International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:2, 2008

An Exploratory Environment for Concurrency Control Algorithms

Jinhua Guo

***Abstract*—**Designing, implementing, and debugging concurrency control algorithms in a real system is a complex, tedious, and error-prone process. Further, understanding concurrency control algorithms and distributed computations is itself a difficult task. Visualization can help with both of these problems. Thus, we have developed an exploratory environment in which people can prototype and test various versions of concurrency control algorithms, study and debug distributed computations, and view performance statistics of distributed systems. In this paper, we describe the exploratory environment and show how it can be used to explore concurrency control algorithms for the interactive steering of distributed computations.

***Keywords*—**Consistency, Distributed Computing, Interactive Steering, Simulation, Visualization

I. Introduction

INTERACTIVE computational steering provides users with the opportunities to tackle new problems in a way that helps them to learn about the computation in a highly engaging, interactive, visual environment. Causal consistency is an important feature of interactive steering of distributed computations, as it is often required to maintain the correctness of the computation. However, due to the asynchronous nature of distributed computations, it is difficult to coordinate steering changes across processes to guarantee that the changes are applied consistently at all processes.

To address the problem of consistent steering, we designed a new approach known as *optimistic steering*. In contrast to a pessimistic approach, which requires that processes agree upon and halt at a consistent cut prior to applying an update, the optimistic approach permits steering changes to be applied at a process without concern for or knowledge of the state of any other process. However, this requires checkpointing, consistency verification, and logging, as well as rollback and re-execution in the case that inconsistency is found.

We are working to design and implement the consistency algorithms that permit optimistic, interactive steering. However, the monitoring and visualization system in which these algorithms are to be deployed is very large, making it difficult to prototype, implement, and debug these algorithms *in situ*, given the complex interactions and low-level, systems-specific code with which these algorithms must be integrated. Thus, it is desirable to have an environment in which we can quickly prototype the algorithms and verify their correctness, but at a higher, more abstract level. We have developed such an environment, which simulates the execution of a distributed computation and its interactions with the interactive steering system. This environment serves as a prototyping system for purposes of development, through which we may develop and debug algorithms for optimistic interactive steering, which include modules for detection of inconsistency, verification of consistency, checkpointing, message logging, rollback, and re-execution.

This exploratory environment also has a visualization component, which is used for debugging, verification, and illustration of the optimistic steering algorithms. The debugging and verification of the distributed consistent steering algorithms can also be a very challenging task. Because they can involve a large amount of distributed state and complex interactions among program processes and control processes. In such cases, visualization may be used to illuminate the execution of the algorithm and help find problems in the implementation. Further, it is also very difficult to explain the steering algorithms to others who do not have the direct experience with the systems. In this case, visualization may be useful in demonstrating the workings of the algorithms. Visualization may also facilitate comparisons between different algorithms, or between different implementations of the same algorithm.

In the remainder of this paper, we first give an overview of the Pathfinder system for monitoring and interactive steering, and describe the optimistic steering strategy. We then present the Exploratory Environment, which consists of simulation and visualization components. Finally, we explain the workings of the two consistency verification algorithms and their visualizations.

II. The Pathfinder System

We have designed and implemented an interactive monitoring and steering system that allows a user to pose queries and visualize program data in a real-time fashion. Through this system, the user may monitor attributes and variables of the distributed computation. This system, known as *Pathfinder*, serves as the base upon which optimistic steering is implemented, see Fig. 1. In this section, we describe the components of the Pathfinder system and their

Manuscript received March 29, 2006.

Jinhua Guo is with Department of Computer and Information Science, University of Michigan, Dearborn, MI 48128 USA (phone: 313-583-6439; fax: 313-593-4256; e-mail: jinhua@umich.edu).

function, and explain how optimistic steering can be integrated into these components. Through the integrated system, users may dynamically manipulate program variables or adjust resource allocation, without compromising the correctness of the underlying computation.

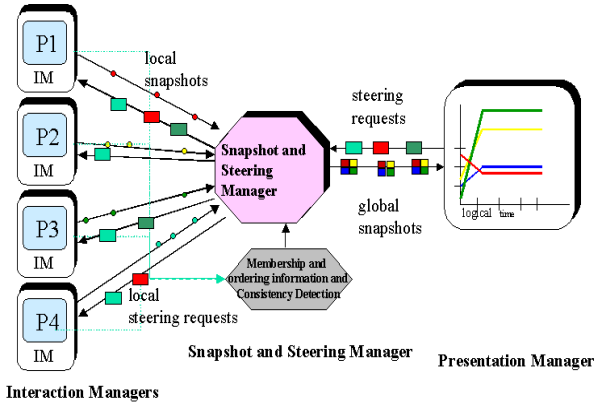

Fig. 1 the Pathfinder System

A. Monitoring

The Pathfinder system is constructed in three parts: Interaction Managers (IM), a Snapshot and Steering Manager (SM), and a Presentation Manager (PM), as seen in Fig. 1. The IM exists as an instrumentation layer that resides between the process and its communication environment. The IM collects local snapshots (sets of local variable values) and transaction labeling information, then sends both to the SM. Transaction labeling information includes information about the processes that participate in each transaction (membership) and the dependence relationships between transactions (ordering) [7].

