CONSISTENT, INTERACTIVE STEERING OF DISTRIBUTED COMPUTATIONS: ALGORITHMS AND IMPLEMENTATION

by

JINHUA GUO

(Under the Direction of Eileen Kraemer)

ABSTRACT

Interactive computational steering provides users with the opportunity 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.

This thesis introduces a transaction-based computation model for distributed computation. This abstract model not only gives users a simple and high-level view of distributed computation, but also simplifies reasoning consistency problem by reducing the amount of information to be handled.

Furthermore, this work investigates two approaches for achieving consistent steering: conservative steering and optimistic steering. The performance of conservative and optimistic steering approaches is evaluated in term of perturbation and lag. Our experiments show that when the percentages of consistency on the first attempt are large enough and the size of checkpoint is not too large, the optimistic approach will achieve better performance; otherwise, the conservative approach will be better.

INDEX WORDS: interactive steering, distributed algorithms, distributed computation, consistency, termination, causality, vector time, transaction, checkpoint, message logging, rollback.

CONSISTENT, INTERACTIVE STEERING OF DISTRIBUTED COMPUTATIONS: ALGORITHMS AND IMPLEMENTATION

by

JINHUA GUO

B.E., Dalian University of Technology, China, 1992

M.E., Dalian University of Technology, China, 1995

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2002

CONSISTENT, INTERACTIVE STEERING OF DISTRIBUTED COMPUTATIONS: ALGORITHMS AND IMPLEMENTATION

by

JINHUA GUO

Approved:

Major Professor: Eileen Kraemer

Committee: E. Rodney Canfield
Thiab Taha
John Miller
Valery Alexeev

Electronic Version Approved:

Gordhan L. Patel
Dean of the Graduate School
The University of Georgia
August 2002

DEDICATION

To my wife Honglei and my daughter Jennifer

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my major Professor, Dr. Eileen Kraemer, for the long hours she spent with me discussing and refining this project and revising a lot of my writings. I really appreciate her encouragement during the arduous odyssey and her willingness to share her experiences with qualitative research. Her support and guidance helped me complete this phase of life and my career.

I would like to thank my thesis committee, Dr. E. Rodney Canfield, Dr. Thiab Taha, Dr. John A. Miller, and Dr. Valery Alexeev, for their effort, advice and guidance.

I would like to thank a lot of people who have contributed to this project: Delbert Hart, Gruia-Catalin Roman, David Miller, Arumugaraja Selvaraj, Yin Xiong, Brandon Kohn, Navin Gupta, and Himabindu Vuppula.

Last but not least, I would like to thank my wife, Honglei, for her devoted love, support, and patient.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Interactive computational steering provides users with the opportunity 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.

Two general approaches exist for achieving consistent steering: conservative steering and optimistic steering. The conservative steering approach avoids inconsistent steering by strictly adhering to the causality constraint. This typically involves blocking the computation before a consensus decision is made and steering changes are applied. On the other hand, the optimistic steering approach 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 be able to detect any inconsistent steering transaction and provide a checkpointing/rollback mechanism to restore the computation to a correct state. Such optimistic techniques have the potential to reduce both the perturbation of the computation and the lag associated with interactive steering.

Given the size and complexity of distributed computations, a formal treatment of a set of events in a distributed computation into higher-level events is crucial in modeling distributed activities to provide different abstract views [Lam86]. In our monitoring and steering system, the overall computation is abstracted to an interleaving of atomic state changes involving one or more processes – by analogy with databases, we call such state transitions transactions. This abstraction not only gives users a simple and high-level view of distributed computation, but also simplifies reasoning consistency problem by reducing the amount of information to be handled.

In this thesis, I present a transaction-based computation model for distributed computation and two approaches for consistent steering of distributed computations: conservative steering and optimistic steering. Optimistic techniques have been used to enhance performance in various areas such as concurrency control and discrete event simulation. This work represents a novel application of optimistic techniques to interactive steering.

1.1 Interactive Steering

The interactive steering of computations permits users to monitor a program's execution and to adjust both application parameters and the allocation of resources in an online fashion. This interactivity provides a powerful tool for application scientists, researchers, and algorithm developers in the process of "charting unknown waters", whether exploring new computational solutions to problems that are not yet well understood, or attempting to select ideal parameters for familiar algorithms applied to new data sets or problem areas. The ability to monitor a program in execution and observe intermediate results, coupled with the ability to tweak application parameters,

select or install new control algorithms, and direct the allocation of resources, provides users with the opportunity to improve the performance of the computation or the quality of solution for a particular run of the program. In addition, the experience of observing and interacting with the running computation can help the researcher to better understand the dynamics of the execution behavior, provide insight into the target problem and data, and build intuition that can lead to the selection of better default parameters and the development of algorithms more highly tailored to the target domain. Applications for which steering is useful are typically long-running, complex simulation, modeling, or control programs executing in parallel or distributed environments.

1.2 Consistency, Perturbation and Lag

Users of an interactive steering system observe visualizations of the program's state, behavior, and performance. Based on these visualizations, users may issue a steering command, an instruction to alter some aspect of the computation. Examples of steering commands include a request to change the value of a variable at one process, a request to change the value of a variable at many or all processes, or to perform a reallocation of resources. A global state change resulting from one steering command is called a steering transaction. A steering transaction can be decomposed into modifications of the local states of one or more processes, known as steering events. The constraints that steering changes must adhere to vary considerably from application to application. Some steering changes may be applied at any of the participating processes at any point in the computation. However, if a steering transaction is intended to update critical control parameters or change the global configuration, causal consistency is often required to

maintain the correctness of the computation. That is, all local steering changes must be applied concurrently across all participating processes.

Other important concerns in interactive steering include the perturbation induced by steering and the lag. Steering perturbs the execution of the application system. Perturbation measures how the underlying application is affected by the presence and use of the steering software. Local perturbation describes the effect on a single process. The primary source of local perturbation is the execution of additional instructions for steering changes of the local process state. The overall effect on the application, including the local perturbations, message traffic, and consistency controls, comprise the global perturbation. Latency or lag refers to elapsed time. Presentation lag is the elapsed time between the existence of a state in the program's execution and the presentation of that state to the viewer. Steering lag refers to the elapsed time between the initiation of a steering command by the user and the application of the associated steering changes at the process of the computation.

The ideal system would feature strong consistency, and low latency and perturbation.

1.3 An Example

Consider, for example, simulation programs that employ the higher KdV equation [BSS87, Tah92]. As seen in Fig. 1 and Fig. 2, numerical simulations of this equation show that its solitary-wave solutions are unstable, and in fact, that neighboring solutions emanating from smooth initial data appear to form singularities in finite time, which means that the solution "blows up" in finite time [BSS87, FW78]. Usually, numerical schemes for solving this equation are based on an explicit timestep method, which extrapolates the solution of the current timestep based on the calculations of previous

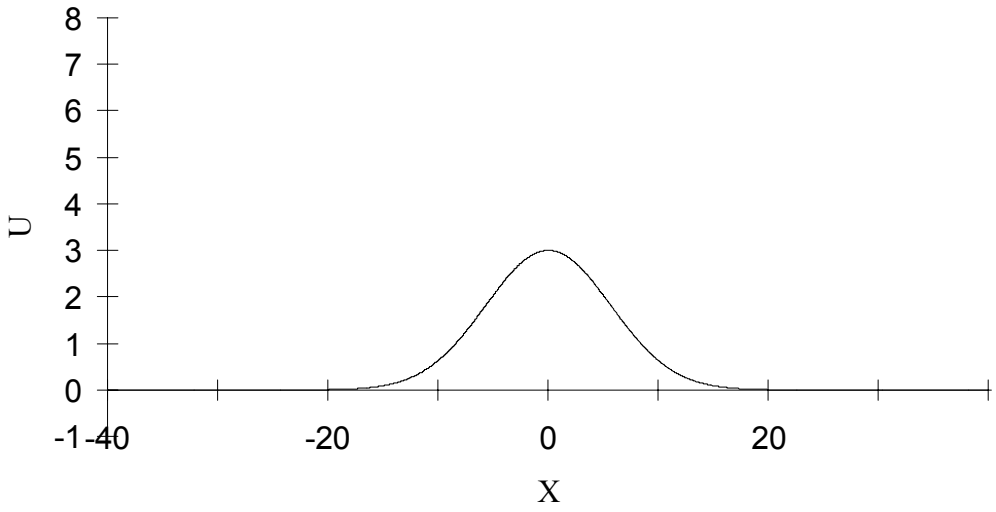


Fig. 1.1 Higher KdV Equation: Initial Conditions T = 0.0

timesteps. Due to the computational complexity, parallel algorithms are often employed to implement these numerical schemes [Tah92, JT02]. Both the mesh size in the spatial direction, Δx, and the size of the timestep, Δt, are tunable. The choice of Δx and Δt in these methods affect the execution time and accuracy of the computation. Typically, small Δx and Δt will produce more accurate results, but will require much longer execution time. Experiments show that when solutions are relatively stable, solutions of higher KdV equation are almost the same no matter how large Δx and Δt are as long as the ratio between Δt and Δx is small enough [JT02]. However, solutions become more and more sensitive to large Δx and Δt when solutions become more and more unstable (i.e. there are more and more peaks in the solution waves, as seen in Fig. 1.2). Therefore, we can choose relatively large Δx and Δt to compute the solutions of early timesteps quickly, and can then alter Δx and Δt to smaller values on-the-fly to get more accurate solutions before they "blow up". By doing this, we could get accurate results with

shorter computation time. However, a change to Δx and Δt may result in incorrect solutions of the equation if applied in an inconsistent manner, i.e., applied at different timesteps in the execution at different processes, or in the midst of a timestep. In this paper, we will show how an optimistic approach can be used to coordinate steering changes across processes so that they are performed concurrently.

