BRANDON KOHN
Practical Considerations in Monitoring and Steering of Distributed Computations
(Under the direction of EILEEN KRAEMER)

Tools that assist users in understanding the complex workings of distributed systems are an important aid to programmers and scientists. Interactive monitoring and steering tools can provide users with insight into their computations. This thesis describes the PathFinder interactive monitoring and steering system and the problems involved in making the system a viable solution for monitoring and steering distributed applications. These problems include MPI support, an Agent Wizard, and a performance analysis of two agent subsystems and their associated milieu, one implemented in Perl and the other in Java.

INDEX WORDS: Agents, Monitoring, Computational steering, Distributed computations.

PRACTICAL CONSIDERATIONS IN MONITORING AND STEERING OF

DISTRIBUTED COMPUTATIONS

by

BRANDON L. KOHN

B.S., The University of Georgia, 1995

M.S., The University of Georgia, 1998

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial

Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2001

© 2001

Brandon L. Kohn

All Rights Reserved

PRACTICAL CONSIDERATIONS IN MONITORING AND STEERING OF DISTRIBUTED COMPUTATIONS

by

BRANDON L. KOHN

Approved:

Major Professor: Eileen Kraemer

Committee: David Lowenthal
Don Potter

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2001

TABLE OF CONTENTS

Page

CHAPTER

1 INTRODUCTION 1

2 MODEL OF COMPUTATION 9

3 AGENTS 14

4 THE AGENT WIZARD 22

5 ADDITION OF THE MPI LIBRARY 27

6 COMPARISON OF THE PERL AND JAVA AGENT SYSTEMS 34

7 DISCUSSION AND CONCLUSIONS 45

BIBLIOGRAPHY 50

CHAPTER 1

INTRODUCTION

As the size and speed of networks have increased, the ability to execute large, long running distributed computations has become increasingly prevalent. The need has arisen for tools that assist in designing and managing these systems, and to promote understanding of the structure and behavior of the computations. PathFinder is a system that uses monitoring and steering techniques as a means to manage, maintain, and understand distributed computations. PathFinder supports these activities through a system for specifying attribute monitoring and steering commands with *agents*.

The work outlined in this thesis details three problems that in some way limited the usability of the PathFinder system. The first problem was the difficulty for non-programmers to create agents. Because agents are actually mobile code, creation of even the most basic agent required that the user know how to program in the language that the agents were implemented. This problem was solved by the creation of an Agent Wizard system designed to automate agent creation from a high level specification. The second problem was that the initial version of PathFinder supported only PVM [1] applications. Many applications use the MPI [2] (Message Passing Interface) as a communication middleware. The problem was addressed by adding support for the MPI library furthering the base of distributed applications compatible with the PathFinder system. The third problem was to investigate performance concerns about the original agent system implemented in PathFinder that used the Perl [3] language. A new agent system

was implemented using the Java language and then tested against the Perl system in order to determine the more efficient system.

Exploratory Visualization with the PathFinder System

Two methods that support distributed program understanding and control are interactive monitoring and steering. Interactive monitoring of a program's state and behavior allows users to learn about how a computation is progressing. Interactive steering allows the user to change the values of variables in the computation at run-time. Using these methods, users may perform steering experiments and study the visualized feedback or simply observe the execution characteristics of a distributed application in order to understand how the application works. We call this methodology *Exploratory Visualization* [4]. An example of this method in use is illustrated as follows. Imagine a distributed application whose function is to simulate a system for routing the movements of abstract entities. Various attributes in the application control how the entities in the simulation are routed. A user presented with a graph visualization of the routing system and current positions of entities could then tune the parameters of the routing junctures and observe the effects of these changes as they occur. Eventually, through experimentation, the user would understand how the various parameters of the application affect the movements of the entities in the simulation.

In order to realize a system that allows *Exploratory Visualization* of any distributed application, tools must be designed to manage coherent monitoring and steering of large and long-running distributed computations in dynamic, heterogeneous environments. Features of these systems should include:

- The ability to dynamically alter what data is collected, as well as where and when it is collected. This feature supports the exploration process, and has the ability to filter the data collected to a manageable level.
- The ability to generate consistent views of the distributed computation, representing some state of the computation that either occurred or logically could have occurred, based on the observed events. Views that are inconsistent can mislead the viewer, and should thus be avoided.
- Consistent steering, in which changes to the running computation are applied in a coordinated fashion that maintains the correctness of the computation.

Related Work

Traditional debugging tools provide the ability to both track and alter program variables, through the use of breakpoints. However, the overhead imposed on the underlying computation is excessive for large and long-running computations. Further, the ability to apply coordinated changes may not be present. Traditional program visualization systems such as BALSA [5], Tango [6], and Polka [7] require that the program source code be instrumented to produce a fixed stream of "event records" upon which visualizations are based. Among these, Polka is capable of producing consistent views of distributed computations, given a definition of the ordering relationships, control, communication, process synchronization among events, and an event stream containing these events. However, the volume of data produced by these approaches can be prohibitive. Further, these approaches lack flexibility in that they require source code alteration, recompilation, and re-execution to alter the monitoring state.

Systems such as CUMULVS [8] and the Progress Toolkit for Falcon [9] permit run-time alteration of the monitoring state and support interactive steering. However, their support for consistency in collected views and steering changes is limited, either to a particular program type (CUMULVS is designed for programs with a single main simulation loop) or the types of changes that may be applied (Falcon supports only local steering, not globally consistent steering.)

JEcho [10] is a high performance communication middle-ware implemented in Java that provides a mechanism for event interaction between distributed processes. JEcho provides these services via a concept called *event channels.* Event channels are constructs that link an event at some producer in the distributed application to a set of observers in the consumer space. The system works by having consumers subscribe to various event channels. When an event occurs the appropriate channel is notified and all consumers subscribed to the channel then execute an event handler in response. JEcho uses components for executing event handlers that are similar to agents and the agent *milieu* called *modulators* and the *MOE* (modulator operating environment). Such a system performs well in providing users data filtering and steering capabilities with low overhead, but lacks the flexibility provided by a system with autonomous agents.

DOVE [11] is a system for visualizing distributed applications that uses CORBA agents to gather process state information and a browser mechanism to create views of the data. The application is prepared such that variables in the computation have their values placed into a data repository called the MIB. Agents interact with an end user and the MIB in order to supply data for visualizations on the state of the distributed computation. DOVE also supports an event model where agents can react to alarms from

other agents or to state changes within the application. DOVE does not currently support user steering of the distributed application; therefore agent operations are limited in scope to filtering or transform operations on the data to be visualized.

The goal of our work is to provide a flexible and powerful system for the monitoring and steering of distributed computations through which users may gain understanding of the workings of a computation without the need to view actual code. *Exploratory Visualization* is a developmental approach to the understanding of large, long running, distributed computations. The methods in *Exploratory Visualization* form the basis of operation for the PathFinder system [12]. PathFinder is a system that can be installed into a distributed application giving the user the ability to monitor and steer any selected variable within the application. PathFinder makes use of *agents* as a mechanism for providing flexibility and power in the specification and deployment of monitoring and steering commands. *Agents* are defined as processes that are situated in some environment and are capable of autonomous action within the *environment* in order to meet their design objective [13]. Implementation of the agents and their operating environment, or *milieu*, requires the use of a programming language that is both platform independent and interpretable at run-time. The initial version of PathFinder made use of agents written in the Perl language. The use of Perl allows the agents to be specified as strings that are interpreted at run time by a Perl interpreter embedded within the native code of the PathFinder system. We define native code as code that is optimized to run on the architecture of the specific system (usually C or C++). An agent framework for monitoring and steering has a number of advantages:

1) **Responsiveness** – agents are able to react quickly to conditions at the application processes because they reside at the same location as the process and may have multiple methods for dealing with a variety of events.
2) **Customization** – since agents may be encoded at run-time they can make use of application-specific information, permitting efficient solutions.
3) **Mobility** – the ability to migrate between processes makes agents well suited to distributed applications in which properties are not necessarily bound to a process.
4) **Transience** – the ability to deploy agents at runtime helps to minimize the cost of monitoring while the user is not visualizing the computation.

This approach provides a framework that guarantees consistent views of the computation, as well as consistent steering changes. Within this framework, two approaches have been implemented: a query-driven approach, and an agent-based approach. In the query-driven approach, users specify queries about the computation. These queries are shipped to wrapper libraries at local processes, where the queries are resolved and responses are returned in a format that permits the assembly of consistent global snapshots. The agent implementation provides greater functionality and responsiveness, permitting multiple responses to various events and conditions to be specified and placing the "reactions" to these conditions closer to the computation and time of the event, at the individual processes where the event or condition is detected.

Problems with PathFinder

PathFinder was designed with the objective of being used for monitoring and steering by a large user base including people who are not trained as programmers. This functionality is provided in part by the agent system that allows users to specify reactions

to events within the application, and by allowing users to specify, filter, and aggregate the data they view from the application. In the initial version of PathFinder users could only get this level of functionality from PathFinder if they were able to implement the agents by hand using the Perl language. This posed a problem because one of the goals of the Exploratory Visualization project is to allow non-programmers to interact with a distributed computation and to be able to specify the agents to perform various monitoring and steering tasks. In order to solve this problem, an Agent Wizard module was built that helps novice and advanced users alike in designing, generating, and deploying agents.

