

1994

# Implementation of a monitoring system for distributed programs

Ali Erkan

*Lehigh University*

Follow this and additional works at: <http://preserve.lehigh.edu/etd>

---

## Recommended Citation

Erkan, Ali, "Implementation of a monitoring system for distributed programs" (1994). *Theses and Dissertations*. Paper 306.

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact [preserve@lehigh.edu](mailto:preserve@lehigh.edu).

**AUTHOR:**

**Erkan, Ali**

**TITLE:**

**Implementation of a  
Monitoring System for  
Distributed Programs**

**DATE: October 9, 1994**

Implementation of a Monitoring System for Distributed Programs

by

Ali Erkan

A Thesis

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering and Computer Science

Lehigh University

September 1994

This thesis is accepted and approved in partial fulfillment of the requirements for the  
Master of Science.

9-27-94

---

Date

Thesis Advisor

---

Co-Advisor

Chairperson of Department

## Acknowledgements

I would like to thank Dr. Madalene Spezialetti for all her help and support. Her understanding and patience allowed me to experiment with many interesting issues of this topic.

I also would like to thank the EECS department of Lehigh University for supporting my graduate studies here at Lehigh.

And, as always, I would like to thank my parents Selma and Metin Erkan and my best friend Naomi for always being so helping and supportive.

# Table Of Contents

<b>Abstract</b>	.....	1
<b>Chapter 1</b>	<b>Introduction</b>	
	1.1 Monitoring System Overview .....	4
	1.2 The Monitoring Programming Interface .....	5
	1.2 The Monitoring System Programs .....	6
	1.3 Conclusion .....	7
<b>Chapter 2</b>	<b>Primitive Events</b>	
	2.1 Process Definitions .....	8
	2.2 Primitive Event Definitions .....	9
	2.3 Event Scripts .....	11
	2.4 Definition Tables .....	12
	2.5 Conclusion .....	13
<b>Chapter 3</b>	<b>The Monitoring Programming Interface</b>	
	3.1 Overview .....	14
	3.2 The IPC Modules .....	15
	3.3 The Integer Module .....	16
	3.4 The Point Module .....	17
	3.5 The Monitor Module .....	18
	3.6 Levels of Monitoring .....	19
	3.7 Sample Code .....	20
	3.8 Conclusion .....	22
<b>Chapter 4</b>	<b>The Monitoring System</b>	
	4.1 Goals .....	23
	4.2 The MS Programs .....	26
	4.3 Machine and Process Registrations .....	28
	4.4 The Event Recognition Scheme .....	29
	4.5 Evaluation .....	32
	4.6 Conclusion .....	34
<b>Chapter 5</b>	<b>Results and Conclusion</b>	
	5.1 Tests and Results .....	35
	5.2 Conclusion .....	39

<b>References</b>	.....	41
<b>Appendix A</b>	<b>Using the Monitoring System</b>	
A.1	Starting the System .....	42
A.2	User Commands .....	43
<b>Appendix B</b>	<b>Interprocess Communication Mechanisms</b>	
B.1	Files .....	49
B.2	Pipes .....	50
B.3	UNIX-Domain Sockets .....	50
B.4	Internet-Domain Sockets .....	51
B.5	Message Queues .....	52
B.6	Semaphores .....	53
B.7	Shared memory .....	54
<b>Appendix C</b>	<b>Characteristics of C++</b>	
C.1	The C++ class construct .....	56
C.2	The constructor/destructor functions .....	57
C.3	Object Declarations .....	58
C.4	Operator Overloading .....	59
C.5	Monitoring Programming Interface .....	60
<b>Biography</b>	.....	62

# List of Figures

<b>Chapter 1</b>	<b>Introduction</b>	
1.1	Overview of the monitoring system .....	6
<b>Chapter 2</b>	<b>Primitive Events</b>	
2.1	Example event script .....	11
<b>Chapter 3</b>	<b>The Monitoring Programming Interface</b>	
3.1	Integer object sample program .....	20
3.2	Client program of the socket example .....	21
3.2	Server program of the socket example .....	22
<b>Chapter 4</b>	<b>The Monitoring System</b>	
4.1	Monitoring system data structures .....	31
<b>Chapter 5</b>	<b>Results and Conclusion</b>	
5.1	Event script for test #3 .....	37
5.2	Event script for test #4 .....	38
<b>Appendix B</b>	<b>Interprocess Communication Mechanisms</b>	
B.1	UNIX-Domain Sockets .....	50
B.2	Internet-Domain Sockets .....	52
B.3	Message Queues .....	53
B.4	Shared memory .....	54



## Abstract

Monitoring is the process of gathering information about a program's execution. In sequential environments, monitoring is a straight-forward operation. Using conventional tools such as debuggers, sequential programs can be stopped and investigated at any time during their execution. With distributed programs, however, the same technique is often not applicable. Since distributed programs have multiple processes executing concurrently (each possibly on a different machine), stopping and analyzing the state of a process changes the execution pattern of the whole program.

An alternative method of monitoring more applicable to distributed programs is to make the program generate information about its state during its execution. This information is then collected and assembled into a global picture for analysis. Since conventional debuggers cannot be used to gather information in this manner, the monitored program is changed to automatically generate runtime information itself.

This project has been the first step in the development of a monitoring system for distributed programs. The result is the development of a small monitoring system constructed in two parts: the Monitoring Programming Interface and the Monitoring System Programs.

The Monitoring Programming Interface is a collection of modules that perform specific programming tasks and provide the monitoring hooks to extract run-time information. The Monitoring System Programs set up a framework over a network of computers to collect the monitoring information provided by the Monitoring Programming Interface.

# Chapter 1

## Introduction

**Monitoring** is the process of gathering information about a program's execution. In sequential environments, monitoring is a straight-forward operation. Using conventional tools such as debuggers, sequential programs can be stopped and investigated at any time during their execution. With distributed programs, however, the same technique is often not applicable. Since distributed programs have multiple processes executing concurrently (each possibly on a different machine), stopping and analyzing the state of a process changes the execution pattern of the whole program.

An alternative method of monitoring more applicable to distributed programs is to make the program generate information about its state during its execution. This information is then collected and assembled into a global picture for analysis. Since conventional debuggers cannot be used to gather information in this manner, the monitored program is changed to automatically generate runtime information itself.

One way of specifying the state information is to define it in terms of events [1]. A **primitive event** is a programming activity that cannot be decomposed into any smaller events. Some typical examples of primitive events are assigning-accessing memory locations, sending-receiving messages, and entering-exiting a function. By building a history of when primitive events occur, the monitoring system creates a trace of the program's execution [2,3,4,5]. Typically, monitoring systems allow the definition of **high-level events** which are built from primitive events using specific temporal and boolean operators. The advantage of high-level events is that they reduce the amount of data while increasing the amount of information

revealed.

There are various ways of extracting information about the occurrence of events from a program [6,7,8]. In some systems, the programmer manually inserts function calls into the code as monitoring statements. Some systems automate this procedure by constructing a translator which accepts a normal program and a set of event definitions to produce a new program containing monitoring statements. In some other systems where monitoring is considered to be an integral part of programming, programs are constructed with monitoring hooks from the bottom up. Finally, if hardware support is available, some systems perform monitoring by inspecting memory-port-bus access [6,9].

Monitoring systems need a global time frame in order to sort the extracted information. There are various choices for selecting an appropriate time frame but the simplest one is the use of the local clock of the machine on which the process runs. The obvious disadvantage of using local clocks is that the set of machines on which the distributed program spreads to may not be synchronized, making the time values from different machines incomparable.

This work addresses the problem of collecting monitoring data from distributed programs. A preliminary system has been implemented which monitors programs for the occurrence of primitive events. A special programming interface has been created to construct monitorable programs. The system time-stamps events using the local clocks of the host machines.

## 1.1 Monitoring System Overview

The implementation of this project has been the first step in the development of a monitoring system for distributed programs. The primary goals of the project were as follows:

- [1] Determining the requirements of constructing a monitoring system.
- [2] Determining a small yet sufficient set of primitive events to extract adequate monitoring information.
- [3] Determining a framework over a network of computers to collect the information generated by the monitored program.
- [4] Implementing a monitoring system which realizes the above three requirements.

The result was the development of a system made up of two parts: the **Monitoring Programming Interface** (MPI) and the **Monitoring System** programs (MS). The MPI is a collection of modules that perform specific programming tasks. If a program uses the MPI, then not only does it have a simple interface to several UNIX services, but also it automatically becomes monitorable. The MS programs are the actual monitoring system executables that set up a framework over a network of computers to collect the monitoring information provided by the MPI. The MS executables were written in C and the MPI was written in C++. The development was done on the IBM RISC/6000 workstations running the AIX operating system.

## 1.2 The Monitoring Programming Interface

Using a programming interface for the purpose of extracting runtime information means that the monitored program always contains monitoring hooks. There are three disadvantages of this approach:

[1] Some monitoring overhead is always present on the user programs even when the program is not being monitored.

[2] The programmer has to use a new set of tools to perform routine programming tasks.

[3] Existing programs cannot be monitored unless parts of them are rewritten using the MPI.

Using the MPI, however, also has substantial advantages:

[1] Since the monitoring statements are always embedded in the code, the runtime behavior of an unmonitored program is closer to the behavior of its monitored form.

