Using Snapshot Streams to Support Visual Exploration

Authors: Delbert Hart, Eileen Kraemer, and Gruia-Catalin Roman

The non-determinism, complexity, and size of distributed software systems present significant difficulties for designers and maintainers. Visualization can help alleviate these difficulties through interactive exploratory tools that allow both novice and experienced users to investigate a distributed computation using a common tool set. Essential to the success of a visual exploration tool is the ability to provide accurate representations of global states. This paper is concerned with the use of snapshots in support of interactive visual exploration of distributed computations. The nature of the visualization process requires snapshots that (1) are consecutive, thus facilitating smooth animation of state changes, (2) vary in content and scope in response to changes in the viewer's interest, and (3) can be generated on demand with minimal impact on long-lived computations that may be monitored only occasionally. The paper describes a monitoring approach that satisfies these requirements and several algorithmic variations. A distributed ray tracer serves as an example.

... Read complete abstract on page 2.
Using Snapshot Streams to Support Visual Exploration

Complete Abstract:

The non-determinism, complexity, and size of distributed software systems present significant difficulties for designers and maintainers. Visualization can help alleviate these difficulties through interactive exploratory tools that allow both novice and experienced users to investigate a distributed computation using a common tool set. Essential to the success of a visual exploration tool is the ability to provide accurate representations of global states. This paper is concerned with the use of snapshots in support of interactive visual exploration of distributed computations. The nature of the visualization process requires snapshots that (1) are consecutive, thus facilitating smooth animation of state changes, (2) vary in content and scope in response to changes in the viewer’s interest, and (3) can be generated on demand with minimal impact on long-lived computations that may be monitored only occasionally. The paper describes a monitoring approach that satisfies these requirements and several algorithmic variations. A distributed ray tracer serves as an example.
Using Snapshot Streams
to Support Visual Exploration

Delbert Hart
Eileen Kraemer
Gruia-Catalin Roman

WUCS97-46

October 12, 1997
Abstract

The non-determinism, complexity, and size of distributed software systems present significant difficulties for designers and maintainers. Visualization can help alleviate these difficulties through interactive exploratory tools that allow both novice and experienced users to investigate a distributed computation using a common tool set. Essential to the success of a visual exploration tool is the ability to provide accurate representations of global states. This paper is concerned with the use of snapshots in support of interactive visual exploration of distributed computations. The nature of the visualization process requires snapshots that (1) are consecutive, thus facilitating smooth animation of state changes, (2) vary in content and scope in response to changes in the viewer's interests, and (3) can be generated on demand with minimal impact on long-lived computations that may be monitored only occasionally. The paper describes a monitoring approach that satisfies these requirements and several algorithmic variations. A distributed ray tracer serves as an example.

Keywords: program visualization, snapshots, distributed computing

Acknowledgement: This paper is based upon work supported in part by the National Science Foundation under Grant No. CCR-9217751 and Grant No. CDA-961931. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Correspondence: All communications regarding this paper should be addressed to

Dr. Eileen Kraemer eileen@cs.wustl.edu
Department of Computer Science office: (314) 935-6621
Washington University in Saint Louis secretary (314) 935-6160
Campus Box 1045 fax: (314) 935-7302
One Brookings Drive
Saint Louis, MO 63130-4899 http://swarm.cs.wustl.edu/cgi-bin/papers.cgi
1 Introduction

As the size and speed of networks has increased, long-lived, distributed applications involving numerous nodes and multiple participants have become increasingly prevalent. At the same time, the need has arisen for tools to assist in the design, maintenance, and management of such systems. However, these tools must meet requirements beyond those addressed by traditional tools for observing, controlling, and debugging software systems; they must be designed to handle the non-determinism, complexity, and size of long-running, highly distributed applications executing in a heterogeneous environment and exhibiting highly dynamic execution behavior.

While traditional debugging tools seek to provide a controlled, observable execution through the use of breakpoints coupled with the textual or graphical display of selected values, the application of this tightly controlled model to highly distributed and long-running applications is overly intrusive, and typically too closely bound to a specific execution environment to be appropriate in the type of heterogeneous, distributed, dynamically changing environment that we address. Similarly, traditional monitoring and visualization techniques, in which the monitored program attributes and their visual representations are statically specified by the user at the start of a monitoring and visualization session, fail to meet the needs of those who must design, maintain, and manage long-running, complex, distributed applications.

In our work, we seek to develop a new family of tools that address these issues of distributed computations, based on the concept of an exploratory environment that treats a computation as a dynamic artifact to be examined for various purposes throughout its lifetime, with minimal interference into its working and with limited impact on its performance. This exploratory environment is characterized by interactive, dynamic specification of both the program attributes to be monitored and controlled and the visual representations of these attributes, accomplished through direct interaction with graphical representations of program entities. Such an approach provides a natural and appealing interface to the user who must understand, maintain, or control the computation.

Tools for this environment address two primary concerns: 1) helping the user to manage and navigate through the vast amount of information that complex, long-running distributed computations may produce, and 2) minimizing the impact of observation and interaction on the underlying computation. Abstraction plays a key role in ensuring that both of these concerns are met. Traditional software visualization uses abstraction to represent programs, typically through a single abstraction represented in each window of the visualization, with the set of abstractions in use chosen before the visualization session begins. Exploratory visualization permits users to dynamically change the abstraction and view the computation from a different perspective and gain new insights. The user modifies the abstraction in use by entering navigation commands based on the current visualized scene. Many different abstractions exist that may be useful in examining the behavior of a running computation, each one a summary that emphasizes aspects of interest and elides irrelevant information.

Abstraction techniques include selection, choosing some subset of available attributes, and aggregation, the derivation of higher order attributes, such as totals, averages, or other functions applied to a group of primitive or aggregated attributes. These powerful techniques may be applied across multiple dimensions, in what we term data abstraction, spatial abstraction, and temporal abstraction. Data abstraction involves the selection or aggregation of the attributes available at a particular process at a particular time. Spatial abstraction involves the selection or aggregation of attributes available at multiple processes at a particular time. Temporal abstraction involves the selection or aggregation of attributes available across multiple time-steps of the computation. The user interacts with graphical representations of the program state to navigate through the state space of the computation as it changes over time. These interactions are translated by the exploratory tool into a query that evaluates to an appropriate combination of abstraction mechanisms. Note that because the desired level of abstraction can be determined from the user's specification (series of interactions), this knowledge of what is to be displayed can be used to limit what is collected, with the felicitous result that the exploratory tool can collect the minimum amount of data necessary to present the desired abstraction, and minimize
perturbation of the underlying program.