Another problem with the initial version of PathFinder was that it only supported applications using the PVM message-passing library. Many distributed applications are written using PVM. However, other message passing libraries exist, and in order for PathFinder to be useful in applications using these different libraries it must be modified to accommodate them. Support for MPI was added to the PathFinder system in order to further the base of compatible applications.

The third problem that is addressed in this work deals with performance issues in the PathFinder system. The conclusions from [14] were that the Perl agent system in PathFinder was responsible for a significant amount of the overhead cost of monitoring the distributed application. Because Perl is not known for having good performance properties it was decided that the agent system should be implemented in another language and tested against the original Perl in order to determine which performs better. As a result of these considerations a new agent framework and *milieu* were implemented using the Java language.

In the next chapter we present a short description of PathFinder's model of the computation, and a high-level description of the Exploratory Visualization approach and its embodiment in the PathFinder system. Chapter 3 describes the agent system, including the agents and the *milieu* in which they operate. Chapter 4 contains a description of the Agent Wizard that is used to generate agents. Chapter 5 contains a description of the version of the PathFinder system that supports the MPI message-passing library. Chapter 6 gives details about the Java version of the agent system and a set of experiments used to measure its performance versus that of the Perl system. Chapter 7 gives a set of conclusions about the work as a whole and some directions that the project will be taking in the future.

CHAPTER 2

MODEL OF COMPUTATION

A distributed computation is a collection of processes in which each process's state changes as a result of *events* – local calculations and message passing activity. The events are partially ordered by Lamport's *happened-before* relation [15]. A subset of a process's state is accessible to PathFinder in the form of readable and/or writable attributes. A *local snapshot* is simply a listing of the readable attributes and their values at an instant in time at a process. In addition, a local snapshot may contain derived attributes and their values. A *consistent global snapshot* is a set of local snapshots such that all processes are represented and none of the local snapshots happened-before any of the other local snapshots [16].

In our research we have focused on a specific class of distributed computation in which the communication patterns are sufficiently structured so that the overall computation can be abstracted to an interleaving of atomic state changes involving one or more processes – by analogy with databases we call such changes *transactions*. A transaction can be thought of as a logically complete block in the code, in which the block can span across processes. Viewing the computation in terms of transactions simplifies the assembly of local snapshots into global snapshots. Transactions also help identify places in the computation that are well suited for visualization and steering, i.e., between logical actions in the application.

PathFinder's Architecture

The PathFinder system (Figure 1) consists of three operational components: Interaction Managers (IMs), a Snapshot Manager (SM), and a User Interface (UI) and visualization system. Process state information is generally transmitted between the components in the form of snapshots. Information also may propagate between the components via agent actions. These "actions" include monitoring and steering operations as well as migration operations in which an entire agent along with its local data space can be transferred between components. The SM merges local snapshots from the IMs into consistent global snapshots that are forwarded to the UI. At the UI, visualizations are created or updated based on these snapshots. The user interacts with the visualizations and user interface panels to direct the monitoring and visualization process and to interactively steer the application program via agents. The UI sends agents to the SM, which coordinates their distribution to the IMs where they are executed.

An Interaction Manager is the interface between the application and the PathFinder system, providing the agent *milieu* access to the attributes and events of the process. To facilitate the monitoring of messages between the application processes, the IM encapsulates the message-passing library used by the process. At present, PathFinder may use either of two communication libraries: PVM or MPI. The IM is also responsible for interfacing *process events* with the agent *milieu. Process events* are events that are common to all applications and include end of transaction (*eot*), send, and receive events. When these events occur, the IM executes the agent *milieu* so that any installed agents can react to these events.Modules that utilize the basic framework of an attribute database and notification of application and communication events provide the IM's functionality.

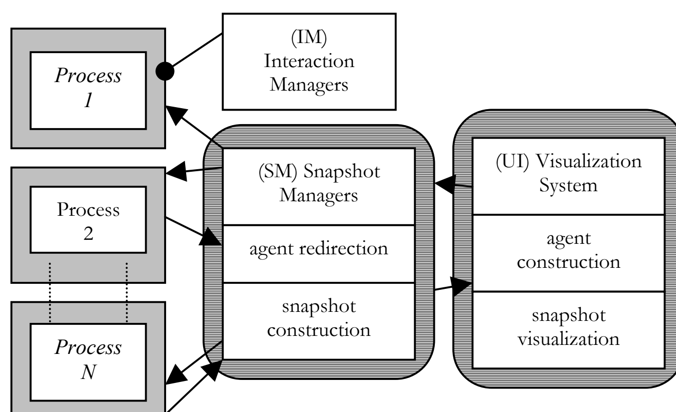

Figure 1. PathFinder System Architecture

The two principal modules used are Transaction Labeling Modules (TLMs) and Attribute Modules (AMs). The TLM is responsible for the collection of the event ordering information the SM needs for the construction of global snapshots. The AM coordinates non-local access to the attributes of a process for either monitoring or steering. In this paper, we focus on an AM that has been implemented to use agents. The agent module, seen in Figure 2, is described in chapter 3.

The Snapshot Manager also has a TLM and an AM. This TLM incorporates information from the TLMs at the IM and uses that data to determine transaction membership and ordering. Based on ordering information from the TLM, the AM merges local snapshots into a global snapshot and may also manipulate the global snapshot by adding, combining, deleting, or updating data fields. The AM is also responsible for interpreting and issuing commands to carry out the user's actions at the processes. The actions can be specifications of what data to collect and how to collect it, or instructions for steering the execution of the application. These steering interactions involve making on-the-fly changes to program parameters, including both application parameters and performance parameters. Both monitoring and steering instructions are instrumented through the use of agents. [17]

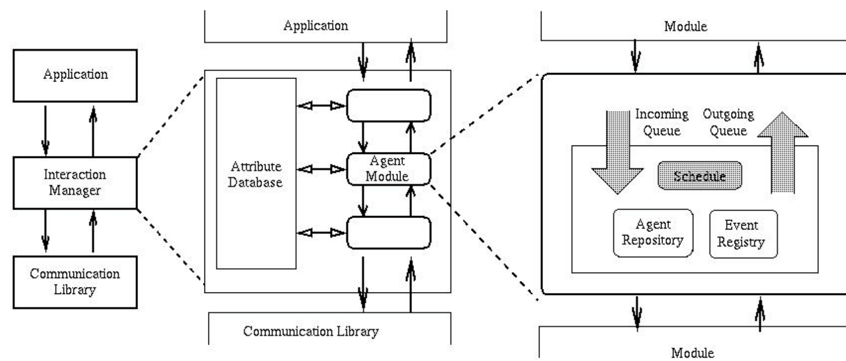

Figure 2. The Agent Milieu

CHAPTER 3

AGENTS

Agents consist of data and procedural responses that are associated with either application events or with events occurring within the agent *milieu*. The data comprises those characteristics used to uniquely identify the agent, such as names and id numbers, as well as local data that is used in performance of the agent's functions. The responses that an agent has to the various events are called *event handlers*. Agents operate by registering their event handlers for given events. When an event occurs, the *milieu* schedules the handlers registered for that event, and each handler is executed sequentially.

The functions that an agent can perform in response to an event are built up from a set of primitive agent operations. These primitive operations include:

Subscribe/Unsubscribe: The subscription operation is used to specify those variables within the computation that are to be included in the snapshots sent from each process. The unsubscribe operation removes the variable from those being written into snapshots.

Register/Unregister: When an agent registers an event handler with the *milieu*, the *milieu* records the identity of this event handler in its event registry. On subsequent occurrences of that event, the handler is invoked. When an agent unregisters a handler for an event, the *milieu* removes the handler from its event registry.

Read/Write Attribute: Agents may access process attributes through the *milieu*. For example, a simple, local steering agent might contain an *eot* handler that reads one or

more process attributes, performs a calculation, and then writes a new value to another process attribute.

Signal Event: The ability to signal an event permits agents to coordinate. For example, a user may wish to create a set of agents that begin monitoring only after a program attribute takes on a certain value. One way to implement this is to place a conditional clause in the *eot* handler for each agent. However, this requires that the *eot* handler of every agent in the set be invoked at every *eot* event. A more efficient approach is to have one agent initially register for *eot*, while the other agents register only for a user-defined event, “attribute-ready”. Now, only one agent is invoked at the *eot* while the condition remains false. When the condition becomes true, the single agent senses this, and responds by signaling the “attribute-ready” event. The set of agents that sense the attribute-ready event are now ‘awakened’. In the case that a one-time execution of a monitoring task is desired, the awakened agents respond by performing their monitoring tasks. In the case that continuous monitoring is desired, the agents also register for the *eot* event. A similar scheme could be employed to turn off these agents when the value of the controlling attribute changes, by having the agents register for an “attribute-not-ready’ event which they would respond to by unregistering for the *eot* event.

Read/Write Agent Data: Agents have the ability to access data contained in other agents, providing the ability to eliminate redundant calculation among agents. For example, in a molecular modeling program, a process may contain an array of the atoms in the molecule. One agent may report the identity and location of all carbon atoms with bond energies above a certain threshold. Another agent may report the identity of all carbon atoms with bond lengths exceeding a particular range. Each of these agents must

first traverse the entire array and select out the carbon atoms. To avoid this, we could create a single agent that traverses the array, selects out the carbon atoms, and records their identities in its local data. The other agents could then access this data directly from the "pre-processing" agent, and avoid the redundant computation.