The SM serves as a central observer. The SM is responsible for merging local snapshots from the IMs to produce consistent global snapshots based on the transaction labeling information [4]. For these global snapshots to accurately reflect the state of the distributed computation, local snapshots must be grouped together and ordered in a manner that does not violate the causal relationships in the distributed computation.

Finally, the global snapshots are sent to the PM to be visualized. The PM oversees the decoding of global snapshots into animation actions, the control of visualizations, and the interpretation of user interactions with the visualizations into monitoring directives.

B. Optimistic Steering

The conservative steering approach avoids inconsistent steering by strictly adhering to the causality constraint. The steering changes can be applied only when a global consistent steering point is reached or detected. This typically involves blocking the computation before a consensus decision is made.

In contrast to the conservative approach, the optimistic approach to steering assumes that the next steerable points are consistent, and invokes the steering change at the next steerable point at each involved process without concern for or knowledge of the state of any other process. This eliminates the need for blocking steered processes. However, the optimistic steering approach must detect any inconsistent steering transaction and provide a checkpointing/rollback mechanism to restore the computation to a correct state if the steering transaction is inconsistent.

A user can issue a steering request at any time during the computation. Upon receiving a steering request from the user, the SM sends out the steering command to the involved processes. Processes receiving a steering command from the SM apply the steering changes at the next steerable points and then report back to the SM. The processes need not know the states of other processes involved in the steering transaction. Upon receiving acknowledgements from all the processes involved in the steering transaction, the SM carries out a consistency check. Based on the local steering times and other transaction membership information, the SM can determine if the steering update was applied consistently. If the steering transaction is consistent, the SM will broadcast an OK message to each process. Upon receiving the OK message, the processes enter a normal state, cease logging and delete all logs. If an inconsistency is detected, the SM issues a rollback command to each process and provides the correct steering time to all the processes involved in the inconsistent steering. Upon receiving a rollback command from the SM, the processes that were affected by the steering transaction will roll back to their previous checkpoints, execute forward, and then reapply the steering changes at the consistent point specified by the SM.

Note that the consistency check and application execution are concurrent. While the SM is verifying the consistency of a steering transaction, the application continues its execution. In optimistic steering, the overhead of state saving and some logging will be incurred for each steering transaction; however, rollback and re-execution will be incurred only in the case of inconsistent steering.

Local checkpointing, rollback and direct steering changes, such as manipulating program variables or adjusting resource allocation, are performed by the IMs. The SM is responsible for coordinating global steering activities. The detection of inconsistency, verification of consistency and calculation of the earliest consistent steering time of a steering transaction all require knowledge of all participating processes. Therefore, the SM performs all these operations. The UI provides the interface through which users may issue steering commands to the computation at runtime.

III. THE EXPLORATORY ENVIRONMENT

The Exploratory Environment (EE) is designed to quickly prototype the abovementioned optimistic steering approach, verify its correction, and evaluate its performance.

The Exploratory Environment consists of two operational components: a *Simulator* and a *Visualizer*, as in Fig. 2. The Simulator models the Interaction Manager (IM) and Snapshot Manager (SM) components found in the Pathfinder system. The Visualizer provides graphical representations of the state

of the Simulator based on information it receives from the Simulator at run time or from log files generated by the Simulator.

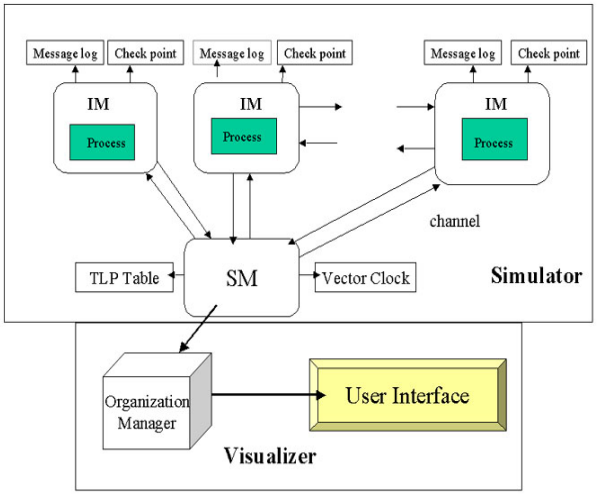

Fig. 2 The Exploratory Environment

A. The Simulator

The Simulator uses Java threads to simulate the processes in a distributed system. Channel objects simulate the communication channels between different application processes. Each process is wrapped with an IM. All incoming and outgoing messages go through the IMs, which are also responsible for applying steering changes, taking checkpoints, logging in-transit messages, and carrying out rollback commands. The SM plays a central role, as it is responsible for issuing steering commands, detecting inconsistency, verifying consistency and sending out rollback orders when necessary.

The application processes may be in one of two states: *normal* or *tentative*. Initially, a process is in the normal state. A process's state changes to tentative when its execution is affected by a steering change. Immediately before a process changes its state to tentative, it must take a checkpoint. Additionally, the process begins logging messages received from normal processes. Similar to the process states described, application messages may be in one of two states: *normal* or *tainted*. A tainted message signals that the sending process is in a tentative state. Therefore, processes receiving tainted messages should also checkpoint and enter a tentative state. Normal messages received by a tentative process are logged. Tainted messages will be resent during rollback recovery and thus need not be logged. This process/message state model guarantees that all local checkpoints belong to a consistent global checkpoint. The state of a process is simply the state of its corresponding Java object. Therefore, local checkpointing can be easily implemented. Similarly, messages are also implemented as Java objects. The message log is implemented as a queue of these objects.