The exploratory approach to understanding and debugging complex distributed applications requires continuous, selective monitoring, the ability to determine globally consistent states, flexible and informative visualizations, and the ability to limit perturbation. Continuity in monitoring prevents gaps in the user's view of the computation, and facilitates the creation of animations, while selectivity permits the tool to handle long running computations and minimize the impact of monitoring. Flexible, informative visualizations have the potential to communicate information quickly and intuitively, and to present graphical representations of a variety of abstraction levels and types. The ability to determine consistent states serves to avoid misleading representations in these visualizations. Excessive perturbation results in tools that are unusable, if not unusable. The perturbation that results from monitoring should depend only on the amount and kind of data being visualized. Finally, implementation of the abstraction techniques of selection and aggregation, along the data, spatial, and temporal dimensions, must be supported by underlying mechanisms.

The strategy we use in this paper is to base exploratory visualization on a stream of snapshots, whose scope may evolve from one snapshot to the next. We refer to these snapshots as evolving snapshots, a sequence of which forms a snapshot stream. To define the data the snapshots contain and when snapshots should be taken, we use queries, derived from the user's interactions throughout an exploratory session. Based on the current queries, an evolving snapshot is created, containing only the needed data. Before presenting a snapshot to the user, its ordering relative to other snapshots must be determined so that it can be visualized at the proper time. The distributed nature of the computation and the fact that we are collecting only a subset of the computation make determining a proper order for these snapshots a challenging task.

This paper describes several algorithms that accomplish this dynamic and lightweight monitoring of distributed computations. Their novelty rests with the fact that they permit the range of processes contributing to the reconstruction of global snapshots to vary rapidly over time. Furthermore, they are designed specifically for the kind of continuous monitoring needed to support animations. To accomplish this, the algorithms presented in this paper assume that distributed computations can be structured in a manner that provides the illusion of a nested transaction system. This is particularly useful when an application may be examined at varying degrees of granularity during its execution. Transaction nesting also supports the implementation of a useful form of temporal abstraction. The combination of an underlying model that supports temporal abstraction (transaction-based) and the specific need to support visualization lead us to new kinds of solutions for a classic problem in distributed computing. This is the main contribution of this paper.

The remainder of this paper is organized as follows: Section 2 provides a formal characterization of the distributed computing model assumed in this paper. Section 3 describes three algorithms for generating continuous snapshots and a discussion of the factors that influence the choice of an appropriate algorithm for a particular application. Section 4 describes how a prototype of the exploratory visualization approach was used to look at a distributed ray tracer. Section 5 discusses related work. Section 6 provides some brief concluding remarks.

2 Computational Model

Our ultimate vision of an exploratory environment for distributed computations is one in which multiple independent users employ visual metaphors to pose questions about a running computation with the answers being reported in the form of continuous animations that encode relevant changes taking place in the computation's state. The kinds of applications that are of interest to us are long lived and involve significant numbers of concurrent processes located on various nodes across a network solving large computational problems or supporting the activities of a distributed enterprise of some sort. While interprocess communication occurs via message exchanges, the application developers and users alike assess progress in terms of global changes in the state of the system rather than relative to low level communication activities. The latter are of no particular interest to users who may not even be aware of the manner in which com-
munication takes place. A bank transaction, for instance, is perceived as a change in the values stored in one or more accounts and not as an exchange of messages. Similarly, in multi-phased computations the emphasis is on understanding what has been accomplished at the end of each phase. Finally, past experience with visual presentation of distributed computations (e.g., termination detection) has taught us that often the presence of errors becomes evident only when one examines high-level abstract properties of the system rather than its low level communication behavior or computational mechanics. For these and other reasons, we believe that the ability to present a viewer on demand with accurate abstract views of global system state is critical to the development of tools supporting dynamic exploration of distributed computations. Other desirable features have to do with providing support for interactive changes in the focus of attention, navigation among levels of abstraction, tracking system changes along the execution timeline, and interactive steering. This last capability is the key for exercising interactive control over the computation and for permitting the user to take corrective action without restarting the application. However, in this paper we focus on monitoring and navigation and leave a discussion of steering algorithms and capabilities to another paper.

In the remainder of the paper we will assume that the computation is structured in terms of a dynamic set of processes that have been instrumented so as to export (if and when it becomes necessary) selected aspects of their local states in the form of attributes which have names and continuously changing values. The attributes of interest to a specific exploratory task are specified by means of queries over the state of the computation. Queries may be provided using a textual or visual language but the only thing of interest to us here is the fact that eventually all relevant processes become aware of the need to supply the values of certain attributes on a continuous basis or to terminate such monitoring as the case may be. Since queries may be issued and retracted concurrently by multiple programmers, the set of processes and attributes being monitored changes over time in unpredictable ways. We chose to use queries as the basis for constructing an exploratory environment because early on we decided to view a network application as a distributed database whose contents is subject to continuous change. Furthermore, queries facilitate both flexibility and generality.

Of course, the contents of a traditional database changes in response to the execution of serializable transactions. This is not usually the case with a distributed computation. Yet this assumption is at the foundation of our monitoring strategy. While average applications are not generally written as transaction systems, they can often be restructured along these lines. Our strategy, however, is much less expensive and less intrusive. It entails a simple set of code annotations to mark completed message exchanges among coordinating applications in the local code of the individual processes. The annotation mechanism does require a good understanding of the application program which may be intellectually expensive but has the potential for automation through static dataflow analysis techniques. Furthermore, the annotation can be easily extended to nested transactions. We will return to the annotation procedure in a later section, but for now it suffices to state that coordinated state changes among cooperating processes can be abstracted as atomic transactions over the global state of the computation but implemented as finite message exchanges among two or more processes. In the remainder of the section we provide precise definitions of all the concepts mentioned above.

### 2.1 Definitions

A distributed computation consists of a set of processes that work together to achieve a common goal. Each process exports a set of attributes that reflect the state of the process. The process's state changes when an event occurs at the process. The event sequence is a history recording the changes undergone by the process. An event \( e \) is characterized by:

- \( \pi.e \): the process with which this event is associated.
- \( \sigma.e \): the state of the process immediately after \( e \) occurs.
- \( \delta.e \): a local sequence number that reflects the event's location in the process's history of events.
- \( \tau.e \): an event type indicating the nature of the event (i.e., send, receive, mark or local).
Figure 1: A portion of a well-formed computation. White circles represent application events, including sends and receives. Black circles represent mark events. Shaded areas represent transactions. The top two transactions are primitive transactions. The dotted line around the transactions represents a compound transaction that contains them.

