction. Since the monitoring application is naturally event-driven, CORBA event

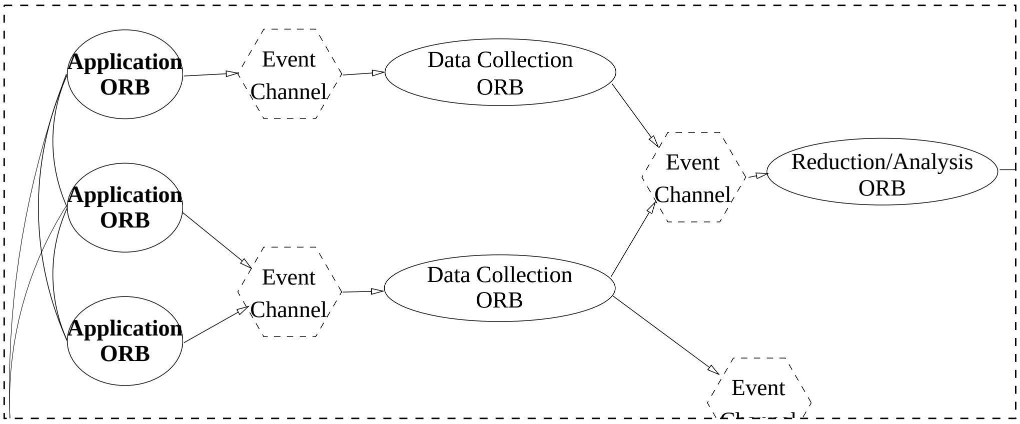

Figure 20: A portion of Figure 19 depicted using event channels as a data transport mechanism.

channels are an obvious choice for communication between these objects. However, even given the basic topology and dataflow description, there are a variety of ways in which event channels could be used. We consider two basic scenarios for decomposing the calculations between the objects in the ORBs: functional and object-oriented.

If the data analysis calculations are decomposed functionally, it is likely that individual objects will either be interested in receiving all events or perhaps a subset of events by type. In this circumstance, it is sufficient to use a single event channel in each place where event transport must occur between ORBs. This arrangement is shown in Figure 20.

However, more complex situations may stretch this simple model. For example, in application-level monitoring where the user is employing monitoring to track the high-level behavior of his application, the data reduction and analysis computations required are more likely to reflect the object-oriented structure of the application. For example, at the first level of interface to the application, objects in the monitoring system may "mirror" certain attributes of objects of in the running program. To support this, it would be necessary for changes in the application objects to generate events which would cause corresponding changes in the mirror objects. Those events in turn might cause changes in other monitoring objects which would have to be reflected in yet other monitoring ORBs or displays. If we assume that applications objects of the same type generate monitoring events with the same type, then we have a situation where just the type of an incoming monitoring event is not sufficient to direct it to the right object in the monitoring system. While it would be possible to designate a single object in each ORB to receive all events of a particular type and then redistribute them to the appropriate object (or simply to have each object receive all events and ignore those that do not apply to it), it seems more natural to employ multiple event channels to segregate event delivery. In this case, instead of single event channels forming the connections between the applications and monitoring ORBs as shown previously, many event channels would be used, each carrying events specific to a particular application object. This is depicted in Figure 21.

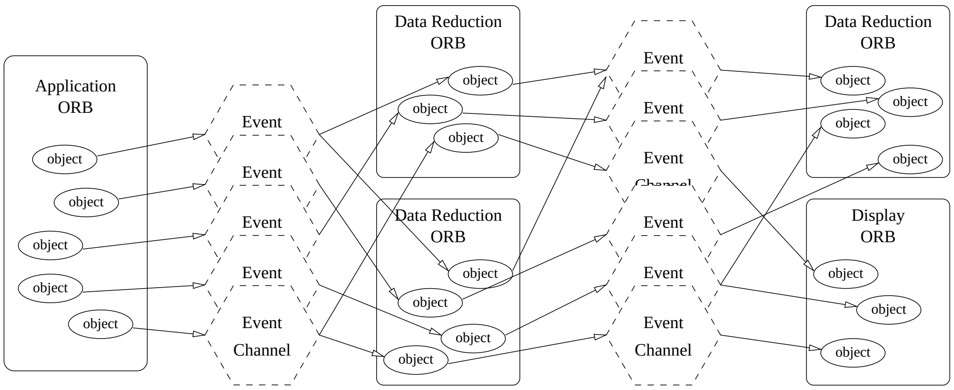

Figure 21: Many event channels may be required to properly segregate events.

4.2.2.3 Some Efficiency Considerations

This analysis of online distributed monitoring leads to some efficiency considerations for COS event channels if this type of application is to be successfully supported. In general, monitoring event delivery needs to be as fast and efficient as possible. Any latency and bandwidth limitations will directly affect the quality and extent of monitoring that is possible without saturating the system. Therefore the overhead involved in inter-ORB delivery of events must be kept as low as possible. The number of event channels potentially required to meet the demands of the object-based decomposition of monitoring computation indicates that the event channel system must also be highly scalable. With these considerations in mind, the next section will present some possible options for event channel implementation.

4.2.3 Event Channel Implementation

In the CORBA Event Services specification, event channels support asynchronous event delivery. Like all of the CORBA Common Object Services, Event Services are designed to be implementable using the basic CORBA object invocation functionality. This section will explain the basic nature of event channels, discuss possible strategies for implementing them using basic CORBA services and consider their performance characteristics.

4.2.3.1 Basics of Event Channels

Abstractly, an event channel defines the domain in which an event is transported. Suppliers attach to an event channel and produce events for it. Consumers attach similarly and receive those events. Producers and consumers can each be either active or passive in the event delivery process. In the CORBA model, event channels themselves are represented by a CORBA object. These objects (and the abstract

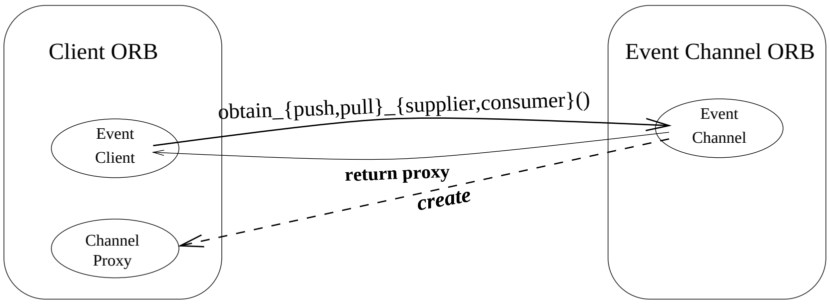

Figure 22: Proxies are created by the event channel and referenced by event clients.

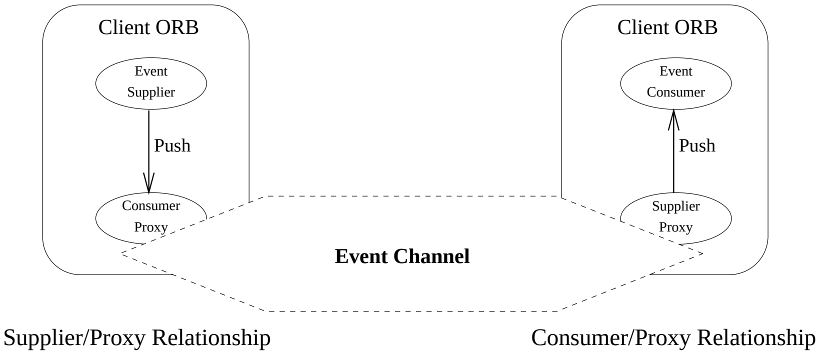

Figure 23: After the initialization phase, event communication is through the proxy.

event transport domains they represent) are created like any other CORBA object. During the attachment phase, producers and consumers invoke a special method in the event channel object to obtain a *proxy* object representing the event channel. This process is depicted in Figure 22. The proxy object is created by the event channel object, and a reference to it is returned to the attaching consumer or supplier. Thereafter, that proxy is the only interface used by the consumer or supplier in their interaction with the event channel. Thus, after the attachment phase, the proxy represents the interface to the abstract event transport domain represented by the event channel. This relationship is depicted in Figure 23. The inner workings of the event channel, such how the events are actually moved from a supplier's proxy to the consumers' proxies is hidden inside the implementation of the event channel proxy. This gives the event channel implementor a variety of options for managing event delivery. The remainder of this section will examine possible implementation schemes.

All examples below are directed towards what COS calls the *Push* model of event communication. In this model, event suppliers call the proxy's Push() method to submit an event to the event channel. To deliver an event to a Push consumer, the consumer's proxy invokes the consumer's Push() method. To avoid overly complicating the implementation discussions we address only the active Push model and assert that the more passive Pull model can be implemented in terms of the Push model.

4.2.3.2 Implementations Based on CORBA

Like most of OMGs Common Object Services, event services were designed to be implementable using standard CORBA requests (object method invocations). The next section will present and discuss several options for implementing COS event channels using the CORBA model.

Terminology The terminology of the COS Event Services specification, while natural when discussing event channels from a client's point of view, is somewhat less natural for discussing event channel implementations. From an event channel's point of view, *ProxyPushConsumer* and *ProxyPullConsumer* objects are where events *enter* the event channel. Events leave the event channel at *ProxyPushSupplier* and *ProxyPullSupplier* objects. Since this terminology is somewhat counter-intuitive, we use the term *supplier's proxy* to name the proxy object attached to clients which supply events to the event channel. Similarly, we use *consumer's proxy* to refer to the proxy object associated with clients which consume events from the channel.

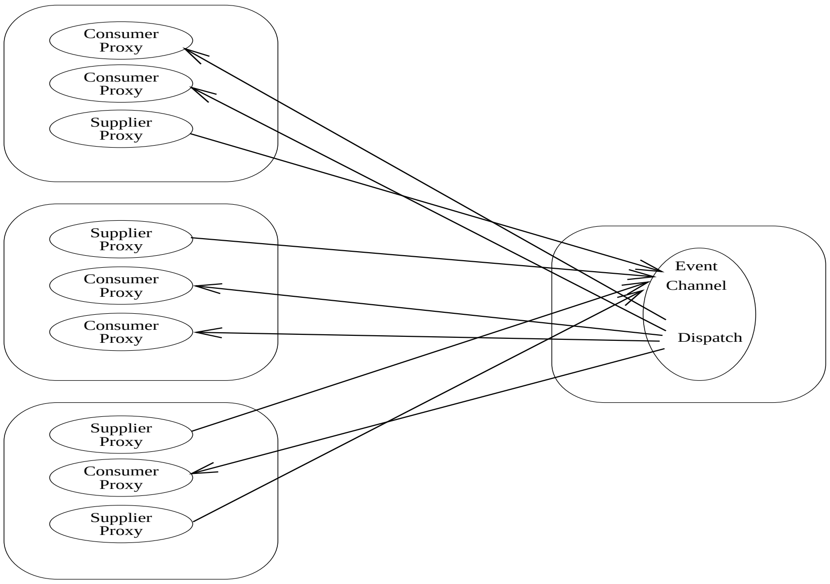

Figure 24: An event channel implementation strategy where channels are active in event distribution.

Active Event Channel Objects Perhaps the simplest possible event channel implementation is shown in Figure 24. In this implementation model, the event channel object itself serves as a hub to which all events are sent for further distribution to event consumers. While this implementation is likely to be the most straight-forward and least complex, it has some significant performance disadvantages. Specifically:

Event delivery is a two-step process: Because events must be first delivered to the event channel and then dispatched to the consumer, latency in event delivery is higher than necessary.