Create New Agents: Agents have the ability to create new agents. This ability provides an alternative solution to the "conditional monitoring" scenario described above. In this approach, a single agent subscribes to the *eot* event. When the condition becomes true, the handler creates a new agent containing the desired monitoring code and submits it to the *milieu* for installation. The *milieu* generates an arrival event for this new agent, and schedules the execution of the new agent's arrival handler. In the arrival handler, the new agent subscribes to the *eot* event. The *milieu* then invokes the newly created monitoring agent on each subsequent end-of-transaction, achieving the desired effect.

Migrate: The ability to migrate to another *milieu* provides agents with the capacity to track phenomena across processes. For example, in the molecular modeling program, it might be of interest to the viewer to track a particular atom as the molecule undergoes changes in conformation. As the atom moves through space, it may be transferred from one processor to another. The agent responsible for tracking this atom would subscribe to the send event, in addition to the usual subscription to the *eot* event. If the type and content of the message sent match the atom of interest, the agent can then move to the destination processor, and continue monitoring and tracking the atom's state.

Leave: When an agent has accomplished all of the functions for which it was designed, it may be removed from the process by specifying the leave operation.

These operations are common to all agents and are implemented in a class called *Agentlib* (Agent Library).

The Agent *Milieu*

The agent *milieu* is the environment in which the agents operate. It consists of a set of structures to house the agents, and a set of methods that provide the mechanisms for event scheduling and agent operation. Specifically, the *milieu* consists of the following components:

- **Incoming queue** – a queue that houses newly arrived agents at the IM.
- **Outgoing queue** – a queue that holds agents that are leaving the IM.
- **Event Registry** – a hash table that maps event names to a list of agent event handlers that are registered to execute when the event happens.
- **Schedule** – a list of agent event handlers that have been scheduled to execute.
- **Scheduler** – the entity responsible for accessing or modifying the schedule, processing all incoming agent's arrival handlers, and executing the appropriate event handlers when their registered events happen.
- **Agent Repository** – a hash table that holds all the agents currently installed at the IM.

The agent *milieu*'s core component is the event scheduler whose purpose is to manage and execute the event handlers of the agents that are registered for the various events. Events may be issued explicitly by agents from within an event handler or by code annotations within the application. These application-initiated events include a set of "stock" events for the common operations that may require agent intervention. These events include end-of-transaction (*eot*), snapshot, send, receive, and message

manipulation calls that occur within the application process. Additional events can also be generated for programmer-defined events, specified through source code annotations.

When an event occurs in the application a special event routine in the IM is executed. The event routines in the IM have the following structure: First, the agent *milieu* is run so that agents may execute their event handlers. Figures 3 and 4 show pseudo-code for the operation of the *milieu* when an application event occurs. The first operation in the *milieu* is to schedule the particular event type with the *Scheduler*. This operation consists of getting the list of event handlers that are registered for the occurring event from the *Event Registry* and then placing each event handler in the list onto the Schedule where it will be executed in turn. Next the *Scheduler* steps through each entry on the *Schedule* and executes it. The *Scheduler* also pulls any newly arrived agents from the *incoming queue* and executes their arrival handlers. When no more agent handlers are scheduled to execute, the *milieu* returns control to the IM where various other tasks associated with the particular event are done. Finally the IM returns, allowing normal operation of the application to resume.

Analysis of Running Time for Agent *Milieu*

The running time for the agent *milieu* has contributions from each of the components described above. Here we present a breakdown analysis of the running time based on the pseudo-code shown in figures 3 and 4. We begin with the run_Event method listed in figure 3. The method contains two blocks of code that iterate over some parameter. In the case of the first block, which schedules the execution of event handlers for the event that is occurring, a hash table lookup is performed on the list of agent event handlers associated with the key corresponding to the current event. This operation is performed

```
Agent_Library.run_Event (Event_Name) Begin

        //First schedule the event and all the registered event handlers
        If events_registry.containsKey(Event_Name)
                list = events_registry.get(Event_Name)//get a list of event handler for the Event
                for i = 0 to list.size Begin
                        EventHandler = list.elementAt(i)//get the ith event handler from the list
                        Scheduler.schedule(Event_Name, EventHandler)//schedule the event pair
                End for
        Else
                Events_registry.put(Event_Name) //if there is no entry for Event_Name create one

        //Next run the Scheduler until it returns false
        boolean step = true
        While step Begin
                step = Scheduler.step() //shown in figure 5.
        End While
End
```

Figure 3. Pseudo-code for the occurrence of an event in the application

```
Scheduler.step () Begin

        While incoming_queue is not empty Begin
                Agent = incoming_queue.dequeue() //pop the first agent off the queue
                Agent.execute_arrival_handler() //run the agents arrival handler
                Agent_Repository.put(Agent) // put the agent into the Agent Repository
        End While

        Current_event = Schedule.next_event() //get the next event/handler pairing

        If Current_event exists
                Current_event.run_agent_handler() //executes the agent handler in Current_event
                Return true
        Else
                Return false
End
```

Figure 4. Pseudo-code for the operation of the Scheduler

in constant time because each event in the system is unique and will be hashed in the table without collision. The second portion of the block iterates over the list of event handlers and schedules each to execute. We call the quantity that designates the number of agent handlers registered for a given event the *event load* or e_L. The event load may vary between none and all of the agents currently installed at the IM. For a worst case scenario we assume that e_L equals n, the number of agents installed at the IM. The next block makes a call to the Scheduler's step method, which executes the handlers that were placed on the Schedule in the previous block. In the worst case, the step method will run n times. This gives us equation 1 for the total execution time, $T(n)$, for n agents:

$$T(n) = n + n * O(step) \qquad \textbf{(1)}$$

Now we calculate *O(step)*. The step function, outlined in figure 4, performs two functions. The first timethe stepfunction is run for a given event, agents may be waiting to be recognized and installed into the IM. This occursonly once for any event because the code is run sequentially (insertion of new agents cannot preempt the execution of an event in the current implementation.) Assuming that m agents are waiting to be installed when any event occurs, we must add the time to process the m agents as well as the time to execute their arrival handlers. This amount can be extracted from the $n*O(step)$ term because it occurs only on the first of the n iterations. We designate the amount of time it takes for agent i to execute its arrival handler as $T_{ah}(i)$. The new equation has the form:

$$T(n) = \sum_{i=1}^{m} T_{ah}(i) + n * O(step) + n \qquad \textbf{(2)}$$

The remainder of the time in *O(step)* consists of getting each event handler from the Schedule (a constant time operation) and executing it. We designate the execution

time for the ith agent event handler as $T_{eh}(i)$. This gives the following equation for the execution time of an event in the agent *milieu*:

$$T(n) = \sum_{i=1}^{m} T_{ah}(i) + \sum_{i=1}^{n} T_{eh}(i) + 2n \qquad \textbf{(3)}$$

In cases where the agent's arrival and event handlers are run in $O(1)$ time, this equation reduces to:

$$T(n) = \sum_{i=1}^{m} O(1) + \sum_{i=1}^{n} O(1) + 2n \qquad \textbf{(4)}$$

Finally we have:

$$T(n) = O(m + 3n) \qquad \textbf{(5)}$$

From equations **4** and **5** we see that if the agent's arrival and event handlers run in constant time, we can expect the agent *milieu* to run in linear time (with respect to both installed and newly incoming agents). If we relax the running time constraint on the agent handlers to include linear or greater executions times we may expect to see the first two terms in equation 3 dominate the total running time of the agent *milieu.* In general we can expect the agent event h9andlers to run in constant time because most agent operations involve performing some constant time operation only once (monitoring and steering operations for example.)

CHAPTER 4

THE AGENT WIZARD

The *Agent Wizard* was created to help users navigate the process of specifying and generating agents. The wizard consists of a series of panels in which the users enter information or select values that ultimately specify how the agent will be created. Users begin by interacting with the Agent Creation panel (figure 5). This panel is used to supply a name for the agent. The name uniquely identifies the agent, and provides a handle through which other agents may interact with this agent. The interface allows users to specify whether they want to create a template or an instance of an agent. Template agents may be partially defined; instance agents must be fully defined. Instance agents are those agents that are complete and ready to be sent out to work. Template agents can be transformed into instance agents when they are completed, and are included as a convenience to avoid redundant agent creation.