A *local* event represents a state transition within a single process. A *mark* event indicates that the process has completed its participation in the current transaction. A matching *send/receive* event pair is called a communication. In addition, the events *init*, *start* and *stop* (special instances of the events *send*, *receive* and *mark*) denote a request to create a process, the start of a process, and the termination of a process, respectively. The happened-before relation [1] over events is a partial order where an event $x$ happened-before an event $y$ if and only if 1) $x$ and $y$ are in the same process and $\delta . x < \delta . y$, 2) $x$ is a send event that matches $y$, or 3) there exists an event $z$ such that $x$ happened-before $z$ and $z$ happened-before $y$.

We then view the distributed computation as a set of events, with a partial order corresponding to the happened-before relation, and an equivalence relation that captures the notion that two events are part of the same transaction. A distributed computation is well-formed (i.e., an appropriate model for transaction-based monitoring) if it satisfies the following properties:

C-1. each process is sequential, it is characterized by a total ordering of events

C-2. processes interact only through message-passing

C-3. every send event has a corresponding receive at some other process

The events of a well-formed computation can be partitioned by defining an equivalence class, a *transaction set*. A *transaction set* can then be used as the nodes of a *transaction graph*. Each transaction in a *transaction set* must satisfy the following properties:

T-1. the send and receive events of a communication belong to the same transaction

T-2. for any two events $a$ and $b$ in a transaction, there must exist a sequence of events from the transaction starting with $a$ and ending with $b$ such that all consecutive events in the sequence are related by the happened-before relation.

T-3. if an event $e_1$ in transaction $t_1$ is before an event $e_2$ in transaction $t_2$, then there can be no event in $t_2$ before any event in $t_1$

Computations that satisfy the above properties permit the calculation of a partial ordering of the members of the *transaction set* such that transaction $a$ happened-before transaction $b$ if
and only if there is an event in a that occurred before some event in b. We can now define a transaction graph for a computation C as $G = (N, E)$ where N is a transaction set over C and E is the transaction happened-before relation.

Transactions form the boundaries in the process executions that we will use to construct consistent global snapshots. Figure 1 shows an example of three transactions. The top two transactions are called primitive transactions. A primitive transaction is a transaction whose set of events cannot be partitioned into two or more transactions. Transactions whose set of events can be partitioned into multiple transactions are called compound transactions. The dotted line around the transactions in figure 1 represents a possible compound transaction. At the end of the next section we will look at how nested transactions can be used. Until then we assume that the transaction set chosen contains only primitive transactions.

3 Algorithms

The algorithms in this section use the computational model described in the previous section to capture consistent and pertinent states of the computation. Transactions define locations in the code at which data is to be collected and how the information collected across processes is to be assembled into a global view. The pertinence of such a global view depends on the user’s current focus; by using a nested-transaction model the user can choose which behaviors to observe. The user selects these behaviors by choosing a subset of the available transactions. With these algorithms in hand it is possible to construct an exploratory visualization tool.

In the following subsection we discuss what the algorithms are to accomplish and the elements common to all three algorithms. The following subsections describe the comprehensive, selective, and exclusive snapshot algorithms. We wrap up the section with a look at how the use of nested transactions affect the algorithms.

3.1 Algorithm Framework

The goal of each snapshot algorithm is to produce a stream of snapshots, with the scope of each snapshot based on the user's current interests. In general, it is not possible to always base a snapshot on the user’s current interest, due to communication delays between the user and the application being studied. The communication delay is referred to as lag, and it is one of the characteristics relevant when choosing a snapshot algorithm to monitor an application. Other characteristics that affect the decision of which algorithm to use are consistency, local perturbation, and scalability. The three algorithms described below all ensure consistency, but they vary in the other characteristics.

To produce a snapshot stream, a snapshot algorithm must fulfill three goals: it must determine transaction membership, determine transaction ordering, and collect local snapshots. The transaction membership is found by deciding which local snapshots were taken at the end of the same transaction. Transaction membership must be known in order to calculate a consistent global view. For instance if a transaction were mistaken for two or more transactions, then the effects of that transaction would not be seen simultaneously. Transaction ordering involves ensuring that the total ordering of snapshots present in the snapshot stream is consistent with the partial order over transactions. Transaction ordering is important because the visualization tool should display a correct sequence of global states, otherwise the behavior of the visualization would not be representative of the behavior of the application.

All messages sent by a process are tagged with the process’s current local transaction id, which is a pair $< P, t >$ where P is the process’s id and t is the sequence number of the last mark event. A global transaction id consists of the set of local transaction ids of the processes that participated in the transaction. A local transaction id can uniquely identify the global transaction to which it belongs.

Each algorithm has two components: a monitoring library that reports information about the application process, and a separate snapshot manager process that collates local snapshots to form the snapshot stream. The monitoring library is present in all application processes and
sends the local snapshot and necessary transaction information to the snapshot manager. In the remainder of this section, the term application process will refer to the monitoring library at an application process. Communication events between a process and the snapshot manager are not considered to be events of the computation. We assume that message delivery is reliable and FIFO, and that each process has a unique id.

Monitored data is data that the user has expressed interest in, usually through the use of a query. Neither the type of queries used nor the amount of data monitored at each process directly affect the algorithms used. For now we will assume that processes can dynamically decide whether or not they are being monitored. This is a generalization of the typical case in which a process, in response to a message from the user, decides if it is monitored. A process is monitored at the end of a transaction if the user has expressed interest in observing the effect of the logical action that the process participated in. A transaction is monitored if it contains a monitored process.

The three algorithms described below differ primarily in the component at which the transaction membership and transaction ordering tasks are carried out. In the comprehensive algorithm, both tasks occur in the snapshot manager, simplifying the monitoring libraries. In the selective algorithm, transaction membership is determined by the application processes and the snapshot manager decides the transaction ordering. The exclusive algorithm distributes both the membership task and the transaction ordering task among the application processes. In all three algorithms, when multiple consistent orderings exist, the snapshot manager selects the order that transactions are presented in.