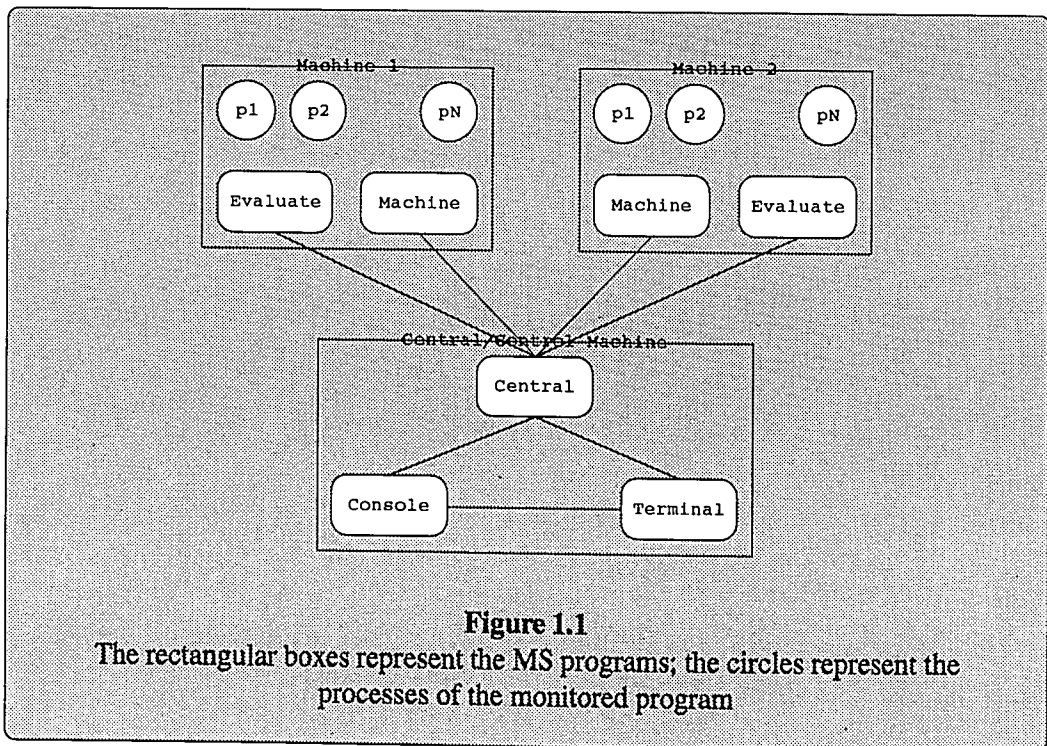
[2] The extraction of the monitoring data is cleaner, more reliable, and more efficient.

[3] The definitions of the events can be changed at run-time, eliminating the need to stop-edit-recompile-reexecute a monitored program just to try out a new set of events.

The MPI time-stamps the occurrence of events with the local time read from the host machine. Although this is not an accurate method for a global time frame, it serves the purpose of achieving some temporal order before a more sophisticated mechanism (i.e. vector clocks, simultaneous regions) is installed.

### 1.3 The Monitoring System Programs

The MS contains five programs. The **Console** and **Terminal** programs handle the user interaction. The **Machine** program is responsible for coordinating the monitoring activities on a single machine; each machine which hosts a process of the monitored program also hosts a process of the **Machine** program. The **Central** program forms the connection node for all



the remote parts of the system and functions as the router of all system wide messages. Finally, the **Evaluate** program monitors the performance of the MS. Figure 1.1 is a high-level view of the MS when two remote machine are connected.

## 1.4 Conclusion

This chapter provided a basic discussion of monitoring and presented an overview of the goals and the components of a monitoring system. The remaining chapters focus on the implementation of the monitoring system. Chapter 2 is a discussion of the primitive events, specifically the way they have been implemented in this project. Chapter 3 focuses on the MPI and shows how it generates state information. Having seen the event definitions and the method of extraction of monitoring information, chapter 4 provides a global description of the project structure. Chapter 5 contains some results and a conclusion.

## Chapter 2

# Primitive Events

As introduced in Chapter 1, primitive events are the simplest types of monitorable events and are used to extract low-level information from user programs. This chapter explains how this project implements primitive events. Since event definitions are based on process definitions, the first section explains process definitions. This is followed by a discussion of primitive event definitions and event scripts. The final section provides some knowledge on how the system maintains the process and event definitions and how the user can make modifications to them at run-time.

### 2.1 Process Definitions

In multitasking systems like UNIX, a file name is not enough to uniquely identify a process since multiple processes can be started from a single program file. The system assigned process IDs also cannot be used since those numbers are not known prior to execution. The solution is to allow symbolic process names to be used in event definitions and later map these names to actual processes. Declaring a process name and associating it with a unique process from a program file is called a **process definition**.

A process definition involves three attributes: a **process name**, a **program name**, and an optional **process count**. The syntax for a process definition is as follows:

```
process <process-name> from <program-name>[process-count]
```



The **process-name** attribute names the new process. The **program-name** attribute is the name of the program file whose execution creates the new process. The optional **process-count** associates the process-name with a specific process of the program file. For example, “**process p from prog[3]**” associates the name “p” with the third process started from “prog”. Without a process-count, the process-name associates with any process started from the specified program file (i.e. “**process p from prog**”). A process definition with a process-count is called a **specific process definition**. A process definition with no process-count is called a **general process definition**.

## 2.2 Primitive Event Definitions

Using primitive event definitions, the MS is instructed about the primitive events for which a program should be monitored. Although the notion of a primitive event is almost an invariant concept for many monitoring systems, each implementation has a specific syntax for their declaration. This project uses an object oriented syntax of six attributes:

```
event <event-name>  
    is <primitive-action>  
    on <object-name>  
    at <machine-name>.<process-name>.<function-name>
```

The **event-name** attribute names the new event. The **primitive-action** attribute is the specification of a programming activity such as `integer_assign` or `socket_read`. The **object-name** indicates what object of the monitored program the primitive-action will be applied to. Since

the monitored program can start a process on any machine, the **machine-name** attribute specifies which machine will contain the process that will trigger the event. Finally, the **process-name** and the **function-name** attributes precisely locate the object on the named machine. The event definition will be rejected if the specified process-name has not been declared by a previous process definition.

There are several silent errors that can take place during an event definition. First, since the system does not know the type of the named object, it cannot determine whether the primitive-action is applicable to it. For example, if the event is `socket_open` on object "S" and "S" happens to be an integer rather than a socket, then the system will not catch this error. Second, the system does not check the source code to make sure that the named object and the named function exist. Finally, the object and the function names of the definitions can be truncated if their length exceed a certain limit, thereby making them different than the corresponding object and function names found in the monitored program. If any one of these cases hold, then the occurrence of the event will not be recognized.

Although the event definition syntax is very specific, there are two generalities that allow an event name to represent multiple events. The first is related to process definitions. Since a general process definition stands for multiple processes, an event based on a general process definition (rather than a specific one) stands for a group of events. The second generality is related to the machine-name attribute of an event definition. Machines must be specified as Internet machine names (i.e. in string form, not the numeric IP address). If, however, an "\*" is used as a machine name, then the event is defined on all machines the monitored program may run on.

## 2.3 Event Scripts

An **event script** is a text file that contains process and event definitions. The MS supports event scripts to facilitate the configuration of the system. An event script contains four types of entries: process definitions, event definitions, comments, and blank lines. Process and event definitions describe the activity for whose occurrence the system will be monitored. Since event definitions are dependent on process definitions, the user must make sure that processes are declared before they are referred to in any event definition. Event scripts are processed line by line and thus no definition can be extended beyond a single line. Any line that starts with the “#” character is assumed to be a comment. Blank lines are automatically skipped.

Figure 2.1 is an example event script. Since event **E** refers to a general process definition, it will be monitored in all executions of “prog”. Unlike **E**, events **E1** and **E2** refer to specific process definitions (namely, the first and the third instances of “prog”). For **E**, **E1**, and **E2** to

```
# Process definitions for P, P1, and P2
process P from prog
process P1 from prog[1]
process P2 from prog[3]

# Event definitions for E, E1, E2, and EAll
event E is integer_assign on i at PL118A.P.init
event E1 is integer_assign on i at PL118A.P1.init
event E2 is integer_assign on i at PL118A.P2.init
event EAll is integer_assign on i at *.P.init
```

**Figure 2.1**

A sample event script with process definitions P1, P1, P and event definitions E, E1, E2, EAll.

occur, program "prog" must be executing on the machine PL118A. However, for EAll, there is no such restriction. Regardless of what machine "prog" is started on or what the current process count happens to be for "prog", EAll is always monitored by the system.

## 2.4 Definition Tables

The system maintains a master table on a designated machine for all definitions. The system also maintains local tables on each machine that participates in the monitoring activity. The master table is used to perform error checking and to resolve dependencies. The local tables are used in the actual monitoring.

When the MS is running, any new process or event definition is immediately effective. This means that new event definitions are forwarded to the appropriate machines as soon as the event is determined to be syntactically valid (as stated before, if the machine name is "\*", then all machines get a copy of the event). New process and event definitions can be entered at any time during the execution of the system.

The user can also remove process and event definitions. When an event is removed, the system notifies the machine which currently contains a local copy of the event. The result of undefining an event is not only discarding the messages that are generated for the occurrence of that event: the system actually turns off the generation of messages, thereby reducing the total monitoring overhead. Undefining a process has more consequence than undefining an event. Since multiple events may be dependent on the process, the system may have to undefine multiple events when a process is undefined.

## 2.5 Conclusion

This chapter expanded on the notion of primitive events that was first introduced in chapter 1. Specifically, the syntax of event and process definitions were explained along with a description of event scripts. The next chapter introduces the Monitoring Programming Interface (MPI) component of the system which generates the information described in the primitive event definitions.