Agent *milieu*x exist at the IM, SM, and UI. Based on the choice of target *milieu*, the wizard provides the appropriate set of choices for stimuli and *milieu* library calls in subsequent panels. Currently, operations are supported at the IM only. Next, the user is presented with a Stimulus-Response panel shown in figure 6 through which the user associates "stimuli" (events) with "responses" (event handlers). On the left is a list of possible stimuli to which the agent can react. To the right is a list of possible responses. The user may add or delete names of event types and event handlers through this panel. The user connects a stimulus to a response by first clicking on a stimulus and then

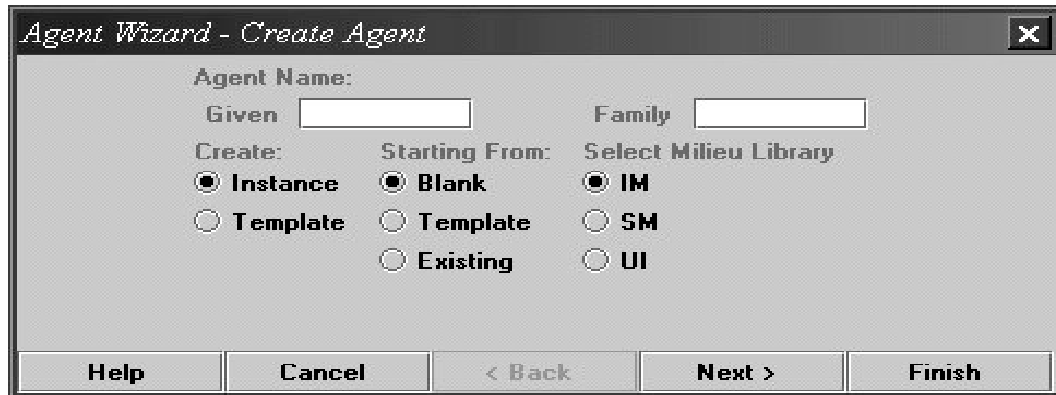

Figure 5. The Agent Creation Panel

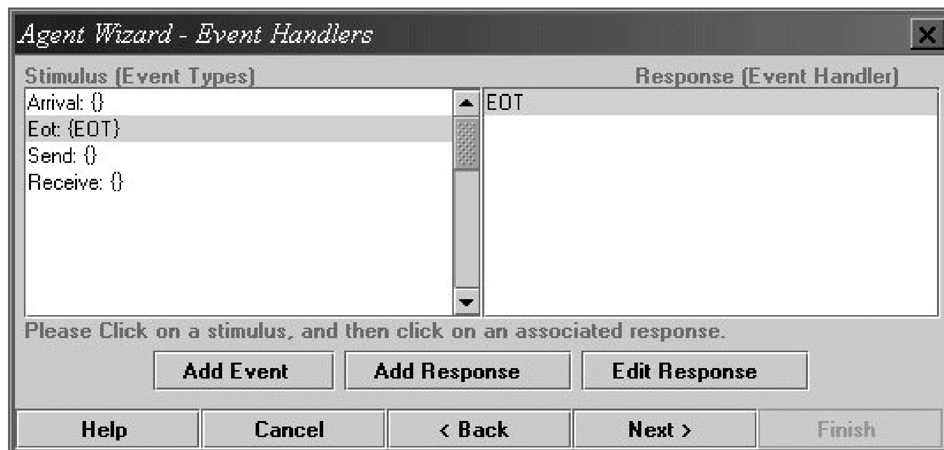

Figure 6. The Stimulus-Response Panel

clicking on a response. Selection of a particular stimulus name causes the associated responses to be highlighted. Such associations are used by the agent creation routine to specify implicit registrations of the selected responses with their corresponding events.

To specify the implementation of an event handler, the user clicks on the "add response" button, and the panel shown in figure 7 appears. Through this panel, the user specifies the sequence of actions to be performed. Actions include *Register*, *Unregister*, *Monitor, Steer, Signal, Migrate, and Leave*. Based on the action selected, appropriate choices appear in the target and value pull-down lists. For example, if the user selects the *Register* action, the target pull-down contains a list of event types and the value pull-down contains a list of event handlers. The order in which the actions appear in the spreadsheet is the order in which they will be executed in the handler. The user may manipulate the order of the actions by moving rows through typical spreadsheet interactions. When all actions have been specified, the user presses "Done" and is returned to the Stimulus-Response panel of figure 6.

After all event handlers have been implemented, and the initial mapping of stimuli to responses specified, the user selects "Next" or "Finish" from the task bar. For an instance agent, the wizard requires the user to provide implementations for all listed event handlers, and to specify at least one stimulus-response connection. The final panel, the Agent Deployment panel, shown in figure 8, is then displayed. The Agent Deployment panel consists of two components, an Agent Control panel and an Agent Editor. The Agent Control panel contains a list of agent objects that are ready for deployment and a special component called a Process Tree. The Process Tree maintains the list of all available processes in the distributed computation and provides a means for

Agent Wizard -hi Handler

Action	Target	Value	Condition
Monitor	attr_x		
Signal	X_Over_100		if(attr_x > 100)
Steer	attr_x	0	if(attr_x > 100)
Leave			

Cancel Done

Figure 7. The Response Handler Panel

Converter Applet/Application

Computations Queries Visualizations Messages Processes Attributes Agent Agent Wizard Java Agent Wizard

Java Agents

File Compile

Processes
0
Basic Agent
1
Basic Agent

Test Agent

TestAgent.java

```
package jmilieu;
import java.io.*;
import attr;
import jc_exec_milieu;


public class TestAgent{
        public static void main(String[] args) {
                String family = "Agent";
                String given = "Test";
                String attribute = "jvalue";
                AgentLib al = new AgentLib();
                //default test
                Agent jag = new Agent(family, given){
                        public void agentArrive(){
                                this.al.register("eot",this, "one");
                                this.al.register(getName()+"_LEAVE",this, "_LEAVE");
                                int smID = jc_exec_milieu.getSMID();
                                jc_exec_milieu.subscribe(smID, "jvalue");
                        }//Arrive

                        public void runEventHandler(String event){
                                if(event.equals("one")){
                                        long attrID = attr.get_attr("jvalue");
                                        attr.DoubleAttr_setValue(attrID,0.0);
                                }//if
                                else if(event.equals("_LEAVE"))
                                        agentLeave();
                        }//runEvent

                        public void agentLeave(){
                                this.al.unregister("eot",this,"one");
                                this.al.unregister(getName()+"_LEAVE",this, "_LEAVE");
                                int smID = jc_exec_milieu.getSMID();
                                jc_exec_milieu.unsubscribe(smID,"jvalue");
                                this.al.leave(this);
                        }//agentLeave
                };
                al.writeAgent("/home/kohn/pathfindernew/agents/java/brandon/"+given+family+".jag"
        }//main
}//TestAgent

```

Load Agent New Agent

1:1 INS

Quit Help

Figure 8. Agent Deployment Panel.

easily deploying and managing the agents at the processes. Through this panel, the user is presented with the code generated by the agent wizard in the Agent Editor. At this point the user may inspect and fine-tune the agent through code annotations in the Agent Editor. The Agent Editor allows users to compile and build agents into agent objects and to load and save the agent code. Agents that are built with the Agent Editor are automatically added to the ready list in the Agent Control panel. Users may also load pre-built agent objects into the ready list. To deploy an agent the user simply drags an agent from the ready list and drops it in the Process Tree at the appropriate node. Agents dropped at the root node of the Process Tree are deployed to all processes listed in the Process Tree. When the agent is installed at the distributed process it sends a message back to the UI informing the Process Tree of its location so that it may be recorded in the proper process node. This functionality is implemented so that both newly created agents from the UI and agents migrating from other processes report upon installation to a new process. This feature allows users to track the current locations of the agents in the distributed process. Users can also use the Process Tree to remove agents from a process by selecting the agent from the Process Tree and hitting the delete key.

CHAPTER 5

ADDITION OF THE MPI LIBRARY

Many distributed applications exist that make use of the MPI (Message-Passing Interface) communication middleware. The issue of adding functionality for MPI to PathFinder arose as a result of the need to use PathFinder for monitoring and steering of a distributed application in simulational physics, implemented using MPI. Both PVM (Parallel Virtual Machine) and MPI have features that make them ideal for various distributed applications. In most applications either library can be used to achieve the same end result. In either case a system for monitoring and steering should be flexible enough to work with any communication middleware. Two problems arise when designing a monitoring and steering system that can support multiple message-passing libraries. First, we must consider the requirements of the monitoring and steering system with respect to its internal communication infrastructure. We do not want to use multiple communication libraries simultaneously and must therefore specify the internal messaging system in an abstract way that can be implemented using any message-passing library. This requires that the monitoring and steering system be implemented using a set of methods that are common to all communication libraries. Secondly, because we want to interface our system to the application with a minimum of annotations, we must support those methods of communication specific to the message library used in the application. The following section outlines differences between MPI and PVM that were

essential to solving the above outlined problems with integrating MPI into the PathFinder system.

Differences Between PVM and MPI

PVM allows users to view a cluster of computers as a single parallel computer. PVM makes use of a simple message passing mechanism that allows users to exchange data between processes. Central to PVM's message passing mechanism is a buffer, an array of bytes, which PVM uses to hold the data contained in a message. Users in PVM are free from concerns about the parameters of the buffer and are presented with a simple scheme for creating and sending messages. In order to create and send a message in PVM the following steps may be taken:

1) **Create a default buffer:** the user begins by calling a method named *initsend*, which creates a default buffer.
2) **Pack data into the message buffer:** the user then calls the appropriate pack methods to add the various types of data to the message. The user may call multiple pack methods to add differing types of data without concern for the type of the underlying buffer, or the size of the buffer. PVM will dynamically grow the buffer to accommodate new data as it is added to the message. Some examples of pack methods include pkint, pkdouble, pkfloat, and their corresponding unpack methods upkint, upkdouble, and upkfloat. These packing methods can be used to pack either single elements or arrays of the given typed data. Methods also exist for packing null terminated strings or arrays of bytes.

3) **Send the message:** this is accomplished using a send method that specifies the recipients of the message and a *tag* identifier that is used by the receiving process(es) to decide how the message should be received.