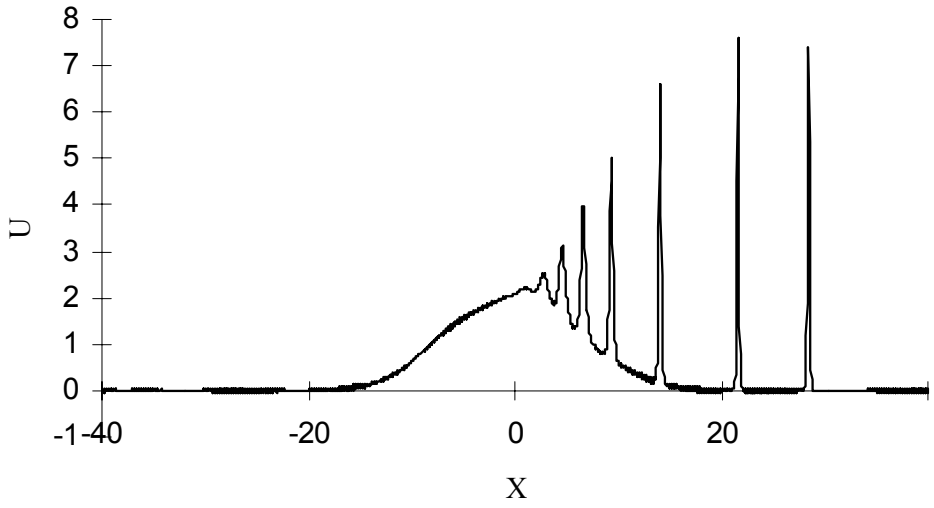


Fig.1.2 Higher KdV Equation: Solutions When T = 0.25

1.4 Conservative Approaches vs. Optimistic Approaches

Globally consistent steering updates typically require that a computation reach quiescence [DS80, Lyn96] before a steering change is applied; the computation blocks before a consensus decision is made and steering changes are applied. This is a nontrivial process in a distributed, asynchronous environment, requiring centralized control of the computation, and with the potential to cause considerable perturbation of the computation

and significant steering lag [KDR98]. In contrast to this conservative approach, we present another approach called optimistic steering. In optimistic steering, the system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the global state of the computation, and specifically, the consistency of the steering transaction, must be checked. If the consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, then the earliest time at which the steering event could be consistently applied at each process is calculated. The computation must then rollback to its state prior to the application of the steering change, execute forward, and then apply the steering change at the consistent time. This approach is expected to perform well in the case that steering is relatively rare and that the processes of the computation tend to remain roughly synchronized because of coordination at the application level. This is often the case, as processes typically wait for messages from one another or synchronize at barriers. Accordingly, while the overhead of state saving and some logging will be incurred for each steering transaction, rollback and re-execution will be incurred only in the case of inconsistent steering, and both logging and re-execution will be of limited scope and duration.

1.5 Related Work

The work related to this research includes: computational steering environments, optimism, causality, vector times, and atomic events in distributed computations.

1.5.1 Computational Steering

An early computational steering environment was VASE, the *Visualization and Application Steering Environment,* developed at the University of Illinois [HBJJ92,

JBBH93]. The VASE system requires special annotation to existing Fortran code and permits the user to alter the values of "key" parameters and to add code at "key" points. However, VASE does not support the coordinated steering of multiple processes. SCIRun supports computational steering in a multithreaded application that runs on a single multiprocessor machine [PG95, PWJ97]. However, it assumes that the underlying program consists of a number of separate modules. The SCIRun system generates a script that controls the invocation of these modules. The steering process involves altering these scripts—thus, changes occur only between modules, not within modules. The script itself is executed sequentially. No support for coordination of distributed changes is required. Progress [VS95] and its successor Magellan [VS97] also provide interactive environments for computational steering. Both these systems were designed to run on multiple multiprocessor machines. Progress does not support coordinated steering of multiple processes. Magellan was extended to support such coordinated steering of multiple processes but requires synchronization points be placed in an application. Before a steering change can take place the application must first halt. Thus, Magellan can be said to support the conservative approach to the interactive steering of distributed computations. CUMULVS was developed at Oak Ridge National Laboratory to support the monitoring and steering of distributed computations [GKP97]. To allow steering, the user interface process creates a loosely synchronized connection with the application, which guarantees that all tasks apply the steering updates at the same time or point in the application, also falling into a conservative steering model. Yet another computational steering environment is the VIPER project [RL97]. VIPER is based on a client/server/client architecture. One client is the parallel computation, the other client is

the visualization unit, and the server acts as a governing body for both information and data extraction and steering application. Each application has synchronization points at which time the server has the ability to consistently apply the steering changes requested by the user. Finally, the CSE environment provides a computational steering environment similar to those already described [LMW97, WL97]. In this system, there exist data manager and satellite worker processes. The data manager is responsible for gathering the monitored data, all communication, and application of steering changes. Like the other environments that support simultaneous steering events, this system also requires its source code be annotated with special synchronization variables. During synchronization, the data manager can consistently apply the steering changes.

1.5.2 Optimism

Optimistic Concurrency Control

Concurrency control is a technique for scheduling concurrent operations in such a way that they appear to have been executed in a single sequence with respect to the data objects that they share. Such a schedule is called serializable [GR92]. However, it is possible to schedule such operations concurrently, so that non-interfering operations are executed in parallel. A conservative scheduling approach, such as two-phase locking [EGLT76] and timestamp ordering [BG80], would look at each operation, and try to find a schedule that does not force any of the operations to roll back. However, there are no deadlock-free conservative protocols that always provide high concurrency [KR81]. Thus the problem of finding a schedule forms a serial bottleneck to the concurrent execution of operation on shared data.

Kung and Robinson's Optimistic Concurrency Control [KR81] is an optimistic algorithm for scheduling concurrent transactions. They make the optimistic assumption that, for sufficiently large sets of data objects, operations usually do not conflict, and so it is possible to optimistically schedule all transactions to be executed concurrently as they arrive for processing, and detect and correct concurrency errors post hoc. If the optimistic assumption is correct often enough, this optimistic approach will achieve higher concurrency than a conservative schedule.

The optimistic method permits transactions to proceed until they are to commit. At commit time, the system checks for conflicts. If conflicts are detected, some transactions are aborted. This results in relatively efficient operation when there are few conflicts, but a substantial amount of work may have to be repeated when a transaction is aborted.

Parallel and Distributed Simulation

Discrete event simulation is a computing application that simulates the behavior of complex systems for the purposes of design and modeling. The number of "events" being simulated in such a system is often very large, and so such simulations may run for a long time. Since most events are independent of one another, discrete event simulation is amenable to speedup through parallelism. However, when simulating a given event, it is often not decidable at the time whether the event in question is independent of other events in the system: is there a factor that this event has not yet learned of ? Conservatively waiting for such confirmation imposes a limitation on the available concurrency in a simulation. Optimistically assuming that such a contributing factor does not exit increases available parallelism.

Jefferson's Virtual Time [Jef85] is an algorithm for distributed process synchronization that has been widely studied. Time Warp is a discrete event simulation system based on the Virtual Time concept that provides the illusion of a globally synchronized clock that can be used to preserve a total ordering across the system as defined by Lamport [Lam78], even though processes actually are being executed out of order. Thus, the semantics that it guarantees are those of a sequentially executed series of computations.

Time Warp optimistically assumes that messages are processed in receive timestamp order, i.e. if the local clock is advanced, no message will ever arrive with a time stamp preceding the new clock setting. Thus every advance of the local copy of the virtual time is a speculative action that may have to be rolled back. If a message arrives with a time stamp that precedes the current local time, then the process is rolled back to a saved state of a simulated time no later than the time stamp of the arriving message.

Variations on the Time Warp model have been proposed [CS89a, CS89b]. The Time Warp protocol uses a detection-and-recovery protocol to synchronize the computation [Fuj89]. The Time Warp mechanism has also been used for optimistic concurrency control in distributed databases, using object-rollback, rather than blocking, as the fundamental means of synchronization [JM86]. Further work on the application of speculative execution to real-time systems has been investigated in [GFS93].

Optimistic methods appear to offer greater hope for general-purpose simulation, if state-saving overhead is kept within a manageable level [Fuj90]. Nevertheless, conservative methods have also been found to offer great potential for certain classes of

applications, particularly when ample application-specific knowledge of the systems being simulated is available, as in [MRR90].

1.5.3 Causality, Time, and Vector Time

Causality is fundamental to many problems in distributed computing. For example, determining a consistent global snapshot of a distributed computation [CL85, FZ90] requires finding a set of local snapshots such that the causal relation between all events included in the snapshots is respected in the following sense: if e' is contained in the global snapshot formed by the union of local snapshots, and e→e' holds, then e must also be included in the global snapshot. Thus, the notion of consistency in distributed systems is basically an issue of correctly reflecting causality. Many important applications of causal consistency are summarized by Schwarz and Mattern in [SM94].

An important characteristic of distributed systems is that there is no global clock. Consequently, ordering the events in a distributed system can be challenging. Lamport [Lam78] introduced an efficient mechanism called logical clocks for totally ordering the events in a distributed system, but the mechanism is not sufficiently powerful to allow concurrent events to be identified. Mattern [Mat89] and Fidge [Fid88] independently developed vector clocks, which precisely capture the causal ordering between distributed events. The main difference between Mattern and Fidge vector time schemes and ours is that we measure logical time in terms of "number of past local transactions" at a process, rather than "number of past events" at that process.

Netzer and Xu [NX95] introduced the notions of Z-paths and Z-cycles, and pointed out that a local checkpoint is useless *iff* it is involved in a Z-cycle. In the transaction-based computational model, if a checkpoint is taken, all participant processes in a global

transaction take the checkpoint at the end of transaction. In this way, no local checkpoint will be involved in a Z-cycle because there is no message in transit at the end of transaction.

1.5.4 Atomic Events in Distributed Computations

In the literature on distributed computations, much attention has been focused on model events in order to reason better about the computations. Thus far, events have been implicitly modeled in the isolated contexts of various applications. A formal treatment of grouping events in a distributed computation into higher-level events is crucial in modeling distributed activities to provide different abstract views [Lam86]. Event abstraction also provides simplicity to the programmer and system designer in reasoning at the appropriate level of atomicity by reducing the amount of information to be handled. Kshemakalyani presents a unifying framework for expressing and analyzing events at various level of atomicity [Ksh98]. Events in distributed computations are defined at four levels of atomicity: primitive send and receive events, send and receive constructs, reactive events, and events between transitless cuts.

The most elementary events are certain "basic" communication actions [CL94] at both processes and communication channels in distributed computations, and are defined as the first level events. This level of atomicity is useful for designing complex communication constructs and for comparing their flexibility with that of primitive communication events at this lowest level of atomicity as a benchmark. At the second level, the events are abstract send and receive events executed at the processes. Modeling events at this level of atomicity has implicitly been done by many applications such as distributed snapshots [CL85], modeling distributed computations [Fid88, Lam78, Mat89],

transfer of knowledge [CM86], leader election, and mutual exclusion [Sin93]. The next level in the atomicity hierarchy has events that are reactive in nature, i.e., each event denotes activity at a process in response to messages received from other processes. Modeling events at this level of the hierarchy has been used for distributed debugging [KF93, NM92] and distributed termination detection [Mat87]. Another level of the hierarchy has events such that each event is affected by and affects the rest the computation only by the process states at the start and end of events, respectively. In this view of the computation, the global state of the system before and after any events is a transitless global state, i.e., there are no messages in transit between any pair of processes in this state. Transitless states are therefore used in applications such as checkpointing and recovery [SY85], atomic transactions [BHG87, FGL82], and fault-tolerant computations [Ran75], in which a past global state may need to be restored.

