

Consistency Considerations in the Interactive Steering of Computations

Delbert Hart, Eileen Kraemer

Abstract—

Interactive steering can be helpful in understanding and managing parallel and distributed systems. Multiple approaches to the implementation of steering systems are under investigation or in development. However, the notion of consistency in both the presentation of system state and the application of steering changes to the executing system varies considerably from system to system. We present these various approaches to steering and to consistency and discuss their costs and benefits. Also, we present PathFinder, a general architecture for monitoring and steering that permits users to configure the steering system to provide varying levels of consistency and that supports highly dynamic and expressive specification of steering actions. PAT(PathFinder: Agents/Transaction), our initial implementation of the PathFinder architecture, employs agent-based monitoring and steering coupled with transaction-based consistency calculations. The flexibility of this agent-based approach is amenable to experimentation with a variety of consistency conditions.

Keywords—Interactive steering, consistency, agents, transactions

I. INTRODUCTION

The interactive steering of computations permits users to monitor a program's execution and to adjust both application parameters and the allocation of resources in an online fashion. This interactivity provides a powerful tool for application scientists, researchers, and algorithm developers in the process of "charting unknown waters", whether exploring new computational solutions to problems that are not yet well understood, or attempting to select ideal parameters for familiar algorithms applied to new data sets or problem areas. Applications for which steering is useful are typically long-running, complex simulation, modeling, or control programs executing in parallel or distributed environments.

The ability to monitor the program in execution and observe intermediate results coupled with interactions that permit the user to tweak application parameters, select or install new control algorithms, and direct the allocation of resources provides users with the opportunity to improve the performance or quality of a solution for a particular run of their program. In addition, the experience of observing and interacting with the running computation can help the researcher to better understand the dynamics of the program's behavior, provide insight into the target problem and data, and build intuition that can lead to the selection of better default parameters and the development of algorithms more highly tailored to the target domain.

A computational steering system consists of a monitoring component, some type of display and user interface, and a mechanism for propagating steering actions back to the executing program. In implementing these features for

distributed systems, issues related to *consistency* and correctness must be addressed. *Consistency* guarantees that displays presented to the viewer represent some valid state of the computation and that steering operations are applied in a way that maintains the correctness of the computation. Note that although some steering actions may be applied at any of the participating processes at any point in the computation, others may be correctly applied only at certain points in the execution of the process, and still others may require some coordination between processes.

For example, in a distributed algorithm for simulated annealing, the goal of the computation is to locate a configuration, defined by the values of a set of variables, that maximizes or minimizes some objective function. In such an application, it may be possible to alter at any time the parameters that control the rate at which the search is conducted. However, a change in the parameters that define the configuration itself may result in an incorrect calculation of the objective function if applied in an inconsistent manner, i.e., applied at different points in the execution at different processes, or in the midst of a section of code that is regarded as atomic with respect to the parameter in question.

Further, some distributed algorithms for simulated annealing actually distribute components of the configuration's state across multiple processors, and parallelism arises from their cooperation in evaluating the objective cost function. Consider a steering action that invokes a redistribution of components across processors to achieve a better load balance. An uncontrolled invocation of this action might result in a transient loss or duplication of some components, causing the computation to fail or produce an incorrect value. It is clear that continued correctness will require some coordination of processes.

Consistency concerns must be balanced against consideration of the *lag* and *perturbation* induced by the monitoring and steering process. *Latency* or *lag* refers to elapsed time. *Presentation lag* is the elapsed time between the existence of a state in the program's execution and the presentation of that state to the viewer. *Steering lag* is the elapsed time between the user interaction that initiates a steering action and the application of that action to the computation. *Perturbation* refers to the degree to which the underlying computation is slowed down or otherwise affected by external forces.

The ideal system would feature strong consistency, and low latency and perturbation. More realistically, the perturbation associated with steering should be small for small numbers of processes with few monitored attributes and displays and rare, local steering interactions, and would in-

crease proportionally as greater numbers of processes and attributes are monitored and visualized, and as the number and scope of steering actions increases. In fact, these desirable characteristics of strong consistency, low latency, low perturbation, and linear scaling are competing characteristics and the choices of algorithms and methods for data collection and analysis, visual presentation of system state, and application of steering actions effectively determine the relative priorities of these goals in the steering system.