4) On the receiving side of the application the user must call unpack methods in the same order as the pack methods were executed in order to restore the data in the message from the underlying buffer.

PVM supports both blocking and non-blocking communication schemes, and allows for dynamic process creation at run-time.

MPI is a newer communication library that supports many of the same features as PVM, although not all are currently available across all platforms. MPI has methods that use a buffered approach to message passing that are similar to PVM. MPI users also have the option of directly sending data in messages without explicitly packing a buffer. Messages sent using this scheme consist only of singularly typed data, single values or arrays of native types. MPI has methods that can extend the range of data types to include user-defined types and structures. This allows users to define custom data types that can be sent without concern for buffering. If the buffered type of messaging is used, the users are responsible for all management of the message buffers. This means that users must allocate buffers of sufficient size to accommodate the data being sent in a message and for data being received from a message. This is problematic for users when packing arbitrary amounts of data into a message. Often the required size of a buffer is not known until all the elements to be packed are considered. In cases where data are packed iteratively such as in a looping construct it may become necessary to reallocate a larger sized buffer and copy the contents of the old into the new. Receiving messages

buffers are easily specified because the users can view the size of an incoming message before allocating the size of the buffer used to receive it. MPI also has a data reduction method that can compute various aggregate values of a variable with respect to the set of all processes in the computation. Other differences between MPI and PVM exist that are less germane to the problems involved with writing a monitoring and steering system, but are worth considering when writing distributed applications in general. A more in-depth discussion of these differences can be found in [18].

Communication Middleware in PathFinder

The PathFinder system is constructed from a series of layers that wrap around a base communication layer. This base communication layer has a set of generic message passing methods that are common to most communication libraries. These methods include blocking and non-blocking versions of both send and receive as well as methods for packing data into a message buffer for sending multi-type content in a single message. The other layers used in the PathFinder system are designed to allow interception and message tracking for purposes of cataloging and constructing globally consistent snapshots. Specifics for the communication layers in PathFinder are found in Chapter 2, under the PathFinder Architecture section. When a message is sent in the application, it is passed through the various layers until finally the message is passed to a base communication layer which makes a method call to the appropriate message-passing library. In the initial version of PathFinder there was only a PVM layer that allowed the system to pass messages. In order to make the PathFinder system compatible with an MPI application, it was necessary to create an MPI layer that encapsulates those message-passing methods used in MPI applications as well as those of PathFinder's internal

messaging system. Figure 9 illustrates the layered structure of the PathFinder system and how it interfaces with an application.

Problem Specifics on Implementing the MPI Layer

The implementation of MPI into the PathFinder system had two distinct facets. First, a series of methods with MPI signatures needed to be added through the PathFinder layers that allowed the system to call MPI methods, using the same method signatures as those found in the MPI library. These methods are used when the application makes message-passing calls that are then propagated through the layers of the PathFinder system. A second version of the message passing methods were needed to mimic the signatures of the message-passing calls as they are used by the PathFinder system for sending internal information specific to cataloging and managing the monitoring and steering operations (i.e. snapshots, and consistency control operations.) The reason the second set of methods are needed is to allow a single version of PathFinder the ability to use either MPI or PVM (each to the exclusion of the other) depending on the type of application. In order to avoid code duplication within the system, a version of the MPI layer that exactly mimicked the functionality and method signatures of PVM, complete with dynamic buffer creation and management was needed. This requirement is made necessary by the fact that MPI expects the user to handle all facets of message buffer management when sending multi-type packed messages. To achieve this goal a standard template library (STL) vector class was used to allow message components to be dynamically added and stored in a linked data structure. Another vector was used to maintain data on the type and size of each component in the message buffer. When all the data to be sent is packed into the vector and the send method is called, a send buffer

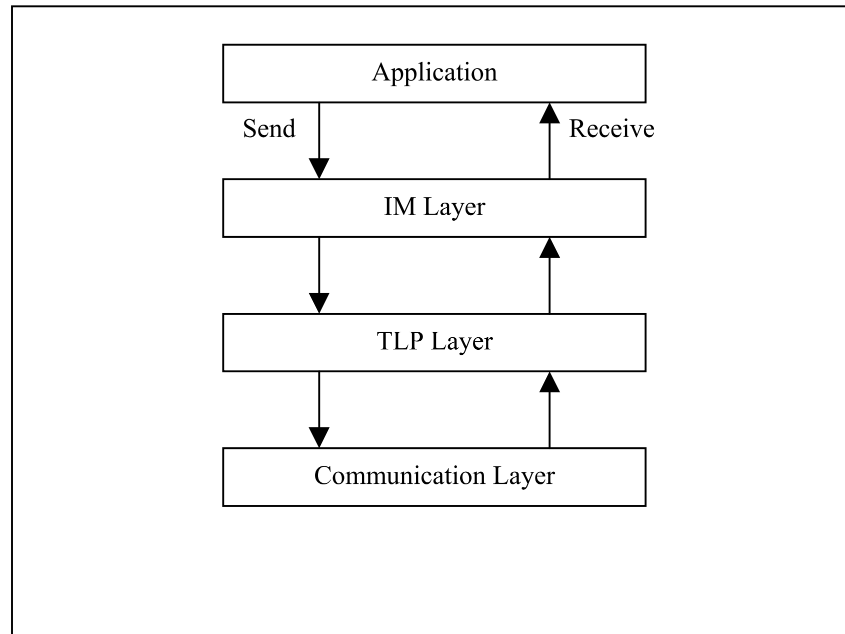

Figure 9. Layers in the PathFinder Architecture.

of type byte array is created with a size equal to the size of the sum of the contents of the data vector. Each component from the data vector is then packed into the send buffer using the MPI packing method appropriate to the type of the current data component. The send buffer is then sent to the appropriate process where it is unpacked in a fashion similar to PVM.

CHAPTER 6

COMPARISON OF THE PERL AND JAVA AGENT SYSTEMS

The algorithm that describes the behavior of the agent *milieu* is the same for both Java and Perl versions, but there are some differences in the basic mechanisms of the *milieu* between the two implementations. The Perl Agent class has data structures that hold data for identifying the agent, defining event handlers, and storing local data. Because Perl can execute by interpreting strings, its event handlers exist as strings in a hash table that associate on an event handler name. When implementing an instance of an agent, the programmer specifies each event handler by creating a hash table entry that associates the event handler name with an array of strings that are the lines of code to be performed. Outside the agent *milieu,* Perl agents exist as string objects. Java agents are defined by an Agent class that is similar to the Perl agents. However, java agents have a different implementation for event handlers. Java agents respond to events via an abstract method called runEventHandler. This method takes the event handler name as a parameter and uses control structures within the code to direct the method to run the correct code based on the event handler name. Java agents exist in the *milieu* as either a Java agent object or a *serialized object* [19]. Serialized objects consist of a snapshot of an object's state as it existed at the time that it underwent serialization. An agent object is an instance of an agent class that has defined abstract methods that describe the event response operations for that specific agent. Agent objects are serialized so that they may

be sent across a network or saved to a file. Both versions of agents allow run-time customization of the agents. Perl agents can be modified by changing all or portions of the string that describes them. Java agents can be altered via an object copy and redefinition of the agent's abstract methods. Because Perl agents exist as strings they can be manipulated and contained in both application native and Perl environments. Java agents exist in the native space of the application as an array of bytes. Java agents have the requirement that they can only be modified within the Java *milieu.* If an operation requires that an agent be modified at a specific node (e.g. the snapshot managers) a Java module must exist where the agent can be accessed and modified.

Experiments to compare Java Agent run times with Perl Agents.

Four experiments were performed on the PathFinder system using both the Java and Perl agent *milieux* in order to determine comparative performance. The version of Perl that was used is 5.00503. The version of Java virtual machine used was from JDK 1.3 version build Blackdown-1.3.0-FCS. Each experiment was run on four Linux machines using two application processes and two PathFinder system processes. The Linux machines used in the experiments were Pentium II 450 processors with 128 Mbytes of memory and were mounted on a shared file system using NFS. The machines were connected on a LAN via a 100 MBit fast Ethernet network.