3.2 Comprehensive Monitoring Algorithm

The first algorithm that we present is the simplest of the three. While all of the algorithms provide consistent global views for any well-formed distributed computation, the choice of which one to use will depend on the performance characteristics desired. The comprehensive algorithm works best for small computations or for situations in which minimizing the lag between the application and the user is a high priority. In the comprehensive monitoring algorithm, all processes are treated as being monitored.

If the snapshot manager knows of all the events that occurred in the computation, then from this global information it can reconstruct the transaction membership and ordering. This could be done by having a process report an event to the snapshot manager immediately after it occurs. If the event information is buffered at the process until a mark event, it does not affect the amount of lag, and it reduces the number of messages that need to be sent to the snapshot manager. The snapshot manager stores each message in a queue corresponding to the process that sent it, until the message is used to identify a transaction.

A process is a neighbor of any process that it communicates with during a transaction; the neighbor relation is specific to the transaction in which the communication occurred. The only information that a process needs to include in the message to the snapshot manager is who its neighbors are and any of its neighbors' local transaction ids that it knows of.

Transaction membership can be determined by choosing a process P, and taking the transitive closure over the set of neighbors, starting with P's neighbors.

1. The snapshot manager chooses a process P, whose queue is not empty. We call the local transaction id at the head of the queue t.

2. A partial global id, GID, is set to contain P's transaction id. A set of local transaction ids, LIDS, is initialized to contain P's transaction id and transaction ids for P's neighbors. If a neighbor's transaction id is not known, then it is set to a special value ⊥, which matches any value.

3. The snapshot manager performs the following loop until there are no unknown values in LIDS.
   (a) Choose a process, X, that is represented in LIDS.
   (b) If the head of X's queue is empty, then start over at step 1.
(c) Check the transaction id of the message at the head of X’s queue. If it does not match X’s transaction id in LIDS or if any of the neighbor information in the message does not match the transaction ids in GID, then start over at step 1.

(d) Add X’s transaction id to GID.

(e) Add the neighbor information from the message to LIDS.

4. GID is now a complete global transaction id. The snapshot manager dequeues the messages at the head of all of the transaction’s members so that the next global transaction can be computed.

5. The GID can be used to look up the local snapshots that need to be accessed to modify the current global snapshot into a new global snapshot.

Intuitively we can see that this algorithm won’t miss a process X, because X must have communicated with someone else, let’s say process Y, so Y will know that X must be in the transaction. Using the same reasoning, we can form a chain of processes from X to F, the process initially chosen by the snapshot manager. The ordering of snapshots is correct because of the FIFO communication to the snapshot manager and the queues maintained at the snapshot manager.

3.3 Selective Monitoring Algorithm

The comprehensive algorithm has the advantages of a relatively low amount of lag and that it is simple. It has the disadvantages that every process, monitored and unmonitored acts alike and that the snapshot manager can become a bottleneck. The comprehensive algorithm works well for small computations, but for larger computations it is useful to distribute more of the work to keep the snapshot manager from becoming overwhelmed with the number of messages sent to it. The number of messages can be reduced, especially when the snapshot manager is monitoring only a subset of the processes in the computation. However, simply eliminating the messages from the unmonitored processes is not a viable alternative.

Figure 2 shows an example of a case where simply eliminating messages from unmonitored processes would lead to an error. There are six processes, A-F. Processes A, C, and F are the only ones monitored. Two problems are illustrated by figure 2. The first one is that transaction membership cannot be determined if only the monitored processes report to the snapshot manager. In transaction t3, the snapshot manager has no way of knowing that processes C and F are part of the same transaction. Creating a snapshot stream that is consistent with the partial order over transactions may also be impossible. This is illustrated in the figure by transactions t1 and t3. Since transaction t2 does not report, because it has no monitored processes, the snapshot manager cannot know that transaction t1 should occur before t3 in the snapshot stream, thus it may choose to order them incorrectly.

In order to effectively monitor large computations, the selective algorithms shift some of the burden of calculating transaction membership to the application processes. The processes compute the transaction membership amongst themselves, and then forward the transaction membership information to the snapshot manager. The snapshot manager then receives only messages containing local snapshots and one message per transaction. When an application process sends a message, it contains a partial global transaction id containing the ids of the processes with a higher id than and including itself, the neighbor information of those processes, and whether or not they are monitored.

1. wait for messages until we have transaction information about all of our neighbors with a higher id

2. send a message with all of the gathered information to the process in the transaction with the highest id lower than the local id. If there is no process with a lower id that participated in the transaction, then this process is the transaction leader, and it will send the information about the transaction to the snapshot manager.
Figure 2: A, C, and F are monitored, consequently transactions t1 and t3 are monitored. Information about t2 is needed to order t1 before t3. Determination of t3's membership requires that the snapshot manager directly or indirectly receives information from D and E.
Once the transaction leader reaches step two it knows the membership and who is being monitored in the transaction. It forwards this information to the snapshot manager who uses it to know the transaction membership and from whom to expect data messages. At the snapshot manager, a set of transaction memberships is maintained and a vector clock representing the time of the last snapshot is inserted into the snapshot stream. To derive a new snapshot, the snapshot manager chooses a transaction whose vector time is exactly 1 greater than the current vector time for every process in the intersection of the transaction membership and the last snapshot. If all of the expected data is present for this transaction, then it creates a new snapshot by replacing the local snapshots of the members of the transaction with the local snapshots in the data messages.

Although the labeling protocol blocks at a process on step 1, it does not block the application from proceeding. When a process finishes its participation in the transaction, it can store the information needed for this protocol and proceed with the application. So, an application monitor may actually be participating in several transaction protocols, each of them independent of each other. The monitoring library can also forward the monitored data to the snapshot manager immediately, so that it does not need to store it at the application.

To see that the application monitors correctly determine the transaction membership, it is useful to construct a tree for the transaction. Each process in the transaction has a node in the tree, and initially there is an edge between two processes if they communicated during the transaction, the process with a higher id points to the one with the lower id. Clearly there is some process with a highest id in the tree and eventually it will finish the transaction. Once this occurs, by action 2 it will send to the process it knows of with the next highest id and remove itself from the tree. When the process receives the new information it integrates it with its own, possibly adding edges to processes with lower ids. Eventually the tree will be reduced to a single process, the transaction leader, who will forward the information to the snapshot manager. Figure 3 shows an example of how the tree changes as the protocol runs.