Local event delivery may require remote operations: Event delivery between producers and consumers that are not in the process of the event channel object is performed via remote operations, even when the producer and consumer are in the same process.

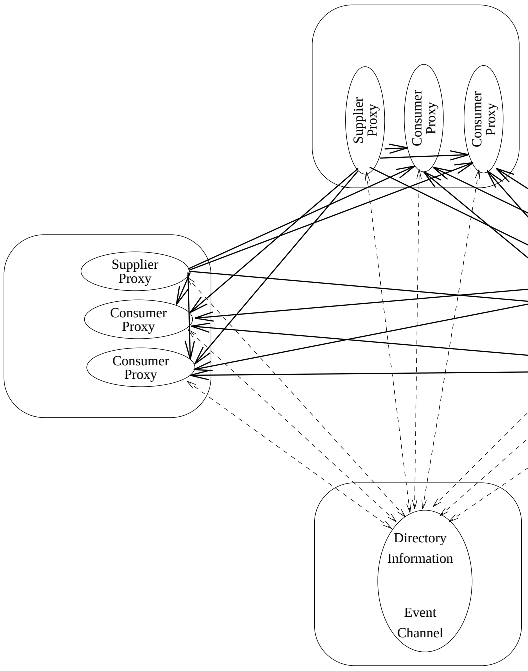

Figure 25: An event channel implementation strategy where proxies directly perform event distribution.

Multiple remote operations to the same process: This channel-centric implementation requires a remote operation for event delivery to each remote consumer. Thus multiple remote operations are required to deliver a single event to remote ORBs containing multiple consumers.

Because event channel subscribers may come and go dynamically, these disadvantages are unlikely to be significantly ameliorated by compile-time analysis of the application or by optimizations of ORB services. However, other implementation approaches show promise for improvement.

Active Event Channel Proxies Figure 25 shows a slightly improved event channel implementation scheme. In this model, the event channel object itself takes no active role in event delivery. Instead, the event channel proxies directly invoke each other in order to deliver events to their destinations. The event channel object is still involved in client subscription and as clients join and leave the event channel, it must update the remaining proxies with current subscriber directory information.

While more complex, this implementation model removes two of the disadvantages which plagued the model of Section 4.2.3.2. Event delivery from proxy to proxy is now performed in one step and delivery between local proxies is performed without remote operations regardless of the location of the event channel object. However, it still requires multiple remote operations to deliver a single event to remote ORBs containing multiple consumers. This disadvantage is addressed in the implementation model in the next section.

Concentrator Objects In our last model for implementing event channels on top of basic CORBA object services, we introduce additional objects to act as message concentrators, as shown in Figure 26. In this implementation, the event channel object

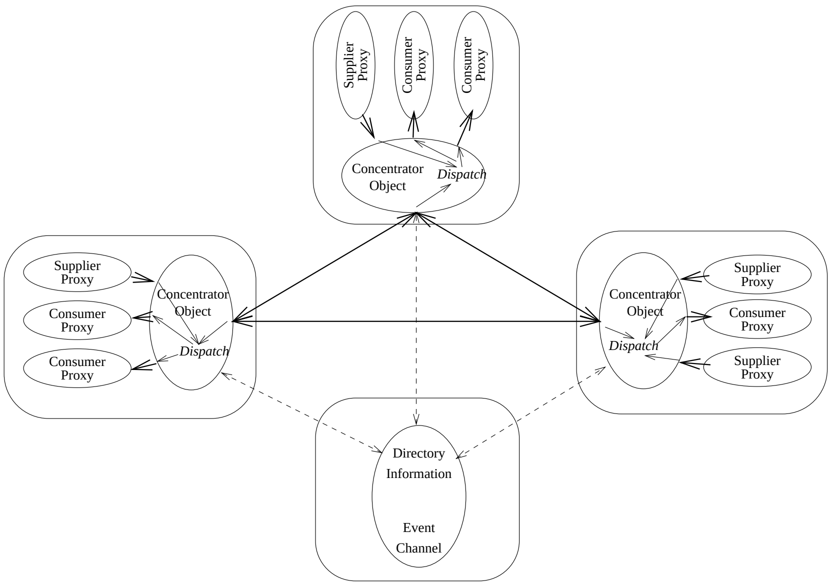

Figure 26: A more message-efficient event channel implementation.

is mostly responsible for maintaining and updating channel subscriber information as it was in Section 4.2.3.2. However, instead of individual proxies performing event distribution directly, they invoke the concentrator objects for event delivery. The event channel has one concentrator object per ORB and these concentrators communicate to accomplish inter-ORB event delivery. Events are sent from the supplier's proxy to the local concentrator object. That concentrator distributes the event to any local consumers as well as sending the event to remote concentrators. Upon receiving an event from another ORB, each concentrator distributes it to local consumers. Because this model uses only a single remote operation per remote ORB to distribute events to remote consumers, it makes much more efficient use of network resources than prior models. Unfortunately, it accomplishes this at a cost of introducing another level of indirection and once again making event delivery a multi-step process.

Because the basic CORBA operation is a one-to-one method invocation, we cannot both reduce the inter-ORB traffic to one invocation per ORB per event and maintain single-step event delivery. In order to accomplish this we have to use facilities outside CORBA. The next sections describe such a facility.

4.2.4 DataExchange Event Channels

Our Event Channel facility is built using DataExchange, which is described in Section 4.1. Event channels are largely built on top of DataExchange facilities. The next sections provide the basic API of Event Channels and discuss their implementation.

4.2.4.1 Channel Creation and Subscription

DataExchange Event Channels are created with the `EChannel_create()` call given below:

```
typedef struct _EChannel *EChannel;
EChannel EChannel_create(DExchange de);
```

Channel creation essentially involves initializing a data structure that will record information about the subscribers to the event channel. The process in which the creation is performed is the permanent contact point for the event channel and participates in the subscription and unsubscription process. However, it does not necessarily participate in event propagation.

When they are created, channels are assigned a character-string global ID. This string can be used to open the channel for use in other processes. The calls involved are given below:

```
char *ECglobal_id(EChannel chan);
EChannel EChannel_open(DExchange de, char *global_id);
```

Opening a channel that was created in another process involves obtaining a DataExchange connection to the creating process and obtaining from that process a list of other processes which have opened the channel. DataExchange connections are then obtained to each process in this list. These direct connections will be used for event delivery. `EChannel_open()` is a synchronous call. That is, it does not return until all data has been received and all connections have been established. In order to do this, it takes advantage of another DataExchange feature, *condition variables.* DataExchange condition variables provide a subset of the features of condition variables in thread packages. One can allocate conditions and wait for them to be signalled. In the span of time between waiting and signalling, other DataExchange events will be handled. One can also associate arbitrary data with a condition variable and later retrieve it. The API for condition variables is as follows:

```
int DECondition_get (DExchange de);
int DECondition_wait (DExchange de, int condition, DEPort dep);
void DECondition_signal (DExchange de, int condition);
void DECondition_set_client_data (DExchange de, int condition, void* client_data);
void *DECondition_get_client_data (DExchange de, int condition);
```

However, the introduction of conditions in DataExchange created a requirement for DataExchange to know something about the overall control flow of the application. In particular, for the DataExchange library to implement `DECondition_wait()` it has to know how to "wait" while continuing to handle events. How that should be accomplished depends upon the overall control strategy of the application. For example, if the application is single-threaded and relies on calling `DExchange_poll_and_handle()` periodically to service the network, `DECondition_wait()` should call the same routine to handle events while waiting. On the other hand, if the application has devoted a kernel-level thread to servicing the network, using that strategy would be at best erroneous and at worst disastrous. Instead, `DECondition_wait()` should allocate a thread-level condition variable and do a thread condition_wait to let the network handling thread execute. `DECondition_signal()` should then do a thread-level signal to awake the sleeping thread.

The need to customize DataExchange behavior based on overall application control style lead to the introduction of the `DEControlList_set_control_style()` call detailed below:

```
typedef enum _DEControlStyle {
    DESingleThreaded, DEDedicatedServerThread
} DEControlStyle;

extern void DEControlList_set_control_style (DEControlList cl,
                                              DEControlStyle style);
```

Currently `DESingleThreaded` and `DEDecidatedServerThread` are the only control styles implemented, but others may be added as necessary. Additionally, to accomodate a variety of threads packages, DataExchange manipulates thread-level condition variables through `gen_threads`, a generic wrapper for thread packages. `gen_threads` allows a library to call thread routines through function pointers which can be initialized at run-time. Thus DataExchange can use thread functions without making assumptions about the particular thread library that will be used in the application.

4.2.4.2 Event Submission

Several routines manage event submission in the Event Channel API. The first is `ECsource_subscribe()`, which derives an `ECSourceHandle` to which an event can be submitted. In the current implementation, `ECsource_subscribe` performs no significant processing, but it is a placeholder for functionality required for future features (such as the derived event channels described in Section 4.2.5.1). Given an `ECSourceHandle`, events can be submitted to the channel with `ECsubmit_event()` or `ECsubmit_eventV()` given below:

```
typedef struct _ECSourceHandle *ECSourceHandle;
extern ECSourceHandle ECsource_subscribe ( EChannel chan );
extern void
   ECsubmit_event ( ECSourceHandle handle, void * event, int event_length );
extern void
   ECsubmit_eventV( ECSourceHandle handle, IOEncodeVector eventv );
```

`ECsubmit_event()` and `ECsubmit_eventV()` differ only in the way they accept the data associated with raising an event. `ECsubmit_event()` accepts the data as a single contiguous buffer with length `event_length`. Alternatively, `ECsubmit_eventV()` accepts data as an `IOEncodeVector`, a vector of `<buffer, length>` pairs. Using this version of event submission can greatly improve efficiency by eliminating a copy step when large data blocks are encoded with the PBIO IOContext calls for submission as events.

Regardless of the mode of event data specification, event submission is synchronous in the sense that before the submission call returns, any local handlers for the event have been called, the event has been transmitted to any remote sinks and the application is free to destroy the data that was submitted.

4.2.4.3 Event Delivery

Events are delivered to event sinks that have registered handlers. Handler registration is accomplished through the `EChannel_subscribe_handler()` function:

```
typedef struct _ECSinkHandle *ECSinkHandle;
typedef void (*ECHandlerFunction) (void *event, int length, void *client_data);
extern ECSinkHandle
  ECsink_subscribe( EChannel chan, ECHandlerFunction func, void *client_data);
```

When an event arrives from the network or is submitted to a channel that has a local sink, the event is queued for later delivery. Event delivery consists of calling the specified handler function and passing the event as a parameter. The `client_data` value specified in the subscribe call is also passed to the handler without interpretation by the library. Note that the same handler is called regardless of whether the triggering event was submitted with `ECsubmit_event()` or `ECsubmit_eventV()`. If the event was submitted remotely via the latter call, the normal operation of PBIO will have caused the buffers to be made contiguous. If the event was submitted locally, the buffers will be made contiguous through a copy operation.

Event delivery is done at two points, just before `ECsubmit_event()` returns and at the completion of a `DExchange_poll_and_handle()` operation. In each case, all pending events are dispatched. During dispatch, handler functions are called directly. Therefore, handler functions that can potentially block or compute for long periods of time should fork a thread to perform those tasks. Preservation of the event parameter data is the responsibility of the handler.

4.2.5 Future Work

The event channel implementation as it currently exists is a complete push-style publish-subscribe event delivery system. However, it is possible to define useful additional capabilities.