In practice, the most desirable configuration of the steering system, depends on the needs of the user. For example, the use of the interactive steering system to explore the performance behavior of a code in order to parallelize it necessarily focuses on minimizing perturbation, while lag and monitoring consistency may be of less importance. On the other hand, user interactions for the purpose of obtaining more highly refined solutions may require reductions in presentation lag and steering lag, at the expense of performance. Similarly, steering interactions for the purpose of *understanding* the effects of parameter changes must place the minimization of lag as a high priority, so that viewers are able to correlate their interactions with the results of those interactions.

It should be noted that the approach to consistency control, including both the degree of consistency, if any, that is attempted or guaranteed, and the method by which consistency control is implemented, is dependent upon other choices, related to instrumentation and data collection methods, data analysis techniques, presentation tool capabilities, and the program model assumed (iterative, transaction-based, message-passing, shared memory, etc.).

In the following section, we describe the approaches of a number of researchers to dealing with consistency concerns, and the balances they strike among the competing concerns of consistency, latency, and perturbation. In section 3 we present PathFinder, an architecture we are developing to provide a modular computational steering infrastructure that meets the needs of a broad class of users and applications and permits the user to configure the steering system to achieve an application-appropriate balance among consistency, lag, and perturbation. Further, we provide details of a structure and methodology for steering agents that permits highly dynamic and expressive specification of steering actions, ranging from the simple setting of a local attribute to the invocation of dynamically defined algorithms capable of evaluating global predicates and applying globally consistent steering actions. Finally, in section 4, we summarize the contributions of the PathFinder architecture and its PAT implementation, and discuss its place along the spectrum of consistency capabilities of steering tools for distributed systems.

II. RELATED WORK

Systems for the interactive steering of computations address consistency concerns at a variety of levels. Some systems leave such concerns entirely up to the user or programmer[1], [2], or rely on the application to provide the

necessary consistency checks[3]. Others, typified by the “scripting approach” described below, limit the points in the program at which steering may take place, and still others limit the steering system to a particular programming model, i.e., simple iterative computations with a single main loop[4]. Application-specific steering systems typically solve the consistency problem in an application-specific manner.

At the other end of the spectrum, more complex schemes for controlling interaction points[5], [6] and detecting changes that affect consistency[7] have been developed. However, few of these systems address the coordination of steering actions across multiple processes in a distributed system.

The “scripting” approach to interactive steering involves breaking up code written in C, C++, or FORTRAN into modules, and using scripts written in Python, Perl, or Tcl to control the execution of these modules. Often, SWIG[8] is used to generate the bindings between the module and the scripting language. Scripts can be changed dynamically. Steering consists of altering the scripts: substituting, adding, or deleting modules or altering control flow code. In such systems, maintaining consistency depends largely on the creator of the modules; they must be written so that modules may be called in any order at any time[9], typically requiring safety checks to ensure correctness. A distinct advantage of this approach is the simplicity of implementation. Further, the approach allows modules to be added in a highly dynamic manner, is portable, and promotes reuse. Successful examples of the use of this approach include the SPaSM molecular dynamics code[9], and Winfield’s virtual laboratory notebook[1]. However, interaction is severely restricted, being limited only to the invocation of modules, with no control over their inner workings.

The dataflow approach is exemplified by SCIRun[7], a scientific programming environment that allows the interactive construction, debugging, and steering of large-scale scientific computations. The primary purpose of SCIRun is to enable the user to interactively control scientific simulations while the computation is in progress. Composing the simulation is accomplished via a visual programming interface to an AVS-type dataflow network. Controlling a simulation involves steering the simulation interactively as it progresses. Both intermodule steering (the adjustment of input parameters to modules), and intramodule steering (direct lightweight parameter changes) are supported. Consistency of intramodule steering actions relies on the implementer of the module. Consistency of intermodule steering actions is achieved through cancellation and re-execution. When the user changes a parameter in any of the module user interfaces, the module is re-executed and all changes are automatically propagated to all connected modules.

CUMULVS[4] for the interactive steering of PVM-based applications, also relies on the placement of instrumentation to ensure consistent steering changes. However, it is more restrictive in that it assumes an iterative computation with a single main loop, and a single point in the it-

eration at which variables are updated. A highly desirable feature of CUMULVS is that it minimizes the transmission of data by keeping track of which data elements are in each task, based a specified decomposition type. Iteration numbers are used to ensure synchronization of steering updates. An optimization of this permits the user to specify a range of iteration numbers during which the update may be applied. Coordination of the actions of multiple users is accomplished through a token scheme.