In our transaction-based computation model, global transactions are another implementations of the fourth level events, events between transitless cuts. However, our transaction-based computation model has some distinguishing features. In the fault tolerance [Ran75], transitless states are created through synchronization at the cost of restricted interprocess communication. The transaction model of [FGL82] uses a non-intrusive scheme of replicating parts of the database to create and record a transitless state. In the Pathfinder system, transitless states are automatically recognized by the system with the proper annotations of source code at meaningful points by the users, and there is no need for global synchronization or data replication.

1.6 Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the Pathfinder system, a system that we have developed for monitoring and steering of distributed computations. Chapter 3 presents the transaction-based computation model. Chapter 4 presents two algorithms for consistent steering of distributed computations: conservative steering and optimistic steering. Finally, conclusions and future work are presented in Chapter 5.

CHAPTER 2

PATHFINDER SYSTEM

We have developed an exploratory visualization 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. 2.1. In this section, we describe the components of the Pathfinder system and their function, and explain how optimistic steering is 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. In Chapter 3, we describe the underlying model of computation.

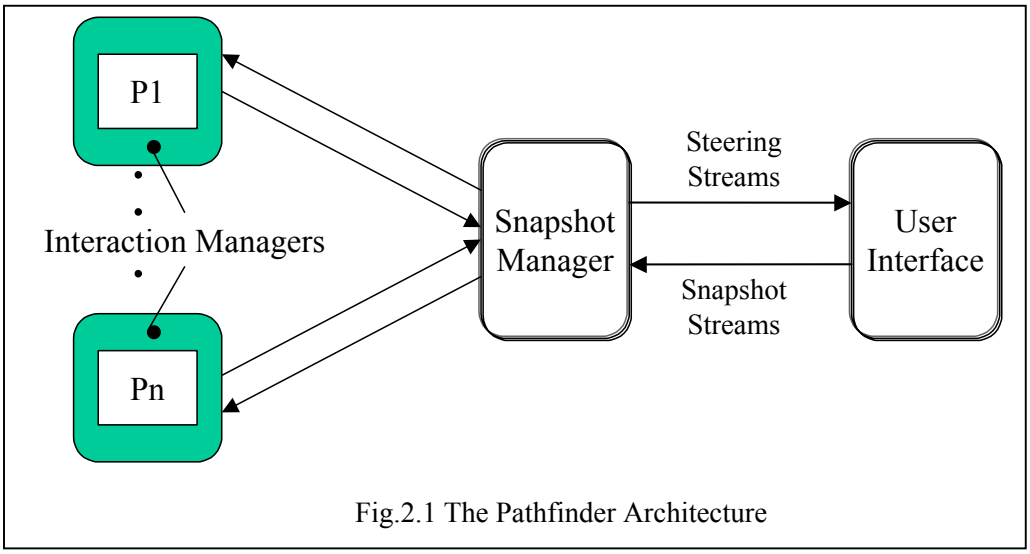


Fig.2.1 The Pathfinder Architecture

2.1 Monitoring

The exploratory visualization system focuses on a class of distributed computations that can be abstracted to an interleaving of atomic state changes involving one or more processes. It is assumed that communication patterns are not affected by steering changes and remain the same during rollback and re-execution. These atomic state changes, or transactions, correspond to logical actions performed by the computation. A transaction boundary is specified explicitly by end-of-transaction (EOT) annotations in an application program [HKR97, Har00, VKH00]. Placement of EOTs controls the granularity of the view of the distributed computation. Annotation may be performed manually or through automated means.

The Pathfinder system is constructed in three parts: Interaction Managers (IM), a Snapshot Manager (SM), and a User Interface (UI), seen in Fig. 2.1. 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) [VKH00].

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 [KHR98]. 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 [Lam78].

Finally, the global snapshots are sent to the UI to be visualized. The UI 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.

2.2 Steering

When users observe a certain state and want to respond to it. He may issue a steering command at any time and make the on-the-fly changes of the state of the computation. The UI provides the interface through which users may issue steering commands to the computation at runtime. The SM is responsible for coordinating global steering activities. The direct steering changes, such as manipulating program variables or adjusting resource allocation, are performed by the IMs.

For optimistic steering approach, local checkpointing, message logging, and rollback are performed by the IMs. 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.

For conservative steering approach, the global synchronization of steered processes are controlled by the IM. While the SM is responsible for check the termination conditions, such as whether or not all steered processes become passive, and whether or not the global transaction involved with steered process are complete.

CHAPTER 3

TRANSACTION-BASED COMPUTATION MODEL

In this section, we define the model of computation upon which our system, algorithms, and definitions are based.

A distributed system consists of a set of processes that cooperate to achieve a common goal and communicate only through message passing. A local computation of a process is a totally ordered sequence of events. The concurrent and coordinated execution of local algorithms forms a distributed computation. Events are classified, as described in [SM94], into three types: send events, receive events and internal events. The event-based causality relation (denoted $\rightarrow$) is a transitive relation satisfying the following conditions: (1) if e and e' are events of the same local computation and e' is the next event after e, then $e \rightarrow e'$; (2) if e is a send event and e' is the corresponding receive event, then $e \rightarrow e'$. If, for two events e and e', neither $e \rightarrow e'$, nor $e' \rightarrow e$ holds, then we say e and e' are concurrent (denoted as $e \parallel e'$) [SM94].

3.1 Transaction

Given the size and complexity of distributed computations, it is important to have a presentation that provides a simple, accurate, and flexible view of an execution. In our monitoring and steering system, the overall computation is abstracted to an interleaving of atomic state changes involving one or more processes – by analogy with databases, we call such state transitions transactions. This abstraction not only gives users a simple and high-level view of distributed computation, but also simplifies reasoning about

consistency problems by reducing the amount of information to be handled. Our transactions are constructed based on end-of-transaction (EOT) annotations placed at appropriate points in the code. We define transactions as follows:

Definition 3.1 A *local transaction* is a sequence of events between EOTs (or between the start of the program and the first EOT) of a process.

Definition 3.2 A *transaction relation* **T** is an equivalence relation on a set of all local transactions such that for two local transactions **a** and **b** of different processes, if **a** contains a send event and the corresponding receive event is in **b**, then (**a, b**) is in **T**.

Definition 3.3 Local transactions can be partitioned in one and only one way into sets called equivalence classes according to the transaction relation, and each equivalence class is called a *global transaction*.

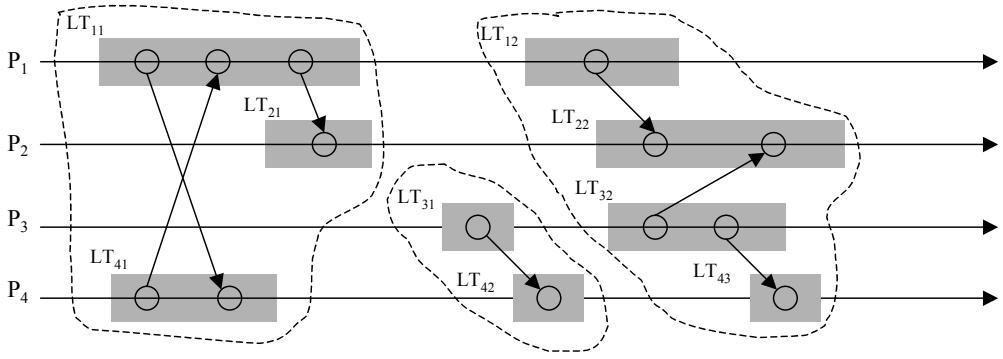


Fig. 3.1 Global Transactions. Processes P_1, P_2, P_3, and P_4 are shown as arrows with time increasing from left to right. Each local transaction is shown as a shaded area containing application events (circles). The boundaries of the shaded areas represent annotations demarking the transaction boundaries. Messages are shown as arrows between process events.

Fig. 3.1 shows an example of a distributed computation consisting of three global transactions, $T_1 = \{LT_{11}, LT_{21}, LT_{41}\}$, $T_2 = \{LT_{31}, LT_{42}\}$ and $T_3 = \{LT_{12}, LT_{22}, LT_{32}, LT_{43}\}$. Local transactions LT_{11} and local transaction LT_{21} are transaction-related because local transaction LT_{11} sends a message to local transaction T_{21}. Similarly, transactions LT_{11} and local transaction LT_{41} are transaction-related. Therefore, local transactions LT_{11}, LT_{21} and LT_{41} all belong to the same equivalence class, a global transaction. Similarly, local transactions LT_{31} and LT_{42} belong to one global transaction and local transactions LT_{12}, LT_{22}, LT_{32} and LT_{43} belong to another.

We then view the local computation of a process as a sequence of local transactions and a distributed computation as a set of partially ordered global transactions. The reason that the model refers to the logical actions taken by the distributed application as transactions is because to the user they appear to exhibit the ACID properties [GR92]:

- Atomicity: To the user, a global transaction is an atomic unit of computation. Each global transaction consists of at most one local transaction at each process.
- Consistency: After the execution of global transactions, process states consist of a consistent global system state because no messages are in-transit.
- Isolation: For global transactions, T_a and T_b, if the local transaction of T_a precedes the local transaction of T_b in one process, then in every other process, the local transactions of T_a must precede the local transactions of T_b.
- Durability: To the user, the values changed by the transaction persist after the transaction successfully completes.

The main difference between Pathfinder transactions and database transactions is that Pathfinder recognizes program transactions and enforces only steering transactions.

Computations that satisfy the above properties are well-formed. Well-formed computations permit the calculation of equivalence classes, reflecting an ordering of the transactions in the computation.