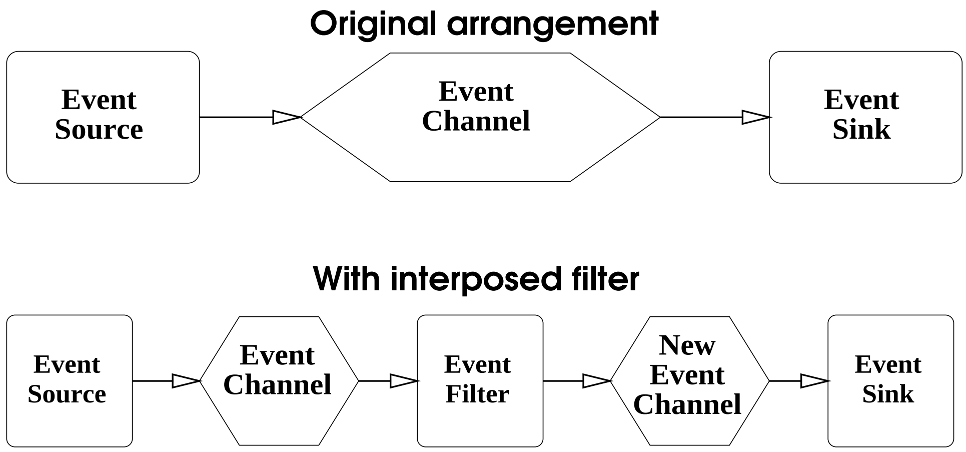

Figure 27: Source and sink with interposed event filter.

4.2.5.1 Derived Event Channels

Consider the situation where an event channel sink is not really interested in *every* event submitted, but only wants every *Nth* event, or every event where a particular value in the data exceeds some threshold. Considerable network traffic could be avoided if we could somehow manage to transmit only the events of interest. One way to approach the problem is to create a new event channel and interpose an event filter as shown in Figure 27.

The event filter can be located on the same node as the event source and is a normal event sink to the original event channel and a normal source to the new, or filtered, event channel. This is a nice solution in that it does not disturb the normal function of the original event channel. However, it fails if there is more than one event source associated with the original event channel. The difficulty is that, as a normal sink, the event filter must live in some specific process. If there is more than one source subscribed to the original event channel and those sources are not co-located, as shown in Figure 28, then we still have raw events traveling over the network from Process A to Process B to be filtered.

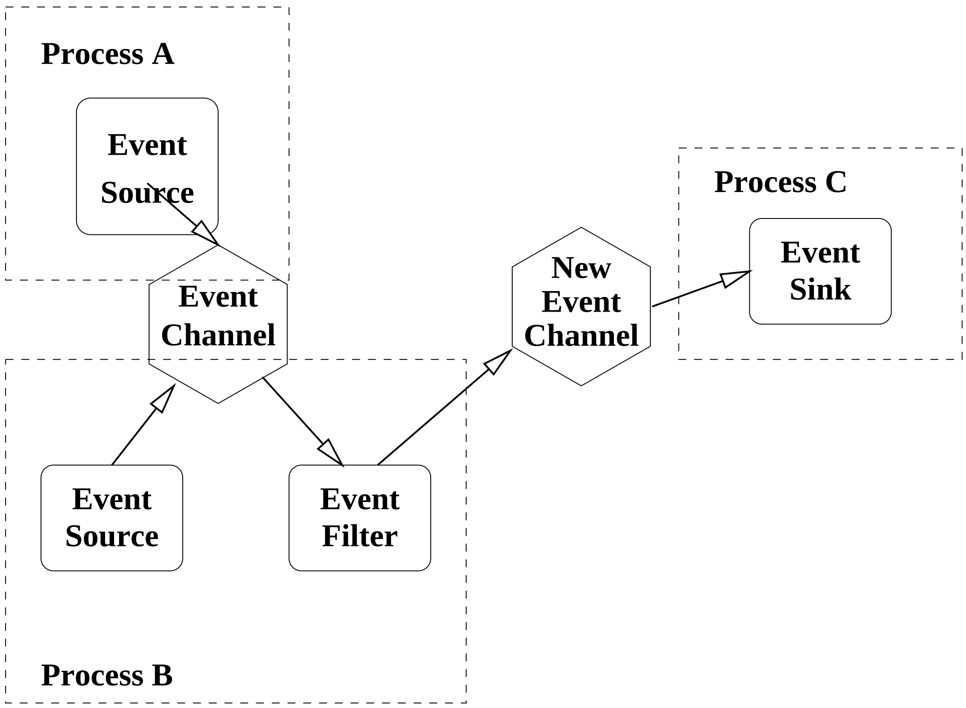

Figure 28: Filter with more than one source.

There is not a good solution to this problem given the normal semantics of event channels, so our approach is to extend event channels. In particular, we introduce the concept of a *derived* event channel. Rather than explicitly creating new event channels with filter objects, applications that wish to receive filtered event data create a new channel whose contents are derived from the contents of an existing channel through an application supplied function, F. The event channel implementation will move function F to all event sources in the original channel, execute it locally whenever events are submitted and transmit any event that results in the derived channel. This approach has the advantage that we limit network traffic as much as possible. In fact, if the derived event channel has sinks that are local to any of the sources in the original traffic, network traffic between those elements is avoided entirely. Figure 29 shows the arrangment of a derived event channel.

Derived event channels are the mechanism through which we plan to implement the steering action migration mentioned in Section 3.2.2 and described more completely in Section 5.2. Approaching the action migration problem at the event channel level is advantageous because this approach leads to solutions which are more broadly applicable than ones attempted strictly within the object domain. The most significant obstacle in implementing derived event channels is the practical difficulty of moving the function F to the original event sources. One could always specify F in an interpreted language, such as Java, TCL or Python, but such a solution requires an interpreter to be present. Another promising approach is the use of dynamic code generation. We are already using dynamic code generation in PBIO to improve the performance of conversion routines, as noted in Section 4.1.5. The code generation techniques used there require relatively little extension in order to be capable of evaluating simple predicates which would be useful as event filters. Steering actions might be more complex than simple filters, but an implementation that works for simple cases would still be of considerable performance benefit.

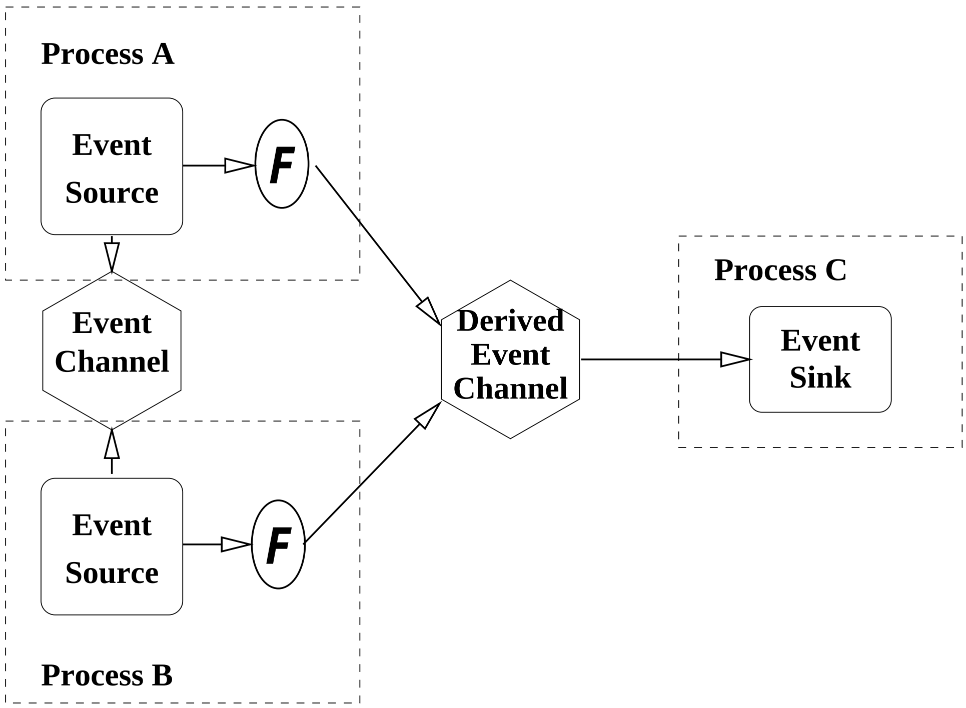

Figure 29: A derived event channel and function F moved to event sources.

4.2.5.2 Derived Subscription

Another important piece of future work is a variation on the derived event channel theme that impacts the behavior of MOSS during the setup stage of monitoring. In order to understand the situation in which it occurs, we will describe parts of the initialization procedure in MOSS.

When a new mirrorable application-level object is created, information about that object is transmitted to all connected MOSS processes. This object information includes: its class, the global ID of the event channel it will use for state updates, and an OTL reference for the object. This information can be conveyed by any number of means, but currently we use traditional monitoring events. When the information reaches an external MOSS process, that process can make a decision about whether or not to create an object which mirrors the application-level object. Creating a mirror object involves creating an object of the appropriate type, subscribing that object's `Update()` method as a handler for update events in the application-level objects event channel, and associating the application-object's OTL reference with the mirror object.

However, some problems arise because this entire process is not synchronous with application object creation. Because of this, even a mirror object that is created as soon as the external clients receive the object information may miss the first few update events put into the event channel. Events submitted to an event channel are delivered only to the subscribers *at the time of event submission.* They are not stored for delivery to subscribers who might come later. These missing updates might be significant to the application. Making the existing process perform synchronously with object creation would involve making the application wait for a response from every connected MOSS process. The resulting application perturbation would be unacceptable in most cases.

`Derived subscription` is an approach to solving this problem. Much like filter predicates can be moved to the source of events to reduce network traffic, we believe that the predicate representing the decision to create a mirror object can be moved to the point where application-level objects are created. If this can be accomplished, then the mirror object creation process could be made synchronous with application-level object creation without overly perturbing application execution. Implementation of derived subscription requires solving problems beyond the code-motion difficulties faced in implementing derived event channels. Because both normal subscription to an event channel and creating the mirror object involve allocating memory in the process of the subscriber, performing those actions at a remote location without causing excessive perturbation may prove difficult. However, in as much as the "event gap" between application object creation and mirror object subscription may be extremely troublesome to some application, derived subscription may be worth considerable effort.

4.2.6 Conclusion and Contribution to Thesis

The DataExchange Event Channel support provides simple, powerful and flexible support for event handling in distributed applications. It is not specifically tied to object-oriented systems or to the CORBA model, but rather provides generic event functionality that could be used in concert with CORBA facilities or independently to support events in non-CORBA distributed programs. It could also provide a mechanism through which CORBA and non-CORBA systems could interact.

Event channels also contribute significantly to attaining the goals set forth for MOSS and the Mirror Object Model. First, they directly support the efficiency of MOSS by suppress events that have no listeners so that they generate no network traffic. This effect is directly responsible for the results in Section 5.4. Second, they automatically route data directly from information sources to sinks. This allows

MOSS to be efficient in ways that the event stream model cannot be, as is demonstrated in Section 5.3.

Essentially, event channels provide the monitoring side of data communication in MOSS. However, they are not employed during steering operation. To support the remote object invocations required for MOSS' steering system, the Object Transport Layer was developed. It is described in the next section.