In other systems, more targeted, refined steering may be achieved through the instrumentation of the code with sensors (for monitoring) and actuators (for steering). Here, consistency is enforced through the placement of these actuators – changes are applied only when the actuator code is executed, at the point in the computation at which the actuator has been inserted into the code. However, such an approach provides no mechanism for the coordination of actions across multiple processes. Systems such as Pablo[10], Falcon[11], Progress[12], Magellan[5] and VASE[13] ensure consistency of steering actions through reliance on the placement of user- or programmer-defined instrumentation at “safe” points in the execution of the code. Progress provides a variety of instrumentation object types including sensors, actuators, probes, function hooks, complex actions and synchronization points. Magellan[5] uses a specialized specification language, ACSL, which provides commands for monitoring and steering using probes, sensors, and actuators.

The MOSS system[3] takes an object-oriented view of steering, and assumes that each “steerable parameter” of an object has a method through which the parameter may be adjusted. Further, it assumes that the object itself encapsulates the consistency concerns, and that the steering system need not deal with it, but rather merely needs to invoke the “set” method on the parameter or parameters of interest, and the application code will handle all consistency or synchronization issues.

In the following section we present the PathFinder architecture for monitoring and steering computations. PathFinder is *configurable*, permitting users to select from among a variety of capabilities (and implementations of those capabilities) for monitoring and steering. These capabilities provide varying degrees of consistency, with varying effects on the lag and perturbation induced by the monitoring and steering process. Of particular interest is PathFinder’s approach to the coordination of steering actions across multiple processes.

III. PATHFINDER

A. Introduction

The Pathfinder architecture for interactive steering of distributed systems attempts to meet the diverse needs of different users, program models, and applications. In the design of this architecture we emphasize configurability (permitting the user to select an appropriate balance among consistency, lag, and perturbation in both monitoring and steering) and support for the coordinated, consistent steering of distributed programs. The Pathfinder

architecture provides a basic framework upon which the desired functionality can be extended in a modular way. In this paper, we describe a particular implementation of the architecture, PAT (Pathfinder: Agents/Transactions), in which the modules are designed for experimentation with different strategies of generating a stream of consistent snapshots[14] and for applying consistent steering actions. In the following subsections we discuss the components of the Pathfinder architecture and the relationships between these components, and describe how they work together to monitor and interactively steer distributed computations. We then present the specifics of the PAT implementation, describe the two layers of functionality in PAT (ordering and interaction), and provide examples of agents for monitoring and steering.

B. Pathfinder Architecture

The Pathfinder architecture consists of an *Interaction Manager*, a *Snapshot Manager*, and a *Presentation Manager*, as shown in figure 1. The *Interaction Manager* consists of libraries installed at each process to provide an interface between the application and the steering system. The *Presentation Manager* contains the visualization component. It processes user commands, which may require it to make a request to the *Snapshot Manager* on behalf of the user. The *Snapshot Manager* is responsible for translating user requests into actions to be taken by the *Interaction Manager*, collating the information collected by the *Interaction libraries*, and coordinating the activities of the *Interaction libraries* installed at the various processes.

Underlying this architecture is an attribute - event model of the computation. Processes possess attributes that are available for monitoring and/or steering. Each *Interaction library* contains a database of these locally available attributes. An event is generated when a given condition exists at a particular point in the code. Typically, when events are added to an application they are done so in a manner that associates some semantic significance with the different types of events. The *Interaction library* is notified of events as they occur in the application.

In the current implementation software annotations indicate the occurrence of events, and the availability of process variables for monitoring, steering, or both. In later implementations, access to attributes through other techniques will also be possible.

Through the *Presentation Manager*, users interact with visualizations to explore and control the distributed computation. In the PAT implementation, these interactions are formulated as, or translated into queries and steering actions. A query is an indication that a user is interested in continuously monitoring a particular attribute. A steering action is a change made to attributes, possibly at multiple processes simultaneously. The *Snapshot Manager* also collates local snapshots into global snapshots and then orders them based on the user’s preferences.