The first experiment was designed to measure the effects of using the PathFinder system on a typical application. The application consisted of two processes that performed multiple iterations consisting of *sin* calculations and one message sent or received. Six cases were considered, varying in the number of iterations performed and in the number of *sin* calculations performed during each iteration. The number of *sin*

calculations was set to 1000 per iteration in one set of execution and 10000 per iteration during another set of execution. Varying the number of iterations provided a context to evaluate the initialization costs. Varying the number of *sin* calculations provided the opportunity to observe the effects of changes in the ratio of computation to communication. The application was executed three times performing 100, 1000 and 10000 iterations on each respective execution. First the application was timed without the PathFinder system installed in order to establish a base time measurement. Next the application was executed for a *milieu* with no active agents, one active agent and with ten active agents installed. This was done to provide a context for determining the costs of running the system with various numbers of agents installed. These three cases also had a single monitoring agent installed. Monitoring agents are not considered active in this context because they only perform a function once upon arrival and add no further overhead during subsequent events. The active agents that were installed performed a single steering operation during each end of transaction event. The transaction frequency was set to one end of transaction event per iteration. The results of the first experiment are shown in Figures 10 and 11. Figure 10 contains a chart showing the increase in execution time of the 1000 *sin* calculation application as compared with the no monitoring case. For 100 iterations, the experiment showed that the application with zero active agents installed executed 10.7 times slower than the base case with no monitoring system did. The version with one active Perl agent took 11.5 times longer to execute, and 18.3 times longer with ten agents installed. As the number of iterations increased the agents performed faster. In the case of 10000 iterations, the relative execution seemed to converge at 2.46 times slower for zero active agents, 3.01 for 1 active agent, and 6.01 for

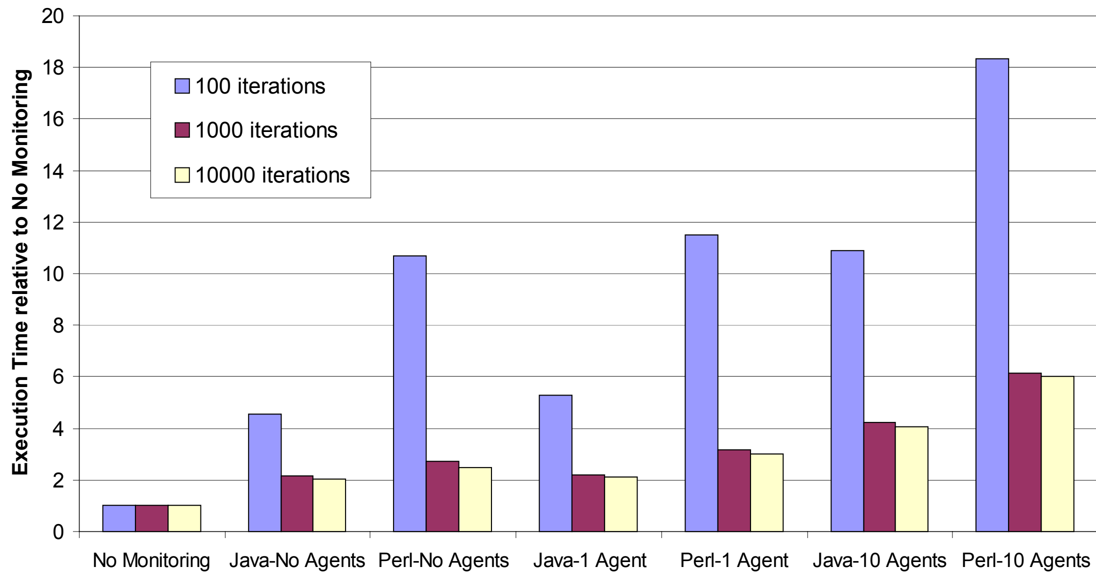

Figure 10. Increase in execution time of application with 1000 sin calculations and under various milieux and agent loads relative to application without Pathfinder installed.

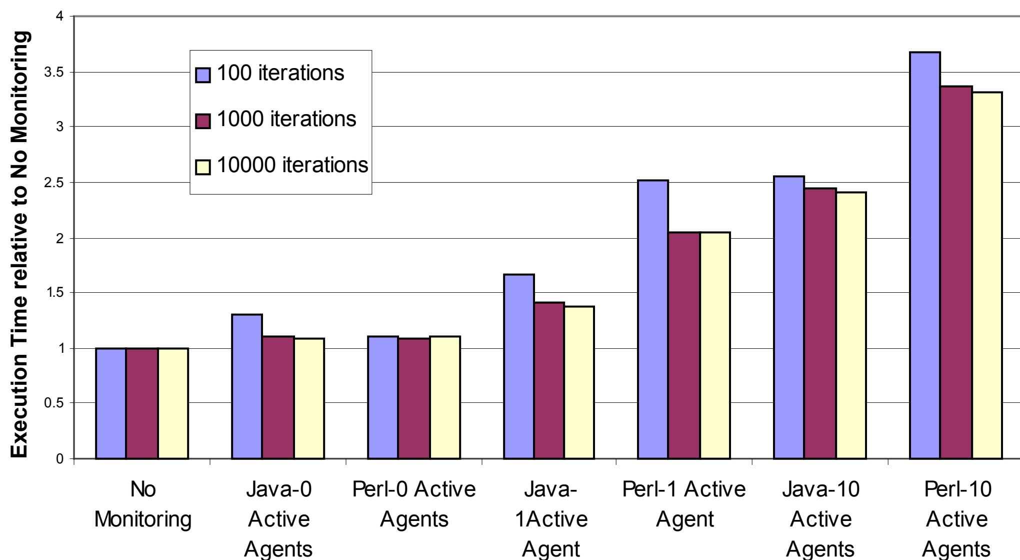

Figure 11. Increase in execution time of application with 10000 sin calculations and under various milieux and agent loads relative to application without PathFinder installed.

ten active agents in the Perl version. For 100 iterations, the Java *milieu* executed nearly twice as fast as the Perl version did. In the case of 10000 iterations, the Java agents performed 2.03 times slower than the base application for zero active agents, 2.1 times slower for 1 active agent, and 4.07 times slower for ten active agents. These results indicate that there is an initial cost to running the agent systems that is amortized over long running times. A typical execution would fall between the case of zero and one active agents, because steering is not generally expected to occur at each transaction. Cases of ten agents steering at each transaction would be extremely rare. Figure 12 shows a chart of the data for the application when run with 10000 *sin* calculations per iteration. In this case the differences between the execution times of the application with no monitoring and under the various agent loads were much smaller. In the case with monitoring and 1one active agent the Java *milieu* performed 1.38 times slower than the no monitoring case. The Perl *milieu* performed 2.05 times slower. In the extreme cases with ten active agents the Java performed 2.41 times slower and the Perl performed 3.31 times slower.

The second experiment was designed to measure the execution time of the agent *milieu* when run with multiple agents. The experiment measured execution times in the *milieu* having from zero to fifty agents installed at each IM. Each installed agent consisted of a single event handler that performed a native method call to subscribe to an attribute in the application, a constant time operation. This event handler was registered on arrival to respond to an *eot* event. Figures 12 and 13 compare the results of the second experiment to measure the execution time of both Perl and Java *milieux* with multiple agents installed at the application IMs. A linear regression analysis of the data from each

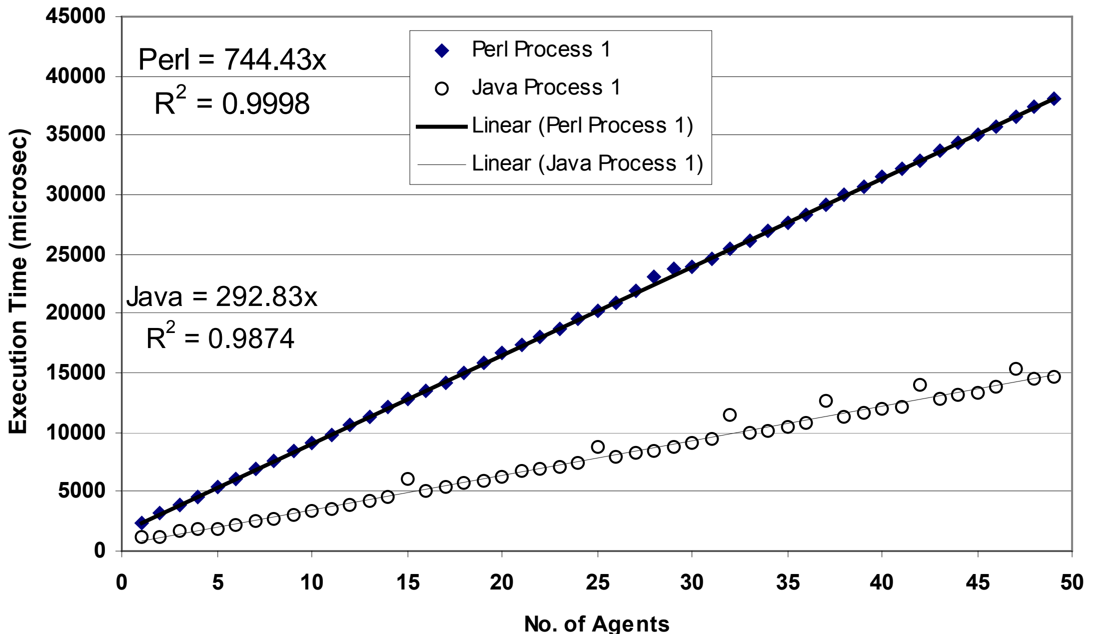

Figure 12. Execution Time for multiple Perl and Java Agents (Process 1)