## Chapter 3

# The Monitoring Programming Interface

The first step in monitoring is the collection of information about the occurrence of primitive events. Since ordinary programs do not automatically provide this type of information, they must be changed at the source code level to emit time-stamped messages as the designated primitive events take place. Considering the difficulty of changing an existing program into a monitorable form, this project takes a simple approach to collect monitoring information.

To make a program monitorable, a programmer must use the **Monitoring Programming Interface** (MPI) to perform specific tasks. Some of these tasks are based on interprocess communication while others are based on data manipulation. The MPI was designed to be as usable and as general as possible so that its inclusion does not become a hindrance to normal programming. This chapter explains the design issues and the structure of the MPI. Since the MPI has been implemented in C++, appendix C contains some background information for the object-oriented issues of the discussion.

### 3.1 Overview

The MPI is a collection of seven modules. Since most events of interest in distributed programs are based on interprocess communication (IPC), five of the seven MPI modules handle various types of IPC: **Signals, Sockets, Semaphores, Message Queues, and Shared Memory** (please refer to appendix B for a general description of these IPC mechanisms).

The sixth module, **Integer**, allows the MS to trace the manipulation of integer values (specifically access and assignment operations). The seventh module, **Point**, generates information about where the program currently is in its source code. In addition, there is the separate **Monitor** module which forms the glue between the MPI modules, the user program, and the MS itself.

Each MPI module is the implementation of a C++ class. To make a program monitorable, objects from these classes must be used instead of the usual C counterparts. For example, in order to create a socket connection between two processes, a monitorable program creates a server socket object in one process and a client socket object in another. The usual C way of doing this would have been to make the appropriate system calls to create an integer descriptor based connection.

## 3.2 The IPC Modules

The first category is an interface to five of the IPC mechanisms provided in UNIX systems (please refer to appendix B for a general description of UNIX IPC support). Aside from the monitoring aspect, the MPI considerably simplifies the use of these IPC mechanisms by providing a consistent interface to all of them. The actual implementation of the MPI modules is broken in two layers: an underlying C module provides the code doing the system calls; a wrapping C++ layer provides the monitoring aspects and error checking. The C++ layer also associates each IPC mechanism with an identity. For example, an MPI socket is not simply an integer, but is an object with a name and a declaration point (the declaration point is the name of the function in which the object is declared). Because objects with

identities can be identified anywhere in the program without ambiguity, event definitions based on IPC objects turn out to be more natural than having them based on integer valued descriptors.

Using the MPI does not make the program dependent on the MS. When used as stand alone modules, the MPI components form an IPC library to various UNIX mechanisms. In fact, considering the uniform interface, simpler function calls (compared to the system calls), and optional error checking to trap programmer errors, it is advantageous to use the MPI instead of making direct system calls.

### 3.3 The Integer Module

Since it is useful to have primitive events that are based on the values of program variables, the MPI includes the **Integer Module** implementing a monitorable integer type. The use of integer objects is very similar to the use of ordinary integer variables in a C program. The module overloads all integer related operators so that the use of an integer object transparently notifies the MS when the integer is accessed or assigned.

The difficulty with monitoring a variable has to do with the many ways the value of that variable can change in a typical program. Ranging from a direct assignment to an indirect change through an alias, each access/assignment must be trapped and reported by the system. In addition, since C allows the nesting of statements (i.e. consider the C way of doing a "while not EOF" loop where the while statement contains an assignment and a boolean expression), a variable can be accessed/assigned more than once in a single C statement.



The MPI overcomes these problems with the Integer module. The use of the integer objects is almost the same as the use of ordinary integer variables since all the operators applicable to integers are overloaded. However, the difference between an integer and an integer object becomes visible when the value of the object is needed in a specific case: if the object cannot be associated with any one of the overloaded operators (i.e. when the object is just by itself and is not in any form of an expression), then the ! operator must be used to retrieve its value.

If the program is attached to the MS while it is running, then an access message is generated each time the value of the object is referred to. Similarly, each assignments causes an assign message to be sent to the MS. Both messages carry the value of the object in case the MS defines events that depend on the value of the integer rather than the knowledge of when it is assigned or accessed.

### 3.4 The Point Module

By using the **Point Module**, a program can generate messages when it reaches arbitrary points in the code. The construction and destruction of point objects generate messages. Therefore, by placing the declaration of a point object in a function, the system knows when the function is invoked. As the function is started, the compiler automatically executes the constructor function of the Point object. Similarly, when the function ends, the compiler automatically executes the destructor function. This means that the monitoring of function invocation requires no monitoring statements other than a simple object declaration.

In order to notify the system when the program reaches arbitrary points in the code, the

Point object can be forced to generate messages. Since arbitrary messages are not part of the object construction / destruction scheme, the generation of the forced messages is at the discretion of the programmer.

### 3.5 The Monitor Module

The **Monitor Module** forms the link between a monitored program and the MS. Each monitored program must declare a single Monitor object in global scope. Upon creation, the constructor of the Monitor object automatically establishes the required connections to transfer monitoring data. Upon exit, the destructor of the Monitor object disconnects the program from the MS. The Monitor object is also responsible for detecting the state of the MS; if the MS exits, then the Monitor object suppresses the generation of all monitoring messages and the program resumes its execution unaffected by the termination of the MS.

When the MPI modules capture the occurrence of a primitive event, they submit a record of the event to the Monitor object. The Monitor object time-stamps the event and sends it to a message queue used system wide (a message queue is a FIFO structure maintained by the kernel and is a generalization of the standard UNIX IPC channels like pipes; please refer to appendix B for details). If the transmission of the message fails, the Monitor turns down all subsequent reports of event occurrences.

## 3.6 Levels of Monitoring

Using the MPI does not force the program to generate monitoring information; it simply makes it possible. The programmer can manually set the level of monitoring information through the use of the MPI classes. It is also possible to write programs that turn the monitoring on or off depending on program specific conditions.

One of the parameters of the constructor for a Monitor object determines whether the Monitor should be connected to the MS at all. If this connection is suppressed, then the program and the included MPI modules work as normal but no events are generated from the program.

The declaration of the MPI objects provides a second level of control. If the object declaration is given a reference to the main Monitor object, then the object produces messages for the occurrence of primitive events. Otherwise, its use is "silent".

The third and final level of control has to do with the use of the objects. Most of the member functions of the MPI objects have a parameter which determines whether the call should cause a primitive event message to be sent. For example, a `socket.read()` creates the a "read" message only if the "report" parameter of the call is on.

Since each of these methods of control depend on the parameters passed to the MPI objects, it is possible to write programs which can dynamically change the amount of monitoring data they produce. For example, if a program encounters an error, it may decide to generate messages for the MS to trace its execution. The program may then turn off these messages if

a satisfactory condition is reached. This way, the MS observes the program only when it is necessary to do so, reducing the overall IPC overhead.

### 3.7 Sample Code

The program in figure 3.1 illustrates the use of the Integer module. It contains a Monitor object "Monitor" in global scope as mentioned above. Integer object "i" is a monitorable integer which has a reference to "Monitor". The "i" object gets an identity by being associated to a name (string "i") and a declaration point (string "main"). Each access/assignment on "i" generates a message containing its value. The only place where the use of "i" looks different than an ordinary integer is when its value is being referred to by itself (i.e. not when

```
#include "monitor.hxx"
#include "integer.hxx"

Monitor_c Monitor( TRUE );

int main( int argc, char *argv[] )
{
    Integer_c i( sMonitor , "main" , "i" , 0 );

    i = 1;
    Print( stdout , "Initial value is %d\n" , i );

    for ( i = 0 ; i < atoi( argv[ 1 ] ) ; ++i )
        Print( stdout , "Loop value is %d\n" , i );

    return( 0 );
}
```

**Figure 3.1**  
Sample program showing the use of an integer object

it associates to an operator). The ! operator is used to retrieve its value in such cases. At all other conditions, (assignment, comparison, expression), the use of an integer object cannot be distinguished from the use of a real integer.

Figures 3.2 and 3.3 show two programs for illustrating the use of the socket class. In both programs, a Monitor object and a Socket object have been declared. The server program additionally has a SocketServer object which sets up a connection request socket to accept the connection request of the client. When connected, the client sends a string to the server and the server echoes the string to stdout.

```
#include "monitor.hxx"
#include "socketx.hxx"

#define Port 30000
#define Message_k "This is the client talking"

Monitor_c Monitor( TRUE );
Socket_c Connect( &Monitor , "Connect" , "global" );

int main( int argc, char *argv[] )
{
    Connect.Open( Port , argv[ 1 ] , 1 );
    Connect.Write( Message_k , strlen( Message_k )+1 , 1 );
    Connect.Close();

    return( 0 );
}
```

**Figure 3.2**  
Client program for the socket example

## 3.8 Conclusion

The last two chapters discussed primitive events in detail. It has been pointed out that the user can define primitive events based on the use of IPC channels as well as the manipulation of data objects. The ability of the system to detect these events depend on the utilization of the MPI part of the system. Now that monitoring information is generated, Chapter 4 presents the framework used to collect this information.

```
#include "monitor.hxx"
#include "socketx.hxx"

#define Port_k 30000
#define Message_k "This is the client talking"

#define Buffer_k 256
typedef char Buffer_t[ Buffer_k ];

Monitor_c Monitor( TRUE );
SocketServer_c Request( &Monitor , "Request" , "global" );
Socket_c Connect( &Monitor , "Connect" , "global" );

int main( int argc, char *argv[] )
{
    Buffer_t Buffer;

    Request.Open( Port_k , 1 , 1 );
    Connect.Open( Request , 0 );
    Connect.Read( Buffer , Buffer_k , 0 , 1 );
    printf( "Message received from client: %s\n" , Buffer );

    return( 0 );
}
```

**Figure 3.3**  
Server program for the socket example

## Chapter 4

# The Monitoring System

The Monitoring System (MS) consists of a set of programs and a set of programming modules. The programming modules are used to construct programs that automatically generate messages as primitive events occur. The MS programs are used to set up a framework to collect this monitoring information.