The *Interaction Manager* provides only a framework for accessing an application’s attributes and learning of its events. The functionality of the monitoring and steering

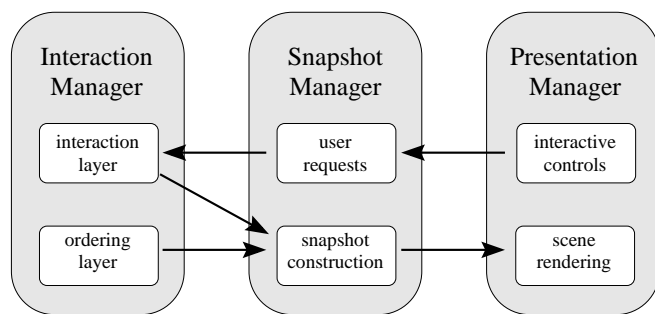


Fig. 1. PathFinder architecture overview.

system is provided by modules installed at the *Interaction libraries* and companion modules installed at the *Snapshot Manager*. A module installed at all of the *Interaction libraries* and the companion module at the *Snapshot Manager* is referred to as a layer. These layers provide services such as ordering information, monitoring or steering capabilities, migration, and rollback. Different versions of a layer may provide similar services, yet have different internal implementations and thus performance characteristics.

The PAT implementation is designed as a platform for experimenting with strategies for monitoring consistency and steering consistency. The PAT configuration has two layers to achieve its mission: an ordering layer and an interaction layer. Monitoring consistency, provided by the ordering layer, is in the form of a stream of consistent snapshots whose order is consistent with the order of events in the application. Various methods of generating this stream of snapshots have been implemented. The ordering layer collects the necessary information to integrate local snapshots into consistent global snapshots and order them. At the *Snapshot Manager* this information is used to generate the snapshot stream. The interaction layer supports the collection of data and the execution of steering actions. Steering consistency relies on the coordination of agents operating within the interaction layer.

Each layer exposes the same type of interface as the application itself: attributes and events. In this way, each layer is relatively independent of the other layers being used. A layer may register attributes, which can be accessed by other installed modules. The modules at an *Interaction library* are totally ordered, and this ordering is used to determine how events are propagated within the library. That is, events are propagated to modules with a higher value. Since the application is not interested in events generated by modules in the *Interaction library*, it is at the lowest layer, see figure 2. Separating the functionality of the system into loosely coupled modules allows the system to be tailored to meet the needs of the user and the application. For example, Pathfinder could be configured with different interaction or ordering layers without any effect on the other layer, or the layers could be left out all together. Some layers require full installation to be effective, for example the ordering layer, while others such as an interaction layer can be installed only where needed.

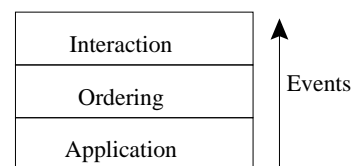


Fig. 2. Events propagate from the lower layers to the higher ones.

C. Interaction Layer

The interaction layer is the medium through which the computation is observed and altered. For the PAT configuration an agent-based interaction layer called FREEDOM (Flexible, Reactive, Extendible Entities for Distributed Observations and Manipulations) is used. The FREEDOM layer uses agents to achieve monitoring and steering and provides a powerful substrate upon which monitoring and steering actions can be realized. In this subsection we describe the FREEDOM layer. Later, we present examples of agents used in the FREEDOM layer to perform consistent monitoring and steering.

The agent-based implementation of the interaction layer enables development of and experimentation with different steering protocols and monitoring strategies, and can provide a steering system that meets diverse demands. Further, the agent framework provides a number of advantages associated with moving functionality to the *Interaction libraries*, has the potential to reduce the amount of lag when reacting to local conditions, and provides a uniform means of specifying actions, while still permitting flexibility and expressiveness.

The FREEDOM layer consists of a *milieu*, (the physical or social setting in which something occurs or develops) at each process, and agents executed within the milieu. An agent is internally structured as a set of event handlers and data spaces. The data spaces can contain any information useful to the agent, such as parameters set by the *Snapshot Manager*, temporary history data of the agent, or pragmas to the milieu indicating how the agent may be more efficiently executed. The agent is reactive in the sense that it responds to events that occur at the milieu it resides in. The most common handler used is an ‘Arrival’ handler, which is executed when the agent arrives at the milieu, and is used to do any necessary initialization. Each event handler is a list of instructions (currently we use Perl[15]), which are evaluated by the milieu. The milieu provides data and procedures for the agent’s functioning. Examples of data available to the agent are references to the currently executing agent, the avatar agent, and the agent library, through which it can access other agents. A milieu provides four services to agents: 1) allowing agents to respond to local events, 2) transportation to other milieus, 3) access to modify other local agents or create new agents locally, and 4) the ability to read the contents of other local agents. Agents within a milieu work cooperatively, such that there is only one agent active in the milieu at any given time. A milieu does not provide any explicit support for inter-milieu communication, other than supporting agent