Orthogonal to the notion of message state is the notion of message type. Two types of messages exist in this system: *application messages* and *control messages*. Application messages are exchanged between application processes while control messages are exchanged between IMs and the SM. Control messages can also be further grouped into *steering* messages, *TLP* messages, *OK* messages and *rollback* messages.

Upon receiving a steering request from the user, the SM sends out the steering command to the involved processes. Processes receiving a steering command from the SM apply the steering action at the next *eot* (end of transaction) event and then report back to the SM. The processes need not know the states of other processes involved in the steering transaction. Upon receiving acknowledgements from all the processes involved in the steering transaction, the SM carries out a consistency check. Based on the local steering times and information available from TLP messages, the SM can determine if the steering update was applied consistently. If the steering transaction is consistent, the SM sends out an OK message to all affected processes. Upon receiving the OK message, the processes enter a normal state, cease logging and delete all logs. If an inconsistency is detected, the SM issues a rollback command and provides the correct steering time to all the processes involved in the inconsistent steering. Upon receiving a rollback command from the SM, the processes roll back to their previous checkpoints, execute forward, and then reapply the steering changes at the consistent point specified by the SM.

To facilitate the testing of algorithms, the Simulator can produce either a consistent or an inconsistent steering transaction. During program execution, significant events such as message send and receive events, steering events, checkpoint events, message logging events and eot events, along with process states, will be recorded in a trace file or directly sent to the Visualizer. Visualization of these event traces and process states can help in developing, debugging and understanding the algorithms.

B. The Visualizer

The Visualizer has two components: a graphical User Interface (UI) and an Organization Manager (OM). The UI presents visualizations, as seen in Fig. 3. Up to four views may be simultaneously displayed side-by-side. In addition, overlapped views can be brought to the front by minimizing the obstructing views. Event trace records are displayed in a text panel near the bottom. Controls for panning, zooming, speed control, etc. are arranged around the canvas.

As the backend of the Visualizer, the OM provides modules that implement the display control functions. The user can choose to invoke a single view at a time or multiple views at the same time. Understanding a complex distributed system entails understanding a variety of distinct behaviors and the relations among them. A single view provides the opportunity for the user to observe particular aspects of the execution in great detail, while multiple views correlate different

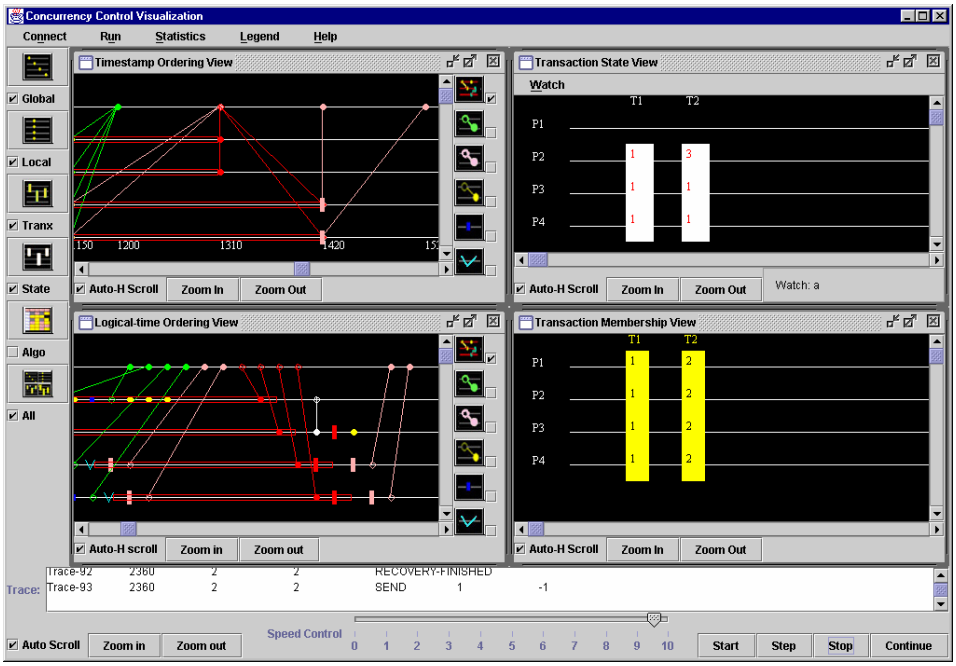

Fig. 3 Multiple Views Displayed in the UI

perspectives to generate a comprehensive global view. Each view is easy to comprehend in isolation, and the composition of several views can be more informative than the sum of their individual contributions. One of the benefits of using multiple views is that each view is usually conceptually simpler and easier to implement. Our emphasis is on providing multiple visualizations that, when executing simultaneously, will help to show the most important aspects of program behavior. A *Single/Multiple View module* controls which of the views should be invoked, in what order, and how the data should be transferred to different views.

When multiple visualizations are running at the same time, information density becomes very high. Information density is a three-dimensional measure that includes not only the size and color of the images, but also the time interval during which the changing display is viewed. A rapid and animated information display can easily overwhelm some users. A *Temporal Control Module* controls the display speed. Users can start, stop or step through the visualization according to their needs.