This chapter explains the design and the structure of the MS, specifically referring to the MS programs. The discussion starts with the goals of the system leading into brief explanations of the individual MS programs. This is followed by the machine and process registration procedures which leads into the recognition scheme of primitive events. Finally, the initial design goals are noted once more in order to show how the MS design is consistent with them.

### 4.1 Goals

The design of any software system requires a set of goals to be specified before the actual implementation starts. These goals determine how numerous development issues are resolved and provide a general road-map for the construction of the system. This section explains some of the important issues relevant in the development of a monitoring system. It is important to evaluate the final product and understand how well the issues have been realized.

#### 4.1.1 The Monitored Program - Monitoring System Connection

Making a program monitorable should not make that program dependent on the MS to execute. That is, even with the code added to extract monitoring data, the program should still be able to run by itself. In addition, the connections between the program and the MS should be easily breakable. This way, if the MS terminates (either because an internal error occurs or because the user stops monitoring), the program is not forced to exit.

#### 4.1.2 The Monitored Program - Monitoring System Interaction

Depending on the implementation, the execution of the MS may effect the execution of the monitored program. For example, if the monitored program uses signals in its normal execution and if the MS starts communicating with this program using the same signals, then signals of different origins will interfere with each other. This will cause the program to execute differently than it would have executed without the MS. To prevent such interference, the execution of the MS should be as isolated as possible from the execution of the application.

#### 4.1.3 Dynamic Event Definitions

The monitored program may run long enough for the user to try different sets of events. For example, if a server program is being monitored, the different service requests received by the server may require different sets of event definitions. Therefore, the MS should provide the means to change event definitions at run-time.



#### 4.1.4 Eliminating Unnecessary Monitoring Messages

Events are defined on processes rather than on program files: for two processes, P1 and P2, of the same program, the user should be able to define different sets of events even though P1 and P2 execute the same code. Since programs are modified at the source code level to generate monitoring information, defining different sets of events on different processes of the same executable causes large amounts of unnecessary messages to be generated. In addition to contributing to the monitoring overhead, the extra data also slows down the evaluation since the system must identify it as "discardable". Therefore, the MS must either stop the generation of unnecessary messages, or at least suppress their transmission.

#### 4.1.5 Measuring the Evaluation Backlog

The intensity of the evaluation depends on many factors, most of which have to do with the dynamics of the monitored program. In the simplest case, if a monitored value is rapidly changing, then the evaluation of the event starts lagging behind the changes that are applied to the value. Because one of the ultimate goals of this project is benchmark the performance of various monitoring strategies, it is important to have the means to measure how much the evaluation lags behind the execution of the program.

#### 4.1.6 Handling the fork() Call

The MS should be able to monitor new processes that are started by the fork() system call. The problem with fork() is that it creates a new process which does not go through its initialization code. This causes the parent and the child processes to be indistinguishable as

far as the MS is concerned since the child does not explicitly register itself. When the two processes are not distinguished, they end up sharing the IPC links which are meant to be unique for each monitored process. As a solution, the MS should provide a modified fork() call which causes the child to be recognized as a new process separate from the parent.

#### 4.1.7 Support for Local Evaluation

In a centralized monitoring system, all monitoring messages must be forwarded to a central node for evaluation. However, depending on the event definition, it may be possible to evaluate an event based on the information available from a single process. Similarly, an event may be resolved by the information that comes from a number of processes all on the same machine. The monitoring system should support evaluation schemes which makes use of the locality of these events.

## 4.2 The MS Programs

The MS contains five programs to set up a network for collecting the messages generated by the monitored program. These programs are:

[1] The **Console** program: **Console** handles the user interaction and is driven by the commands entered by the user. **Console** is responsible for storing all the process and event definitions. These definitions are stored in shared memory so that the **Terminal** and **Central** programs also have access to them. **Console** commands are explained in detail in appendix A.

[2] The **Terminal** program: **Terminal** displays the messages generated by the monitored program. Through **Terminal**, the output of the system can be channeled to the screen or to a log file.

[3] The **Central** program: The **Central** program connects all remote parts of the MS into a single network. When a new definition is entered by the user, it is the responsibility of **Central** to relay it to the appropriate machines. **Central** also keeps a simple load statistic for each connected machine in order to give an idea about the performance of the MS. **Central** is automatically started by the **Console** program. Since **Central** does not have its own output window, it always communicates with the user by sending display messages to **Terminal**.

[4] The **Machine** program: An instance of the **Machine** program is started on each remote machine that the distributed program spreads to. The responsibilities of **Machine** include handling of the shared data structures, the registration of the machine, the registration of the monitored processes, etc. A **Machine** process is not expected to do any evaluation; it is simply a coordinator.

[5] The **Evaluate** program: The **Evaluate** program is meant to perform machine scope evaluations on the host machine. However, it is currently limited to waiting on the system message queue and relaying the accumulating messages to the **Machine** process. **Terminal** also keeps track of the heaviest load seen on the message queue as a simple performance figure of the system.

Regardless of the size and the distribution of the monitored program, one process is created

from the **Console**, **Terminal**, and **Central** programs. The machine which hosts these three processes is called the **central machine**. The user controls the MS from the central machine.

### 4.3 Machine and Process Registrations

The MS is started when the user starts the **Console** and **Terminal** programs. The rest of the system automatically starts as various components of the monitored program are started. There are two kinds of registration procedures that establishes the network connections to monitor a distributed program: **machine registration** and **process registration**.

When a monitored program is started on a host machine, the first part of its initialization is to register itself with the **Machine** process of that host. A process registration involves two steps. First the process presents some runtime information about itself to **Machine**. **Machine** in turn incorporates the new process in the monitoring scheme and notifies the new process which symbolic process name it corresponds to from the process definitions. When the new process is given an identity from **Machine**, it is able to determine which primitive events it should report to the MS. This procedure is called a **process registration**.

When a **Machine** process is started, part of the initialization is creating a connection to the central machine. When the central machine accepts the connection request, **Machine** sends some information about the host the **Central**. **Central** in return sends all the relevant events that are to be monitored on any user process started on this new host. This procedure is called a **machine registration**.

It is the process registration of the first program started on a new machine that triggers the machine registration procedure. When a monitored user program is started, the monitoring code attached to that program realizes that the machine does not have a **Machine** process. It first starts the **Machine** program and then starts the **Evaluate** program. When **Machine** is started, it completes the machine registration and notifies the user program that the machine setup has been completed. Any subsequent process registration finds **Machine** to be running and thus skips the host machine setup phase. This scheme is repeated on any machine that the user program spreads to.

## 4.4 The Event Recognition Scheme

As explained in earlier, each primitive event is associated with a five element description:

- [1] **<action>**: The primitive event action (i.e. read, write, assign, etc)
- [2] **<object>**: The name of the object the action is applied to
- [3] **<function>**: The name of the function the object is declared in
- [4] **<process>**: The name of the process the event is defined on
- [5] **<machine>**: The name of the machine the process runs on

The **<machine>** attribute is used when the event definition is sent to the appropriate machine during registration. The **<process>** attribute is used when assigning processes to a subset of the events defined on a single machine. The **<action>/<object>/<function>** attributes are used by the monitoring code attached to monitored programs when recognizing the occurrence of primitive events.