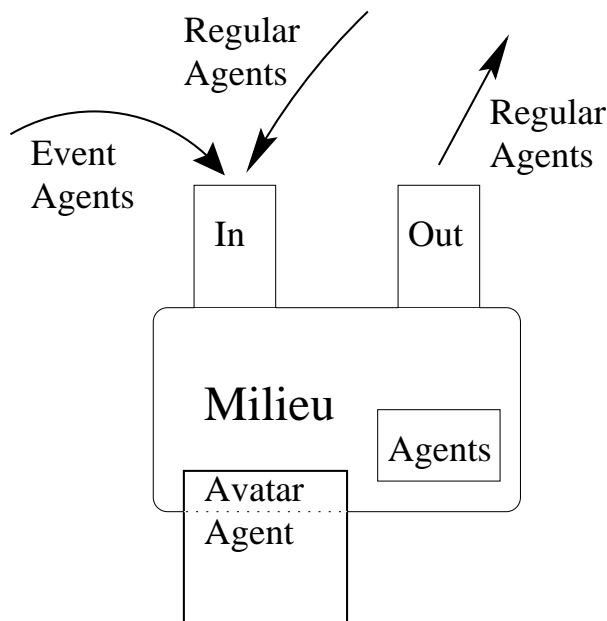


Fig. 3. The input and output queue are a milieu's explicit interface, and avatar agent and event agents are an implicit interface.

migration.

A milieu resides within a FREEDOM module, installed in an *Interaction library*. At least logically, the milieu has a separate thread of control from the application process. An input queue and an output queue of agents provide the only explicit interface that the milieu has with the outside world (see figure 3). The input queue is the channel through which agents can enter the milieu. Likewise, the output queue is traversed by agents going to other milieux. Implicitly, the milieu is connected to the application process via two types of agents: an avatar agent, that represents the state of the process within the milieu, and event agents that are generated by the FREEDOM module to coincide with events that it receives. Event agents are enqueued onto the input queue, and thus don't appear any different to the milieu than any other arriving agents. When the event agent arrives in the milieu it triggers other agents that have been waiting for the event that the event agent represents.

The avatar has attributes which are linked directly to the application's attributes. To read a value in the application an agent simply inspects the corresponding attribute in the avatar agent. Steering is done similarly. The agent modifies the value within the avatar agent, which propagates the change back to the application process. Synchronization with the application process occurs when the FREEDOM module generates event agents in response to events received by the library. The application can choose whether or not it is willing to synchronize. If the application chooses to synchronize, then it will be blocked within the FREEDOM module. It will unblock once an agent within the milieu calls the *resume* method on the avatar agent.

Since the application affects only the avatar and event agents, the milieu itself is independent of the program

Register	Allows the agent to register an event handler to be executed on the occurrence of an event.
Migrate	Sends an agent to another milieu.
Read Agent	Allows inspection of the contents of an agent at the same milieu.
Write Agent	Allow the creation of a new agent or the modification of an existing one, at the same milieu.

Fig. 4. Summary of services provided to the agent by the milieu.

model. The program model affects the behavior and properties of the avatar agent, the generated event agents, and whether the application is willing to synchronize with any modules.

D. Ordering Layer

The Pathfinder architecture is capable of operating under a variety of program models. While some layers, like the FREEDOM layer, are independent of the program model of the computation, other layers are closely tied to the program model. The ordering layer in use in the PAT implementation, TRANSACTION LABELING, is closely tied to the program model.

The consistency requirement supported by PAT is for the presentation of global states to be consistent with the partial ordering defined by the happened-before relation [16]. The presentation of snapshots is totally ordered by the time at which they are shown to the user. The TRANSACTION LABELING layer guarantees that this total ordering of the snapshots is consistent with the partial-order defined by the computation.