An *Online/Post-mortem module* connects the Visualizer with the data source selected by the user. Online visualization provides the benefit of an up-to-the-moment view of the computation's progress, while post-mortem visualization gives the user repeatable displays.

A *Data-Image Mapping module* maps the data to specific images according to predefined mapping rules. Colors, shapes and positions are the most important factors in the mapping rules, each of which conveys some specific information of the data being visualized. Shapes serve as the visual means for primary categories while colors are used for subdivisions of the same category. For instance, ovals are used to represent messages with yellow ovals for application messages, green ovals for TLP messages, pink ovals for steering messages and red ovals for rollback messages.

Three kinds of visualizations have been developed in this environment: program visualization, performance visualizations, and algorithm visualizations. In the following sections, we describe the visualizations provided with the Exploratory Environment and show how they can be useful in understanding, debugging, and performance analysis of concurrent control algorithms.

IV. PROGRAM VISUALIZATION

Four program visualization displays are available: the *Timestamp Ordering View*, the *Logical Time Ordering View*, the *Transaction Membership View*, and the *Transaction State View*.

For purposes of illustration, consider a simple calculation. At each transaction, process P_1 sends three numbers x_2, x_3, x_4 to P_2, P_3 and P_4, respectively. P_2, P_3, and P_4 do a simple computation $y = a * x_n$ and then send y back to P_1, which adds up all three y to get *sum*. Initially, a is set to *1*. The first steering action sets a to *3* and the second steering action sets a to *5*. The state of the computation is consistent if a is the same across all processes at any instant.

First we start up the Visualizer and select the views we wish to display by selecting the checkboxes beside the icons on the left border (see Fig. 3). Then we start up the Simulator. From the Visualizer's "Connection" menu, we select "Online" and then visualization begins.

As shown in Fig. 3, multiple views are displayed in smaller

windows, or a single view can occupy the entire window. We can resize, relocate, minimize, maximize or close any of the windows at any time. Trace records are displayed in a scrollable text box at the bottom of the screen.

In this case, we select all the views at once and choose to log the trace. In doing so we can examine the views one by one in great detail by maximizing one view at a time, switch between and correlate the different views, and re-run the visualization with the same program trace repeatedly at a later time.

We begin our detailed examination with the Timestamp Ordering View and the Logical Time Ordering View. Both of these views are event-based and use the same set of icons to represent program events. Ovals represent messages and lines are drawn between a pair of send-receive event icons. An oval becomes hollow when its message is received. Small rectangles represent starting and ending events, with blue rectangles for end-of-transaction events, red rectangles for rollback events and pink rectangles for steering start events. Checkpoints are represented by cyan ticks. The two views both have an event selector on their right side. The user can check or uncheck any of the checkboxes to obtain a variety of event combinations. Some combinations of event selection can reveal patterns that may be missed otherwise.

The Timestamp Ordering View (upper left in Fig. 3) is displayed as a time-space diagram with the *x* axis indicating the elapsed time. Icons for events with the same timestamp that happened at different processes are aligned vertically while icons for events with the same timestamp that happened at the same process are overlapped with one another. To view the details of the overlapped event icons, we can zoom in to enlarge the icons and stretch out the distances between them.

From the Timestamp Ordering view, we can obtain information on the time of the occurrence of events as well as the starting and ending times of transactions. We can also compute how much time elapsed between events. Timestamps are assigned by the IMs and are not necessarily synchronized across processes, but instead represent the local clock times at individual processes.

The Logical Time Ordering View (lower left in Fig. 3) organizes and displays the events according to their logical times. The logical time of a process advances only when an event occurs, so this view eliminates long gaps between events and highlights patterns. The logical time ordering is a consistent ordering, as it enforces the "happened-before" [5] constraint: icons for send events must precede icons for corresponding receive events. As [3] noted, the advantages of this ordering are that it "can produce valid, comprehensible visualizations in the absence of high-resolution, global timestamps and that it can, to the extent possible, maintain perspective on the duration of inter-event periods."

The Transaction Membership view (lower right in Fig. 3) displays transactions as columns of yellow rectangles. Each process that participated in a transaction is represented with a yellow rectangle aligned vertically with its fellow member processes. On each yellow rectangle is a number showing the logical time at which the process participated in the transaction. This view is based on a highly specialized ordering that captures the inter-process communications of the computation. This ordering conveys the information about which processes were involved in a transaction and at what time they were involved.

In addition, the Transaction Membership View illustrates how the computation was done. At each transaction, P_1 updates x_2, x_3, x_4 and sends them to P_2, P_3, and P_4 respectively, thus involving all four processes in each transaction. Note that a transaction does not necessarily involve all processes.

The Transaction State View (upper right in Fig. 3) has a "watch" menu at its top-left corner. The user can pull down the menu and see a list of the observable variables. He can select a variable to watch at any time during the execution. The values of the variable are displayed on the white rectangles representing the processes in a transaction. At the bottom of the view, the name of the variable under observation is displayed.

By comparing the Transaction Membership View to the Logical Time Ordering View, we can gain some understanding about how transactions are specified. In Fig. 3, we see that all four processes participated in the first two transactions. The Transaction Membership view confirms the above conclusion. The Transaction Membership view is especially helpful in preparation for understanding the concurrency control algorithms since both of the algorithms are based on transactions.