3.4 Exclusive Monitoring Algorithm

The selective algorithm reduces the degree to which the snapshot manager is the bottleneck, but it potentially increases the amount of lag by the amount of time required to send a message to each process in the transaction and have it forward information to the next process. This algorithm reduces the number of messages to the snapshot manager from one from each process
in the transaction to one from each monitored process (plus one). The selective algorithm is useful in many cases, but for very large computations the snapshot manager may still be too much of a bottleneck, especially if only a small number of transactions are being monitored relative to the size of the computation.

The selective monitoring algorithm sends the snapshot manager a message for each transaction regardless of whether or not the transaction is monitored. In order to reduce the load on the snapshot manager, the selective monitoring algorithm can be changed so that only monitored transactions send a message. In order to accomplish this, the monitoring libraries must track which monitored transactions must precede them. By having the processes do their own dependency tracking, a single message per monitored transaction is sufficient as long as it contains the transaction membership, which processes are monitored, and which monitored transaction the transaction depends upon.

The exclusive monitoring algorithm represents an extension of the selective algorithm, in which the dependencies that each process in the transaction has are included in the message sent to other monitoring libraries. Once the transaction leader has all of the information, it transmits the dependency information to all unmonitored processes for use in the labeling of subsequent transactions. Monitored processes already know that future transactions they participate in will depend on the current transaction, and can identify the current transaction uniquely with their local transaction id.

1. wait for TLP messages until we have transaction information about all of our neighbors with a higher id and dependency information from the transaction leader of the last transaction, if we were not monitored.

2. send a TLP message with all of the gathered information to the process in the transaction with the highest id lower than the local id. If there is no process with a lower id that participated in the transaction, then this process is the transaction leader, and it will send the information about the transaction to the snapshot manager.

3. the transaction leader sends the dependency information to all unmonitored processes.

When the snapshot manager wants to choose a new transaction, it checks to make sure that all of the monitored transactions that it depends upon have already occurred in the snapshot stream.

The membership determination is unchanged from the selective algorithm. The correctness of the snapshot stream ordering can be seen simply from the fact that the snapshot manager won't insert a transaction until all of its dependencies have been satisfied, i.e. already appeared in the stream. The dependencies are determined correctly by the transaction leader because it simply does a union over all of the dependencies of the individual processes of the transaction; step 1 ensures that the labeling protocol will not proceed until it has the current dependency information.

The exclusive algorithm is able to better handle computations that are large, and in which only a small number of processes are being monitored. Lag is higher in this algorithm because unmonitored processes must wait for a message from the transaction leader before proceeding with the protocol for the next transaction. The application is not delayed by the change in the transaction labeling protocol.

3.5 Nested Transactions

Transactions represent the occurrence of logical actions in the computation, thus presenting good locations at which to observe the computation. In order to support temporal abstraction, the algorithms must efficiently support different transaction granularities. In order to take global snapshots at varying levels of temporal granularity, we allow transactions to be nested. The freedom to choose the granularity of time-steps is complementary to the ability of the user to choose which data to monitor. It allows the monitoring of only those aspects of the computation's behavior that are of current interest to the user.
While presenting the snapshot algorithms, we limited the transaction set to contain only primitive transactions. We now allow the transaction set to contain compound transactions. The transactions that make up a compound transaction are called sub-transactions. This subsection describes modifications that can be made to support multiple levels of transaction granularity.

Although the choice of which transactions to monitor is made by the user, the nesting of transactions is embedded in the computation. Recall that a transaction corresponds to some logical action made by the application, so sub-transactions are merely logical component actions that are taken to accomplish the composite logical action associated with the compound transaction. Mark events in the computation indicate the end of a primitive transaction, but may be contained within a compound transaction. When supporting nested transactions, a different implementation of logical time is needed. A local transaction id was defined by the process's id and the sequence number of the mark event that preceded the current transaction. With nested transactions, there are multiple current transactions, so the local transaction id becomes a pair containing the process id and an array of mark sequence numbers. Each element of the array holds the sequence number of a mark event that preceded the current (compound or primitive) transaction. The array is ordered from largest transaction to smallest, so the sequence number of the mark event that ended the most recent primitive transaction is always the last element of the array. This more sophisticated local transaction id is used to communicate the transaction nesting information to both the snapshot manager and the process's neighbors. Figure 4 shows an example of how the local transaction id is structured.

The simplest way to support multiple transaction granularities is to use the algorithms after each primitive transaction. The snapshot manager can then use the information embedded in the local transaction ids to reconstruct the nesting information and suppress the insertion of snapshots into the snapshot stream that correspond to transactions that the user is not interested in. The snapshot manager ensures that the resulting snapshot stream is consistent with the happened-before relation over the transaction set used.

If the user is only interested in compound transactions, then the above approach may generate
an unnecessarily large number of messages. The messages sent by each of the algorithms can be delayed until either a monitored transaction has occurred or some other condition becomes true. Delays in the comprehensive algorithm simply mean that a compound message containing the delayed messages is sent to the snapshot manager. Delaying messages in the selective or exclusive algorithms means that again a compound message is sent, but step one must be satisfied for all messages that are sent, and the message is forwarded to the process with the highest id from all of the messages’ information. For example, in figure 4 t1 is a monitored transaction which has two sub-transactions t2 and t3, both of which are unmonitored. In this case the labeling protocol would not be run for either t2 or t3, only for t1.

Intuitively, the correctness is not affected because we are only affecting the time at which information arrives at the snapshot manager/monitoring library. The trade-off in deciding when to run the snapshot algorithms is between how quickly the user learns about fine-grained transactions and in how much network overhead is incurred.

4 Global Surveyor

Our current prototype is Global Surveyor, an exploratory tool based on the Query-Based Visualization (QBV) model[2]. Global Surveyor was designed to demonstrate the ability to interactively observe running distributed computations. This first prototype focuses on the general use of exploration and how evolving global snapshots can be efficiently collected to support exploration. Later prototypes will incorporate more sophisticated data extraction, steering, data filtering and user interaction into the exploration process. This section describes the organization of Global Surveyor and then illustrates its use with the example exploration of a distributed ray tracer.