In the program model that we have been working in, we view the computation as consisting of a set of (possibly nested) logical actions. The user can view these logical actions as occurring atomically, at whatever level of granularity is appropriate for them. We refer to the logical actions as transactions.

Showing actions that correspond to the logical construction of the application as occurring atomically simplifies and enhances the presentation of the computation to the viewer. Transaction boundaries represent good locations at which to both collect data and apply steering changes. For example, in a phased computation, each phase would perform a single logical action across multiple processes. So, it is advantageous to be able to show the results of this action to the user as occurring simultaneously rather than piecewise across processes. A steering change across multiple processes then could be applied such that it takes place between the same phases at each process.

Events in this model consist of communication events, and the start and end of transactions. The latter mark the boundaries of the logical action the process is currently participating in. The communication events, send and receive, are generated automatically by a wrapper around the communication library. Information from the communication events is used to correlate the actions taken at different

processes. In the transaction model a send/receive pair always occurs within the same transaction.

We have developed three algorithms (comprehensive, exclusive, and selective [17]) that the ordering layer can use to recognize and order transactions. Each of the algorithms has different performance trade-offs, but they all have the same consistency guarantees. Thus any of the three algorithms can be used, the selection will depend upon the desired performance characteristics.

E. Agent Examples

The FREEDOM layer is a particular instantiation of an interaction layer, and consists of an architecture based on agents executing in milieux that provide certain services. The structure of this agent architecture and the services of the milieux are program model independent.

However, in the PAT implementation, we are aware that we are dealing with programs that may be modeled as a series of transactions. So, while the agent services and architecture remain the same regardless of the program model, the bodies of the agents may be written to take advantage of knowledge about the semantics of certain event types and of information about the attributes in the application. Thus, in the PAT implementation, the agents (and the groups they work in) can make use of the semantics of transaction events and utilize the consistency information available through the TRANSACTION LABELING layer.

This subsection gives examples of how agents can work to perform monitoring and steering actions. The first example is of a simple agent, monitoring a single attribute at a particular process. The other examples show how a group of agents can cooperate to perform user actions at multiple processes. Of these, the first are examples of cooperating monitoring agents. The last two examples describe how agents can be coordinated to perform consistent steering.

E.1 Monitoring

To introduce the use of agents for monitoring and steering we begin with a simple agent that collects the value of a given attribute at a particular process. To retrieve the value of 'X' at a particular process during any point in the computation the agent shown in figure 5 could be used. The agent's 'Arrival' handler is executed when it arrives at the milieu. It then simply reads the value of X from the avatar and returns to the *Snapshot Manager*, whose address would have been included when the agent was created.

The simple agent described above represents a one-time query, where the value at a process is polled. Typically though, agents are used to set up persistent queries that report information after the end of each transaction. In such cases the agent remains installed until it is explicitly removed. For efficiency, instead of having every agent installed at a milieu send an agent back to the *Snapshot Manager* separately (to report data), the agents bundle the information together into a single export agent, representing the state of the local process after the transaction

```

Arrival:
  $X = $avatar->{'X'};
  Migrate ($self, Snapshot Manager);

```

Fig. 5. Monitoring agent to retrieve a value and return to the *Snapshot Manager*.

```

gatherer
  End of Transaction:
    add X to the export agent

coordinator
  Arrival:
    send out gatherers
  Snapshot:
    if all present
      take average
      add to global snapshot

```

Fig. 6. Pseudo-Code for continuous monitoring agents (gatherer) and the agent to coordinate their data collection.

completed. The export agent then migrates to the *Snapshot Manager*.

Now, let us examine a more complex case. Suppose the user wants to find the average value of the attribute X, which is present in each process. One way to achieve this is to have a 'coordinator' agent at the *Snapshot Manager* send out a 'gathering' agent to each milieu. Each gathering agent then reports back the value of X after the end of each transaction. The coordinator agent then reads the reported value of X for each process and inserts the average into the global snapshot that is generated. See figure 6. Communication between agents at different milieux occurs via an agent migrating to the destination milieu. In this case, the export agent carries the information from the *Interaction library* to the *Snapshot Manager*.

Because of the unpredictability in message delivery times, it may be that some gathering agents begin reporting sooner than others. The coordinator can choose either to report an average which reflects only a subset of the processes or not to report anything until it has information from all processes. The advantage of simply not reporting the average until all of the values of X are known is that it gives the illusion that the query was enacted all at once, and avoids the situation in which presentation of partial information is misleading.