The Transaction State view is a visual aid in finding out if a steering transaction is inconsistent. Fig. 3 displays a snapshot of the State view immediately after the first steering transaction, which sets *a* to 3, was carried out and before the rollback began. We can see that the value of *a* was changed to 3 at process P_2, while it remained 1 at P_3 and P_4. This was caused by the inconsistent steering transaction. Therefore, the computation will need to bring back to a consistent state by rollback and re-apply the steering changes at the earliest consistent time.

V. PERFORMANCE VISUALIZATION

The Performance View contains four smaller windows, as seen in Fig. 4. The top-left window shows the distribution of the program events in a pie chart. Colors are related to specific events and have their unique meanings across different views. For example, yellow is always associated with application messages, green with TLP messages, and pink with steering messages. A key, not seen here, is displayed to the user. Below the pie charts, the total number of events is displayed. The bottom-left window displays the distribution of events among different processes. The horizontal line at the bottom of the window is marked with event numbers. The information provided by this view can be used to evaluate load balancing and communication patterns. For example, P_1 has a greater number of events than any other process, which shows that this process communicates more. The top-right window shows

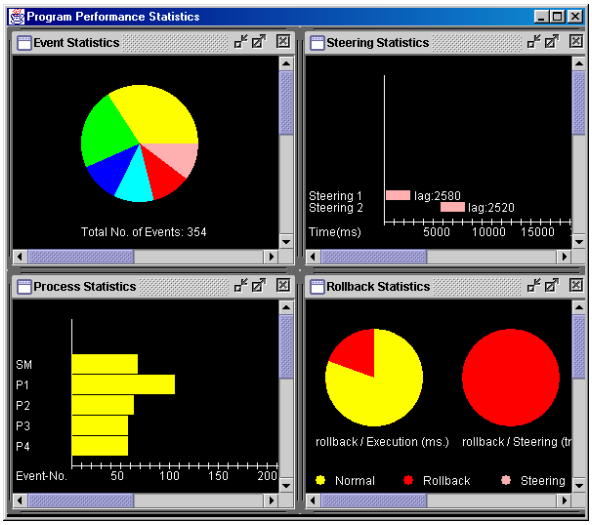

Fig. 4 the Performance View Window

the latency of steering transactions. At the bottom of the window is a timeline running from the left to the right. Steering transactions are displayed in pink bars with their starting and ending times correlated to the timeline. Immediately after each bar is the textual display of the latency time for that specific steering transaction. In this case, only two steering transactions were applied. The first steering transaction took 2580 milliseconds while the second one required 2520 milliseconds. Such information is very useful in evaluating the two concurrency control algorithms. The bottom-right view presents information about rollbacks. The two pies in the window represent two sets of data about the rollbacks. The left pie chart shows the ratio of rollback time to the total program execution time while the right pie chart shows the percentage of steering transactions that required rollback. As shown in Fig. 4, all the steering transactions so far are inconsistent, so the rollback percentage is 100%. Note that this is not typical of the expected ratios for optimistic steering.

VI. ALGORITHM VISUALIZATION

The Exploratory Environment uses algorithm visualization for two purposes: (1) to help algorithm developers design and test concurrency control algorithms for Pathfinder; and (2) to help users understand both these algorithms and distributed computations in general. Two views have been created for the concurrency control algorithms that we developed in this environment: the History-based Algorithm View and the Vector-time-based Algorithm View.

A. The History Based Algorithm

The history-based algorithm [6] depends on a specific subset of computation history. It compares the time stamps of a steering transaction with the time stamps of the program transactions in the history. If a steering transaction can be placed between program transactions, then it is said to be consistent. Key data structures and concepts related to this algorithm are described below.

TLP Table: A TLP table stores a subset of the computational history. Fig. 5 shows the graphical representation of a TLP table. Table elements represent the logical local time at which a process participated in a transaction. Each row represents a transaction. A "-" indicates that a process is not involved in that transaction. Time runs from top to bottom.

Steering Transaction: Computational steering can be viewed as the modification of the local states of one or more processes. Each modification is a steering change or a steering event. A steering transaction is a set of steering events issued by the user in a single request. In the visualization, a steering transaction is represented as a row of pink rectangles. Each pink rectangle has on it a number showing the logical time at which the steering event was applied.

Consistent Steering: A steering action is considered consistent if and only if a steering transaction is not concurrent with any program transaction with which its process set intersects. In other words, a steering transaction is considered inconsistent if some of its timestamps are earlier while others are later than the corresponding time stamps in a program transaction.

The Earliest Consistent time: If a steering transaction is inconsistent, we must calculate the earliest time at which these steering changes can be consistently applied. This is the time of the earliest consistent cut after the initial steering transaction. We need to roll back the steered processes to the checkpoint taken prior to steering, execute forward to the calculated time, apply the steering changes, and continue executing forward. By doing so, the computation will be brought back to a consistent state and the steering will be carried out in a way closest to the user request.