4.3 Object Transport Layer – Steering Enactment

The Object Transport Layer (OTL) supports remote object invocation in the Mirror Object Steering System. Is a relatively thin layer on top of DataExchange and was designed to support a highly configurable distributed object system[1, 38]. Many OTL features support explicit needs in the IDL compiler. OTL's research contributions have not been fully explored, but this section will detail its interfaces and basic operation and describe how it contributes to the thesis.

4.3.1 Object References

In order to support configurability both above and below OTL, object references in OTL are lists of name-value attribute pairs. Certain of these attributes are interpreted by OTL in order to support basic remote object invocation. For example, the OTL:HOME_HOST attribute is a string which identifies the hostname and IP port through which an object can be invoked. The set of attributes which together give enough information to distinguish an object is stored in an attribute list called the *object name*. For efficiency, OTL supports the caching of additional attributes which might be related to the object name but which should not be transmitted with the reference if it were moved between processes. For example, the OTL:DEP attribute holds the `DEPort` value which represents the connection between the current process and the object's home process. Once this attribute is set, invocations can proceed without the string-level processing required to interpret the OTL:HOME_HOST attribute. Attributes such as OTL:DEP are kept in a separate list, the *name translation cache*, in order that they can be easily culled if the object reference must be marshalled as a parameter or return value. Some situations require caching of attributes specific to the particular object method being invoked. For this reason, OTL object references augment the *object_name* and *name_translation_cache* lists with a third element, the *method_cache*. This list contains attributes associated with each object method. The attribute values are themselves the attribute lists which serve as the method-specific attribute cache for that method.

The following subsections will describe the functionality of atoms and attribute lists in more detail.

4.3.1.1 Atoms

In order to avoid the string comparisons in searching attribute lists for needed attributes, OTL encodes attribute names and enumeration values as *atoms*. Atoms, an idea borrowed from the X windowing system, are 32-bit quantities with a unique one-to-one mapping to strings. The mapping from strings to atoms and back again is uniform across at least the entire application. In this way, rather than doing string comparisons to see if an attribute is present, OTL can translate the attribute name into an atom and then search for a matching atom in the attribute list using 32-bit compares instead of the more expensive `strcmp()`. In X, the X server is responsible for handing out atoms and maintaining a uniform translation (on a single X display). In OTL, this function is performed by the *atom server*. The atom server creates and maintains the mappings between strings and atoms, and all participants in an application must use the same atom server. The subroutines `str_to_atom()` and `atom_to_str()` translate between strings and atoms, contacting the atom server where appropriate and using a local atom cache to limit network traffic. In the current implementation, a process' first contact with the atom server results in its cache being initialized with every mapping known to the atom server. The process will only have to communicate with the atom server again if it needs a translation for something that was not known to the server at the time of initial contact. Given that the set of attribute names is relatively static, communication with the atom server is a relatively rare occurrence in a long-running application.

4.3.1.2 Attribute Lists

As described above, attribute lists are sets of name-value pairs. Attribute names are atoms, but the values can take on a number of different types. The set of calls that create and fill attribute lists are enumerated below:

```
typedef enum _attr_value_type { Attr_Undefined, Attr_Int4, Attr_Int8,
                                Attr_String, Attr_Atom, Attr_List } attr_value_type;
typedef struct _attr_list_struct *attr_list;
extern attr_list create_attr_list();
extern int add_attr (attr_list attrs, atom_t attr_id, attr_value_type val_type,
                     attr_value value );
extern int set_attr (attr_list attrs, atom_t attr_id, attr_value_type val_type,
                     attr_value value );
extern int replace_attr (attr_list attrs, atom_t attr_id,
                         attr_value_type val_type, attr_value value );
```

Add, set and replace_attr() are all similar. `add_attr()` adds an attribute of name, type and value even if another attribute of that name is already present. `replace_attr()` changes the type and value associated with an existing attribute. `set_attr()` does a replace if the named attribute already exists and does an add otherwise.

In addition to the basic mechanisms for adding attributes to lists, OTL defines mechanisms for joining lists. The efficiency of these mechanisms is important since the separate caching of various attributes means that lists must be joined on every invocation. Because of this, lists are joined by reference. That is, a joined list is created which incorporates the original sublists by reference. Because this mechanism can lead to lists being referenced in a variety of places, we use a reference count mechanism to determine when to free an attribute list. The `free_attr_list()` decrements the reference count and only frees the list when it hits zero. `add_ref_attr_list()` increments the reference count when necessary. `attr_join_lists()` creates a new attribute list with two specified lists as sublists. `attr_add_list()` adds list2 to list1 if list2 is already a list of lists (such as returned by `attr_join_lists()`. Otherwise it is the same as `attr_join_lists()`. All list creation routines increment reference counts appropriately. Programs calling the attribute list routines need to exercise caution in managing reference counts to guard against leaks or premature deletion. `dump_attr_list()` is available to print the contents of attribute lists to aid in debugging.

```
extern void free_attr_list (attr_list list);
extern void add_ref_attr_list (attr_list list);
extern attr_list attr_join_lists (attr_list list1, attr_list list2);
extern attr_list attr_add_list (attr_list list1, attr_list list2);
extern void dump_attr_list (attr_list attrs);
```

4.3.2 Initiating an Invocation

Invocations are achieved through the cooperation of compiler-generated stub routines and routines from OTL. Figure 7 in Section 3.2.1 gave a simplified version of a generated attribute set routine. Figure 30 gives a more complete listing of the invocation stub generated for a simple parameterless method, in this case, the `disconnect_push` method of the `PushConsumer` interface in the `CosEventComm` module. Most of the code in this figure is inside `if` blocks that will only be executed once. The first `if` establishes the translation of attribute names to atoms and stores the results in static variables which will be used in every subsequent call. The predicate in the second `if` is locating the method-specific attribute cache. The `if` body is executed if no method-specific cache exists for this object/method. The body creates such a cache and stores a reference to it in the name translation cache.

The final `if` block in the routine is actually the heart of the invocation. The call to `otl_direct_call()` returns true if the invocation can be accomplished by directly calling the code that implements the body of the method. We can only safely do a direct call if the referenced object is local and if the object we are invoking has not been subclassed. If we are invoking an object of a derived class through a superclass interface, that derived class will have different methods and we must

```
extern void
CosEventComm_PushConsumer_disconnect_push(CosEventComm_PushConsumer object,
                                          CORBA_Environment *ev)
{
    /* invoke the disconnect_push method */
    attr_list method_attrs;
    otl_handle tmp_handle;
    IOEncodeVector encoded_params;
    static atom_t otl_method_id_atom = 0;
    static atom_t otl_interface_atom = 0;
    static atom_t this_interface_atom = 0;
    static atom_t method_cache_atom = 0;
    if (otl_method_id_atom == 0) {
      otl_method_id_atom = attr_atom_from_string("OTL:METHOD_ID");
      otl_interface_atom = attr_atom_from_string("OTL:INVOKE_INTERFACE");
      this_interface_atom = attr_atom_from_string("CosEventComm_PushConsumer");
      method_cache_atom =
       attr_atom_from_string("CosEventComm_PushConsumer_disconnect_push_cache");
    }
    if (query_attr(object->name_trans_cache, method_cache_atom,
                   NULL, (void**)&method_attrs) == 0) {
      method_attrs = create_attr_list();
      add_attr(method_attrs, otl_method_id_atom,
               Attr_Int4, (attr_value)
                   COBS_method_id_CosEventComm_PushConsumer_disconnect_push);
      add_attr(method_attrs, otl_interface_atom,
               Attr_Atom, (attr_value) this_interface_atom);
      add_attr(object->name_trans_cache, method_cache_atom,
               Attr_List, (attr_value) method_attrs);
    }
    if (otl_direct_call(object->obj_name, method_attrs)) {
        impl_CosEventComm_PushConsumer_disconnect_push(object, NULL);
     } else {
        encoded_params = NULL;
        tmp_handle =
            otl_invokev(object->obj_name,NULL,
                        method_attrs,
                        encoded_params, NULL, 0);
        otl_wait(tmp_handle);
        free(encoded_params);
        otl_free_handle(tmp_handle);
    }
}
```

Figure 30: Stub code generated for a parameterless method invocation.

go through the `otl_invoke()` interface to locate them. We determine whether or not the object interface and the invoking interface match by comparing the value of the OTL:INVOKE_INTERFACE attribute (which the invocation code has placed in the method-specific cache) with the value of the OTL:OBJECT_INTERFACE attribute (which was placed in the object name attribute list during object creation). If they match and the object is indeed local, then we can directly call the implementation body. Also, to make sure the determination is reused for the next invocation, `otl_direct_call()` caches the result in the OTL:DIRECT_CALL attribute. Object locality is determined by checking for the OTL:IS_LOCAL attribute. If that attribute is not found, its value is set to the result obtained from comparing the value of the OTL:HOME_HOST with the name and IP port of the current process.

If the process described in the previous paragraph results in the determination that no direct call can be made, the `else` clause is executed and `otl_invokev()` is called to perform the invocation. The first action of `otl_invokev()` is to determine if the object is local. If it is, then the invocation is handled as if it were an incoming invocation, described in the next section. If the object is not local, then a DEPort value connected to the object's process is obtained. (i.e., a connection is opened if one has not yet been opened.) The DEport value is cached for later reuse as the OTL:DEP attribute. An `OTL_INVOKE` message is then composed and sent to the object's process. `otl_invokev()` returns without awaiting a response, returning a handle to its caller. This handle can be used in a subsequent `otl_wait()` call to await the completion of the remote invocation. In fact, handles are always returned from `otl_invokev()` whether the object invoked was local or remote and they can always be used in `otl_wait()` to await the completion of the invocation. (Local invocations will have already completed when `otl_invokev()` returns if they were not threaded, but threaded invocations may complete asynchronously.)

4.3.3 Receiving an Invocation

The process of receiving an invocation from the network, or performing an invocation to a local object, is similarly driven by object attributes. Two values are of importance to OTL in carrying out an incoming invocation, the dispatch address and whether or not the invocation is to be carried out in a thread of its own. The latter is specified by the OTL:IS_THREADED attribute. The former is somewhat more complex. It can be specified directly and cached in the OTL:DISPATCH_ADDRESS attribute. More commonly the dispatch address is specified less directly. The method of specification of the dispatch address is itself specified by the OTL:DISPATCH_STYLE attribute. The only currently valid value for this attribute is OTL:DISPATCH_BY_VIRTUAL_FUNCTION. In this dispatching style, the actual dispatch address is determined by an entry in a virtual function table. The attribute OTL:VIRTUAL_FUNCTION_TABLE_BASE gives the base address of the virtual function table. The OTL:METHOD_ID attribute is the index into the table that yields the dispatch address. The dispatch style and virtual function table base attributes are added to the object name at creation. The OTL:METHOD_ID attribute is added to the method cache as part of the second `if` block described above. This style of dispatch by virtual function table is sufficient to support single inheritance. Future support of multiple inheritance may require additional mechanisms to be added.

The dispatch address determined by this mechanism is actually that of yet another compiler-generated stub routine, this one devoted to receiving invocations. For parameterless invocations like `disconnect_push`, the stub shown in Figure 31, is relatively simple.