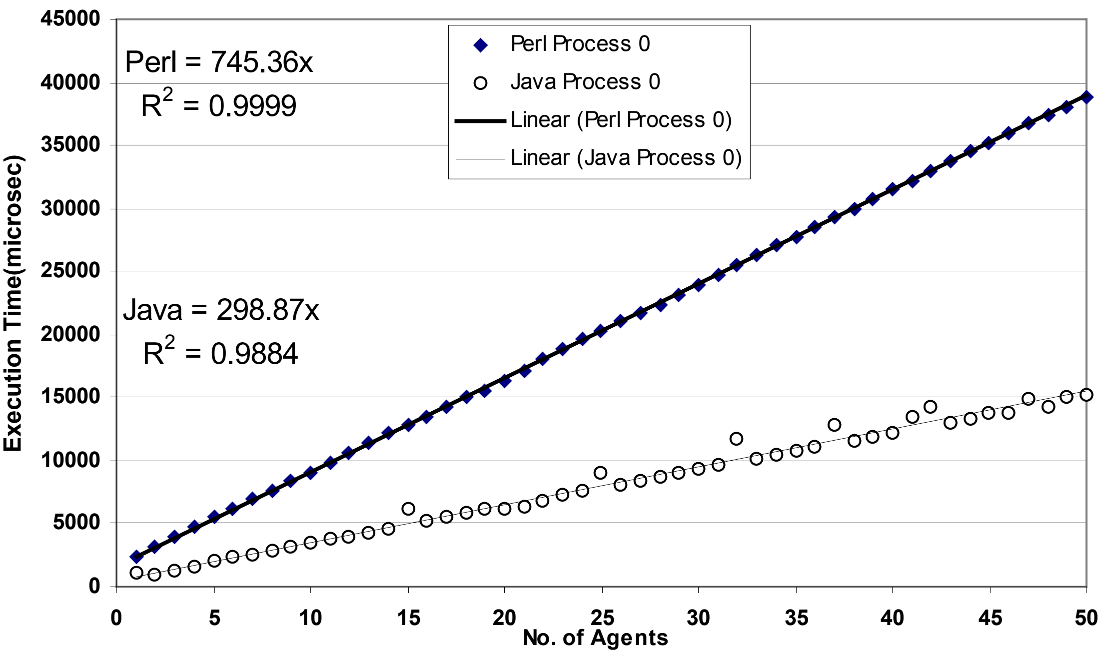

Figure 13. Execution Time for multiple Perl and Java Agents (Process 0)

process shows that the execution time of the agent *milieu* grows in a linear fashion. The regression analysis for Perl process one shows an additional cost of 744.4 µseconds for each additional agent. The analysis of Java process one1 shows an additional cost of 292.1 µseconds for each additional agent. The results for Perl process zero are similar with an additional 745.4 µseconds for each agent. Java Process zero shows an additional 298.9 µseconds for each additional agent. The correlation, R^2, for each linear regression analysis shows good agreement between the data and the fitted line. The results show that Java agents run more efficiently than the Perl agents do when multiple agents are installed at the *milieu.*

The third experiment was designed to measure the execution time of the agent *milieu* with a single agent installed that executes one event-handler. Fifty agents were tested each with an increased amount of work in their event handler in order to find how an additional workload in the event handler affects the overall performance of the *milieu*. The work in the event handler consisted of a *for*-*loop* whose maximum number of iterations varied between zero and 49000 in 1000 iteration increments (the 0th agent performed zero iterations and the 49th agent performed 49000 iterations.) For each iteration the agent performs two load-and-storeload and store type calculations. An additional version of this experiment was performed with maximum iteration varying from zero to ninety-eight and the agent performing a subscribe operation as in the multiple agent experiment. From these two versions of this experiment we hoped to learn about the performance of an agent when running only code from its particular language versus when the agent is running native code from the application. Figure 14 shows the results for the third experiment where a single agent executes at the *milieu* with a varying

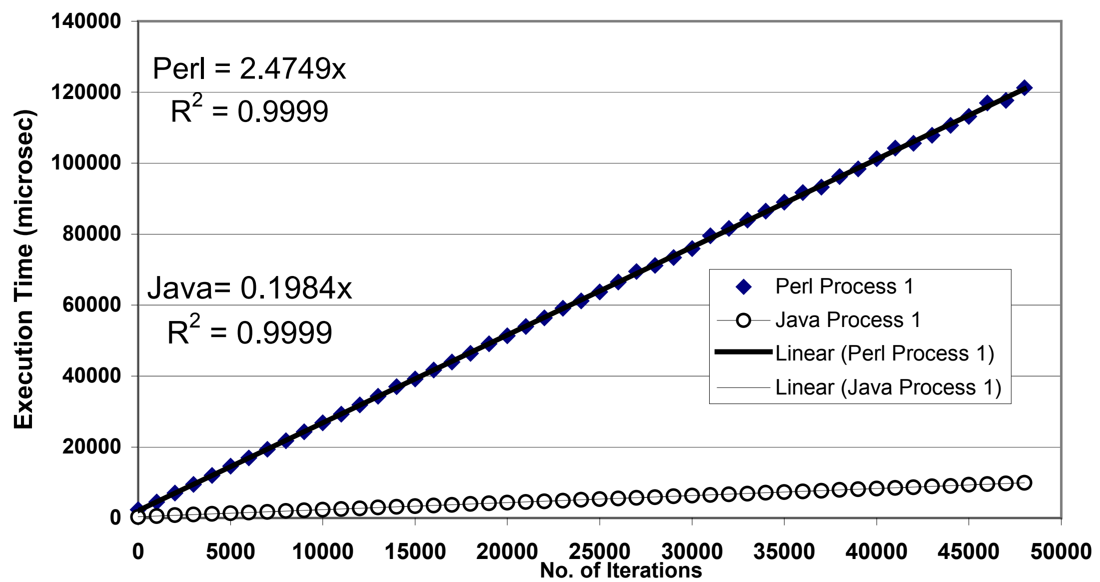

Figure 14. Execution Time for Perl and Java Agents, varying workload

amount of work in the event handler for both the Perl and Java *milieux* respectively. The Perl version shows an additional 2.5 μseconds for each additional iteration performed by the agent's event handler. The Java *milieu* shows a cost of 0.2 μseconds for each additional iteration. This result indicates that the Java version of the *milieu* performed roughly ten times faster than the Perl when performing multiple iterations of purely computational operations. Figure 15 shows the results for the second version of this experiment. The Perl and Java *milieux* had similar performance in this case with the Java version performing slightly better. This is likely due to the fact that both implementations incur the overhead due to the *milieu* only once (only one agent event handler executes) and that the agent code performed in multiple iterations benefits from caching.

The fourth experiment was done to characterize the execution times of the components involved in running the agent *milieu*. These components include:

- Context switching between the application and the agent *milieu*
- Building the schedule of all registered event handlers
- Executing each event handler entry on the schedule

The cost of context switching to the *milieu* was determined by taking four time measurements as shown in Figure 16. The four measurements consist of two measurements in the IM, t_1 and t_2, made before and after the *milieu* executes, and two measurements, m_1 and m_2 made immediately upon entering and before exiting the *milieu.* The difference t_2 - t_1 is the total execution time of the agent *milieu* plus the time spent in context switching. The difference m_2 - m_1 is the total execution of the agent *milieu* without the context switch. By subtracting these two differences we have the amount of time it takes to context switch both to the *milieu* and then back again to the IM. By

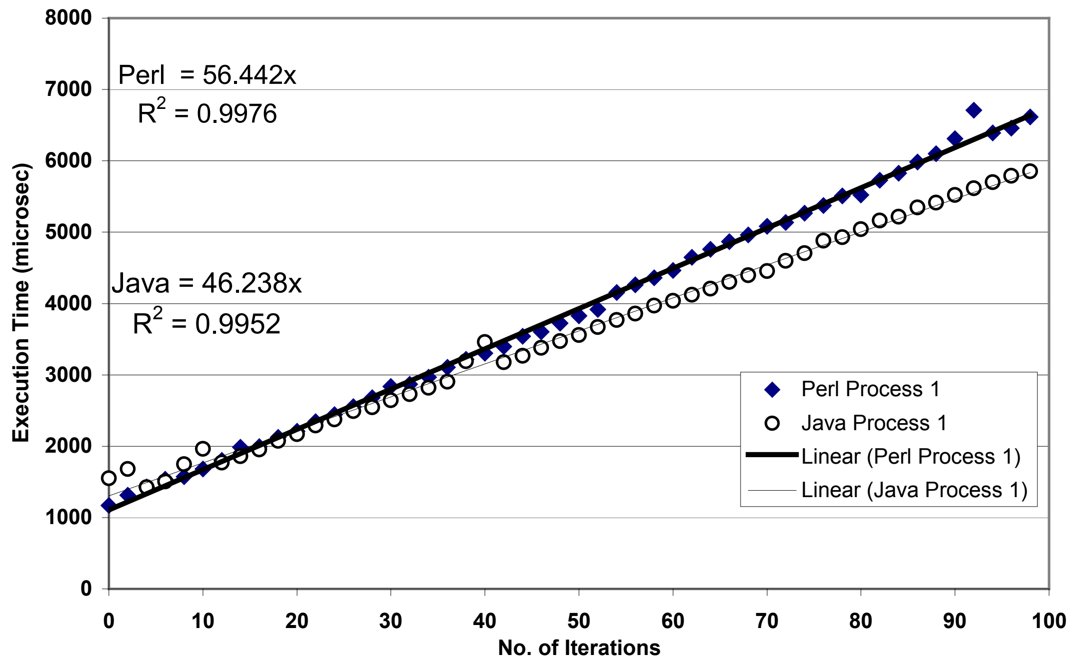

Figure 15. Execution Times for Perl and Java Agents, multiple iterations running native code

taking ½ of this amount we arrive at the cost, ***C***, of a single context switch as shown in the following equation:

$$C = 1/2((t_2 - t_1) - (m_2 - m_1))$$

Building a schedule of events is accomplished by executing a *schedule_event* operation from the agent library of methods. In order to measure the cost of building a schedule, ten agents were registered for a given event, and then the *schedule_event* method was timed as it processed the registered agents. The cost of executing a typical event handler was found by timing the execution of each of the ten agent's event handlers. The event handlers performed a single monitoring operation that was implemented in native code. The results of the fourth experiment showed that Perl took 550.5 µseconds to context switch for the Perl *milieu* and 115.1 µseconds for the Java *milieu*. These results indicate that the Java agent system is 80% faster at context switching from the IM to the *milieu* than the Perl system. These differences are likely due to the fact that the Perl *milieu* has to interpret the code for the agent *milieu* each time that it is executed. Building the schedule took 408.0 µseconds for the Perl *milieu* and 297.0 µseconds for the Java *milieu*. In this case the Java system performed on 27% better than the Perl system. In both *milieux,* the building of the schedule involves associating a list of event handlers with a key name for the event. This smaller difference in performance may be due to optimizations in hashing, which is often used function in Perl applications and is likely more optimized than the Java hashing implementation. The execution time of each event handler under the Perl *milieu* was 353.0 µseconds on average. The Java agents executing similar event handlers had measurements of 48.0 µseconds on average. The Java event handlers performed an average of 86% faster than the Perl handlers did.