Consistency Validation: A vector is maintained to store the consistent times at which a steering action can be applied. In Fig.5 this vector is represented as a row of green rectangles at the top of the display, labeled as "CV". Validation begins with the most recent transaction in the history. The logical times for each of the program transactions are compared with the logical times of the steering transaction. For a program transaction T_i, if its logical times are all equal to or greater than those of the steering transaction, we know that the steering transaction happened before T_i. Then the non-null values of T_i are written to the consistency vector. Working back through the history, the vector is updated with the non-null values of each transaction whose logical times are equal to or greater than those of the steering transaction. If some or all of the logical times in a program transaction are earlier than their counterparts in the steering transaction, this transaction is concurrent with or happened before the steering transaction. In this case, a flag is set for each process in the program transaction indicating that no earlier steering event time for those processes could logically occur.

The comparison goes on with each program transaction until one of the two conditions is realized:

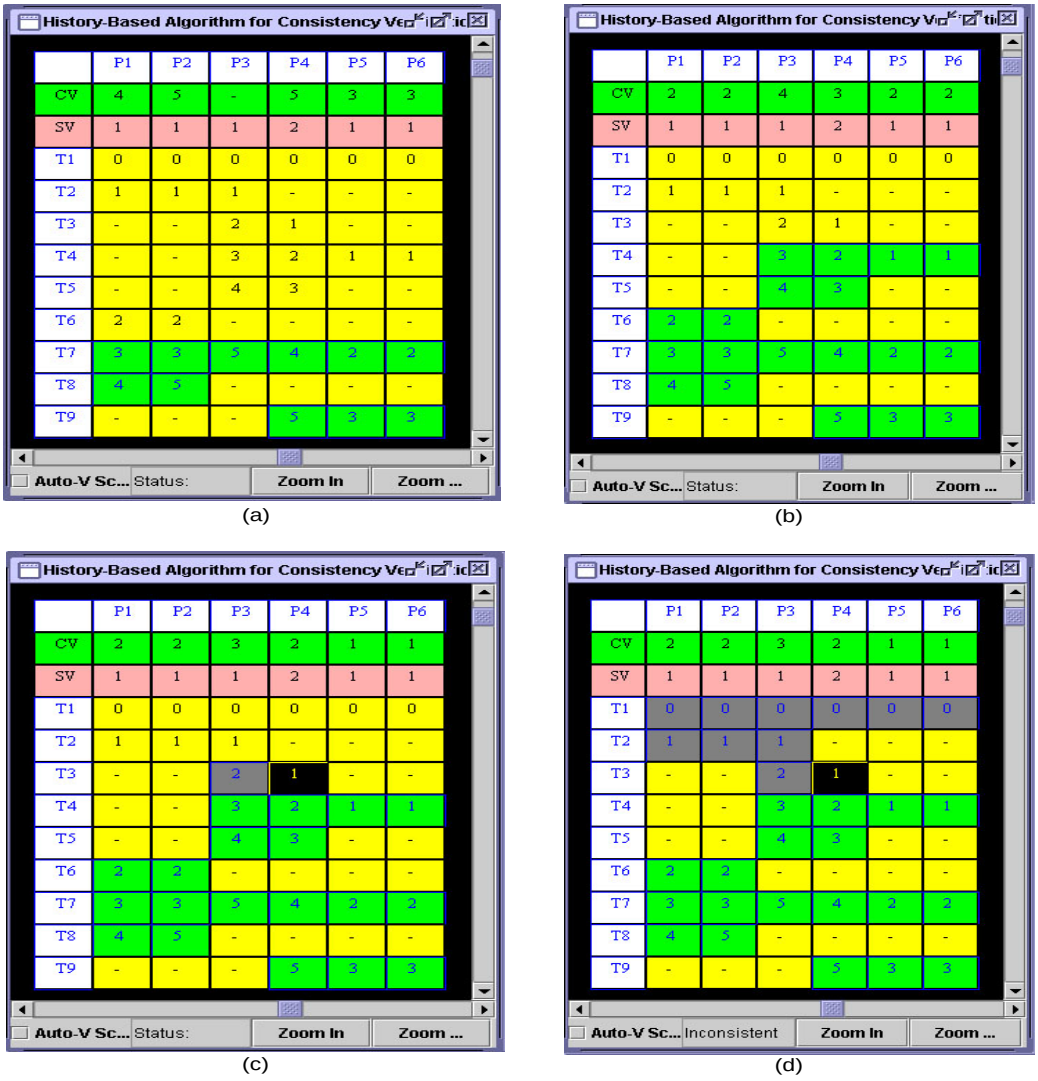

Fig. 5 The History-based Algorithm: Screen Shots of a Sample Run

(1) All the processes in the steering transaction have been flagged; or (2) All the transactions in the TLP table have been checked.

The vector now contains the local times for the earliest consistent steering transaction.

The Simulator sends the TLP table and the steering transaction to the Visualizer as the algorithm begins to run. The Simulator records the result of each comparison in a vector and sends it to the Visualizer. A result vector contains the transaction ID and one or more of the three code words: TRUE, FALSE and AFFECTED, where

FASLE indicates that the logical time of the process is equal to or greater than its counterparts in the steering transaction;

AFFECTED indicates that the logical time of the process is less than its counterpart in the steering transaction; and

TRUE indicates that the logical time of the process is equal to or greater than its counterpart in the steering transaction but some of its fellow processes in the same transaction or some of the logical times for this process in later transactions has been marked as AFFECTED.