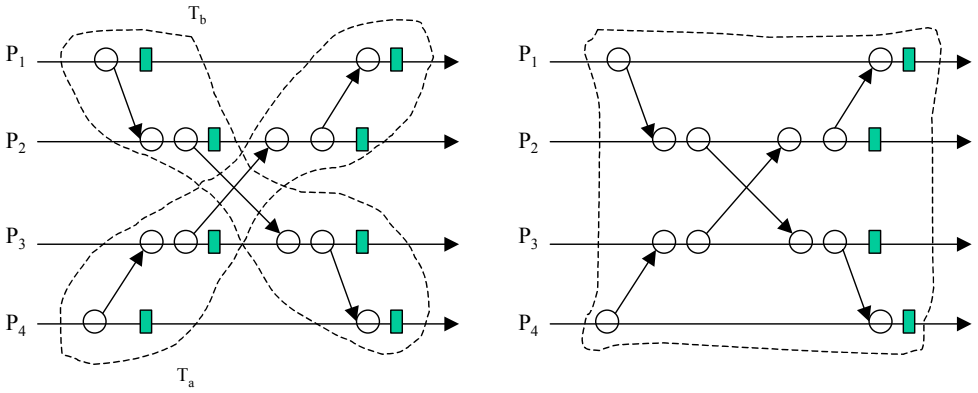


Fig. 3.2 Transaction Annotations. The solid block indicates the annotation of the end-of-transaction. (a). This shows an improper annotation. Local transactions of transaction T_b precede local transactions of transaction T_a in process P_1 and P_2, while local transactions of transaction T_a precede local transactions of transaction T_b in process P_3 and P_4. (b) This shows a proper annotation after removing the first EOT annotations of all processes.

Improper annotation may result in the appearance of a computation that is not well-formed, as seen Fig. 3.2 (a). In processes P_1 and P_2, local transactions of T_b precede local transactions of T_a, while in processes P_3 and P_4, local transactions of T_a precede local transactions of T_b. This violates the isolation requirement. However, we note that this is a result of the annotation, rather than structure of the computation itself. If we re-annotate, removing the first EOT annotations of all processes, then all events of all processes belong to a global transaction and the computation is well-formed, as seen in Fig. 3.2 (b). Careful placement of end-of-transaction annotations will allow nearly any computation to exhibit the "well-formed" property. Although program instrumentation

requires extra work for the programmer, it is not difficult for the programmer to identify the logical units of source codes.

3.2 Transaction-based Causality and Concurrency Relations

In evaluating the consistency of a steering transaction, we may need to evaluate the concurrency and causality relations between pairs of steering events, pairs of transactions, and between steering events and transactions. In this section, we define the relations necessary to evaluate these relationships. Further, we give a definition of a consistent steering transaction. Since we view transactions as atomic state changes, steering changes are constrained to be applied **between** transactions. For example, in the program for solving higher KdV equations discussed in Chapter 1, changes of Δx or Δt can only be applied between timesteps, modeled as transactions.

Definition 3.4 The transaction-based causality relation $\boldsymbol{t}\rightarrow$ between two global transactions T_a and T_b is a transitive relation satisfying the following condition: if there exists a process P_i that participates in both T_a and T_b, and the local transaction of T_a in P_i precedes the local transaction of T_b in P_i, then $T_a \boldsymbol{t}\rightarrow T_b$.

For example, in the execution depicted in Figure 3.3, $T_1 \boldsymbol{t}\rightarrow T_2$ because P_2 participated in both global transactions, and participated in T_1 before T_2.

Definition 3.5 The transaction-based concurrency relation $\boldsymbol{t}\|$ between two global transactions is defined as: $T_a \boldsymbol{t}\| T_b$ iff $\neg(T_a \boldsymbol{t}\rightarrow T_b)$ and $\neg (T_b \boldsymbol{t}\rightarrow T_a)$.

For example, in Fig. 3.3, $T_0 \boldsymbol{t}\| T_1$. However, we note that the transaction-based causality relation is more constrained than the event-based causality relation; the concurrency relation does not exist between T_1 and T_3. In Fig. 3.3, in terms of the event-based causality and concurrency relations, $s_1 \| s_3$, $s_1 \| r_3$, $r_1 \| s_3$, and $r_1 \| r_3$; therefore,

global transaction T_1 would be considered concurrent with global transaction T_3. However, in terms of the transaction-based causality and concurrency relations, T_1 $\boldsymbol{t}\rightarrow$ T_2, and T_2 $\boldsymbol{t}\rightarrow T_3$, thus T_1 $\boldsymbol{t}\rightarrow$ T_3. The transaction-based causality relation is stronger because of the view that global transactions are logically atomic actions.

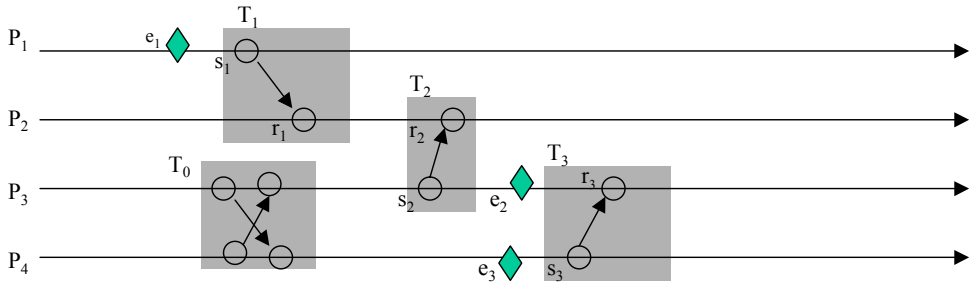


Fig. 3.3 Transaction Relations. Each global transaction is shown as a shaded area containing application events (circles). The boundaries of the shaded areas represent annotations demarking the transaction boundaries. Messages are shown as arrows between process events. Steering events are shown as diamonds.

In the remainder of this section, we define transaction-based causality and concurrency relations between a steering event and a global transaction and between two steering events. Further, we give a definition of a consistent steering transaction.

Definition 3.6 The transaction-based causality relation $\boldsymbol{t}\rightarrow$ between a steering event e in process P_i and a global transaction T is a transitive relation satisfying the following condition: if process P_i participates in global transaction T, and the local transaction of P_i in T precedes the steering event, then T $\boldsymbol{t}\rightarrow$ e; if process P_i participates in global transaction T, and the local transaction of P_i in T is after the steering event, then e $\boldsymbol{t}\rightarrow$T.

For example, in Fig. 3.3, e_1 $\boldsymbol{t}\rightarrow T_1$ and T_2 $\boldsymbol{t}\rightarrow$ e_2.

Definition 3.7 The transaction-based causality relation $\boldsymbol{t}\rightarrow$ between two steering events e_i and e_j is defined as: $e_i \boldsymbol{t}\rightarrow e_j$ iff there exists a global transaction T, such that ($e_i \boldsymbol{t}\rightarrow T$) and ($T \boldsymbol{t}\rightarrow e_j$).

For example, in Fig. 3.3, $e_1 \boldsymbol{t}\rightarrow T_1$ and $T_1 \boldsymbol{t}\rightarrow T_2$, thus $e_1 \boldsymbol{t}\rightarrow T_2$. Further, $T_2 \boldsymbol{t}\rightarrow e_2$, so $e_1 \boldsymbol{t}\rightarrow e_2$.

Definition 3.8 The transaction-based concurrency relation t|| between two steering events e_i and e_j is defined as: $e_i \boldsymbol{t}\| e_j$ iff $\neg(e_i \boldsymbol{t}\rightarrow e_j)$ and $\neg(e_j \boldsymbol{t}\rightarrow e_i)$.

For example, in Fig. 3.3, since neither $e_1 \boldsymbol{t}\rightarrow e_3$ nor $e_3 \boldsymbol{t}\rightarrow e_1$ holds, then $e_1 \boldsymbol{t}\| e_3$.

Definition 3.9 Let e_i denote the steering event of process P_i, and let SP denote the set of processes that participate in a steering transaction. A steering transaction is said to be causally consistent if: for all i, j, $i \neq j$, $i \in$ SP and $j \in$ SP, $e_i \boldsymbol{t}\| e_j$.

That is, a steering transaction is causally consistent if and only if all steering changes of that transaction are applied concurrently. For example, in Fig. 3.3, suppose steering events e_1, e_2 and e_3 belong to the same steering transaction, the steering transaction is not consistent because $e_1 \boldsymbol{t}\rightarrow e_2$.

3.3 Transaction-based Vector Time

A straightforward approach for consistency detection is to identify the concurrency relation between all steering events of a given steering transaction. If all steering events are concurrent, then the steering transaction is consistent; otherwise, it is inconsistent. Vector time schemes [Fid88, Mat89] have proved a good way for identifying concurrent events in a distributed system. Here, we describe a vector time based algorithm for identifying the concurrency relation between steering events. In our transaction-based system, the logical time of each process is measured by the *number of past local*

transactions that have occurred at the process. The vector time of process P_i can be defined as follows:

Definition 3.10 Let P_1, …, P_N denote the processes of a distributed computation. The vector time V_i of process P_i is maintained according to the following rules:

(1) Initially, $V_i[k] = 0$ for k = 1,…,N

(2) On each steering event e, process P_i increments V_i as follows: $V_i[i] = V_i[i] + 0.5$;

(3) At the start of each new local transaction, process P_i increments V_i as follows:

$$V_i[i] = \lceil V_i[i] + 0.5 \rceil;$$

(4) At the end of a global transaction T, update $V_i[j] = \max_{k \in SP}(V_k[j])$, for j = 1, …, n.

Since at the end of a global transaction T, all participating processes will exchange their knowledge about logical times of other processes and synchronize their vector clock, as described in the rule (4), then the vector clocks of all participating processes will be the same when the global transaction T is completed, which is denoted as V(T). V(T) is said to be the vector timestamp of global transaction T. Let V(e) denote the vector time V_i that results from the occurrence of steering event e in process P_i. V(e) is said to be the vector timestamp of steering event e. Informally, the component $V_i[i]$ of process P_i's current vector time reflects the accurate logical time at P_i (measured in "number of past local transactions" at P_i), while $V_i[k]$ is the best estimate P_i is able to derive about P_k's current logical clock value $V_k[k]$.

Definition 3.11 Let u, v denote vector times of dimension m [SM94].

(1) $u \leq v$ iff $u[k] \leq v[k]$ for k = 1, …, m

(2) $u < v$ iff $u \leq v$ and $u \neq v$

(3) $u \parallel v$ iff $\neg(u < v)$ and $\neg(v < u)$.

Theorem 3.12 For two global transactions T_a and T_b of a distributed computation, we have

(1) $T_a \ t\rightarrow T_b$ iff $V(T_a) < V(T_b)$

(2) $T_a \ t\| T_b$ iff $V(T_a) \| V(T_b)$

Proof.