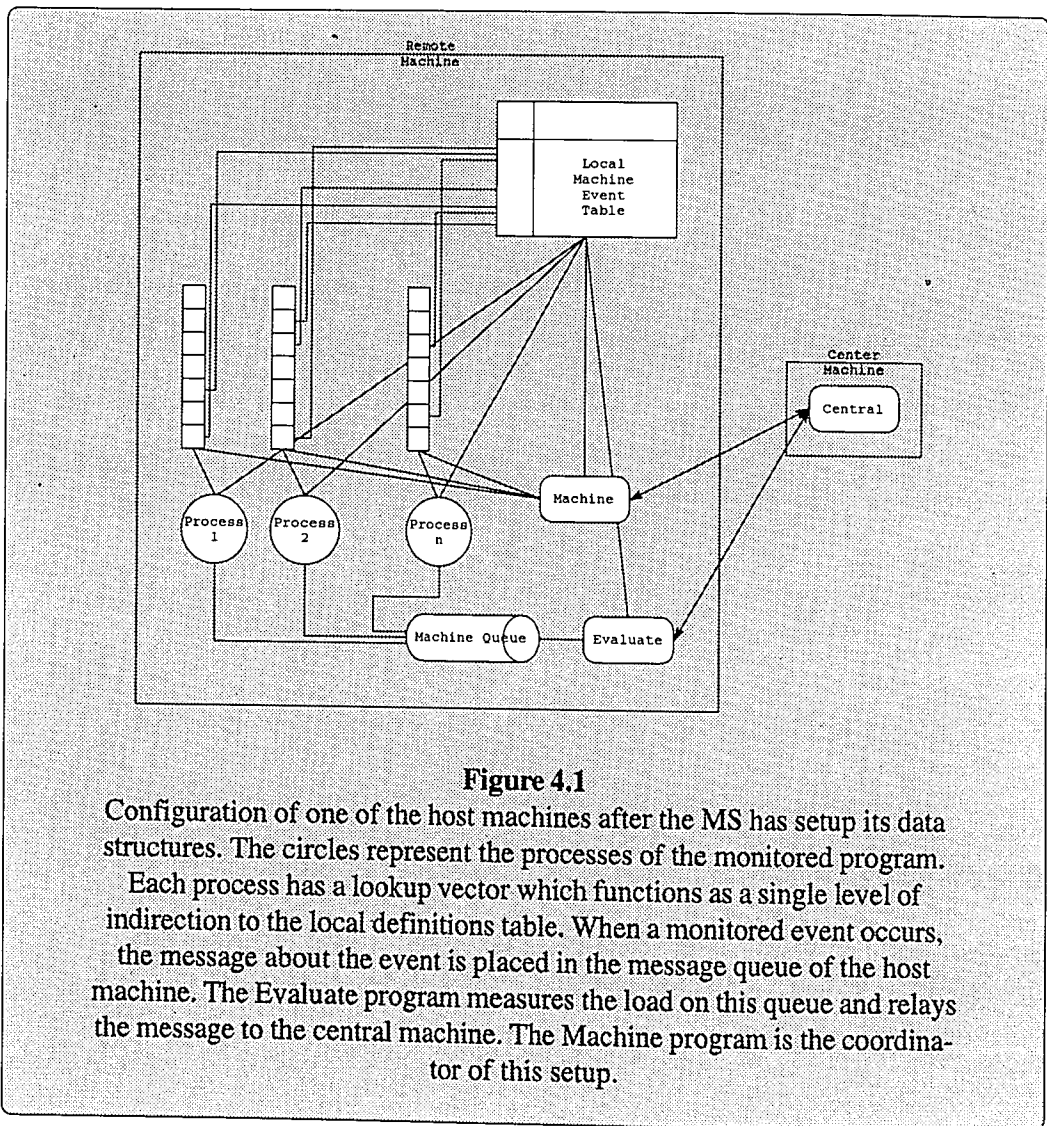
Monitored programs have the potential to generate messages for every type of primitive event that the user could be interested in monitoring. But only those events that are currently specified in the event definitions are reported. This protocol is implemented via a lookup scheme that each monitored program goes through before reporting an occurrence. The lookup scheme forms the core of many of the features of this project.

When a process registration completes, the **Machine** program stores the relevant information about the process in a process-record in shared memory. This process-record contains a lookup vector which is a set of references to the events to be monitored on that host machine. By controlling the contents of the lookup vectors, the MS is able to control exactly what each process of the monitored program should report. Monitored processes are given references to their own lookup vectors during their registration phase.

When a primitive event occurs, the part of the MS that is attached to the user process computes an index from the <action>, <object>, and <function> attributes of the event. This index is used to refer to the lookup vector. If the indexed position contains a reference to a valid event definition, then the process reports the occurrence. Otherwise, the event is ignored.

When a new event is defined by the user, the event definition is forwarded to the appropriate **Machine** process by the content of the <machine> attribute of the definition. When a **Machine** process receives a new definition, it evaluates the <process> attribute of the event to determine which lookup vector to update. If the specified process is currently running, then **Machine** updates the lookup vector of that process to collect information about the occurrence of the new event. Similarly, when an event is undefined, the appropriate **Machine**

process is notified so that the lookup vector of some process can be updated. In this case, a reference is removed from the lookup vector. Figure 4.1 gives an overview of this setup based on the lookup vectors.



**Figure 4.1**

Configuration of one of the host machines after the MS has setup its data structures. The circles represent the processes of the monitored program.

Each process has a lookup vector which functions as a single level of indirection to the local definitions table. When a monitored event occurs, the message about the event is placed in the message queue of the host machine. The Evaluate program measures the load on this queue and relays the message to the central machine. The Machine program is the coordinator of this setup.

## 4.5 Evaluation

Having presented the structure of the MS, this section examines how the system present an evaluation on how the system meets the following requirements outlined in section 4.1:

- [1] The Monitored Program - Monitoring System Connection
- [2] The Monitored Program - Monitoring System Interaction
- [3] Dynamic Event Definitions
- [4] Eliminating Unnecessary Monitoring Messages
- [5] Measuring the Evaluation Backlog
- [6] Handling the fork() Call
- [7] Support for Local Evaluation

A monitored program and the MS communicate with each other through shared memory. Part of the MS that is attached to the user processes always checks for specific life signs of the MS. When these life signs are not found, the programs detaches itself from the shared data structures and continues its execution without generating any monitoring messages. Thus when the MS terminates, the program is not forced to exit.

Because the interaction between the monitored programs and the MS is through shared memory, the execution of the MS does not interfere with the program's execution. This makes it possible to monitor programs that setup several processes which communicate with each other using signals, pipe, and socket.

Dynamic event definitions are handled by using a single level of indirection to the event



definitions. Under the control of the **Machine** program, user processes can be attached and detached from event definitions at run-time. New events add references to the lookup vectors of processes while the removal of process and event definitions remove references.

The use of the lookup vectors also provide a way of stopping unnecessary monitoring messages. To illustrate this by example, let P1 be the first process from a program file and let P2 be the second. If the set of events defined on P1 is different that the set of events defined on P2, then the MS adjusts the lookup vectors of the two processes differently. P1's lookup vector refers to only the events that P1 is expected to report, while P2's lookup vector refers to only the events that P2 is expected to report. Thus, although P1 and P2 share the same executable code with the same embedded monitoring statements, they generate process specific messages only.

The problem of measuring the system's performance is handled by collecting the occurrence messages using message queues. The **Evaluate** program is responsible for monitoring these queues to determine the heaviest load the system sees during the execution of each user program. Whenever a new load-high is observed, the **Evaluate** of that machine sends a special load message to **Central**. **Central** in turn keeps track of the loads of all the connected machine. The user can ask the system about these load figures to get a feel for how well the system is keeping up with the execution of the monitored program.

To handle the monitoring of the new processes created by the monitored program, the MS's version of the fork() function must be called. The monitored fork() first disconnects the child from the system and then submits a new connection request. This allows the child process to be recognized as a distinct process from its parent.

Finally, the support for local evaluations is achieved by keeping the event definitions in shared memory. This way, each monitored process has access to the monitoring information from the other processes running on that machine. Thus an evaluation agent can determine whether machine-scope events have occurred without initiating a single message transfer.

## 4.6 Conclusion

Chapters 2, 3, and 4 have focused on the main components of the implementation and outlined the use of the system. In summary, the monitoring system is made up of two main parts: the **MS programs** and the **MPI**. User programs automatically become monitorable when they use the MPI for some of the distributed programming tasks. The user defines primitive events using **event scripts**. The MS is started by executing the **Console** and **Terminal** programs; the rest of the system is automatically started as the monitored program runs. The event messages are collected over the network and presented to the user for inspection.

The next chapter presents a few of the test cases the system was experimented with.

## Chapter 5

# Results and Conclusion

The MS was tested on a number of monitoring situations to verify its functionality. The first section of this chapter summarizes the results of these tests. The second section concludes by pointing out how this project can be expanded into a practical tool for monitoring distributed programs.

### 5.1 Tests and Results

Since monitoring requires extensive interprocess communication, it is important to understand the IPC throughput of the underlying system to better judge the performance of the monitoring program. Although many forms of IPC mechanisms are used in this project, only one mechanism involves transferring information over the network: Internet-domain sockets. This naturally makes Internet-domain socket connections the bottleneck of the system. Using a few test programs, the throughput of Internet-domain sockets were estimated to be about an integer (four bytes) per millisecond. Therefore, if the rate of monitoring is higher than this rate, the monitoring system slows down the execution of the monitored program. It was also illustrated that, up to a practical limit, the rate of communication on Internet-domain sockets has little to do with the message size; the real cost is the overhead of setting up a point-to-point communication link for each transfer.

### 5.1.1 Testing the MPI Overhead

The first experiment determined the overhead imposed on the monitored program by the MPI. The test program contained a for loop performing a simple computation. The control case was using a normal integer variable as the loop counter. The first test case ran the same loop with an integer object instead of an integer variable **with no monitoring**. The second test case ran the loop with an integer object again, but this time, **with monitoring**.

It was seen that the loop slows down by about four times when monitoring is off and by about 50-100 times when monitoring is on. The reason for the enormous overhead of the second case has to do with the string processing involved in using lookup tables. However, the results can be misleading since the spinning of a tight loop is not a fair test for a tool that is meant to be used for the monitoring of distributed programs. This is especially true considering the fact that the real events to be monitored will mostly be interprocess communications and these events are orders of magnitude slower than a single iteration of a simple loop. Nevertheless, this test was valuable for pointing out the worst case.

### 5.1.2 Testing Machine Loads

As mentioned in the previous chapters, each machine contains a message queue in which all monitored processes deposit their occurrence messages. The purpose of these message queues is to get an estimate on the accumulation of the event messages and to benchmark the performance of the monitoring system. The second test was done to check the capacity of the message queues and to see the effect of monitoring while the message queues are completely filled.

The test program contained a loop with an assignment statement on an integer object. As the number of iterations were increased, the number of stacked messages in the message queue was counted. It was observed that at around 700 iterations, the message queue was saturated. Beyond this, monitored processes had to wait for the message queue to be emptied before writing further messages to it. As mentioned earlier, the backlog of the message queue is a result of the slowness of the Internet socket connection that relays the messages to the central machine.

### 5.1.3 Testing the Dynamic Event Definitions

The results of this test was more qualitative than quantitative. The test involved a server program that expects strings to be sent to it from its clients and a client program that sends strings to its server. The events were defined by the event script shown in figure 5.1.