Fig.5 shows the screen shots of the animated visualization of the algorithm applied to a small TLP table. In Fig. 5, (a) is the screen shot after the comparisons with T9 and T8 have been done and while T7 is being processed. Note that CV has been updated to reflect the times in T9 and T8, and will soon be updated to reflect the consistent cut at T7. Screen shots (b) and (c) represent the algorithm after comparisons with T6 and T5, respectively. Again, note that the vector was updated for every transaction found to have happened after the steering

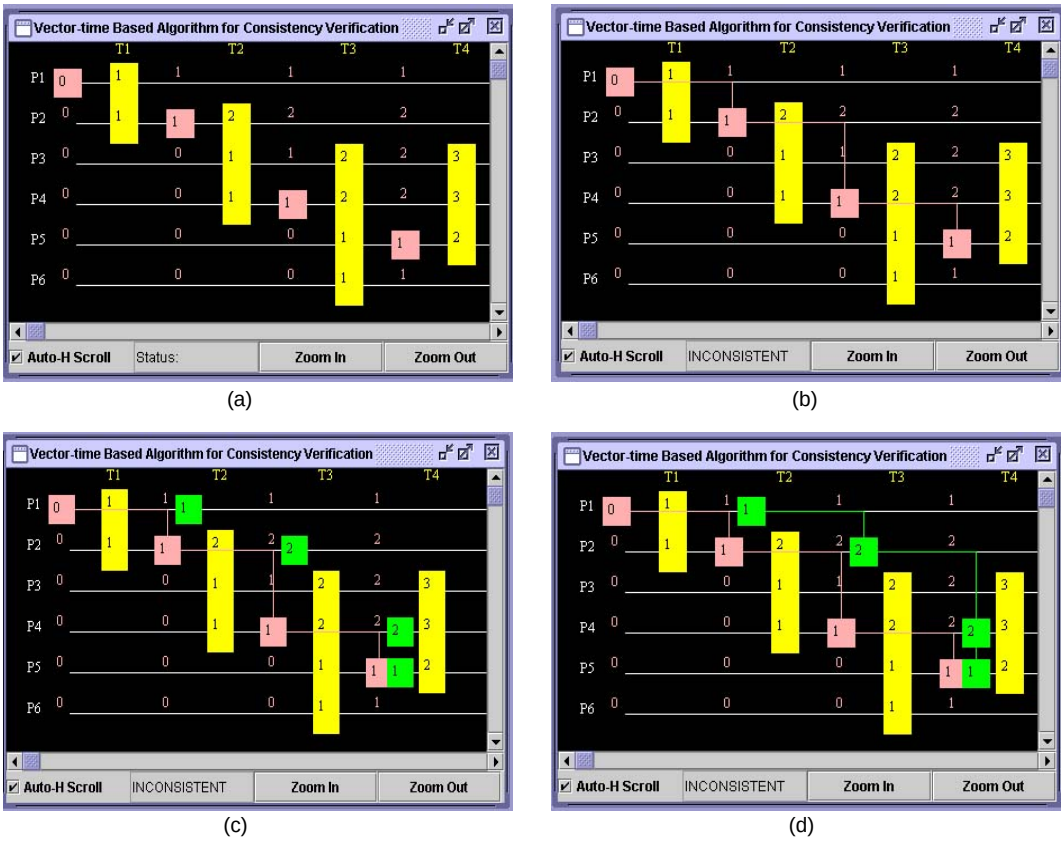

Fig. 6 Vector Time Based Algorithm: Screen Shots of a Sample Run

transaction; (d) is the final screen shot when the algorithm reached its end. Note in T3, P_4 has a smaller logical time than its counterpart in the steering vector. A flag is raised for P_4 (indicated by the black color) and P_3 is marked as TRUE (indicated by gray color). All the three processes in T2 are marked as TRUE because P_3 in T2 is causally related to P_3 in T3, which is already marked as TRUE. The comparison stopped at T1 when all the processes had flags on and coincidentally when all the transactions in the TLP table had been checked.

B. *Vector Time Based Algorithm*

Unlike the bottom-up algorithm that takes a steering transaction as a whole and relies on the history of inter-transactional relations between a steering transaction and the program transactions, the vector-time based algorithm [1] focuses on the relations among the steering events in a steering transaction. In the vector time based algorithm, the consistency of a steering transaction is detected by directly comparing the vector timestamps of steering events. Further, the earliest consistent transaction can be obtained by checking the vector time of each steering event in a steering transaction. The following are the key data structures and concepts that need to be captured in the visualization of the vector-time based algorithm:

Steering Event and its Vector Time: In the vector time based algorithm, each process is associated with a vector clock V of size *n*, where *n* is the number of processes in the system. Each element in the vector corresponds to a process in the system. The value of V[i] denotes the number of past local transactions at that process as known by this process. The vector time of a steering event in process P_i is the vector time that results from the occurrence of steering event in process P_i. A detailed description of transaction-based vector time can be found in [1].

The Transaction-based vector time is represented as a column of numbers in a time-space diagram, as seen in Fig. 6. A steering event is displayed as a pink rectangle with its vector time shown in pink numbers along the conceptual time line.

Consistent Steering: In the vector time algorithm, we decide whether or not two steering events are concurrent by checking if there exist any causal relations among the steering events in the steering transaction. The lack of causal relations among the steering events indicates that the steering changes were applied at a consistent cut; otherwise, the steering transaction is deemed inconsistent.