CHAPTER 7

DISCUSSION AND CONCLUSIONS

Exploratory visualization is a highly visual and interactive approach to helping users understand and manage long-running, distributed computations. Concerns about performance, flexibility, consistency, and correctness provide challenges in the implementation of such a system. In this work three problems were identified in the PathFinder system and addressed in an effort to increase the usability and efficiency of the system in monitoring and steering distributed applications.

Target users of the PathFinder system include both programmers and non-programmers. To permit non-programmers to take advantage of the agent capabilities of the PathFinder system, an agent wizard was developed. Through this agent wizard a non-programmer user is able to create a high-level specification of an agent's behavior. The wizard then generates the agent code from this high-level specification and sends it to be installed at the appropriate system components. In this way, the agent capabilities of PathFinder were made available to a wider range of users.

Another factor limiting the usefulness of the PathFinder system to users was the communication middleware used by the system. The first version of the PathFinder system made use of the PVM message-passing library. Many distributed applications and scientific computations use a newer middleware called MPI. In order to increase the scope of applications usable with the PathFinder system, a version of PathFinder was

implemented that allows users to monitor and steer applications using either PVM or MPI.

Research on a previous version of PathFinder found the Perl agent system to be a significant source of the performance penalty incurred when using the PathFinder system. This finding sparked interest in developing another agent system using different languages in order to find the best available. This work presented a version of the PathFinder system that makes use of a Java agent system for monitoring and steering of distributed applications. It also presented results from a series of experiments that were performed to determine which language performed better at efficiently operating the agent system.

The results from each experiment showed that the Java agent system had faster performance than the Perl system. The first experiment investigated the overall cost of using the PathFinder system with either agent *milieu.* The results showed that the Java system performed 20 – 30% faster than the Perl system did. The second experiment measured how the agent *milieu* performed under multiple agents and found that the Java agents performed roughly two times faster than the Perl agents did. The third test involved running a single agent with increasing amounts of work in its event handler. In the case where the agents were performing multiple subscription operations from within a single event handler it was found that the performance gap between Perl and Java became much smaller. This result was likely due to the fact that much of the overhead due to the agent *milieu* was obviated when only a single event handler was executed. When the agent *milieu* executed only a single agent, its running time was reduced to 3 constant time operations, building the event schedule, pulling the event off the schedule and then

beginning execution of the event handler. In cases where the agents performed multiple iterations of pure computation without calling native methods the Java agent system performed over ten times faster than the Perl system. These results indicate that the way users design agents can have a dramatic effect on how the system performs. By installing fewer agents with more work in their event handlers users can expect to see faster performance in the application. These results also indicate that future work in optimizing agent operations at the *milieu* can be performed by combining the works of multiple event handlers.

Other factors that must be considered when choosing an implementation for the agent system include ease of specification and power of the resulting agents. Perl and Java provide a means of implementing agents that would be considered easy to a trained programmer. Java may have a slight advantage over Perl in this respect because Java has a much cleaner specification for defining classes. Both Java and Perl are fully functional programming languages and as such an agent written in either language is equally powerful. Perl has an advantage in the fact that its agents exist as strings and may be modified at run time from either the native side of the PathFinder system or within the *milieu.* Java agents can be modified only from within the *milieu.*

Future Work

A concern that remains to be addressed in future work is the means by which agents are deployed to the distributed application. Currently, the agents are created at the UI and sent to the snapshot manager whose job is to direct the agents to the appropriate IM. In large-scale applications this scheme poses a scalability problem because a single

SM will become a bottleneck between the multiple users and their access to the many processes.

The problem is that there may be n processes in the distributed computation and m users attempting to monitor and steer the application from m different user interface programs. Agents sent from each UI must pass through a snapshot manager in order to be redirected to the appropriate processes as specified by the users. If the number of snapshot managers does not scale with m and n, a bottleneck will exist in the deployment of the agents. This problem can be solved by specifying a set of snapshot managers S where the $\mathbf{i}^{\mathbf{th}}$ snapshot manager in the set, s_i, is responsible for communication with a subset of the processes in the distributed application. The set of processes that a snapshot manager communicates with is called a *region.* The *region* of the $\mathbf{i}^{\mathbf{th}}$ snapshot manager, r_i, consists of a set of processes $\{ p_l , p_{l+1} ,\ldots, p_{l+k} \}$, where l and k are parameters that specify the subset of the processes. Similarly each of the m user interfaces is restricted to communicate with a specific snapshot manager. The set of user interfaces that a SM communicates is called u_i. Using this approach, the communication group for $s_i = \{S, r_i, u_i \}$. If an agent arrives at a snapshot manager whose destination lies within the region of a different snapshot manager, the agent will be forwarded to the appropriate manager. This adds the requirement that each snapshot manager maintain a table mapping the regions of the distributed process to the set S. Using this scheme, agents will arrive at the appropriate process by travelling through at most two SM nodes and avoids bottlenecks because no one SM is responsible for delivering all agents.

Other work on the Java agent system includes the investigation of multi-threading the agent system in order to gain better performance in I/O intensive applications. As the

agent *milieu* is run it may be possible for some of the agents to be executed in parallel with execution of the distributed application. In order to benefit from multi-threading, this would require that the application make somewhat frequent use of disk I/O or messages using blocking semantics. In these cases the agents could make use of the processor time while the distributed application awaits completion of I/O operations. Implementing this feature would also necessitate a study of how agents interact with the system in terms of monitoring and steering consistency.

BIBLIOGRAPHY

[1] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315-339, 1990.

[2] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI – The Complete Reference, 2nd Edition, *The MIT Press,* 1999.

[3] R. Schwartz and L. Wall. Programming Perl. O'Reilly and Associates, 1994.

[4] Stasko, John T. and Kraemer, Eileen, A Methodology for Building Application-Specific Visualizations of Parallel Programs, *Journal of Parallel and Distributed Computing*, *Vol. 18, No. 2 :*pp 258-264, June 1993.

[5] Brown, Marc H. Exploring Algorithms Using BALSA-II. *IEEE Computer,* 21(5):14-36, May 1988.

[6] Stasko, John T., TANGO: A Framework and System for Algorithm Animations, *IEEE Computer, Vol. 23, No. 9* :pp 27-39, September 1990.

[7] Geist, G. A. and Kohl, J. A. and Papadopoulos, P. M. CUMULVS: Providing Fault-Tolerance, Visualizations, and Steering of Parallel Applications. *SIAM*, August 1996.

[8] Vetter, J. and Schwan, K. Progress: A Toolkit for Interactive Program Steering. *Proceeding of the 24th International Conference on Parallel Processing.* 139-142. 1995.

[9] Gu, Weiming and Eisenhauer, Greg, and Kraemer, Eileen and Schwan, Karsten and Stasko, John and Vetter, Jeffrey and Mallavarupu, Nirupama. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. *College of Computing, Georgia Insitute of Technology, Atlanta, GA Technical Report* GIT-CC-94-21, April 1994.

[10] The Distributed Object Visualization Environment -- draft. [web page] April 2001; http://www.cs.wustl.edu/~schmidt/dove.html [Accessed 23 April 2001].

[11] A High Performance Java Distributed Event Delivery System – draft. [web page] April 2001; http://www.cc.gatech.edu/~zhou/jecho/index.html [Accessed 23 April 2001].

[12] Delbert Hart, Eileen Kraemer and Gruia-Catalin Roman. Interactive Visual Exploration of Distributed Computations. *Proceedings of 11th International Parallel Processing Symposium*,121-127, April 1997.

[13] Michael Wooldridge. Intelligent Agents. In G. Weiss, editor: Multiagent Systems, *The MIT Press*, April 1999.

[14] Navin Gupta. Performance Considerations in the Monitoring and Visualization of Distributed Computations. *The University of Georgia Press*, August 2000.

[15] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, 1978.

[16] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63-75, 1985.

[17] Delbert Hart and Eileen Kraemer. Consistency Considerations in the Interactive Steering of Computations. *International Journal of Parallel and Distributed Systems and Networks*, to appear.

[18] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A Comparison of Features. Calculateurs Paralleles Vol. 8 No. 2, 1996.

[19] Sun Microsystems Java -- draft. [web page] April 2001; http://www.javasoft.com [Accessed 20 April 2001].