The system successfully monitored all usage of the socket connection between the two processes. When strings were sent rapidly from the client to the server, monitoring on either end of the socket could be temporarily undefined causing the throughput of the system to double

```
process Client from lines_c
event ClientOpen is socket_connect on Connect at pl118e.Client.global
event ClientWrite is socket_write on Connect at pl118e.Client.global
event ClientClose is socket_close on Connect at pl118e.Client.global

process Server from lines_s
event ServerAccept is socket_accept on Connect at pl118e.Server.global
event ServerRead is socket_read on Connect at pl118e.Server.global
event ServerClose is socket_close on Connect at pl118e.Server.global
```

**Figure 5.1**  
Event script for test #3

as expected.


#### 5.1.4 Testing the Handling of the fork() Call

The ability of the system to monitor the creation of new processes with the fork() call was tested with one of the classic introductory programs of parallel computation. A program reads a matrix from a disk file to sum up the coefficients. However, the processing is done by assigning a child process to the summation of each vector of the matrix while the parent process collects the results of the children to compute the matrix total. In this program,  $N+1$  processes are involved for the summation of an  $N$ -by- $N$  matrix.

The program is written to print only the matrix sum and not the vector sums. But using the simple event script of figure 5.2, the monitoring system retrieves each of the vector sums and displays them through the Terminal program. Although the monitoring system does not know the number of processes that will be started, the general process definition "p" refers to all of them.

```
process p from parallel
event E is integer_assign on Show at p1118e.p.Sum
```

**Figure 5.2**  
Event script for test #4



## 5.2 Conclusion

This thesis has completed the first step in the development of a monitoring system for distributed programs. A protocol has been established for extracting monitoring information from executing programs. A simple syntax has been defined for process and event definitions. The resulting implementation has been tested with various programs. The rest of this section describes in what ways the project can be improved towards making it a practical tool.

The first important issue is related to the time-stamping of occurrence messages. Currently, a message for an event is time-stamped **after** that event has occurred. Because the occurrence and the time-stamping is not handled as one atomic step, the scheduling decisions of the Operating System leads to inconsistent views of the program. For example, in some of the test cases, the reception of a message is reported earlier than its sending even when the sender and the receiver are located on the same machine. If, on the other hand, the occurrence and the time-stamping is handled as an atomic step, then it is possible for the system to lead into deadlock.

Another critical issue is the construction of a global time frame. Currently, all time values are based on the local clocks of the machines on which the program runs. However, since these clocks are not synchronized, time values recorded on different machines are not truly comparable. An alternative time scheme that is more applicable to distributed systems is **vector clocks**. The advantage of vector clocks is that the system establishes its own virtual time frame in which events can be partially ordered in the time domain. Vector clocks, however, are based strictly on IPC. Another method is to use **simultaneous regions** which allows a broader set of events to be defined and monitored for.

The third and final improvement is to add the definition of high-level events to the monitoring system. A monitoring system that accepts only primitive event definitions is too low-level to be used in many practical cases. The system should be able to summarize the data using higher abstractions of programming activities.



## References

- [1] P.C. Bates and J.C. Wileden, "High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach," *J. of Systems and Software*, 3(4):225-264, 1983.
- [2] J. Fidge, "Timestamps in Message Passing Systems that Preserve the Partial Ordering," *Proc. 11th Australian Computer Science Conference*, pp.55-56, 1988.
- [3] L. Lamport, "Time, Clocks and the Ordering of Events in Distributed Systems," *CACM*, 21 (7):558-565, 1978.
- [4] F. Mattern, "Virtual Time and Global States of Distributed Systems," In M. Cosnard, editor, *Parallel and Distributed Algorithms*, pp.215-226, Elsevier Science Publishers, Amsterdam, 1989.
- [5] M. Spezialetti and J.P. Kearns, "Efficient Global Snapshots," *Proc. 6th ICDCS*, pp.382-388, 1986.
- [6] D. Haban, "DTM - A Method for Testing Distributed Systems," *Proc. 6th Symp. on Reliable Distributed Software and Database Systems*, pp.66-73, 1987.
- [7] J. Joyce, G. Lomow, K. Slind, and B. Ungar, "Monitoring Dist. Systems," *ACM TOCS*, 5(2):121-150, 1987.
- [8] R. LeBlanc and A. Robbins, "Event Driven Monitoring of Distributed Programs," *Proc. 5th ICDCS*, pp.512-522, 1985.
- [9] J.P. Tsai, K.F. Fang and H. Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems", *IEEE Computer*, 23(3):11-23, 1990.

# Appendix A

## Using the Monitoring System

When a program is built with the monitoring programming interface (MPI), the monitoring system (MS) automatically collects information about the occurrence of primitive events. This information is presented to the user through the **Terminal** program. In response, the user controls the behavior of the MS through the **Console** program. Thus the **Console/Terminal** pair forms the user interaction component of the system. This section explains the use of these two programs.

### A.1 Starting the System

The MS has two types of output: the response of the system to user commands and the notification of the occurrence of events. The system's response to user commands is displayed in the window of the **Console** program while the occurrence of events are displayed in the window of the **Terminal** program.

When the system is being started, the user should open a new window and execute **Terminal**. **Console** should be started in a separate window so that the output of the two programs do not interfere with each other. If the system successfully initializes, then **Console** prints a command prompt, notifying the user that the MS is ready to be used.

## A.2 User Commands

The user controls the MS with the commands that are entered from **Console**. Currently the system accepts the following commands:

**Help, Define, Undefine, Read, Show, Remote, Log, Stop, Ps, Exit**

### A.2.1 Help Command

The help command provides some online help for using the system. If the command is issued without any parameters, then the list of available commands are displayed. In order to get specific help, the user must enter **help** followed by the name of one of the valid commands. Help information is retrieved from a text file which makes it possible for the user to change the contents of the help messages.

### A.2.2 Define/Undefine/Read Commands

The **Define/Undefine** commands are used to add/remove process and event definitions from the keyboard. With these two commands, the user has the means to change the definitions without stopping the system.

The **Read** command is used to read a text file of process/event definitions. The **Read** command must be followed by a file name.

A process/event definition follows the same syntax regardless of whether it is being typed in from the keyboard or being read from a file. The user must make sure that the system sees the definition of a process before seeing the definition of an event that depends on that process. New definitions merge with the current set of definitions unless there is a name conflict (i.e. a new definition name is the same as an the name of an old definition). In this case, the new definition overwrites the old one.

All new event definitions are immediately relayed to the machines they apply to. Upon receiving new definitions, the **Machine** processes update their local definition tables and the system starts monitoring the currently executing processes for these new events.

All events that are undefined with the **Undefine** command are also relayed to the machines on which they are currently being referred to. This allows the system to suppress the occurrence of events that are no longer needed in order to reduce the monitoring load.

When the MS is tracing the execution of a program which rapidly generates monitoring messages, then a define/undefine will not be immediately effective due to the queuing of messages.

### A.2.3 Show Command

The **Show** command is used to query the system about various types of information. The **Show** command accepts one or two parameters:

**show process** [**<process-name>**]: With the optional **<process-name>** parameter, this command displays the definition of the named process. Without a process name, the definitions of all processes are shown.

**show event** [**<event-name>**]: With the optional **<event-name>** parameter, this command displays the definition of the named event. Without an event name, the definitions of all events are shown.

**show support** [**<process-name>**]: With the optional **<process-name>** parameter, this command lists the names of all events that depend on the named process. Without a process name, the dependency list is given for all currently defined process names.

**show machine** [**<machine-name>**]: With the optional **<machine-name>** parameter, this command displays the connection information about the specified machine. Without a machine name, this information is printed for all currently connected machines.

Part of the information displayed after the "show machine" command is a simple monitoring-load statistic that the system keeps track of. The monitoring-load for a machine is the highest number of messages that has ever accumulated in the message queue of that machine. Since the rate with which monitoring messages are generated is faster than the rate they are transferred, each machine typically ends up with a back-log of messages. The machine load is meant to be used in determining an approximate activity rate beyond which the monitoring system cannot keep up with.

## A.2.4 Remote Command

The **Remote** command is mainly a debugging tool but it was not removed from the program since it reveals some interesting state information:

**remote <machine-name> event:** The named machine displays its set of local event definitions.

**remote <machine-name> process:** The named machine displays its set of process records and the associated lookup vectors.

If the event/process parameters are left out, then the **Machine** process displays both the event definitions, process records, and the lookup vectors. It should be noted that **Machine** processes do not have their own display window and thus the output comes from the window of one of the monitored processes on the named machine.

## A.2.5 Log Command

The **Log** command controls the way monitoring data is captured by the **Terminal** process. All system output goes to the **Terminal** process window by default. However, the user has the means to turn off the screen display and forward the messages to a file.

**Log on:** Turns on screen display of **Terminal**

**Log off:** Turns off screen display of **Terminal**

**Log on <file-name>:** Directs **Terminal** to save the monitoring information to the named file. The data is appended to <file-name> if it already exists.

**Log off <file-name>:** Directs **Terminal** to stop saving the monitoring information to the named file. The file can be reopened with the “Log on <file-name>” command.

**Log erase <file-name>:** Erases the specified log file.

### A.2.6 Stop Command

The **Stop** command is used to terminate **Machine** processes. One of the goals of this project was to establish a loose connection between the MS and the monitored application so that an MS exit (whether fatal error or intentional quit) does not effect the execution of the monitored application. Thus, when a **Machine** process is stopped, the monitored application simply continues its execution without generating any monitoring data.

**stop <machine-name>:** If a machine name is given, then the **Machine** process of that machine is stopped. Without a machine name, all **Machine** processes are stopped.