Upon completion of an invocation which had been received from the network, an `invoke return` message is composed and sent back to the calling process. This `invoke return` message contains the address of the handle which had been returned

```
extern void
STUB_CosEventComm_PushConsumer_disconnect_push(self, pblock, plen, pvector, handle)
void *self;
void *pblock;
int plen;
IOEncodeVector pvector;
otl_handle handle;
{
    /* handle the disconnect_push method */

    object_ref_struct this_obj_ref;
    this_obj_ref.obj_name = otl_get_name(handle);
    this_obj_ref.name_trans_cache = create_attr_list();
    this_obj_ref.method_specific_cache = otl_get_cache(handle);
    impl_CosEventComm_PushConsumer_disconnect_push(&this_obj_ref,
                                                          NULL);
    free_attr_list(this_obj_ref.name_trans_cache);
}
```

Figure 31: Stub code generated for receiving a parameterless method.

from `otl_invokev`. When an `invoke return` message arrives, or when a local invocation completes, OTL sets the handle's `terminated` variable and signals the thread-level condition variable associated with the handle. These two acts will awaken any thread already waiting on `otl_wait()` and/or cause any future `otl_wait()` on that handle to return immediately. Additionally, the `invoke return` message carries cached attribute values back to the invoking process. These are added to the attribute caches used in the invocation (references to which have been saved in the handle). In this way, the cached attributes are available for the next invocation and do not have to be recomputed. Like attribute lists, the handles used in `otl_invokev` keep reference counts, so both the calling application and OTL must free the handle before it is actually deallocated.

```
extern otl_handle otl_invoke (attr_list object, attr_list name_cache,
                              attr_list method_cache, char *pblock,
                              int plen);

extern otl_handle otl_invokev (attr_list object, attr_list name_cache,
                               attr_list method_cache,
                               IOEncodeVector pvector,
                               void *local_pb, int pb_size);
```

Figure 32: The OTL invocation interface

4.3.4 Parameters and Return Values

The addition of parameters and return values complicates the stubs given in Figures 30 and 31. OTL provides two largely separable kinds of support for parameters and return values.

First, OTL will transport parameter blocks to the invocation destination and return any results. The manner in which parameter blocks are handled is the only difference between the `otl_invoke` and the `otl_invokev()` calls. `otl_invoke()` accepts a parameter block as a contiguous memory block of a given length and base address. This was used extensively in previous versions of the compiler, but has recently been supplanted by `otl_invokev()` which accepts the parameter block as as `IOEncodeVector`, a null-terminated vector of `<buffer, length>` pairs. The latter is more efficient when large blocks of data are encoded with the PBIO IOContext routines because it avoids the copy operation required to make the encoded block contiguous. The profiles of `otl_invoke()` and `otl_invokev()` are given in Figure 32.

While `otl_invoke()` only accepts the encoded parameter block (in the pblock and plen parameters), `otl_invokev()` actually accepts both the encoded (pvector) and the unencoded parameter blocks (local_pb and pb_size). This is done for efficiency in the case where the call is local, but not direct (i.e., we are invoking a derived object through a superclass interface). In the remote case, OTL relies on the normal operation of PBIO to ensure that the blocks in parameter vector are contiguous. But in the local case, there is no efficient way to reverse the encoding process performed by `encode_IOcontext_to_vector()`. Therefore, `otl_invokev()` accepts the original unencoded parameter block strictly for use in the situation where the call is to be performed locally. The length of the block is also necessary because OTL will copy the block to preserve it in case the local call is threaded.

In addition to this support for parameter passing, return values are supported through attribute specification. Upon call termination, the attribute cache is checked for the presence of the attributes OTL:RETURN_BLOCK and OTL:RETURN_LENGTH. If these are present, the block of memory they specify is returned to the calling process as part of the `invoke return` message. On the called side, the return block is passed to `free()` before the invocation terminates. On the calling side, memory is allocated for the block and the OTL:RETURN_BLOCK attribute is adjusted to point to the block's new location. After the OTL invoke call returns, freeing the block is the responsibility of the application.

The remainder of the OTL calls supporting parameters and return values are largely convenience routines for the compiler. Their use is demonstrated in Figure 33 which gives the core of invoke-side stub code for a routine with parameters.

The principal differences between the code in Figure 33 and that in Figure 30 are the `if(a_SetB_format == NULL)` block, the setup of the parameter block `p`, and the call to `otl_pbio_type_encodev()`. The additional `if` block makes certain that the types and subtypes used in the parameter block are known to the PBIO `IOContext` used by the OTL convenience routines. The body of the block is executed only the first time this routine is called. It is followed by an assignment for each parameter that fills in the parameter block. Atomic data types and structures are passed by value. Arrays are passed by reference. Finally, the call to `otl_pbio_type_encodev()` encodes the parameter block using PBIO and returns the resulting `IOEncodeVector` value.

```
if (otl_direct_call(object->obj_name, method_attrs)) {
    impl_a_SetB(object, y, NULL);
} else {
    if (a_SetB_format == NULL) {
        otl_set_pbio_type(&a_complex_format, "a_complex", a_complex_fields);
        otl_set_pbio_type(&a_SetB_format, "SetB", a_SetB_fields);
    }
    p.y = *y;
    encoded_params = otl_pbio_type_encodev(a_SetB_format, &p);
    tmp_handle =
        otl_invokev(object->obj_name,NULL,
                    method_attrs,
                    encoded_params, &p, sizeof(p));
    otl_wait(tmp_handle);
    free(encoded_params);
    otl_free_handle(tmp_handle);
}
```

Figure 33: The core of a compiler-generated stub for a routine with parameters.

Additionally, the vector of `encoded_params` is freed after the invocation completes.

In addition to changes in the invoke-side stub, the receiving stub must also deal with the encoded parameters. The receiving stub for a method with parameters is shown in Figure 34.

The routine shown in Figure 34 has two principal execution paths. If `plen == 0`, then this stub is actually being called locally, but not directly (see Section 4.3.2). In this case we have a local parameter block, and the call `otl_copy_local_pb()` copies the value of that block into the structure `p`. If the call is not local, then we must decode the encoded parameter block, which is delivered to us as a contiguous block of memory with address `pblock` and length `plen`. In PBIO encoding, conversions must be established for every different type of architecture that has encoded data. Therefore, if N different architectures remotely invoke this routine, the call to `otl_check_conversion()` will return false at most N times. When it does return false, the `otl_set_pbio_types()` call will establish architecture-specific conversions

```
extern void STUB_a_SetB(self, pblock, plen, pvector, handle)
void *self;
void *pblock;
int plen;
IOEncodeVector pvector;
otl_handle handle;
{
    /* handle the SetB method */

    object_ref_struct this_obj_ref;
    struct a_SetB_params p;

    if (plen == 0) {
        /* We have a local pblock! */
        otl_copy_local_pb(&p, handle);
    } else {
        if (!otl_check_conversion(pblock)) {
            otl_list_of_formats format_list[] = {
                {"a_complex", a_complex_fields},
              {"SetB", a_SetB_fields},
              {NULL, NULL}};
            otl_set_pbio_types(pblock, format_list);
        }
        otl_pbio_decode(pblock, plen, &p);
    };

    this_obj_ref.obj_name = otl_get_name(handle);
    this_obj_ref.name_trans_cache = create_attr_list();
    this_obj_ref.method_specific_cache = otl_get_cache(handle);
    impl_a_SetB(&this_obj_ref, &p.y, NULL);
    free_attr_list(this_obj_ref.name_trans_cache);
}
```

Figure 34: A compiler-generated stub for receiving a method with parameters.

for every subtype involved in the parameter block. Finally, the `otl_pbio_decode()` call fills the structure `p` with the results obtained by decoding the parameter block.

4.3.5 Contributions and Future Work

As was mentioned previously, the general research contributions of OTL have not been fully explored. It was designed to be a highly flexible environment, with attribute lists being allowed to specify such complex invocation details as : network transport characteristics, invocation scheduling, quality of service, real-time requirements, and object fragmentation. These all represent areas of future work for OTL.

With respect to MOSS and this thesis, however, the contributions of OTL are clear. OTL provides the basic mechanism through which steering actions are performed, as well as providing general support for the object infrastructure. It's efficient integration with DataExchange and PBIO make possible the performance results described in Section 5.2.

In future development OTL is likely to play a more prominent role. For example, OTL is key in allowing real-time interaction with running programs. While much user-based program interaction occurs in a non-real-time environment, automatic (or closed loop) program steering may require the ability to interact with the program in well-defined ways with time-constrained limits to ensure that the steering actions are both safe and appropriate.

CHAPTER 5

Evaluation

This chapter evaluates the Mirror Object Steering System in the context of the thesis statements and contributions enumerated in Chapter 1. In particular, the following evaluations will be presented:

- **Sensor production microbenchmark** – To support MOSS' claim to be a monitoring and steering system suitable for high-performance applications, we must show that support for higher-level object abstractions does not overly impact the basic functions of a monitoring and steering system. One such measure is the amount of overhead involved in monitoring. This experiment compares the time required to generate monitoring events of various sizes in each of three systems: MOSS, Audobon (a variant of Falcon, a previous Georgia Tech monitoring system) and Autopilot from the University of Illinois.

- **End-to-end steering benchmark** – a second basic measure of the function of a monitoring and steering system is the time required to complete a round-trip steering action triggered by a monitoring event but with the steering action initiated by a process on another machine. Again, the goal is to show that MOSS does not impose substantial additional overhead. To demonstrate this we perform similar steering tasks with MOSS and Autopilot and compare their performance.

Additionally, to show the importance of steering action migration, the results of the basic end-to-end two-machine steering benchmark are compared to the time required to perform a steering action when that action has migrated from an external client into the application itself. Steering action relocation, as described in Section 3.2.2, does not occur automatically in MOSS, but the MOSS framework is designed to support such optimizations. This experiment demonstrates the value of this particular optimization.

- **Data parallel data reduction** – another important claim of the Mirror Object Model is that it is effective in scaling the interactivity system because it allows automatic routing of monitoring information to the proper data reduction point. This contrasts with more traditional event-stream based monitoring systems which must broadcast events. Routing updates only to interested clients becomes important in several situations. One of these is when the quantity of monitoring data requires parallelizing the data reduction system in order to meet throughput requirements. We show that the Mirror Object Model and MOSS can significantly reduce network throughput requirements.

- **Automatic data suppression** – an equally important capability for scaling in a monitoring and steering system is the suppression of unwanted monitoring events. The schemes employed by many systems do not handle multiple clients well because they involve explicit switches to turn monitoring sensors on or off. This experiment demonstrates the ease with which the Mirror Object Model handles complex situations. The experimental setup involves a data source exporting data objects which represent a grid of values which is updated regularly. Element-by-element data is available as well as aggregate values which represent the average value of some region of the grid. The clients of this data source are interested in receiving element-by-element data for a portion of the grid around their point-of-interest and average values for the remainder of the grid. Each client's point-of-interest can change over time, and there may be more than client receiving updates at the same time.

5.1 Sensor Production Microbenchmark

One possible criticism of the Mirror Object Model is that support for the extended functionality it offers will increase the costs of basic monitoring and steering operations. In order to address this concern, this section examines the computational costs of a basic monitoring operation in MOSS as well as two other systems which involve monitoring. The first of these external systems is Audobon, a variant of the Falcon[16] monitoring system developed at Georgia Tech. Audobon was chosen because it is a basic event-stream-based monitoring system. The second external system chosen for evaluation is Autopilot[34], a program monitoring and steering system recently developed at the University of Illinois. Like MOSS, Autopilot is object-based and implements per-object data routing facilities.