4.1 Architecture Overview

In this section we distinguish between the programmer who inserts annotations into a distributed computation’s source code and a viewer who explores an already annotated computation. In some cases, this may be the same person. In other cases, in which the source code has already been annotated, it may not be. The annotation process is not implicitly part of exploration, but source code annotation is the way that transaction boundaries and application-specific attributes are defined in Surveyor. Figure 5 shows the logical organization of Surveyor. The viewer begins exploring by selecting a running computation through Surveyor’s user interface. A snapshot manager is then started, which in turn contacts the monitoring libraries at each of the application processes to find out what attributes are available for visualization. The snapshot manager reports to the viewer the identities of the available processes and attributes. The viewer then specifies the areas of interest by submitting a query to the snapshot manager. The snapshot manager decodes the query and sends out attribute subscription requests to the monitoring libraries. As local snapshots arrive, the snapshot manager synthesizes consistent global snapshots that are inserted into a snapshot stream. The stream provides data for visualizations that the viewer has launched. The monitoring libraries are able to handle requests from different snapshot managers, so it is possible for multiple viewers to independently view the computation. Monitoring can begin at any time while the computation is running. There is little perturbation of the computation while the computation is not being monitored.

Global Surveyor is able to monitor running distributed computations that use the PVM[3] library for message passing. The Surveyor’s monitoring libraries can be used in C, C++, and Fortran programs. We do not need to modify the PVM library to support Surveyor; instead, Surveyor’s monitoring library intercepts calls made to the PVM library.

**Monitoring Library**: Data about a program can be extracted in many ways. It is possible to extract data from the runtime system, a custom library or through debugging hooks. An overview of data collection methods can be found in [4].

Two types of attributes are made available by Surveyor’s monitoring library: common attributes and application-specific attributes. Common attributes are characteristics of the application process that are possessed by all processes. Examples of common attributes are general
process characteristics such as CPU time used, wall clock, and memory used, and characteristics derived from the use of a common library (i.e. PVM), such as message sends and receives. Application-specific attributes are made available through program annotation. The annotations declare attributes that are available for monitoring. The example in the next subsection will show the specific function calls used to annotate the source code.

In addition to making information about the message activity available for monitoring, Surveyor intercepts calls made to PVM in order to determine transaction membership. This allows the monitoring library to transparently gather the neighbor information. Outgoing messages are tagged with the local transaction id, which is then removed by the monitoring library on the other side, leaving the original message unchanged. Figure 6 shows how the monitoring library relates to the application and message library.

Ideally, annotations are inserted by the programmer when the program is written. Program annotations, like comments, are easier to make at the time of creation rather than afterwards. Unfortunately, this will not be true for most applications. Consequently, Surveyor annotations can be added in an incremental manner, i.e., as the programmer learns about the computation, he will increase the number of annotations. Initially, only a small number of annotations are
necessary to start exploration. These initial annotations indicate transaction boundaries. After indicating where transaction boundaries are, the viewer has access to common attributes, which can be used with predefined visualizations. Then as the programmer learns more, she can add annotations to monitor application specific attributes and define more sophisticated transactions, allowing more extensive visual exploration, and leading to greater understanding of the computation.

The monitoring library has two primary components: the transaction labeling protocol (TLP) and the application monitor (AM). The TLP is responsible for sending information to the snapshot manager so that it can determine the transaction membership and ordering. The AM maintains the identities of available attributes and fulfills requests from the snapshot managers for data. The TLP is based on one of the algorithms described in Section 3. Currently, Surveyor has TLP modules that implement the comprehensive and selective monitoring algorithms. The current AM module supports subscription to attributes at the application process. The AM sends the local snapshot to the snapshot manager, tagging it with the local transaction id.

It is possible that errors can be made when annotating the code to indicate transaction boundaries. Such errors can usually be detected by the monitoring library, snapshot manager, or the viewer. For example, incorrect nesting of transactions can be easily detected by the monitoring library. An example that could be detected by the viewer would be transactions that were a different size than expected. The effects of errors made in annotating are confined to monitoring, and will not affect the computation other than the amount of overhead normally associated with monitoring. The only errors that could occur while making attributes available would be to try to monitor data that no longer existed, because the monitoring library maintains a pointer to the data, which would become invalid.

**Snapshot Manager:** The snapshot manager uses information from the TLPs to decide transaction membership and ordering. The snapshot manager sends subscription requests to the AM and receives local snapshots. The snapshot manager also processes the viewer’s queries, and distributes attribute subscription requests to fulfill the query. The snapshot manager also filters the snapshots, so that the data associated with a query is not seen in a snapshot until all parts of the query are fulfilled. For example, suppose a query requests information from two processes. Snapshots inserted into the snapshot stream would not contain the data from either process until data from both processes were available. Since the user submits a query as one command, it is appropriate not to show partial effects of the command, which could be misinterpreted by the viewer as being complete.

**Visualization System:** The visualization system consists of two parts: a user interface for the interactive specification of queries, and a set of visualizations to render the data being monitored. The user interface is written in Java to allow viewers to monitor distributed computations remotely from a variety of platforms. Visualizations can be any program visualization system that can use a stream of snapshots. We have written an adaptor so that Paveletz visualizations can be used. Some visualizations have also been integrated into the user interface. Surveyor can support an arbitrary number of visualization displays. Generally, the displays are synchronized between each snapshot.

### 4.2 Exploration

Exploration is an interactive endeavor, in which the viewer evaluates the data currently available and based on that data can decide what action to take next. He refines his location in the visual environment via navigation. In Global Surveyor the navigation commands are primarily accomplished through refining the query evaluated and choosing appropriate visualizations. An exploratory session can be saved at any time, and then resumed later. This is especially useful since your location in the exploration is defined by the queries and visualizations that you are using. It enables the viewer to easily periodically check-in on long running computations and for viewers to share queries and visualizations.

To illustrate how an exploratory session might proceed, we give an example exploration of a distributed ray tracer. The example we present is a case where the computation had no initial
annotations. In such cases annotations can be made incrementally, as knowledge about the application is gathered. This is mainly an artifact of Surveyor's use of hand annotation of the source code. If the computation is already annotated or if it can be automatically annotated, then the incremental nature would only apply to the creation and refinement of visualizations and queries. Since we can only build visualizations upon data that is available, an incremental approach is appropriate.

4.2.1 The Application

A ray tracer takes a description of a scene, and a point of view, and computes a visual representation of the scene from that point of view. It does so by casting rays back from each pixel to be shown into the scene. When a ray intersects an object the ray tracer computes the amount of light reflected off that object back to the pixel. Since the computation of each pixel is independent, ray tracing can be distributed for performance benefits. The only coordination needed between processes is to decide which process computes which pixels and to reassemble the image.