Suppose $T_a \ t\rightarrow T_b$ holds. Then, by the definition of the transaction-based causality relation, there exists a sequence of global transactions such that $T_a = T_0 \ t\rightarrow T_1 \ t\rightarrow \ldots \ t\rightarrow T_s = T_b$, where there exists a process P_j that participates in both T_i and T_{i+1}, and T_{i+1} is the next global transaction after T_i in P_j. According to the definition of vector time, we have $V(T_{i+1})[j] = V(T_i)[j] + 1 > V(T_i)[j]$ and $V(T_{i+1})[k] \geq V(T_i)[k]$ for k = 1, …, n, which implies $V(T_{i+1}) > V(T_i)$. Therefore, $V(T_a) = V(T_0) < \ldots < V(T_s) = V(T_b)$.

Conversely, suppose $V(T_a) < V(T_b)$ holds. Assume P_i is a process that participated in global transaction T_a. Then, $V(T_a)[i]$ is the accurate number of local transactions that happened at P_i when T_a happened, and $V(T_b)[i]$ is the best estimate that participant processes in T_b were able to derive about the number of local transactions that happened at P_i. Since $V(T_b)[i] \geq V(T_a)[i]$, then either T_a must happen before T_b, or T_a is the same as T_b. Since $V(T_a) < V(T_b)$, then they cannot be the same. Therefore, $T_a \ t\rightarrow T_b$.

Property (2) follows immediately from (1) and the definition of the transaction-based concurrency relation. ■

Theorem 3.13 For two steering events e_i and e_j of a distributed computation, we have

(1) $e_i \ t\rightarrow e_j$ iff $V(e_i) < V(e_j)$

(2) $e_i \ t\| e_j$ iff $V(e_i) \| V(e_j)$

Lemma 3.14 For two steering events e_i in process P_i and e_j in process P_j, we have,

(1) $e_i \ t\rightarrow e_j$ iff $V(e_i)[i] \leq V(e_j)[i]$

(2) $e_i \ t\| e_j$ iff $V(e_i)[i] > V(e_j)[i]$ and $V(e_j)[j] > V(e_i)[j]$

Lemma 3.14 states that we can restrict the comparison to just two vector components in order to determine the precise causal relationship between two steering events if their origins P_i and P_j are known. The intuitive meaning of the lemma is easy to understand. If the "knowledge" of steering event e_j in process P_j about the number of local transactions in P_i is at least as accurate as the corresponding "knowledge" $V(e_i)[i]$ of e_i in P_i, then there must exist a chain of transactions which propagated this knowledge from e_i at P_i to e_j at P_j, hence $e_i \ t\rightarrow e_j$ must hold. On the other hand, if event e_j is not aware of as many events in P_i as is event e_i, and event e_i is not aware of as many events in P_j as is event e_j, then both events have no knowledge about each other, and thus they are concurrent.

CHAPTER 4

CONSISTENT STEERING

In this research, we investigate two approaches for achieving consistent steering: conservative steering and optimistic steering. The conservative steering approach avoids inconsistent steering by strictly adhering to the causality constraint. This typically involves blocking the computation before a consensus decision is made and steering changes are applied. On the other hand, the optimistic steering approach 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 be able to detect any inconsistent steering transaction and provide a checkpointing/rollback mechanism to restore the computation to a correct state.

In this chapter, we first present the two approaches for consistent steering of distributed computations, the conservative approach and the optimistic approach. Both approaches have been implemented in the Pathfinder system. We then discuss the relative costs and benefits of the optimistic and conservative approaches.

4.1 Conservative Approach

The conservative steering approach avoids inconsistent steering by strictly adhering to the causality constraint. That is, no two steering events in the steering transaction may

be causally dependent. Thus, in conservative steering, steering changes cannot be applied until the next steerable points are confirmed to be concurrent.

4.1.1 Conservative Approach Overview

A user can issue a steering request at anytime during the computation. Upon receiving a steering request from the user, the SM sends out the *prepare-to-steer* to all steered processes. A process receiving a *prepare-to-steer* message from the SM will block its execution at the next steerable point (EOT point), report back to the SM, and wait for the confirmation from the SM. A blocked (passive) process may become active again when another process requests it either to send a message or to receive a message in order that all steered processes reach a consistent point. When the SM has determined that all steered processes have blocked at a consistent point, it then issues a *steering-change* command to all steered processes. Upon receiving the steering-change command, the steered processes then apply steering changes and continue execution. Thus, the conservative steering approach requires that all steered processes be synchronized at the consistent steerable point before steering changes are applied. In conservative steering, the computation blocks while the SM is checking the consistency.

In the following section, we will discuss process state and some control messages for conservative steering. Consistency checking and process synchronization will be discussed in section 4.1.3.

4.1.2 Process State and Control Message

Within the Pathfinder system, communication traffic exists between both the executing processes and between the various control modules of the Pathfinder system. We refer to these messages, respectively, as application messages and control messages.

For the conservative approach, control messages include *prepare-to-steer* messages, *steering-change* messages, *message-send-request* messages, and *message-receive-request* messages. A *prepare-to-steer* message informs a process that it is involving a steering transaction. A *steering-change* message informs a process that it can go ahead to apply the steering change. Both *message-send-request* and *message-receive-request* messages are used for synchronizing steered processes. A *message-send-request* informs a process that it must send the requested message before it blocks (become passive). Similarly, A *message-receive-request* informs a process that it must receive the requested message before it blocks (become passive).

A steered process is defined as a process that has already received the *prepare-to-steer* command. An affected process is defined as a process that participates in the same transaction with a steered process. A process may be in one of two states: *active* or *passive*. *Active* processes are those currently working on a computation; *passive* processes are waiting. However, a *passive* process can still be able to process any control messages, such as message-request, steering-flag, etc. A process may go through several state changes from passive to active and from active to passive before it reaches a consistent point. A steered active process becomes passive when it reaches a steerable point, no process is requesting additional messages, and no process is requesting it to receive a message. A passive process becomes active if a *message-send-request* message or a *message-receive-request* message arrives, forcing it to participate in another transaction.

When all steered processes in the system are passive and every global transaction that steered processes were involved in has completed (this guarantees that passive processes

will not become active), steering changes then can be applied concurrently across all involved processes.

4.1.3 Consistency Checking and Process Synchronization

Within the Pathfinder system, the SM is responsible for checking consistency. When a steered process becomes passive, it will report its state to the SM. A consistent point is reached when (1) all steered processes in the system are passive and (2) each global transaction that involved steered processes has completed. The first condition guarantees that all steered processes have reached a steerable point. The second condition guarantees that no two steering events can be causally dependent on each other.

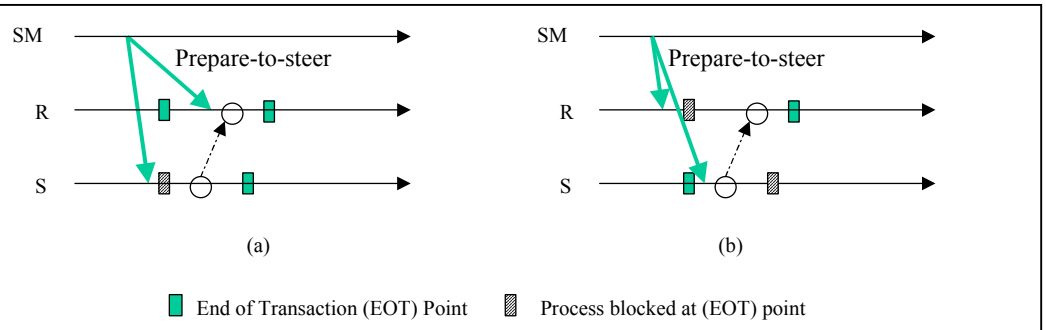


Fig. 4.1 Process Synchronization. (a) A passive process S must become active again when it need send a message to a steered process; (b) A passive process R become active again when it need receive a message from a steered process.

Consistency checking appears very simple. However, a steered process may not be able to reach a steerable point if it is waiting for a message from a passive process, as seen in Fig. 4.1 (a). In this case, the sender must become active and execute forward to send the message. For this reason, when a steered process R attempts to receive a message M and M has not arrived, R sends a *message-send-request* message to the sender S. The message M may not have arrived because its sender S is passive or because the

execution of S is slow. If M has not been sent because S is passive, then S becomes active and executes forward to send M. Process S remains active and continues execution until it sends the message M and reaches the next steerable point. If M has not been sent because S is slow, then S remains active and continues execution until it sends the message M and reaches the next steerable point. If M has already been sent, process S ignores the request. Here, we assume that the receiver always knows which process is the message sender.

Another potential problem is that a global transaction that involves steered processes may not be able to complete if the receiver of a message from a steered process is blocked at an earlier steerable point, as seen in Fig. 4.1 (b). In this case, the receiver must become active and executed forward to receive the message in order to complete the global transaction. For this reason, before a process become passive, it will broadcast a **message-receive-request** message to all its neighbors, processes with which it has communicated. This informs its neighbors that all messages it has sent should be received in order to complete the global transaction. If M has not been received, then R becomes active and executes forward to receive M. Process R remains active and continues execution until it reaches the next steerable point. If M has already been received, process R ignores the request.

In a *message-send-request* message, it is necessary to specify which message must be sent. The *message-send-request* contains an identifying message number, the number of messages that have been received from S by the receiver. Similarly, in a *message-receive-request* message, it is necessary to specify which message must be received. The *message-receive-request* contains an identifying message number, the number of

messages that have been sent by the sender. The IM counts the messages sent to or received from every process and uses this count to uniquely identify messages.

Here, we borrow some ideas from the halting algorithm for a global Conjunctive Predicate [MI92], a variation of the distributed termination detection problem [DS80]. However, there are several special properties when the computation is transaction based. First, termination is detected when all steered processes in the system are passive and each global transaction that involved steered processes has completed. Second, it is only necessary to block steered processes, not all processes. Finally, a passive process must become active again when it needs to receive a message from a steered process or when it needs to send a message to a steered process in order to complete the current transaction.

When termination is detected, the SM then knows a consistent point has been reached and issues a *steering-change* command to all steered processes. Upon receiving the steering-change command, the steered processes then apply steering changes and continue execution. Thus, the conservative steering approach requires that all steered processes be synchronized at the consistent steerable point before steering changes are applied. A detailed description of the conservative steering algorithm is shown in Fig. 4.2.