Because the amount of overhead imposed by network transmission is both potentially large and mostly outside the control of the monitoring system, the experimental setup separates the network from consideration and concentrates on computational costs which precede the network `write()` call. In each of the three systems, this is accomplished by interposing a null network send routine into the monitoring system infrastructure after allowing the monitoring system to establish a normal connection to an external client. In the case of Audobon and MOSS, both of which use PBIO[6, 8] for network transmission, this interposition is accomplished by using `set_interface_IOfile()` to modify the basic functions through which PBIO accesses the network and supplying a `write_func` which returned a success code. In the case of Autopilot, which uses Nexus[13] for network transmission, internal Nexus data

Table 1: Computational costs for sensors of various sizes

Data Size	MOSS	Audobon	Autopilot
12	14.4 μsec	26.3 μsec	24.9 μsec
100	14.4	27.8	25.6
1024	14.4	33.5	38.0
10240	14.4	88.6	157.2
102400	14.4	641.2	1264.4

structures were traversed and the protocols `send_rsr` function pointer was changed. Instead of a pointer to `tcp_send_rsr()`, this pointer was redirected to a function which deallocated the buffer and returned. (Buffer deallocation would normally occur at the completion of the network write in Nexus.)

Because the Mirror Object Model targets include large scientific simulations which may produce very large events, sensor costs were measured for a variety of sensor sizes. This experiment was run on a Sun Sparc Ultra 1 workstation and the results are shown in Table 1.

The striking aspect of the results in Table 1 is that while the non-network computational costs in MOSS do not grow with sensor size, those of Audobon and Autopilot do. This dramatic difference in behavior is not *per se* a reflection of the broad design characteristics in the systems, but can instead be traced to more subtle differences in semantics and operation.

In Autopilot, the principal performance impact comes from two copy operations that are performed on the data before it reaches the network `write()` call. Both of these copies are the result of the choice of Nexus as the transport mechanism. The first copy is the result of reliance on the Nexus put/get operations for filling the buffer with primitive data types. Nexus does not impose a "network format" for data, so the put operation on a primitive data type does not actually transform or byte-swap it, but it does result in a copy of the data into a send buffer. The second copy operation is internal to Nexus and is a result of a Nexus design decision. Rather than allowing the application thread to write to the network directly, the Nexus `tcp_send_rsr()` call instead queues the data for later sending by a network handler thread. However, since the application is allowed continue executing and the network write may not happen immediately, Nexus must first preserve the data to be sent by moving it to another buffer and incurring a second copy operation. This "out-of-line" network send operation has the advantage that the application thread is not blocked for network operations, but in this case has the consequence of imposing additional copy overhead. Neither of the two copy operations which impact Autopilot sensor performance is necessary in the sense that it is unavoidable in terms of Autopilot semantics, and further optimization of the Autopilot transport system might significantly reduce this overhead.

In Audobon, the additional overhead as data size increases can also be traced to a copy operation. Audobon, like Falcon, has sensors which accept data as subroutine parameters. Additionally, the functional requirement for an Audobon sensor requires that it add certain header information to the data. However, its sensors were designed to accommodate the limitations of an earlier version of PBIO which required at least the base data block to be a "C"-structure-style contiguous block. A copy operation was necessary to achieve contiguity. However, the semantics of Audobon also do not intrinsically require a copy operation. It is built on top of PBIO, which is now designed to write application-level data in-place, so a reimplementation of Audobon and its sensor mechanism could also potentially eliminate data-dependent growth in sensor costs.

In contrast to the two systems above, MOSS does not normally incur additional non-transmission costs as such things as array sizes grow in monitored data. This is due to careful design and interaction in the data marshalling mechanism (PBIO) event system which routes and transports. First, the event system accepts data payload for an event as either a single contiguous buffer, or a list of buffers and sizes (i.e., a buffer vector). The latter dovetails with PBIO's ability to generate an encoding of a complex data structure and represent that encoding as a buffer vector. In particular, PBIO's `encode_IOcontext_to_vector()` call is used to marshal the sensor data. This encoding requires the creation of a small amount of data describing the structure to be encoded. Additionally, any substructures containing pointers (to strings or dynamic arrays) must be copied in order that their pointers can be rewritten into offsets in the message. However, the majority of the data, particularly large arrays of atomic data types, is left at its application address, with only its address and size appearing in the buffer vector. Since the event channel operates with the semantics that the complete processing of the event data payload before the event submission returns, no data copying is necessary.

This experiment shows that MOSS is more efficient than Autopilot, a monitoring and steering system which is also object-based, and even more efficient than Audobon, a much less featured basic monitoring system. It also underscores the importance of such basic techniques as minimizing copying in data transfers.

We also note that minimizing transmission overhead is only one aspect affecting a monitoring system's ability to handle large amounts of data. Other potentially useful techniques include the buffering of updates, periodic sampling instead of push-style update generation, and the local reduction of data through averaging and other means. These facilities were explored in Falcon[15] and are available in Autopilot. It is possible to employ such techniques to the object state update mechanism in MOSS as well. Additionally, one could reduce overheads in MOSS by only transmitting changed attributes when an update occurs. However, as prior research has already explored these techniques their application to MOSS is left for later development.

5.2 End-to-end Steering Benchmark

The basic steering functionality in MOSS will be evaluated by comparing its performance with that of Autopilot. While Autopilot is also an object-based monitoring and steering system, these systems do differ in the amount of overhead inherent in their approaches. Autopilot's actuators have just a single method, `processActuatorData`, that can be triggered by remote clients. Parameters to that method consist only of a single array of atomic data types. The run time system needed to support this functionality is not as complex as the MOSS run-time system, which must be capable of invoking any application-level method.

To evaluate this cost, similar steering situations were created in Autopilot and MOSS. Each involved an application-level sensor triggering a steering action from a remote client. The processing on the client is minimized, consisting of only initiating the steering action, passing as a parameter the data it has received from the sensor. The application-level steering action consist only of copying the parameter data into an application-level data structure. The experiment was first run between two Sun UltraSPARC 170's connected with 100Mbps ethernet. The results are shown in Table 2. The times reported are elapsed real-time between sensor initiation and the completion of the steering action.

Table 2: Elapsed real-time between sensor initiation and steering completion.

Data Size	MOSS	Autopilot
12	18.6 msec	13.4 msec
100	19.8	14.3
1024	22.2	18.1
10240	45.6	43.5
102400	266.	297.

The results in Table 2 show that the steering response times for MOSS and Autopilot are comparable. Autopilot is 30% faster than MOSS for the smallest data set measured. But for large data exchanges, more efficient data handling gives MOSS the advantage. The use of a more efficient distributed object system than OTL could further reduce the MOSS round-trip steering time. However, we believe that the additional flexibility and functionality offered to MOSS clients outweighs the performance differences in the current implementation.

A more ambitious optimization also has great potential for improving the performance of MOSS as was mentioned in Section 3.2.2. Other systems, such as Autopilot and Progress, make a clear distinction between the sensors which produce monitoring data and the actuators which can be used to steer the program. In the Mirror Object Model, sensors and actuators are combined in the form of the mirrored application-level object. This provides a semantic link, missing in other systems, that can be exploited for automatic optimization. In particular, this link gives MOSS the potential to migrate external steering computations into the application. As mentioned in Section 3.2.2, avoiding the round-trip across the network can potentially reduce steering latency dramatically and open new domains of application to steering. General application steering migration is almost certainly impractically complex, but in certain simple cases it is relatively easy. For example, consider the following `Update()` method from a steering client. The method is represented in C++ and the variable `request_rate` is an object state variable mirrored from the application object. The method `Replicate` is also mirrored from from the application object and represents a call that steers the application object.

```
SteeringClient::Update()
{
    if (request_rate > 100) {
        Replicate();
    }
}
```

This is a very simple, but useful type of steering computation. It references nothing outside of the object itself, but it does accomplish a basic steering goal. It monitors an important internal aspect of object operation and reconfigures the object when a threshold is exceeded. This type of steering computation would be an ideal candidate for migration into the application. Because of its simplicity, a compiler could easily determine that this method had no external side-effects and depended only on mirrored data. If the migration could be accomplished, steering latency could be dramatically reduced and network traffic eliminated. As an example of the gains possible, Table 3 shows the steering latency numbers of Table 2 with an added column that represents the time required if the steering action in the earlier example was migrated into the application. The numbers shown here were **not** produced through automatic steering migration, but are meant to show the benefits that can be gained through that optimization.

Table 3: Elapsed real-time between sensor initiation and steering completion with steering action migration.

Data Size	MOSS	Autopilot	Steering Migration
12	19.8 msec	13.4 msec	0.304 msec
100	20.7	14.3	0.363
1024	22.2	18.1	0.538
10240	45.6	43.5	1.91
102400	266.	297.	16.3

The results in Table 3 show clear and dramatic benefit from migrating steering computations. The results above were generated by manually linking the steering code into the application and causing it to be called in response to a sensor event. While MOSS is not yet capable of performing this important optimization automatically, this potential optimization is still a valuable contribution of MOSS and will be an active area of future work. Section 4.2.5.1 discusses the approach that will be taken in implementing this migration of steering actions. The most significant obstacle in achieving these gains automatically is the practical difficulty of moving the `Update()` method code into the application. One promising approach is the use of dynamic code generation, such as is employed in MDL in the Paradyn project[26]. As is noted in Section 4.1.5, PBIO is already capable of using dynamic code generation to create customized data conversion routines. The performance improvements created by this dynamic code generation of conversion routines is documented in that section, as well as in [8]. These generated routines must be able to access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Generating a simple `Update()` routine like the one shown above merely requires adding expressions and simple control structures to PBIO's repertoire.

5.3 Data Parallel Data Reduction

In order to demonstrate the abilities of MOSS with respect to data parallel data reduction, we will describe an application situation and detail the manner in which it could be implemented in MOSS and with a more traditional event-stream-based monitoring and steering system. Then we will calculate the network bandwidth required by each solution.

For the application, we once again consider the atmospheric model described in Section 2.1. This model performs most of its computations in the spectral domain, with atmospheric species concentrations represented by superimposed waves with different weights. Yet the most meaningful representation for display of this data requires the spectral representation be converted to a grid-based representation. This requires evaluating the spectral representation at each grid point and is a computationally costly conversion. If the atmospheric model is executing rapidly, it can produce spectral data more quickly than the display processor can convert it to grid form. In this situation, it is natural to want to parallelize the spectral-to-grid conversion in order to achieve the required conversion speed. In particular, because species data for each atmospheric level can be converted independently, a data-parallel approach leads to an embarrassingly parallel solution.