The ray tracer we explore is the PVMPOV [6] ray tracer. Persistence of Vision Ray-Tracer (POV-Ray) is a popular ray tracer available for a wide variety of platforms. PVMPOV is an extension to POV-Ray that allows a group of processes to cooperate in computing the POV-Ray scene using the PVM message passing library. We had no prior experience with either POV-Ray or PVMPOV, we chose it as a testbed because of the clearly defined logical actions that occur within it. The distributed version works by having a master process allocate rectangular regions, blocks, of the screen to different processes, slaves. Work is balanced by having slaves request blocks to work on. In response to a request, the master then sends a block to the slave, who will later send back the completed block. The master then assembles the image and allows the display of blocks as they arrive. Figure 7 shows an example of a POV scene that is partially rendered. For more information on POV-Ray see the reference manual [7]; or for more information on PVMPOV see [6].
4.2.2 Initial Survey

We want to minimize the number of annotations needed for viewers to start exploring an unfamiliar computation with Surveyor. The annotations that are needed are transactions to tell the monitoring library when to take a local snapshot. We initially only insert the annotations necessary to define high-level transactions. This will allow the use of prepackaged visualizations that use common attributes.

Initial Annotations: Surveyor uses source code annotations to define transaction boundaries. Initially we use only two annotation functions:

- start-transaction ("transaction instance", "transaction type");
- stop-transaction ("transaction instance", "transaction type");

The start-transaction's arguments are an instance identifier to help the viewer know where in the code a particular transaction is taking place, and a type argument that describes the process's role in the logical action. Thus, for different processes in the same transaction, the labels may differ. In the distributed ray tracer we annotated the main loop in both the master and slave code, with a start transaction at the top of the loop and an end transaction at the end of the loop. In general, the beginning and end of loops are good locations in which to put transaction boundaries.

Initial Queries: The viewer connects to a running computation through the user interface. There are a number of options that a viewer can exercise when she first monitors a computation. The attributes and processes that are available are automatically gathered. Figure 8 shows the user interface's display of the currently available processes. A panel (figure 9) shows the currently available visualizations. Often an initial query will collect data from all of the processes in order to provide a global overview of the computation. The common attributes that are monitored through a query are those needed to support the visualizations that the viewer is interested in.
Figure 9: The left panel shows which visualizations are currently in use. In this case the grid and text visualizations. The right panel shows which visualizations are available for use.
Starting Visualizations: A number of visualizations are prepackaged with Surveyor and are applicable to all computations. These standard visualizations are based upon common attributes. Their applicability to a wide variety of applications causes viewers to become familiar with these visualizations, and allows the viewer to more quickly understand the visualization and its relationship to the properties of the computation it represents.

Two examples of standard visualizations are shown in figure 10. The visualization on the left shows the messages passed between processes, with the current transaction highlighted in the dark blue box. The visualization on the right displays the amount of processor time used by each process. Another standard visualization is shown in figure 11. This visualization is a grid of buttons with processes along the top and attributes along the left. The visualization shows which attributes are monitored in which processes. It also allows the viewer to interact with it to refine the query being used by simply clicking on the appropriate buttons.

Initial Observations: One of the most fundamental tasks that a viewer would do to start exploration would be to use Surveyor to learn what processes are in the computation, the general message pattern between them, and what attributes are available for monitoring.

We choose not to look at global states with messages in transit, in order to present simpler global views of the computation. This does not mean that we ignore messages or that they are unimportant in understanding the behavior of a distributed computation. A viewer can easily visualize the messages that occur within a transaction. For example, by looking at the message pattern the viewer notices that there is a single process that communicates with all other processes. The single process is the master process and the other processes are slaves. This can be seen using a prepackaged "message" visualization, that shows the messages embedded in a transaction.

We can look at a performance visualization to see how well balanced the processes are, i.e. that there is roughly an equal amount of work being done by each process. This would use a CPU visualization of all of the processes.
Figure 11: The grid visualization allows an alternative way to specify queries. The attributes are along the left and the processes are along the top. To query all of the attributes at a process/a single attribute at all processes the user toggles the process id/attribute name. Green indicates that the attribute at the process is currently being monitored; yellow indicates that the query has been submitted.
4.2.3 A Closer Look

Further Annotation: After becoming familiar with the general characteristics of the computation, the viewer may wish to examine more application-specific attributes about the application. To achieve this the programmer can declare attributes within the application that can be available for monitoring. This would be done using the other two annotation functions:

- id = watch (x, "attribute name", "Attribute Description");
- unwatch (id);

The watch annotation’s first parameter is the variable’s address (the watch function is overloaded on the variable’s type so it is not necessary to include its type as a parameter). The name and a description of the attribute are given in the following two arguments. The attribute name describes what the variable semantically represents. The description is used as a reference for a viewer who wants to know more about the attribute. The watch call needs to be made only once, to let the monitoring library know that the variable is available for monitoring. The id returned by the watch function is the attribute’s id. The id is needed if the programmer no longer wants the attribute to be available for monitoring. For instance, this would be necessary for attributes that are dynamically created and destroyed.

Queries: The addition of application-specific attributes does not change the options available to the viewer, as far as the types of queries possible. Common and application-specific attributes are treated the same by the snapshot manager and the user interface.

Queries can be specified with mechanisms other than textual queries submitted through the query panel. The grid visualization mentioned earlier provides a simple interface for specifying the attributes to be monitored at each process. For example, figure 11 shows a visually specified query that focuses on the master and one slave.

In computations that use a master and a set of slaves, like the ray tracer, focusing on the interaction between the master and a single slave can provide insight without having to monitor all of the slaves, which produces extra perturbation and a more complex visual scene.

Visualization Templates: The disadvantage of prepackaged visualizations is that the scope of data that they can be applied to is limited. They are useful for performance visualizations, but they are not as useful for algorithm understanding or debugging, because they do not utilize any application-specific data. To provide more flexibility and ease in creating visualizations, visualization templates can be defined. These visualizations are parameterized by viewer specified data and create a visual display of that data.

This has the advantage of supporting the visualization of data that is specific to a particular application in a form that is familiar to the viewer. Conceptually, these are similar to the types of graphs or charts that spreadsheets use. Data is used to customize the visualization to show the desired data.

For the distributed ray tracer we may want to use an integer viewer to view how many blocks are processed by each process, and compare it to the amount of time used (both actual and CPU).

Observations: The addition of application-specific attributes allows a view that provides more details into what the computation does. The use of visualization templates gives viewers flexibility in creating custom visualizations without the need to invest time into creating one from scratch.