Fig. 4.2 Conservative Steering Algorithm

Snapshot Manager:

```
Send prepare-to-steer to all steered processes;

IF (all steered processes are passive) AND
        (global transactions that each steered process P_i was involved in at LVT_i are completed)
    Send steering-change to all steered processes;
```

Interaction Manager / User Process:

```
// Initialization
state := active;
steeringFlag := false;

for i := 1 to N do {      // N = number of processes
    s[i] := 0;        // the number of messages sent to P_i
    r[i] := 0;        // the number of messages received from P_i
    rs[i] := 0;       // P_i requested to send up to rs[i]
    rr[i] :=0;        // P_i requested to receive up to rr[i]
}
```

When message arrives from the SM:

```
SWITCH message type
    CASE  prepare-to-steer:
        steeringFlag := true;

        IF the process is waiting for a message from P_i THEN
            k := r[i] + 1;
            send a message-send-request(k) to P_i;
        END IF

    CASE  steering-change:
        apply the steering change;
        steerFlag := false;
        state := active;
        continue execution;
END SWITCH;
```

EOT:

```
IF steeringFlag = true THEN
    IF for all i (s[i] >= rs[i]) AND (r[i] >=rr[i]) THEN
        FOR each process P_i DO
            send message-receive-request (s[i]) to P_i;
        END FOR
        state := passive;
        send (passive, LVT_i) to the SM;
        sleep;

    END IF;
END IF
```

Message Send to P_i:

```
s[i]++;
send the message;
```

Message Receive from P_i:

```
IF the message is not in the queue THEN
    IF steeringFlag = true THEN
        k := r[i] + 1;
        send a message-send-request(k) to Pi;
    END IF
    wait for the message until it arrives;
END IF
r[i]++;
process the message;
```

When control message arrives from P_i:

```
SWITCH message type
        CASE  message-send-request(k):
            rs[i] := k;

            IF steeringFlag = false  Then
                steeringFlag := true;
            END IF;

            IF state = passive THEN
                IF s[i] < rs [i] THEN
                    state := active;
                    continue execution;
                END IF
            END IF

        CASE  message-receive-request(k):
            rr[i] := k;
            IF steerFlag = false THEN
                steerFlag := true;
            END IF

            IF (k > r[i]) AND (state = passive) THEN
                state := active;
                continue execution;
            END IF
END SWITCH;
```

4.2 Optimistic Approach

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 be able to detect any inconsistent steering transaction and provide a checkpointing/rollback mechanism to restore the computation to a correct state.

In optimistic steering, IMs are 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.

4.2.1 Optimistic Steering Algorithm Overview

A user can issue a steering request at anytime 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 action at the next EOT (end of transaction) 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 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.

In the following section, we discuss two consistency detection algorithms: the vector time based algorithm and the history-based algorithm. The checkpointing/rollback algorithms will be discussed in detail in section 4.2.4.

4.2.2 Process State and Application Message Type

The application processes may be in any one of three states: **normal, tentative,** or **recovering,** as seen in Figure 4.3. A *normal* state implies that neither the consistency of a steering change is in question nor is the correction of an inconsistency underway. A tentative state implies that a process is executing speculatively, its result will be canceled if the steering transaction is determined to be inconsistent. A process enters a tentative state from a normal state when its state is affected by a steering transaction. A process can be affected by a steering transaction, either directly by applying a steering change or indirectly by receiving a message from a process already in a tentative state. In order to

be able to cancel the speculative execution if the steering transaction is determined to be inconsistent, the process must take a checkpoint before entering the tentative state. Finally, a process may only enter a recovering state if it is in a tentative state and the associated steering transaction is determined to be inconsistent.

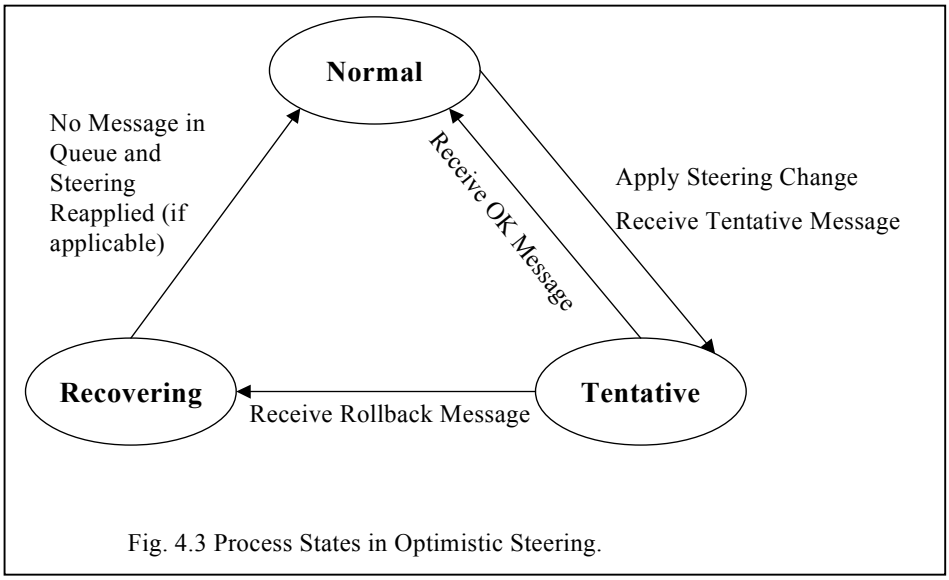


Fig. 4.3 Process States in Optimistic Steering.

For optimistic steering, control messages include *Steering* messages, *ACK* messages, *OK* messages and *Rollback* messages. These control messages will be discussed in the following section. A *Steering* message informs a process that it involves in a steering transaction and should apply the change at the next steerable point. An *ACK* message tells the SM that the process has applied a steering change at the certain time. OK messages inform processes that the steering changes have been applied consistently across processes. Rollback messages inform processes that the steering changes have been applied inconsistently, the computation must roll back to the state before applying steering changes.

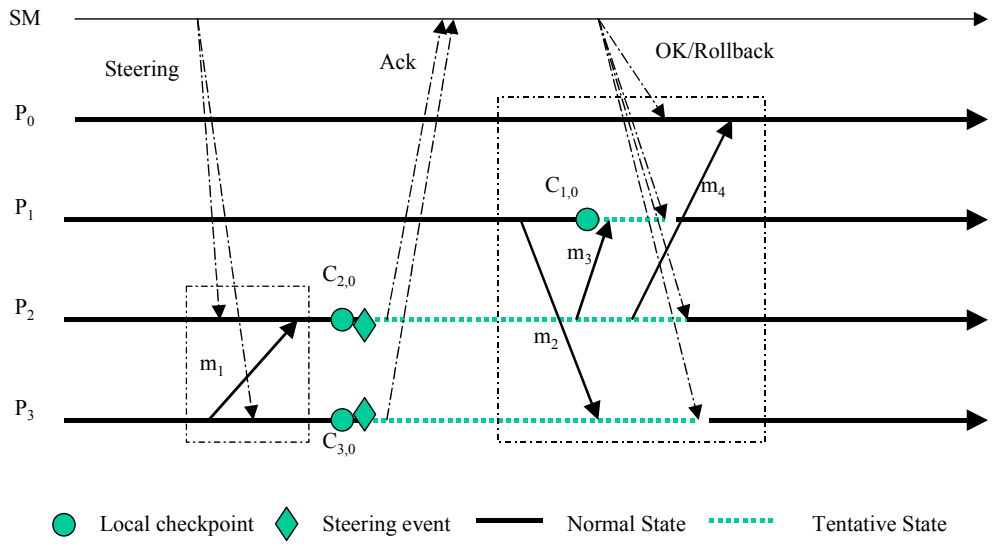


Fig. 4.4 Optimistic Steering Example

Application messages can be further grouped into *normal* messages**,** *tentative* messages**,** and *in-transit* messages. A message is a tentative message if the sender is in the tentative state. Depending on the state of the steering transaction, tentative messages may be in any one of three states: *unconfirmed*, *confirmed, obsolete*. Upon receiving a tentative message, if the associated steering transaction has not been confirmed, the tentative message is in an *unconfirmed* state; if the associated steering transaction has already been confirmed to be consistent, the tentative message is in a *confirmed* state; if the associated steering transaction has been determined to be inconsistent, the tentative message is in an *obsolete* state. A process in the normal state will enter the tentative state when it receives an unconfirmed tentative message. A confirmed tentative message will be considered as a normal message. An obsolete tentative message will be discarded by the receiver because it is out of date and will be resent during recovery. A message is an in-transit message if the sender is in the normal state and the receiver is in the tentative

state. An *in-transit* message cannot be resent during recovery. Thus, it must be recorded by the receiving process. Within the Pathfinder system, the recording of the in-transit message is referred as message logging.

For example, in figure 4.4, message m_1 is a normal message because both the sender P_3 and receiver P_2 are in the normal state. Message m_2 is an in-transit message because the sender P_1 is in the normal state and the receiver P_3 is in the tentative state. Message m_3 is an unconfirmed tentative message because the sender P_2 is in the tentative state and the consistency of the associated steering transaction has not been determined when m_3 is received by P_1. Tentative message m_4 is received after the steering transaction has been confirmed, so its state will depend on the state of the steering transaction. If the steering transaction is determined to be consistent, message m_4 will be in the confirmed state, and process P_0 will merely receive it and remain in the normal state. If the steering transaction is determined to be inconsistent, message m_4 will be in the obsolete state, and process P_0 will discard this message and wait for a new copy of the message.

4.2.3 Consistency Detection

As described earlier, in optimistic steering, the system invokes each steering event in the steering transaction at the respective process without concern for or knowledge of the state of any other process. Therefore, the consistency of the steering transaction must be checked. If the consistency is verified, the computation continues. If the steering transaction is found to be inconsistent, then the earliest time at which the steering event could be consistently applied at each process is calculated. In this section, we describe two algorithms for the detection of inconsistency, verification of consistency and calculation of the earliest consistent steering time.

4.2.3.1 Vector Time Based Approach

To detect the consistency of a steering transaction, we can compute the causality and concurrency relations between steering events by comparing the vector timestamps of all steering events according to theorem 3.13. If they are concurrent, then the steering is consistent, otherwise not. To do this, we need m * (m-1) / 2 vector comparisons in the worst case, where m is the number of affected processes. This is easy, but not efficient. The following theorem gives a more efficient way to detect inconsistency, verify consistency and calculate the earliest consistent steering time.

Theorem 4.1 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 $i \in SP$. 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 $i \in SP$. Then,

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

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

Proof

(1) We show that CV is consistent.

For any $i, j \in SP$, let e_i' denote the steering event that is applied at time CV[i] at process P_i, and let e_j' denote the steering event that is applied at time CV[j] at process P_j. We have $V(e_i')[i] = CV[i] = \left\lfloor \max_{k \in SP} V(e_k)[i] \right\rfloor + 0.5 > V(e_j')[i]$, which implies $V(e_i')[i] > V(e_j')[i]$. Similarly, we have $V(e_j')[j] > V(e_i')[j]$. By lemma 5.5, we have $e_i' \parallel e_j'$

(2) We show that CV is the earliest consistent steering time.