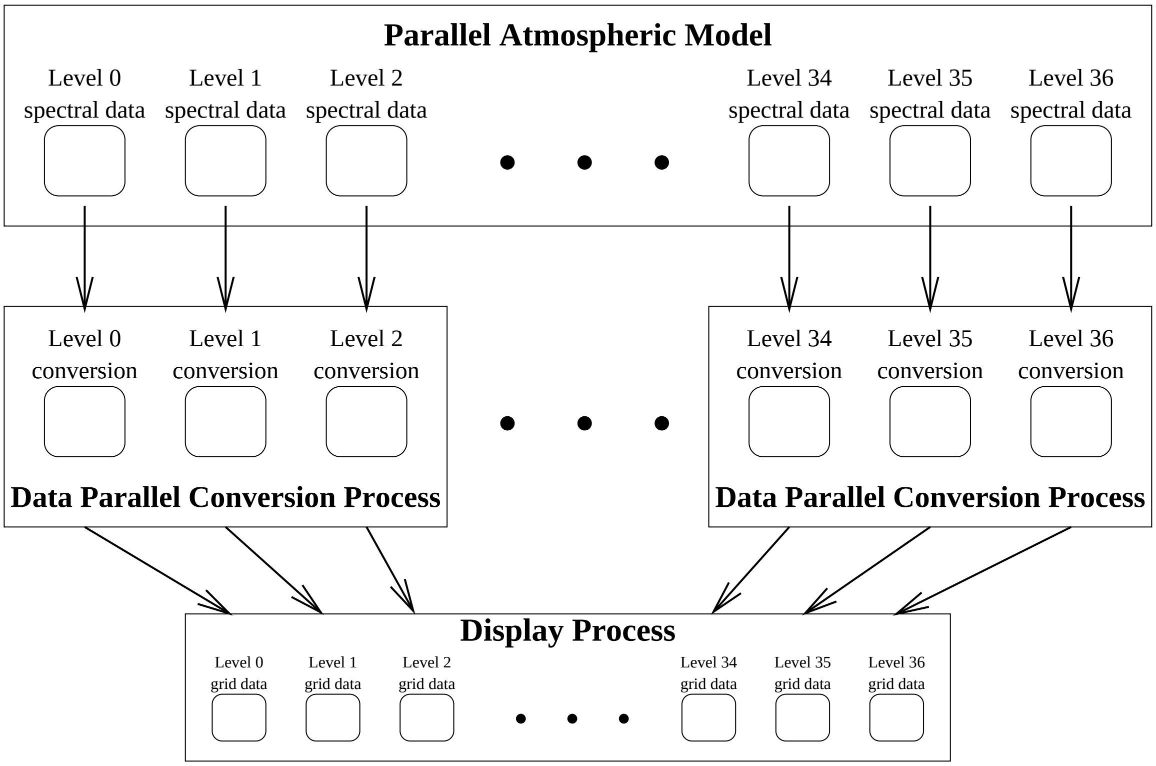

Figure 35: Logical data flow in data parallel monitoring reduction.

Figure 35 shows data locations and message flow for a data parallel decomposition of the spectral-to-grid conversion. The parallel atmospheric model is depicted as a single process as it might be on a large shared memory multiprocessor. If it were decomposed across distributed memory processing elements, the data flow between the application and the conversion processes would not change. Each arrow represents a transmission of data associated with a single atmospheric level, with the upper set of arrows representing spectrally encoded data and the lower set representing data in grid form. In this figure, we have arbitrarily chosen to assign three levels to each conversion process, but any decomposition is possible.

In MOSS, implementation of data-parallel reduction is relatively straightforward. We enclose the spectral data for each level of the atmosphere in an application-level object. Each of those objects is mirrored in some data parallel conversion process. The placement of the mirror objects determines where the computation is performed. Finally, the data conversion objects are themselves mirrored in the display process. This results in a data flow that is the same as the logical data flow shown in Figure 35.

However, in an event-stream-based monitoring implementation, such as Falcon or Progress, the results are quite different. When these systems provide for differential routing of events, it is typically only routing based on event type, not event contents. Because of this, the straightforward implementation of data parallel reduction leads to a data flow depicted in Figure 36. Here, each arrow again represents the transmission of data about a single atmospheric level, but the termination points of the arrows are more clustered to represent a single event-stream subscription for the process. There is no quantitative change in the final event flow to the display process because there is only one subscriber at that level. But the event flow from the atmospheric model to the conversion processes has increased dramatically. Because of the lack of data-driven routing, events associated with each level must be broadcast to all the conversion clients. These clients, who have knowledge of what levels they are assigned, then discard the levels that they are not interested in. However, the broadcast imposes a considerable cost in terms of network bandwidth, increasing the cost of the first level

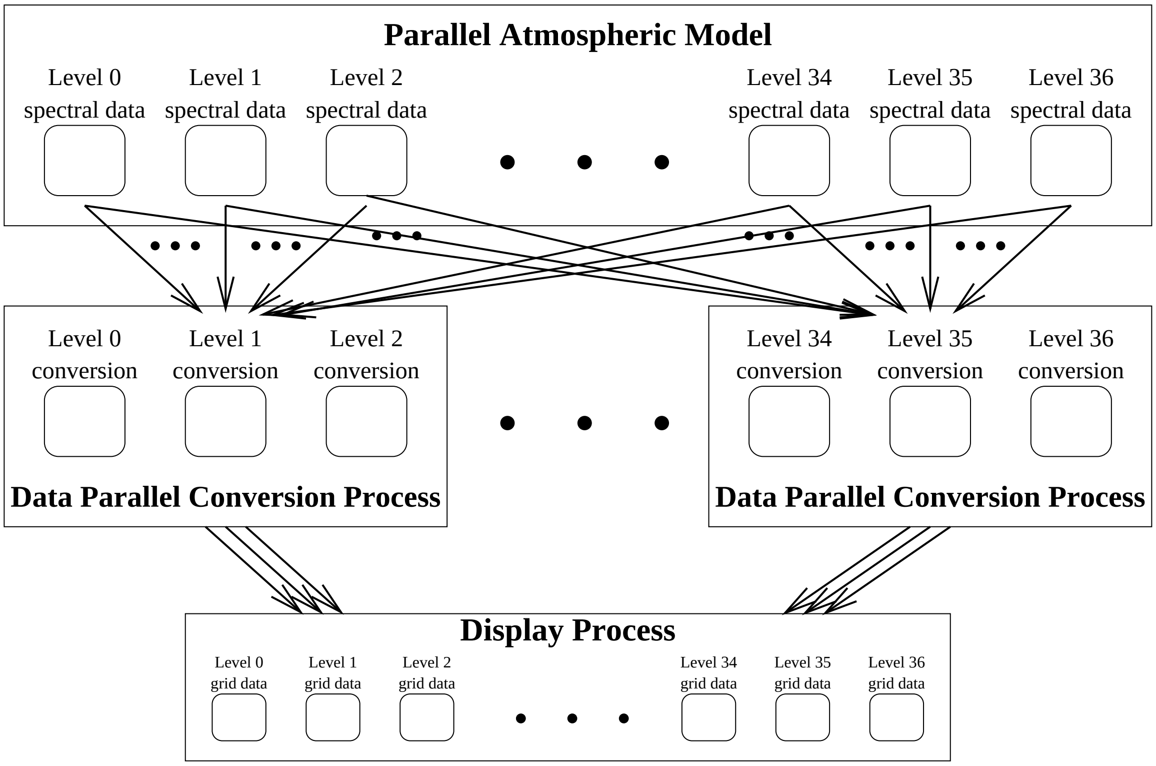

Figure 36: Event stream data flow in data parallel monitoring reduction.

of data transfers by a factor of N, where N is the number of processes involved in the data parallel monitoring reduction.

On a Sun Sparc Model 170, the conversion of one level of atmospheric species data from spectral to grid form requires 2.5msec, consuming 253 complex numbers (about 2 Kbytes). If the conversion of all 37 levels were decomposed across 37 processors and run at full rate, MOSS would require a network bandwidth of roughly 30Mbps between the application and the conversion processors. An event-stream based system would require 37 times as much bandwidth, or over 1 $Gbps$. Conversely, if we were to assume that bandwidth were limited to 100 $Mbps$, MOSS could easily still support a 37 processor system which would process the model output at 400 Hz. However, an event stream-based solution would saturate the network at 11 processors, a level of parallelism which would result in a 100 Hz processing rate.

This analysis indicates that the event routine provided by the Mirror Object Model is an important contribution to the scalability of interactivity systems.

5.4 Automatic Data Suppression

In addition to appropriate data routing, data update suppression is an important technique for scaling an interactivity system. In order to demonstrate how important, we consider a different aspect of the data flow shown in Figure 35. In particular, consider the level of traffic between the spectral-to-grid conversion processes and the display process. Species concentration for an atmospheric level in grid form consists of 2048 floating point numbers, in this case about 8 Kbytes. At the fastest event rate possible with MOSS and the conversion parallelized across 37 processors, this amounts to 120Mbps. But in the distributed laboratory environment, there may be more than one display monitoring the application at a time. If each is sent the entire data set, they would *each* require 120Mbps of network bandwidth, enough to tax most networks. However, many displays will not use the entire 300Kbytes that each update will bring. Many displays are one- or two-dimensional and only need a particular column, row or level of atmospheric data. This indicates a need for a mechanism for specializing what data are sent based on the actual requirements of the data consumer.

This need has been recognized and met by different mechanisms in other systems. Falcon has a mechanism for enabling and disabling sensors. Paradyn inserts and removes instrumentation points as they are needed. However, most previous mechanisms are inadequate for the distributed laboratory environment. For example, the Falcon and Paradyn sensors essentially allow only that particular collection points be turned on and off. This type of mechanism is not easily adaptable to the situation in the previous paragraph, where we need to suppress update of parts of the atmosphere, but not all data of a particular type. Another difficulty with the simple on-off switches in systems like Falcon is that they do not work well when multiple data consumers are present. While each client knows what data it needs, no client has the global knowledge required to properly operate the switches. MOSS avoids this difficulty through the operation of event channels. Because there is one event channel per application-level object, and because the event channel properly handles multiple event consumers and suppresses network traffic when there are no consumers, the channels provide a natural update suppression mechanism that is data-specific and performs in a multi-consumer environment.

For example, in the situation described above, if each display is interested in data from only a single level and therefore creates a mirror object for only the level it is interested in, MOSS will transmit only the minimum 8Kbytes per update per display. Compared with almost 300K per update per display for a Falcon-style event-stream implementation, this represents a considerable savings in network load.

CHAPTER 6

Conclusion

This thesis presents the Mirror Object Model, a new higher-level model for programming distributed monitoring and steering systems for high-performance applications. Existing monitoring and steering systems suffer from a variety of problems. Many event-based systems have properties that prevent them from scaling to high event rates and large quantities of data. Additionally, no existing monitoring and steering systems provide a semantic model of computation that unifies application-level monitoring semantics, monitoring data reduction, and application-level steering enaction. The Mirror Object Model supplies the unifying semantic link between monitoring, externally triggered computation and steering enaction.

One of the principal results of this thesis is the Mirror Object Steering System (MOSS), a set of tools and libraries that provide an implementation of the Mirror Object Model. Chapter 3 describes the Mirror Object Model and presents details of its realization in MOSS. MOSS is built upon several middleware packages described in Section 4, including a distributed object system, a communications manager, and a binary I/O package. While these packages have research merit in their own right, this thesis concentrates on the features and merit of the Mirror Object Model and MOSS. Examples throughout the thesis have shown how the Mirror Object Model can be used in monitoring and steering situations. The performance analysis in Sections 5.1 and 5.2 show that MOSS provides the additional functionality of the Mirror Object Model without significant performance degradation. In fact, Section 5.1 shows that

MOSS imposes considerably *less* overhead than other sampled monitoring mechanisms. Additionally, Section 5.3 and 5.4 show how the basic properties of the Mirror Object Model significantly reduce network overhead when scaling event rates through basic mechanisms such as data-parallel monitoring reduction.

MOSS runs on the same systems as the packages identified in Section 4. Currently, that list includes Sun Sparc SunOS 4.1.3, Sun Sparc Solaris 2.x SGI MIPS IRIX 5.x, SGI MIPS (32 and 64-bit) IRIX 6.x, IBM RS6000 AIX 3.2, x86 Linux, x86 Solaris 2.x, and x86 Windows NT. Additionally, the PBIO and DataExchange packages are adaptable to a variety of network infrastructures. This adaption can be carried out by replacing their lower-level network interface routines with alternative implementations. The evaluations in this thesis use standard TCP/IP interfaces, but ports are in progress to other networks, including ATM using bandwidth reservation for QOS.