To ensure that the coordinator is seeing an accurate view of all of the X attributes the gathering agents include the local logical time at the process at the end of the transaction. The coordinator agent can then use the information provided by the ordering layer to make sure that the global view of Xs is consistent, i.e., that we have a correct average of the values.

Accessing consistency information is a frequent task that can be handled by the *Snapshot Manager*, rather than by individual agents. Typically, instead of waiting for indi-

```

changer
  Arrival:
    start logging
    make change and report

fixer
  Arrival:
    rollback
  End of Transaction:
    if time to apply change
      apply change
    clean up logging

coordinator
  Arrival:
    send changers
  Snapshot:
    if all changers have reported in
      if change ok then send cleanups
    else calculate and send fixers

```

Fig. 7. Pseudo code for optimistic agent consistent steering.

```

blocker
  Arrival:
    prevent initiation of transaction
    send ok to coordinator
    for each transaction
      send not ok to coordinator
      participate in transaction
      send blockers to each member of transaction
    send ok to coordinator

changer
  Arrival:
    remove blocker
    apply change

coordinator
  Arrival:
    send blockers
  Snapshot:
    if all are ok
      send changers

```

Fig. 8. Pseudo code for pessimistic consistent steering.

vidual agents to arrive, an agent at the *Snapshot Manager* will be activated by the preparation of a new global snapshot. It can then look for the information it needs amongst the export agents associated with the transaction between the current snapshot and the new one. This prevents the redundant calculation of consistency information.

In this next example of the use of monitoring agents, we wish to monitor for significant changes in the value X. Again, agents are sent to all of the processes. Once installed at their respective processes, the agents signal the coordinator that they are installed. Subsequently, they only send information when the value of X has changed by a “significant” amount from its most recent value, i.e., its value at the end of the previous transaction. If the export agent for a particular transaction does not contain information from the gathering agent, the coordinating agent at the *Snapshot Manager* knows that the value of X has not changed significantly.

E.2 Steering

Consistent steering requires that agents coordinate their actions. In the PAT implementation, several approaches to coordinated steering exist, implemented by groups of agents. Typically, a group of agents to carry out coordinated steering would not be written from scratch. Rather, groups of agents for steering would be created and stored in a library. These agents would then be supplied with appropriate input parameters to carry out the user’s steering action. Groups of monitoring agents are similarly available in libraries.

First, we present an optimistic steering approach. (See figure 7.) In this technique, we employ a ‘changer’ steering agent at each participating process, and a coordinator agent at the *Snapshot Manager*. Each changer agent ap-

plies the steering update, and reports to the coordinating agent the local transaction number at which the update was applied. The coordinating agent waits to hear from each of the changer agents. Based on the reported local transaction numbers, and information available from the ordering layer, the coordinator can determine if the steering update was applied consistently. If consistent, the processes may continue executing. If inconsistent, the coordinator calculates the local transaction number at which each of the steering agents should have acted, initiates a rollback, and the processes re-execute and apply the steering update at the appropriate time. Implementation of this technique requires logging and rollback facilities.

The main assumptions of this approach are that most steering actions will be consistent, and that such coordinated steering will be employed rarely. The overhead of state saving and logging will be incurred for each steering action of this type. However, rollback and re-execution, although expensive, should be rare. The benefit of this approach is that the application need not be halted at a barrier-type synchronization point in order to achieve consistent steering. This optimistic scheme is appropriate when steering a small subset of the application processes. The greater the number of processes that must be coordinated, the more likely that rollback will be required.

Another approach to coordinated steering, which is more pessimistic, is to block application processes until it is safe to apply the steering change. The processes at which the steering change is to be made are asked to block, by the ‘coordinator’ sending a ‘blocker’ agent to those processes. (See figure 8.) A process may not be able to stop immediately because another process, to be blocked, may need the process to participate in a transaction so that it can reach

the end of its own current transaction, and stop. Hence, processes are permitted to participate in transactions if forced to, i.e. they receive a message, but they may not initiate any new transactions. Once all of the needed processes are blocked, then the coordinator sends a 'changer' agent to remove the blocker and apply the steering change. The blockers are also removed from other processes which became blocked.

While this group of agents works for some applications, this steering approach is not appropriate for all computations. It can cause all of the processes in a computation to become blocked before the steering change can be made. It also has the potential to cause deadlock in some computations.