4.2.4 Detailed View

More Transactions: The programmer can iteratively add more refined transactions and additional application-specific attributes in order to create a more complete exploratory environment. Suppose that we want to see each pixel as soon as possible. This requires a much finer grain transaction than we initially used. To do this we simply nest start and stop transactions, inside the higher level transactions, inside the innermost loop. Adding these transaction annotations does not force us to view the computation at this low level, but if we decide to begin viewing
Block Loop

- start-transaction ("outer", "processblock");
- request block
- receive block
- Line Loop
  - Pixel Loop
    * start-transaction ("inner", "processpixel");
    * Calculate Pixel value
    * stop-transaction ("inner", "processpixel");
- send block
- stop-transaction ("outer", "processblock");

Figure 12: Pseudo code showing the location of transaction annotations. The inner transaction is nested inside of the outer transaction.

at this level of detail as the computation is running, then it would be possible to switch temporal granularities. Also needed would be an application-specific attribute to make the slave's block available for monitoring. Figure 12 shows pseudo code of the placement of the needed annotations.

Transaction Queries: Until now we have been monitoring all transactions in the computation since we only had one level of transactions. In order to change the temporal granularity, we need to specify which transactions are to be monitored in the computation. For example, if we called the two transaction types outer and inner for the outermost and innermost loops respectively, then a query that wanted to see each pixel as it is computed would monitor inner transactions, and queries that only wanted to see the computation progress in whole blocks would monitor the outer transactions.

Application Specific Visualizations: The disadvantage of prepackaged visualizations and visualization templates is that full flexibility is not available in how the data is shown. Complex relationships specific to the application may not be compatible with the visualization templates available. Application-specific visualizations are useful since they allow a visualization to compactly represent the important aspects of an application. For frequent visualization of application-specific data, an application-specific visualization can be worth the extra effort needed to create it.

We define an application-specific visualization for the ray tracer that creates a visual representation of a block of work being done by a process. Figure 13 shows the application-specific visualization created. Each pixel is drawn in the visualization, similar to how it would be shown in the display that comes with POV. In addition the custom visualization includes information about which processes are computing which blocks.

5 Related Work

Exploratory tools draw upon research from program visualization and distributed systems to support designers and maintainers of distributed computations. The starting point when using an exploratory tool is often a prepackaged visualization. Prepackaged visualizations are available in a wide variety of visualization systems, for example the Pablo system[8] uses presentation modules for visualizations. Each presentation module provides a prepackaged visualization for inspecting data at various points in a data analysis graph. Performance information is often visualized with prepackaged visualizations, since there are many performance characteristics that are common to all computations. Application-specific visualizations also play an important role in exploratory tools; they allow a user to see data in a form tailored to the application's domain.
Application-specific visualizations are available in a large number of systems, such as Pavane[5], Polka[9], and Zeus[10]. Many visualization systems provide the ability to use both application-specific and prepackaged visualizations.

Regardless of what visualizations are used, obtaining a consistent global view of the computation is important to prevent unnecessary misunderstandings or confusion. One way to obtain global data is to execute a checkpoint algorithm. A checkpoint is a representation of the state of a single process at a given time. A collection of checkpoints can be used to form a global snapshot[11] if there are checkpoints for each process in the computation and if there are no causal dependencies between the checkpoints. Surveys of checkpoint and snapshot algorithms can be found in [12] and, respectively. Without any modifications, snapshot and checkpointing algorithms are too expensive, in terms of perturbation, to be used for the continuous monitoring of distributed computations needed by exploratory tools. Our selective and exclusive snapshot algorithms were designed for the collection of a sequence of snapshots where the scope of the data that needs to be present in the snapshot may change from one snapshot to the next.

An alternative approach to achieve continuous monitoring of computations is to have each process produce a stream of events. When merging the events of these streams into a single event stream, the events can be ordered to meet consistency criteria needed by the exploratory tool. The Animation Choreographer[13], part of the PARADE system[14], is a graphical, interactive tool that supports the calculation of alternate feasible event orderings of the program execution under study. This allows visualization of the program under a variety of temporal perspectives. A different approach to viewing different possible orderings is taken by Perspective Views, implemented in the Belvedere system[15]. Perspective Views has the user choose a subset of the causal relationship and user defined orderings, to base the total ordering on. This allows the user flexibility in how the computation is visualized. Care must be taken when using this approach, because while it may be useful to do this to make visualizations more understandable, the resulting visualizations may communicate misleading or inaccurate information. The Falcon
system for interactive program steering [16] relies on the existence of an ordering filter placed at
the point at which the event streams are merged to ensure a valid ordering of events collected by
the monitoring system. This causality filter[17] is based on the causal relationships between the
events in the program.

Event streams can also be used to detect higher level events. This extraction of higher level
events is a form of temporal abstraction. A system based on this approach is the Ariadne
debugger[18] for the pC++ language. The low level events for Ariadne are communication actions
between processes, language specific events, and user defined events. The events are then matched
against the expected behavior of the program, that the user specified in terms of both low and
high level events.

Steering tools, in addition to monitoring an application, allow feedback to be sent to the com-
putation. Because of this, these tools need visualizations based on accurate global information,
so users can make well informed decisions about what steering commands to invoke. The CU-
MUVS system[19] for the steering of PVM programs assumes that the application is structured
around a main simulation loop. A data transfer routine is placed in this loop. When this routine
is executed, the equivalent of a local snapshot is collected, and marked with an iteration number.
Local snapshots are then combined into global snapshots on the basis of the iteration number.

6 Concluding Remarks

Understanding distributed computations can be challenging because of the non-determinism,
complexity, and size of the applications. We believe that exploratory tools will play an important
role in meeting the needs of designers and maintainers for these systems. In this paper we
described a nested transaction model, upon which algorithms for collecting evolving snapshots
were given. Abstractions help the user to focus in on interesting aspects of the computation.
The snapshots algorithms in this paper make use of the abstractions employed by the user to
make data collection more efficient, thus allowing exploratory tools to be more easily used with
large and long-lived distributed computations. We also presented Global Surveyor, a prototype
exploratory tool, which served as a testbed for the development of the snapshot algorithms. An
example exploratory session of a distributed ray tracer was also described. Future work will
look at classes of distributed predicates that can be used in an exploratory environment, the
application of consistent steering actions to distributed computations, and the interpretation of
interactions with visualizations.

References