### A.2.7 Ps Command

The **ps** command is an interface to the UNIX **ps** command. The **ps** command along with the

parameters the user provides is turned into a shell command and executed. The MS makes no interpretation on the command string.

### A.2.8 Exit Command

The Exit command stops the MS. Prior to exiting, all Machine processes are terminated and the **Terminal** and **Central** process are removed from the system. An alternative way to exit is to press Ctrl-D which leaves **Terminal** running. If multiple monitoring sessions will be made, then exiting with Ctrl-D saves the user the extra step of starting **Terminal** in a separate window each time the MS is started.



## Appendix B

# Interprocess Communication Mechanisms

This appendix is an overview of the Interprocess Communication (IPC) methods available on UNIX systems. The Monitoring System (MS) utilizes each one of these methods in constructing a network for collecting information. In addition, a substantial part of the Monitoring Programming Interface (MPI) is the implementation of a C++ layer on top of these IPC mechanisms.

### B.1 Files

File IO in UNIX is descriptor based. A descriptor is an index to a system table and forms a reference to disk file. Reading from a file is receiving a stream of characters and writing to a file is sending a stream of characters. The sent/received data has no type or structure; the interpretation of the data is entirely application specific. File IO is handled primarily through four system calls:

**open()** Opens a new file descriptor and connects the calling process with a disk file

**read()** Reads from the file

**write()** Writes to the file

**close()** Close the descriptor and disconnect the calling process from the file

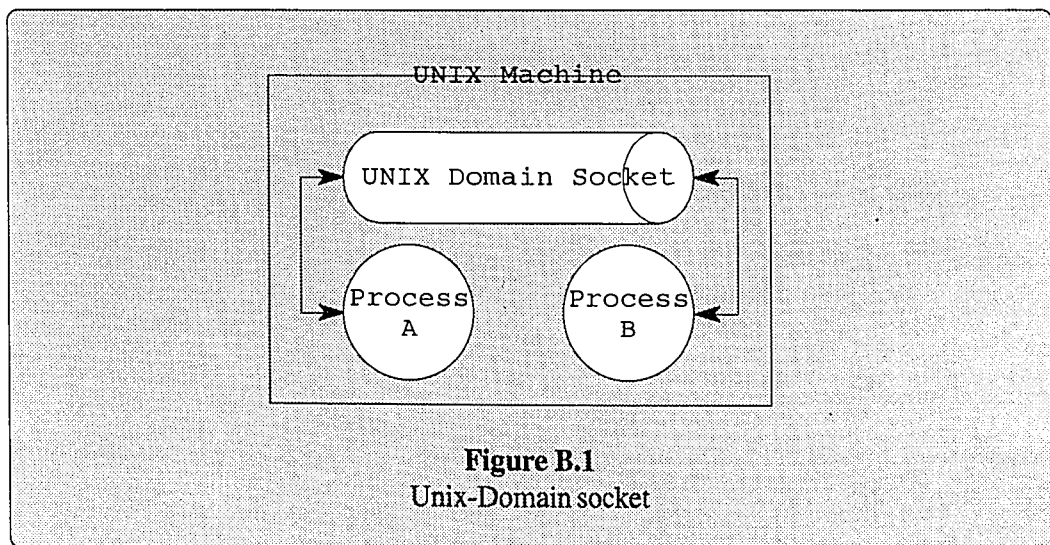
The semantics of file usage in UNIX has been generalized into using other forms of IPC.

## B.2 Pipes

A pipe is a communication channel between two processes. Like files, the use of pipes is descriptor based and the same `read()/write()` system calls are used to receive/send information. Each pipe has one “write” descriptor and one “read” descriptor. Pipe IO is half-duplex; therefore, if two processes need to communicate in both ways, they need to setup two pipes. Unlike files, pipes are nameless entities. The main restriction with pipes is that they may be formed only between related processes (that is, processes which can be associated with each other through a sequence of `fork()` and `exec()` calls).

## B.3 UNIX-Domain Sockets

A UNIX-domain socket is very similar to a pipe; in fact, most of their implementation share the same kernel code. However, UNIX-domain sockets have two improvements over pipes:



[1] A **UNIX-domain socket is a named entity**; when a process asks the system to create/open a UNIX-domain socket, that process provides a socket name.

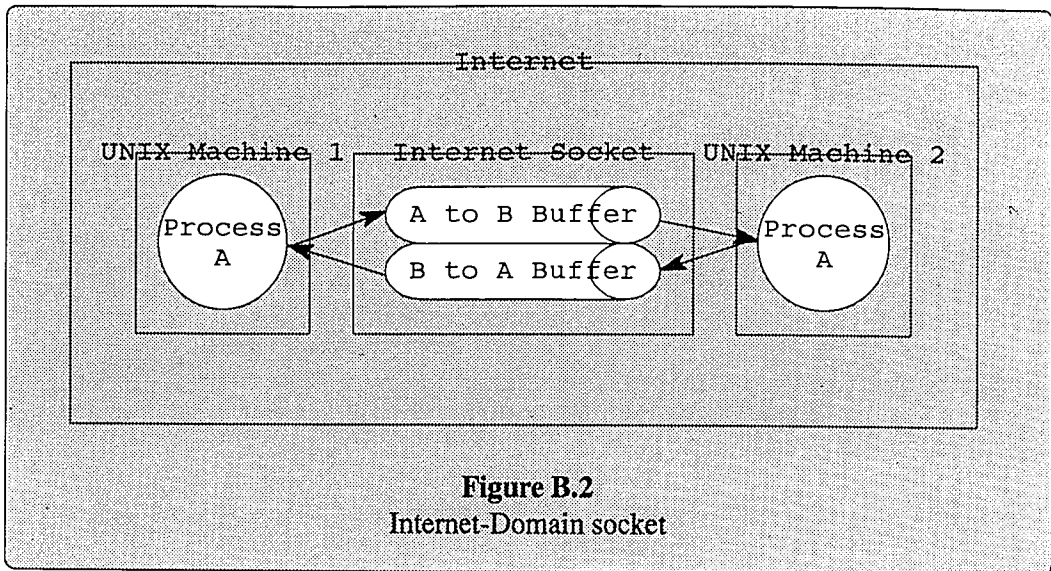
[2] A **UNIX-domain socket can be formed between any two processes on the same machine**. This is possible since both processes have reference to the same socket name file.

UNIX-domain sockets are full-duplex connections if the communicating processes can coordinate their socket use in a one-to-one manner; otherwise, a process may end up reading its own message. Once a UNIX-domain socket is created, the same read()/write() calls can be used to receive/send information. The UNIX-domain socket structure is outline in figure B.1.

## B.4 Internet-Domain Sockets

Like UNIX-sockets, Internet-sockets allow any two unrelated processes to communicate through a communication channel. The main difference is that while the processes using UNIX-sockets must be on the same machine, the Internet-socket processes may run on separate machines. Another difference is that Internet-sockets are full-duplex channels regardless of the order of use.

Once an Internet-socket is created, its use is very similar to files and pipes. The same read()/write() calls can be used to receive/send information over an Internet-socket. The Internet-socket structure is outline in figure B.2.



## B.5 Message Queues

In principle, message queues are similar to pipes and sockets. They are FIFO structures where information is sent and received by sets of processes. However, message queues provide four major improvements over pipes and sockets:

- [1] **Message queues preserve message boundaries**; pipes and sockets do not.
- [2] **Message queues support out-of-band messages**; this is available with pipes and sockets only in a limited way.
- [3] **Message queues can be multiplexed to simultaneously carry unrelated streams of messages**; pipes and sockets cannot.
- [4] **Message queues are persistent structures kept in the kernel space**; pipes and

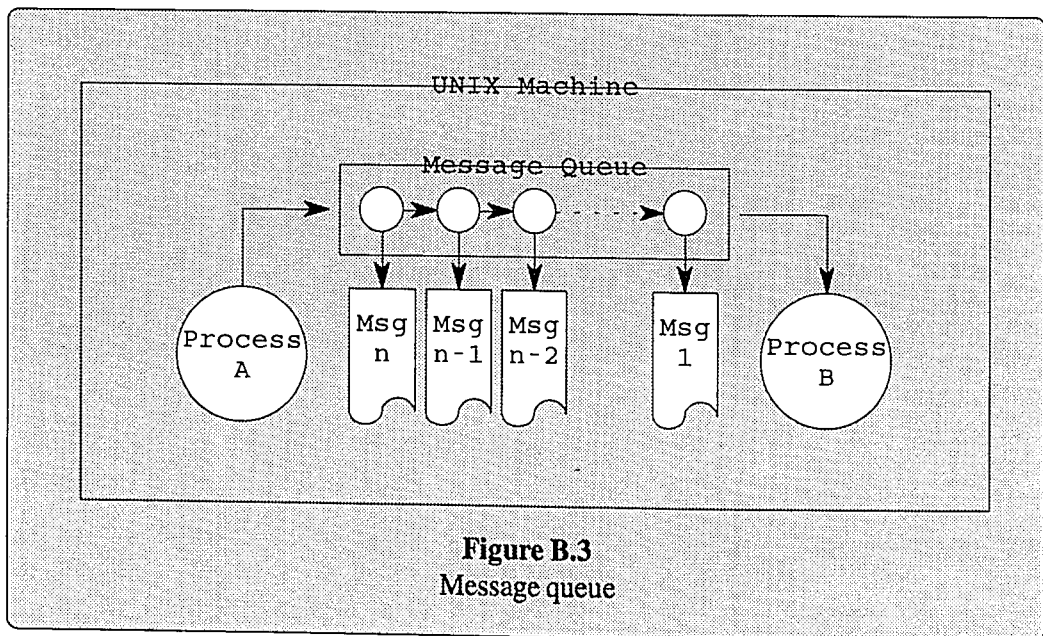
socket are temporary structures outside of the kernel space.