If SV = CV, then CV is the earliest consistent steering time.

If SV < CV, then for any vector SV' such that SV ≤ SV' < CV, we show that steering events applied at SV' are not consistent. Let e_i' denote the steering event that is applied at time SV'[i] at process P_i. Since SV' < CV, then there exists at least one element j, SV'[j] < CV[j], then SV'[j] + 1 ≤ $CV[i] = \left\lfloor \max_{k \in SP} V(e_k)[i] \right\rfloor + 0.5$. This implies SV'[j] + 0.5 < $\left\lfloor \max_{k \in SP} V(e_k)[i] \right\rfloor \leq \max_{k \in SP} V(e_k)[i]$, let's say that V($e_k$)[j] = $\max_{k \in SP} V(e_k)[i]$. Then, we have V($e_j$')[j] = SV'[j] < V($e_k$)[j] ≤ V($e_k$')[j], which implies V($e_j$')[j] < V($e_k$')[j]. According to lemma 3.14, we have e_j' ***t***→ e_k'. Therefore, SV' is not consistent. ■

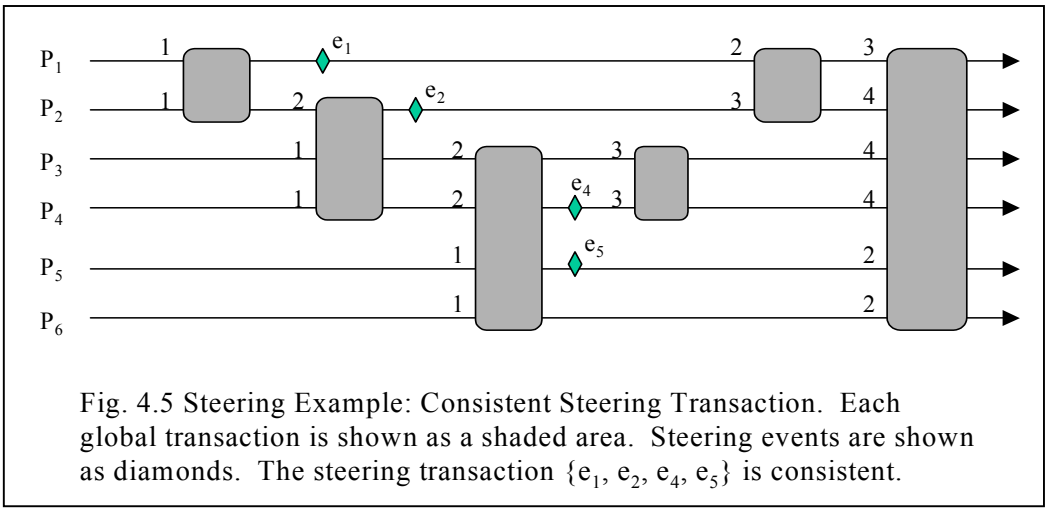


Fig. 4.5 Steering Example: Consistent Steering Transaction. Each global transaction is shown as a shaded area. Steering events are shown as diamonds. The steering transaction {e_1, e_2, e_4, e_5} is consistent.

Fig. 4.5 shows an example of a consistent steering transaction, consisting of four steering events: e_1, e_2, e_4, and e_5. SV = {1.5, 2.5, -, 2.5, 1.5, -}; V(e_1) = {1.5, 1, 0, 0, 0, 0}; V(e_2) = {1, 2.5, 1, 1, 0, 0}; V(e_4) = {1, 2, 2, 2.5, 1, 1};V(e_5) = {1, 2, 2, 2, 1.5, 1}; According to theorem 4.1, $CV[i] = \left\lfloor \max_{k \in SP} V(e_k)[i] \right\rfloor + 0.5$, for all i ∈ SP. Then, we have CV = {1.5, 2.5, -, 2.5, 1.5, -). Since SV =CV, the steering transaction is consistent.

Fig. 4.6 shows an inconsistent steering transaction, consisting of four steering events: e_1, e_2, e_4, and e_5. SV = (0.5, 1.5, -, 2.5, 1.5, -); $V(e_1)$= {0.5, 0, 0, 0, 0, 0}; $V(e_2)$= {1, 1.5, 0, 0, 0, 0}; $V(e_4)$= {1, 2, 2, 2.5, 1, 1}; $V(e_5)$= {1, 2, 1, 1, 1.5, 0}; thus, CV = {1.5, 2.5, -, 2.5, 1.5, -}. Since SV ≠ CV, the steering transaction is not consistent.

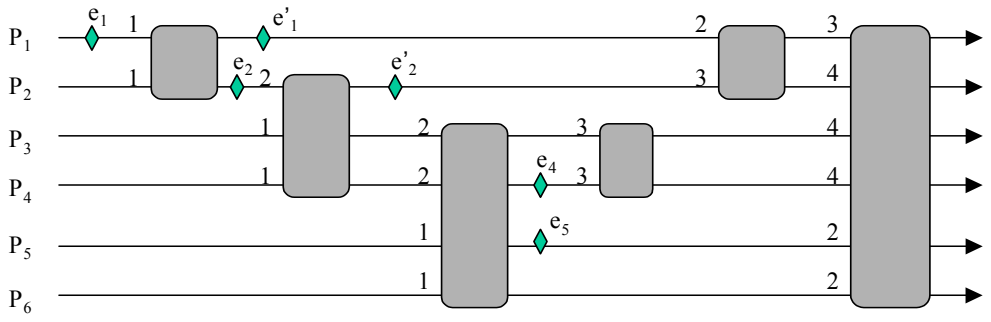


Fig. 4.6 Steering Example: Inconsistent Steering Transaction. Each global transaction is shown as a shaded area. Steering events are shown as diamonds. The steering transaction {e_1, e_2, e_4, e_5} is inconsistent. The earliest consistent steering transaction is shown {e'_1, e'_2, e_4, e_5}.

4.2.3.2 History-Based Approach

According to definition 3.4, we say that two steering events are causally dependent if there exists a transaction T, such that (e_i $\boldsymbol{t}\rightarrow$ T) and (T $\boldsymbol{t}\rightarrow$ e_j). Further, a steering transaction is causally consistent if and only if all steering changes of that transaction are applied concurrently. An equivalent view of consistency considers a steering transaction as a program transaction; the consistency criterion is that the computation, including the inserted steering transaction, still meets the *well-formed* requirement. This equivalence suggests that consistency of a steering transaction can be detected by checking the consistency between a steering transaction and each program transaction. The history-based algorithm is based on this idea.

In order to determine the consistency of a steering transaction, the system maintains a chronological list of vectors representing a partial ordering of a subset of the program transaction history. The program transactions contained in this subset, or window of interest, are all program transactions that occur between the first and last steering events of a steering transaction. To determine the consistency of a steering transaction, a vector representing the time of the steering transaction is compared against the partial program transaction history. If the steering transaction is consistent, the algorithm returns such; however, if the given steering transaction is inconsistent, the algorithm returns the earliest consistent time after the given steering transaction for which the steering changes could logically occur.

To accomplish consistency detection, the algorithm creates a consistency vector representing a *consistent cut* [Lyn96] at which a steering transaction could be applied. The algorithm works backward, comparing the steering transaction with each program transaction, generating vector times for consistent cuts. The algorithm stops when all program transactions have been checked or the system can decide that all remaining transactions happened before the original steering transaction. The generated consistency vector is the earliest time at which the steering transaction could have been consistently applied. If the consistency vector is the same as the steering vector SV, then the steering is consistent; otherwise, it is inconsistent.

This algorithm, seen in figure 4.7, requires six data structures and one Boolean variable. First, a *TLP (Transaction Labeling Protocol)* table is used to maintain the chronological history of program transactions. Next, there are four vector times, a Boolean vector, and a Boolean variable*: TV (Transaction Vector), SV (Steering Vector),*

CV (Consistency Vector), CVTemp, *Verified*, and *consistent*, respectively. Figure 2.7 shows an example of the initialized data structures. The *TV* vector holds the row of the *TLP* table currently being analyzed. The *SV* vector contains the timestamps representing the steering transaction. The *CV* vector represents the time of a consistent steering transaction. The *CVtemp* vector provides a temporary holder of possible new timestamps for the *CV* vector. The values of *CVtemp* should not be committed to the *CV* vector until all elements of the *SV* vector have been compared against corresponding elements in the *TV* vector. Both the *CV* vector and *CVtemp* vector are initially empty. The Boolean *Verified* vector contains flags signifying that the earliest timestamp for a steering event to occur at each respective process has been verified. If a TRUE flag is present, then no new timestamp for that process should be added to the *CV* vector. *Verified* is initialized with all elements set to FALSE. Finally, the Boolean variable *consistent* indicates whether the values stored in *CVtemp* should be committed to the *CV* vector.

At the beginning of each iteration of the WHILE loop beginning on line 16, *consistent* is set to TRUE. This WHILE loop is used to determine the stopping point for the algorithm. The algorithm terminates once all elements of the *Verified* vector that correspond to elements of the *SV* vector have been set to TRUE. The key point of analysis occurs in the FOR loop starting on line 20. Here, each non-empty element of the *TV* vector is compared with the corresponding non-empty element of the *SV* vector. If the element compared in the *TV* vector holds a timestamp equal to or later than that of the element in the *SV* vector, then that timestamp is entered into the corresponding element in *CVtemp*. If *consistent* remains TRUE through all iterations of the FOR loop, then the values of *CVtemp* are committed to the *CV* vector. However, if any element of the *TV*

vector occurred earlier than the corresponding element in the *SV* vector, then *consistent* will be changed to FALSE and the loop will terminate, as seen in lines 33 and 34. All entries in *CVtemp* are then purged and all elements of the *Verified* vector corresponding to elements of *TV* are marked as TRUE. This later action indicates that the present *TV* vector was concurrent with the *SV* vector. As explained, any concurrency implies an inconsistent steering transaction.

One other condition will cause the FOR loop to terminate without completing all iterations. If any element of the *TV* vector corresponding to an element of the *Verified* vector has already been set to TRUE, then all elements of the *Verified* vector corresponding to elements of the *TV* vector will be set to TRUE, and the loop terminates. As above, if any element of the *TV* vector has already been verified, then the earliest time at which a steering event could have occurred for that process has happened. Therefore, no other process having direct or transitive communication with that process could consistently apply a steering action during the program transaction represented by that *TV* vector or any earlier *TV* vector.