6.1 Future Research

The Mirror Object Model and MOSS leave open many promising avenues for future research. Two potentially rewarding subjects which may be pursued in the future are described in Section 4.2.5. While derived event channels and derived subscriptions are essentially optimizations at the level of the event channel, their implementation would result in considerable improvements in MOSS' functionality. In particular, derived event channels are the mechanism through which steering actions can be migrated from the interactivity system into an application. Section 5.2 shows that such migrated steering actions can proceed at a much higher rate than external steering actions. This optimization has the potential to open whole new domains of applicability for closed-loop steering. Additionally, performing the optimization at the event-channel level, as opposed to the mirror-object level, means that it is more likely to benefit non-object programs as well. A variety of applications, from distributed event simulation to distributed virtual reality make use of event-based communication abstractions and could potentially benefit from an event channel mechanism that was able to intelligently migrate computation.

In the area of visualization, MOSS has potential to ease the construction of some types of displays of program state. As compared with remote visualizations based upon event-stream-style program monitoring, MOSS provides considerable additional support, including:

- automatic construction of remote data structures (the mirror objects in the visualization client)
- event interpretation to update state
- easy interaction with the application through remote method invocation to application objects
- plug-and-play attachment to the application.

Code to support the actions and features above can make up a significant fraction of simple visualization clients, indicating MOSS may be of significant benefit in their construction.

In the area of programmability and software engineering, Section 3.3 described strong similarities between the programming model proposed in IBM's Subject Oriented Programming system [19] and the Mirror Object Model. Subject Oriented Programming is promoted as an enhancement to object oriented programming styles that allows decentralized class definition with resulting software development benefits in: requirement-based development, extensibility, customizability, unplanned composition and feature-based configuration management[30]. While SOP does not address distributed systems, it creates a programming model that very closely resembles that created by the Mirror Object Model and MOSS. We have not explored the relative programmability of MOSS-based and event-stream-based interactivity systems, but this resemblance between the programming models implies that MOSS-based systems may have significant advantages in terms of programmability and extensibility. Future work may explore and evaluate these advantages.

Another avenue of future work is the implementation of MOSS in other object-oriented languages. In particular, a Java-based MOSS implementation able to interact with a non-Java application through IIOP or OTL would be a valuable addition to the MOSS system. This would allow the simple construction of browser-based visualizations and interaction systems. While less research-oriented, this type of development has the potential to open program monitoring and steering to a new class of users and aid in rapid development of customized visualizations and steering interfaces.

The potential utility of recursive application of steering has also not been extensively explored in this thesis. The Mirror Object Model is the first steering system with a programming model that could be applied to itself. The data-parallel monitoring data reduction described in Section 5.3 and 5.4 shows the use of multiple levels of data processing in monitoring analysis, but the utility of multiple levels of steering has not been evaluated. While this functionality may be important in application-specific roles, it may play a more general role in providing meta-level control to the interactivity system itself. In particular, recursive steering could provide a mechanism through which one could perform the types of load-balancing and rate control envisioned in [24].

While the strength of the Mirror Object Model is that it allows the easy creation of relatively complex systems, this is a two-edged sword. Complex systems, particularly distributed, multiprocessor, multithreaded, multiobject systems like MOSS-supported interactivity systems, have complex behaviour and semantics. One basic area of complex behavior is synchronization. While MOSS doesn't prescribe a synchronization model, each call to `Update()` in a mirror object is independently threaded, making some kind of synchronization imperative in all but the simplest sample applications. Some programming models (such as Java) may impose their own style of synchronization on a particular Mirror Object Model implementation, but I believe that the general topic of appropriate synchronization in a Mirror Object system is one that deserves additional research.

Finally, the semantics of monitoring updates in the Mirror Object Model are narrowly defined. In the current implementation of MOSS, if any object attribute is modified, all attributes are sent to any existing mirror objects. This can have some performance penalties, particularly when objects have large state and a lot of updates are performed one after another. A relatively simple implementation optimization would be to limit the mirror update to send only the changed attributes. This would help to limit overhead, but a more general approach may be valuable. Mirror objects should be able to customize the list of attributes from which they want updates as well as the circumstances under which those updates are performed. The sensor policies of Falcon[16] might provide a model for attribute update semantics, but another promising approach would be to use distributed shared memory style consistency specifications to model the relationship between between application objects and mirror objects. Such an approach might leverage the approaches of years of DSM work and bring a new formalism into the area of program monitoring and steering systems.

Bibliography

[1] Mustaque Ahamad and Karsten Schwan. The COBS Project. http://www.cc.gatech.edu/systems/projects/COBS.

[2] Scott B. Baden. Software infrastructure for non-uniform scientific computations on parallel processors. *Applied Computing Review*, May 1996. ACM Special Interest Group on Applied Computing.

[3] Ken Calvert, Robin Kravets, and Boby Krupczak. An extensible end-to-end protocol and framework. In *Proceedings of the 1995 IFIP International Working Conference on Upper Layers Protocols, Architectures and Applications (ULPAA)*, December 1995.

[4] *CORBAservices: Common Object Services Specification*, chapter 4. Object Management Group, 1997. http://www.omg.org.

[5] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, Sept 1990.

[6] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[7] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu. Falcon – toward interactive parallel programs: The on-line steering of a molecular dynamics application. In *Proceedings of The Third International Symposium on High-Performance Distributed Computing (HPDC-3)*. IEEE, IEEE Computer Society, August 1994. Also available as Georgia Institute of Technology Tech Report GIT-CC-94-08.

[8] Greg Eisenhauer, Beth Plale, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. to appear in *Journal of Parallel Computing*, 1998.

[9] Greg Eisenhauer and Beth Schroeder. The DataExchange library. Technical Report GIT-CC-96-17, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, 1996. *(anon. ftp from ftp.cc.gatech.edu)*.

[10] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[11] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. http://www.mcs.anl.gov/mpi/mpi2/mpi2.html.

[12] MPI Forum. MPI: A Message-Passing Interface. Technical Report CS/E 94-013, Department of Computer Science, Oregon Graduate Institute, March 94.

[13] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[14] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users Guide and Reference manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[15] Weiming Gu. *On-line Monitoring and Interactive Steering of Large-Scale Parallel and Distributed Applications.* PhD thesis, Georgia Institute of Technology, Atlanta, GA 30332, 1995.

[16] Weiming Gu, Greg Eisenhauer, and Karsten Schwan. Falcon: On-line monitoring and steering of parallel programs. to appear in *Concurrency: Practice and Experience.*

[17] Weiming Gu, Jeffrey Vetter, and Karsten Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29(9):140–148, September 1994.

[18] Lars T. Hansen. Sneezy note #5: Proposal for multiclient support. Unpublished design note: http://www.cs.uoregon.edu/research/paracomp/proj/tau/sneezy/note5.ps.gz.

[19] William Harrison and Harold Ossher. Subject-oriented programming - a critique of pure objects. In *Proceedings of 1993 Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1993.

[20] Harold H. Hersey, Steven T. Hackstadt, Lars T. Hansen, and Allen D. Malony. Viz: A visualization programming system. Technical Report CIS-TR-95-05, University of Oregon, Department of Computer and Information Science, April 1996.

[21] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonçalves, Oscar Naim, Zhichen Xu, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, November 1997.

[22] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[23] LabSpace. Creating virtual laboratories for scientists. URL:http://www.cs.northeastern.edu/research/labspace.

[24] Vernard Martin and Karsten Schwan. Ili: An adaptive infrastructure for dynamic interactive distributed applications. In *Proceedings of the 4th International Conference on Configurable Distributed Systems (ICCDS'98)*, May 1998.

[25] Jim McDonald and Karsten Schwan. Ada dynamic load control mechanisms for distributed embedded battle management systems. In *First Workshop on Real-time Applications, New York*, pages 156–160. IEEE, May 1993.

[26] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 1995.

[27] B. Mohr, A. Malony, and J. Cuny. Tau. In G. Wilson, editor, *Parallel Programming using C++*. MIT Press, 1996.

[28] Bodhi Mukherjee and Karsten Schwan. Experiments with a configurable lock for multiprocessors. In *Proceedings of the International Conference on Parallel Processing, Michigan*, pages 205–208. IEEE, Aug. 1993.

[29] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[30] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, July 1994. http://www.research.ibm.com/sop/soppubs.htm.

[31] Scott Pakin, Mario Lauria, and Andre Chien. High performance messaging on workstations: Illinois fast messages (FM) for myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[32] S. G. Parker and C. R. Johnson. Scirun: a scientific programming environment for computational steering. In *Proc. Supercomputing '95*, 1995. San Diego.

[33] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society Press, 1993.

[34] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, special issue (Performance Data Mining), submitted for publication, November 1997.

[35] G.-C. Roman, K. C. Cox, C. D. Wilcox, and J. Y. Plun. Pavane: A system for declarative visualization of concurrent computations. *Journal of Visual Languages and Computating*, 3(1):161–193, 1992.

[36] Beth Schroeder, Greg Eisenhauer, Karsten Schwan, Jeremy Heiner, Vernard Martin, and Jeffrey Vetter. From interactive applications to distributed laboratories. Technical Report GIT-CC-96-23, College of Computing, Georgia Institute of Technology, 1996. http://www.cc.gatech.edu/tech_reports.

[37] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley & Sons, Inc., 1996.

[38] Dilma M. Silva, Karsten Schwan, and Greg Eisenhauer. Configurable distributed retrieval of scientific data. In *Proc. 2nd International Conference on Configurable Distributed Systems (ICCDS)*, May 1998. Also available in http://www.ime.usp.br/ dilma/papers/iccds98.ps.

[39] Richard Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[40] J. Stankovic and K. Ramamritham. The design of the spring kernel. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 146–157, December 1987.

[41] P. David Stotts and James M. Purtilo. Virtual environment architectures: Interoperability through software interconnection technology. In *Proceedings of the Third Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 221–224, Morgantown, W.Va., April 17-19 1994. IEEE Computer Society Press.

[42] Jeffrey Vetter. Computational steering annotated bibliography. *ACM SIGPLAN Notices*, 36(6), 1997.

[43] J.S. Vetter and K. Schwan. Progress: a toolkit for interactive program steering. In *Proc. 1995 Int'l Conf. on Parallel Processing*, pages II/139–42, 1995.

[44] J.S. Vetter and K. Schwan. High performance computational steering of physical simulations. In *Proceedings of IPPS'97*, pages 128–132, 1997.

[45] Mark D. Wood and Bradford B. Glade. Information servers: A scaleable communication paradigm for wide area networks and the information superhighway. In *Proceedings of the 7th SIGOPS European Workshop.* Association of Computing Machinery, 1996.

Vita

Greg Stephen Eisenhauer was born on July 15, 1962 in Carbondale,Illinois. He entered the University of Illinois in Champaign-Urbana in 1980, graduating in 1983 with a Bachelors of Science in Computer Science with high honors and in 1985 with a Masters of Science in Computer Science. Upon graduation, he joined Honeywell's Systems and Research Center in Minneapolis, Minnesota. While at Honeywell, Greg's work included Distributed Ada, a project to create an Ada compiler for distributed target environments. In 1990, after five years at Honeywell, he joined the graduate program in Computer Science at the Georgia Institute of Technology. In 1996, Greg accepted a postition as a Research Scientist at Georgia Tech. Under the supervision of Dr. Karsten Schwan, he finished his dissertation, titled "An Object Infrastructure for High-Performance Interactive Applications", in June 1998.