The message queue structure is outline in figure B.3. The diagram implies that system queues are implemented as generic linked lists in the kernel space.

## B.6 Semaphores

Although semaphores are synchronization tools, they are also considered IPC mechanisms for two reasons:

[1] **Synchronization itself is a limited form of communication:** this makes the participating processes become aware of each others' states.

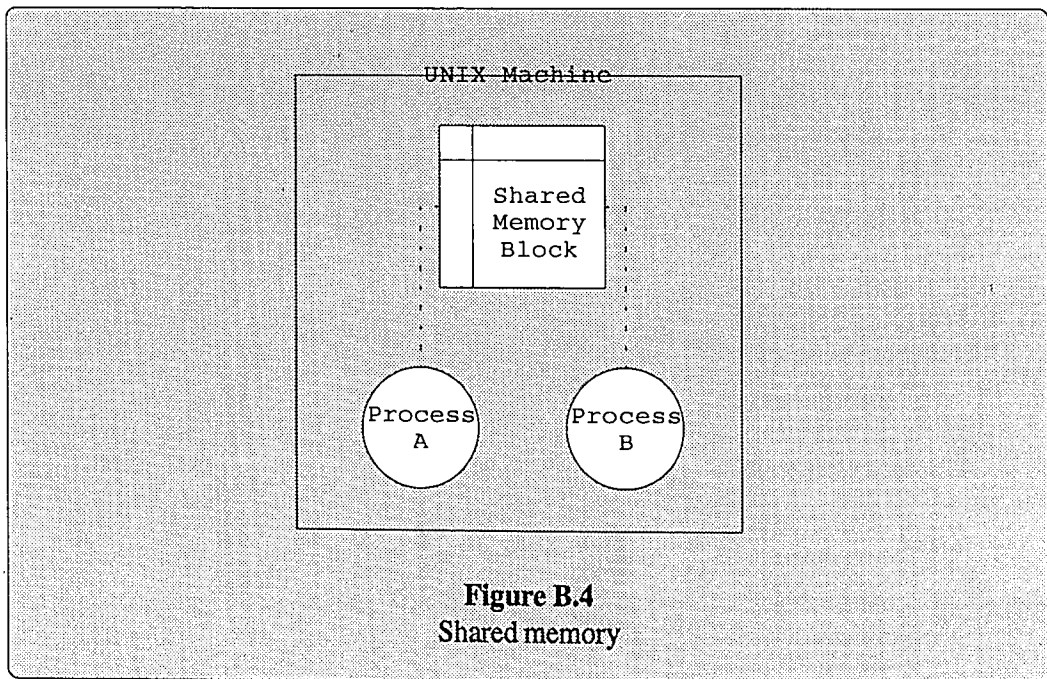


[2] **Semaphores are typically used along with other forms of IPC:** For example, they are used with files and shared memory to establish an order of use.

The AIX semaphore implementation is more general than the usual binary and counting semaphores. AIX uses semaphore sets where a semaphore set is very similar to a set of counting semaphores. The benefit of gathering several counting semaphores into a semaphore set is that the process can issue several semaphore operations and all of those operations are completed atomically.

## B.7 Shared Memory

A UNIX user process is always enclosed in a specific segment of the total address space. The



system enforces the restriction that all the memory access performed by the process is within the limits of this segment. With shared memory, a process can ask the system to create a block of memory outside its address space. Several processes may share this memory block for reading and writing. Shared memory is the most general form of IPC available to UNIX processes since it can be used in place of all the other IPC mechanisms except for Internet-sockets.

Once a shared memory block is created, its access is no different than that of local memory; there are no specific function calls to write to it or to read from it. The use of shared memory is outlined in figure B.4.

Figure A.4

## Appendix C

# Characteristics of C++

Although the MS was written in C, the MPI was implemented in C++. An object oriented view of the programming elements allows a more natural representation of the events. It also leaves no holes in the system where events can occur and yet not be monitored. This appendix discusses those aspects of the C++ language that are relevant in the design of an object oriented MPI.

### C.1 The C++ Class Construct

The C language provides the **structure** construct with which the programmer can group together existing data types. The resulting new type is then treated as a single entity. Once new data types are defined, the programmer then writes the code which manipulates the new data type.

The C++ **class** is a generalization of the C structure and it takes the data abstraction issue two steps further:

[1] A class declaration binds data with the code which manipulates the data. In order to associate a new data type with the code which manipulates that data type, the programmer includes the function prototypes in the class declaration. Thus, in addition to its data fields, a class has function fields.



[2] The class construct establishes access privileges to its members so some parts of the class can be hidden from its outside.

The data fields of a class are referred to as **member data** and the function fields of a class are referred to as **member functions**. If a member (either member data or member function) is not visible from outside of the class, then it is called a **private member**. If a member is visible from outside of the class, then it is called a **public member**. The issues of visibility and member type lead to four combinations:

[a] **Public member data**: The meaning and use of public data members are identical to the use of the fields of a normal C structure.

[b] **Private member data**: These fields cannot be accessed directly by the program. If indirect access is needed, then the class provides member functions to retrieve their values. This is basis of information hiding in C++.

[c] **Public member functions**: These functions form the interface of the class to the outside world.

[d] **Private member functions**: These functions handle the internal issues of the class that are to be hidden from the outside world.

## C.2 The Constructor/Destructor Functions

Each C++ class has two special member functions. The first one is called the **constructor**

**function** and is automatically invoked with object declarations. As the name implies, the constructor function is responsible for any initialization needed by the object. For example, if a class definition contains a pointer declaration, then the constructor may contain a call to `malloc()` to allocate memory. The call to the constructor is never explicitly seen in the source code; the constructor of an object is automatically called when the object's declaration is encountered in the execution of the program.

The dual of the constructor function is the **destructor function**. The destructor function is responsible for any termination steps that have to be taken before an object is destroyed (i.e. the loss of a local variable as a function ends or the loss of global variables as the program ends). For example, if a class definition contains a pointer declaration, the destructor function would have a call to `free()` so that the dynamically allocated memory by the object is returned back to the system before the object dies.

### C.3 Object Declarations

The declaration of class type variables is similar to ordinary C variable declarations. A class type variable is called an **object** of that class. In C++ terminology, an object is also referred to as an **instantiation** of its class.

Ordinary C variable declarations are of the format "type name;" and this is also one way of declaring C++ objects. For example, if a class called "Complex" was defined for complex numbers, an object of type "Complex" would be declared as "Complex C;".

However, the C-style declaration format is not sufficient if the class contains constructor functions with parameters. Since there is never an explicit call to the constructor of an object, it was not obvious where these parameters would actually be specified. With C++, the parameters to the constructor function are given at the point of declaration.

An example should clarify this C++ feature. Since complex numbers have a real part and an imaginary part, complex numbers are initialized with two values: a real number and an imaginary number. Therefore, the constructor for the Complex number class would be defined with two parameters. Since the programmer cannot call the constructor, he/she specifies these two parameters at the point of declaration with the statement "Complex X(1,2);". When the object X goes into scope, its constructor is automatically called and the constructor initializes the real and the imaginary parts of X to 1 and 2, respectively.

## C.4 Operator Overloading

Operator overloading already exists in C. When "a+b" is evaluated, the code that is invoked for the addition is different if a and b are integers than if a and b are floating point numbers. In a context sensitive way, the compiler understands which meaning of the + operation the programmer is implying.

While C stops short of offering overloading to programmer defined data types, C++ makes it possible. The programmer can write special member functions in the definition of a class which are named by the overloaded operators. When that operator is used within the context of an object derived from that class, the overloaded function is executed. For example, with

the above complex number definition, the + operator can be overloaded for the complex number class so that the expression "a = b+c;" invokes the complex number addition function to add two complex numbers.

## C.5 Monitoring Programming Interface

As described in Chapter 3, the MPI has seven modules. This final section explains how an object oriented design has helped the construction of these modules.

Each IPC mechanism of the MPI is a class definition. The C level IPC details (such as descriptor values and file names) are all hidden from the user by making them private members of the classes. For this reason, these modules can be used only through the set of public function calls provided by the MPI. These function calls all contain monitoring statements which generate information about the use of the modules.

The Integer module is another C++ class. By overloading every integer operator of the C++ language (i.e. +, +=, ++, <, etc) Integer objects are made to function like ordinary integer values. As noted in chapter 3, however, the only exception to this is the retrieval of the value of an integer object when the object is not associated with any one of the overloaded operators.

When an object is being declared in the program, the MPI insists that the programmer supplies an object name and a declaration function name for the new object. By keeping track of these two string names, the MPI has a simple way of associating programming elements with

the references made in event definitions.

## Biography

**Name:** Ali Erkan

**Birth:** February 19, 1967 (Istanbul, Turkey)

**Mother:** Selma Erkan

**Father:** Metin Erkan

**Undergraduate Degree:** Lehigh University, BS in Computer Engineering, January 1991

**Teaching Assistance:** Fall 1992 - Spring 1995 (CSc 17, CSc 303)

**Research Assistance:** Summer 1993 (Distributed Systems, Dr. Madalene Spezialetti)

**Teaching Award:** Arthur E. Humprey Teaching Assistant Award, 1993-1994

**Academic Award:** Beta Pi Chapter of Phi Beta Delta Honor Society for International Scholars

**END OF**

**TITLE**