IV. CONCLUSIONS

The ability to steer an application in an interactive and consistent way is an important ability. There is no consensus though as to the best way to achieve it.

We have outlined the PathFinder architecture for the monitoring and steering of distributed computations. PathFinder is designed to support configurable monitoring and steering, highly dynamic interactions, and consistent steering, coordinated across multiple processors. PAT, an implementation of the PathFinder architecture uses agents to perform the monitoring and steering functions, and relies on transaction events from the application for consistency calculations.

Our future work will use the PAT configuration for the exploration of different steering techniques, the investigation of the benefits of an agent-based monitoring/steering framework and the development of libraries of agents for consistent monitoring and steering.

REFERENCES

- [1] A.J. Winfield, "A virtual laboratory notebook for simulation models," in *Proceedings, Pacific Symposium on Biocomputing '98*, Maui, HI, 1998, pp. 177-188.
- [2] D.M. Beazley and P.S. Lomdahl, "Extensible message passing application development and debugging with python," in *Proceedings, International Parallel Processing Symposium (IPPS'97)*, Geneva, Switzerland, 1997.
- [3] Greg Eisenhauer and Karsten Schwan, "An object-based infrastructure for program monitoring and steering," in *2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, Welches, Oregon, USA, August 1998.
- [4] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "CUMULVS: Providing fault-tolerance, visualization, and steering of parallel applications," *SIAM*, Aug. 1996.
- [5] J. Vetter and K. Schwan, "High performance computational steering of physical simulations," in *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.
- [6] M. Oberhuber, S. Rathmayer, and A. Bode, "Tuning parallel programs with computational steering and controlled execution," in *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*.
- [7] S.G. Parker and C.R. Johnson, "SCIRun: A scientific programming environment for computational steering," in *Supercomputing '95*, 1995.
- [8] D.M. Beazley, "SWIG: An easy to use tool for integrating scripting languages with C and C++," in *Proceedings of the 4th USENIX Tcl/Tk workshop*, 1996, pp. 129-139.
- [9] D. M. Beazley and P. S. Lomdahl, "Building flexible large-scale scientific computing applications with scripting languages," in *8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, Minnesota, March 1997.
- [10] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera, "An overview of the Pablo performance analysis environment," in *Proceedings of the Scalable Parallel Libraries Conference*, Mississippi State, MS, USA, Oct. 1994, pp. 104-113.
- [11] Greg Eisenhauer, Weiming Gu, Karsten Schwan, and Niru Mallavarupu, "Falcon - toward interactive parallel programs: The On-line steering of a molecular dynamics application," in *Proceedings of High Performance Distributed Computing (HPDC-3)*, Aug. 1994.
- [12] J. Vetter and K. Schwan, "Progress: a toolkit for interactive program steering," in *Proceedings of the 24th International Conference on Parallel Processing*, Urbana, IL, 1995, pp. 139-142.
- [13] D. Jablonowski, J. Bruner, B. Bliss, and R. Haber, "VASE: The Visualization and Application Steering Environment," in *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993, pp. 560-569.
- [14] Delbert Hart, Eileen Kraemer, and Gruiia-Catalin Roman, "Using snapshot streams to support visual exploration," Tech. Rep. WUCS97-46, Washington University in St. Louis, 1997.
- [15] R. Schwartz and L. Wall, *Programming Perl*, O'Reilly and Associates, 1994.
- [16] Leslie Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, July 1978.
- [17] Delbert Hart, Eileen Kraemer, and Gruiia-Catalin Roman, "Query-based visualization of distributed computations," in *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

Delbert R. Hart is completing his D.Sc. in Computer Science at Washington University in St. Louis. His research interests include the monitoring and steering of distributed systems and software visualization. He received his M.S. in Computer Science from Washington University in St. Louis (1995) and his B.A. in Computer Science and Mathematics from SUNY Plattsburgh (1993).

Eileen T. Kraemer is an Assistant Professor of Computer Science at Washington University in St. Louis. Her research interests include parallel and distributed systems, software visualization, and tools for computational biology. Dr. Kraemer received the Ph.D. in Computer Science from the Georgia Institute of Technology (1995), the M.S. in Computer Science from Polytechnic University, Brooklyn (1986), and the B.A. in Biology from Hofstra University (1980). She is a member of ACM

and IEEE.