With the help of the visualization, we can gain knowledge about the processes by checking their positions in the steering transaction. A line is drawn to connect two steering events if a causal dependency exists. If the lines cut through a transaction, then we know that this steering transaction is inconsistent.

The Earliest Consistent Time: If the steering events in a steering transaction are not concurrent, we know that there exists some causal relation between at least two of the steering events. To avoid violating such causal relations, the simplest method is to apply the steering action at the later time of the two. If we can find the latest time for each process involved in the steering transaction and apply the steering at that time, the new steering transaction will be consistent and it will be the earliest consistent transaction, as any steering transaction applied before it is inconsistent.

Let $V(e_i)$ denote the vector time of a steering event in process i. Let SP denote the set of processes that participate in a steering transaction. Let SV be a time vector and SV[i] denote the local time at which the steering event actually happened at process P_i for all i ∈ SP, . Let CV be a time vector and CV[i] denote the time at which the steering event could be consistently applied at process P_i for all i ∈ SP. Then,

$$CV[i] = \left\lfloor \max_{k \in SP} V(e_k)[i] \right\rfloor + 0.5, \text{ for all i } \in \text{ SP}$$

IF CV[i] = SV[i], for all i ∈ SP, then the steering is consistent, otherwise not.

Taking the vector times for the steering events in Fig. 6: $V(e_1)$={0.5, 0, 0, 0, 0, 0}, $V(e_2)$= {1, 1.5, 0, 0, 0, 0}, $V(e_4)$= {1, 2, 1, 1.5, 0, 0}, $V(e_5)$= {1, 2, 2, 2, 1.5, 1}, then we have the earliest consistent time CV = {1.5, 2.5, -, 2.5, 1.5, -}. Comparing the steering transaction SV= {0.5, 1.5, - , 1.5, 1.5, -} with the earliest consistent transaction, we find that the steering transaction was applied at a time earlier than the earliest consistent time, which indicates that the steering transaction is not consistent. Note that, for the processes that did not participate in the steering transaction, a null sign "-" is placed at its position as a placeholder.

To improve performance, we compute the earliest consistent transaction first and then compare it with the steering transaction. If they are identical, then the steering transaction is consistent; otherwise, it is inconsistent.

The animated visualization of the algorithm captures all the important steps in the verification of consistency, the detection of inconsistency, and the computation of the earliest consistent times. Fig. 6 displays 4 screen shots from a run of the vector-time-based algorithm where (a) shows the transactions and steering events, (b) displays the vector times for the steering events, (c) adds the information of a cut, and (d) was captured after the earliest consistent times were computed.

VII. CONCLUSIONS

The primary goal of this research is to build an exploratory environment for the development and understanding of concurrency control algorithms. We prototyped and experimented with versions of these algorithms, compare their performance, and thoroughly test the algorithms in the Exploratory Environment before we implemented these algorithms in the Pathfinder system. To accomplish this goal, we built a simulation of the Pathfinder system and a visualization component to monitor and visualize the computation process as well as the execution of the algorithms in the simulation.

This approach has been very effective. We have successfully integrated the consistent interactive steering algorithms, including the history-based algorithm and the vector-based algorithm, into the Pathfinder system [1].

In this paper, we have seen that the visualization can be very useful for understanding the execution of distributed algorithms. Visualization can provide support for program debugging and correctness verification. It can also be employed to demonstrate and teach the workings of algorithms and to compare the behavior of different algorithms.

REFERENCES

[1] J. Guo, "Consistent, Interactive Steering of Distributed Computations: Algorithms and Implementation," Ph.D. Dissertation, Department of Computer Science, University of Georgia, 2002.

[2] D. Hart and E. Kraemer. "Consistency Considerations in the Interactive Steering of Computations", *International Journal of Parallel and Distributed Systems and Networks*, 2(3), 1999, pp 171-179.

[3] E. Kraemer and J. T. Stasko. "Creating an accurate portrayal of Concurrent Executions" *IEEE Concurrency*, 6(1), 1998, pp 36-46.

[4] E. Kraemer, D. Hart, and G-C. Roman, "Balancing Consistency and Lag in Transaction-Based Computational Steering," *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences*, pp 137-147, 1998.

[5] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21(7): 558-565, 1978.

[6] D.W. Miller, J. Guo, E. Kraemer and Y.Xiong, "On-the-fly Calculation and Verification of Consistent Steering Transactions", *IEEE/ACM Super Computing 2001* (SC2001), Denver, CO.

[7] H. Vuppula, E. Kraemer, and D. Hart, "Algorithms for Collection of Global Snapshots: An Empirical Evaluation," *Proceedings of the ISCA Conference on Parallel and Distributed Computing*, pp 197-204, 2001.

Jinhua Guo is the director of Vehicular Networking Systems Research Laboratory and an Assistant Professor in the Department of Computer and Information at the University of Michigan-Dearborn, USA. He received his Ph.D. in Computer Science from the University of Georgia in 2002. He received the B.Eng. and the M.Eng. in Computer Science from Dalian University of Technology, China in 1992 and 1995, respectively. His current research interests include wireless networks and mobile computing, vehicular networks, mobile agents, and distributed systems. He is a member of ACM and IEEE.