Once the condition has been satisfied that all elements of the *Verified* vector corresponding to elements of the *SV* vector have been set to TRUE, the WHILE loop will terminate and a comparison between the *CV* vector and *SV* vector occurs. If the *CV* vector and *SV* vector are found to be identical, the algorithm returns TRUE. The IS system can then purge all checkpoints and stop any message logging. If the *CV* vector is not equal to the *SV* vector, then the algorithm returns the *CV* vector. The IS system can then issue a command for each process to rollback to the checkpoint at the time specified

```
1.  Table TLP /*Table containing transaction history*/
2.  Vector TV /*Vector holding information about present transaction in TLP*/
3.  Vector SV /*Steering Vector containing list of processes involved in steering transaction*/
4.  Vector CV /*Consistent Vector representing when the steering transaction should take place*/
5.  Vector CVtemp /*Temporary vector to hold information until it is verified SV has not made TV
6.      inconsistent*/
7.  Vector Verified /*A Vector of Boolean values set to true when a process listed in SV is at its
8.     earliest logical time to have invoked the steering command*/
9.  Boolean consistent /*Boolean flag to indicate if SV has made TV inconsistent*/
10.
11. BEGIN
12.
13. set all elements of Verified to FALSE
14. set TV equal to last vector of TLP table
15.
16. WHILE (all processes in SV have not been set to true in Verified)
17.       BEGIN
18.            consistent set to TRUE
19.
20.            FOR (compare each corresponding, non-empty cell in TV and SV)
21.                 BEGIN
22.                      IF(any non-empty cell in TV corresponds to a cell marked TRUE
23.                      in Verified)
24.                           THEN mark all cells in Verified corresponding to non-
25.                            empty cells in TV TRUE
26.                            set consistent to FALSE
27.                            break
28.
29.                      IF(TV is greater than or equal to SV)
30.                           THEN set corresponding cell of CVtemp to TV
31.                      ELSE

32.                           Mark Verified cells corresponding to non-empty TV cells to TRUE
33.                           and mark consistent to FALSE
34.                           break
35.                 END

36.
37.            IF(consistent)
38.                 THEN
39.                      FOR(each non-empty element of CVtemp)
40.                           set corresponding cells of CV to CVtemp
41.
42.            IF(all cells of Verified corresponding to all cells SV are marked true)
43.                 break
44.            ELSE
45.                 set TV equal to previous vector in TLP
46.       END
47.
48. IF(SV equals CV)
49.       Return Consistent
50. ELSE
51.       Return CV
52.
53. END
```

Figure 4.7 – History-Based Consistency Detection Algorithm

4.2.4 Checkpointing, Message Logging, and Rollback

When an inconsistent steering transaction is detected, the system must restore the computation to a correct state. In this section, we discuss the checkpointing, message logging, and rollback mechanisms used in the Pathfinder system to permit recovery of state.

Checkpointing and restart of a process is a standard technique used to protect work in progress from hardware or software failure. While the checkpointing mechanisms used in restart are similar to those used in rollback, rollback is distinguished from restart in that the process *P*, instead of being killed and recreated, is simply returned to some previous state. Process *P* retains its process ID and all associated kernel resources. By retaining its process ID and kernel resources, other processes in the system can still communicate with *P*, and its communication channel and files stay open.

4.2.4.1 Local Checkpoint

The computational state of a process is recorded in the form of a checkpoint before a steering change can be invoked or before a tentative message can be processed. The Pathfinder system maintains a checkpoint that includes:

- The state of the execution stack and all local variables
- All dynamic and global variables explicitly listed for protection by the programmer
- The CPU state, including but not limited to the program counter (PC) and stack pointer (SP).

Maintaining the execution stack and the CPU state allows the Pathfinder system to seamlessly restart a process's execution at the exact point at which the checkpoint was

taken. Since the mechanism is rollback instead of restart, there is no need to restore code space and kernel space state data.

As described in Chapter 2, in the Pathfinder system each process in a distributed computation is "wrapped" by an IM, communication layers between the process and its communication environment. Included in these layers is the optimistic steering module, which maintains process state, message logs, and checkpoint file handles. Each layer of the IM is loaded dynamically at runtime and is thus not part of the local set of variables. During a checkpoint, it is neither desirable nor necessary to record the state of the IM, as this would introduce both wasted storage for the checkpoint and excessive checkpoint restoration time during a recovery. Therefore, to allow dynamic memory protection, the IS system provides the programmer with an explicit protocol through which dynamic variables created during a process's execution can be specified for inclusion in checkpoints. Figure 4.8 provides a code sample illustrating this.

```
QBV *qbv = new QBV(...); //Allocate first IM layer
char *mem = new char[100]; //Dynamically allocate an array of chars
qbv->protectMem(mem, sizeof(char) * 100);//Memory to include in checkpoints
```

Fig. 4.8 Memory Protection Example

All checkpoint data is written to two binary files per process. The local variables and execution stack are written into one file while all protected dynamic memory is written to a second file. Since the local set of variables and execution stack both lie contiguously in the program's stack, a direct memory dump can be made from the program stack to file

during a checkpoint. However, a slightly more elaborate scenario exists for dynamic memory checkpointing.

Through a Pathfinder protocol, a programmer can explicitly protect any dynamic memory that needs to be checkpointed. The programmer specifies the base memory address for a variable and the number of contiguously associated bytes, as seen in Fig. 4.8. This information is then stored in a data structure. During checkpointing, the data structure of each process is traversed and all specified memory is contiguously dumped to the second binary file.

Finally, the state of the stack context/environment must be stored out so that on a rollback the process state is just as it was when the checkpoint was taken. Fortunately, the *setjmp.h* library provides an API for both storing out and recovering such environmental states.

4.2.4.2 Consistent Global System State

Checkpointing and restoring the state of a single process is relatively simple, but processes interact with other processes. There are mainly two ways for recovering and restoring the state of a distributed computation: a consistent global checkpoint scheme, and a dependency tracking scheme. In our implementation, a globally consistent checkpoint mechanism is used. A consistent system state is one in which every message that has been received is also shown to have been sent in the state of the sender [CL85].

A global state includes the state of all processes and state of communication channels. In our implementation, a checkpoint is used for saving process state; message logging and replay are used for saving and recovering channel state.

The goal of optimistic steering is to avoid global synchronization. Therefore, coordinated checkpointing that involves global synchronization is not desirable. However, uncoordinated checkpointing is susceptible to the "domino effect" [Ran75], in which cascading rollback propagation may force the system to restart from the initial state. Communication-induced checkpointing allows processes in a distributed computation to take independent checkpoints and to avoid the domino effect. Therefore, communication-induced checkpointing is favorable in optimistic steering.

A consistent global state includes process state and channel state. While checkpointing is used for recording process state, message logging is used for recording channel state. As described earlier, a process must take a checkpoint before it enters the tentative state from the normal state. A process enters the tentative state from the normal state when its state is first affected by the steering transaction, either by applying a steering change or receiving an unconfirmed tentative message. Every in-transit message should be logged by the receiving process. To correctly take a checkpoint and log the in-transit messages, a process must be able to recognize different kind of messages. A simple index will solve this problem.

Within the Pathfinder system, each process maintains a checkpoint index, initially 0. Each steering transaction and its steering actions are associated with an index, which is greater than 0 and sequentially increasing. Every message piggybacks the sending process's checkpoint index. A message is an in-transit message if its piggybacked index is less than the receiving process's checkpoint index. A message is a tentative message if its piggybacked index is less than the receiving process's checkpoint index. Upon applying a steering change whose index is greater than the local index, a process takes a

checkpoint and sets the local index to the index of the steering action and sets its state to *tentative*. Upon receiving a tentative message, the receiver is forced to take a checkpoint before processing the message to avoid inconsistency. A process in the *tentative* state records all incoming in-transit messages. The detailed checkpointing algorithm is shown in Fig. 4.9.

Fig. 4.9 The index-based communication-induced checkpointing algorithm

a. Each process maintains a checkpointing index, initially 0. Each steering transaction and its steering actions are associated with an index, which is greater than 0 and sequentially increasing.
b. Upon applying a steering change whose index is greater than the local index, a process takes a checkpoint and sets the local index to the index of the steering action and sets its state to *tentative*.
c. When a process is in the *tentative* state, the checkpoint index will be piggybacked on every post-checkpoint outgoing message.
d. Upon receiving a message with a piggybacked index greater than the local index, the receiver is forced to take a checkpoint before processing the message to avoid inconsistency. It then updates its local index to the piggybacked index, and sets its state to *tentative*.
e. A process in the *tentative* state records all incoming normal messages (in-transit messages).

By doing this, all local checkpoints avoid involvement in any Z-cycles [NJ95], which have the potential to produce "useless" checkpoints. The "useless" checkpoints are checkpoints that cannot be incorporated into any consistent global checkpoint. Also, each process need take at most one checkpoint for each steering transaction. Further, only those processes whose execution is dependent on the consistency assumption need take a checkpoint.

Therefore, this protocol does not need any extra control messages for synchronization purposes and all forced checkpoints are needed for recovery. The only synchronization overhead is the piggybacked index on top of the application message.

4.2.4.3 Rollback and Re-execution

In optimistic steering, the system must have the ability to restore the state to what it was before the inconsistency occurred and re-execute a portion of the computation to a

consistent state and reapply the steering changes. The SM is responsible for verifying the consistency of each steering transaction. If the steering transaction is determined to be consistent, the SM will issue an *OK* message to all processes, all checkpoints will be discarded, message log queues cleared, and all processes will transition back into a normal state. If the steering transaction is determined to be inconsistent, the SM issues a *Rollback* message to all processes; any process presently in a tentative state will begin the recovery process and transition into a recovering state.

Because the SM has no knowledge of processes that have been indirectly affected by a steering change through the receipt of a tentative message, it must issue a *Rollback* message to *all processes* within the system. Once a process in a tentative state receives a *Rollback* message, it transitions into a recovering state and restores its previous state from the checkpoint. The system resumes execution at the point the checkpoint was taken. Once a process begins re-executing, on each *Receive* it first attempts to replay a logged message from the queue associated with the sender. If no logged message exists, the system then resorts to performing a normal *Receive*.

An interesting scenario exists in which a process may send a tentative message that the receiving process does not process until after the SM has issued either an OK or Rollback message. To address this, each process maintains a mapping indicating which steering transactions it knows of, and whether they have been determined to be consistent or inconsistent. If a process receives a tentative message associated with a steering transaction that was deemed consistent (*confirmed tentative message*), it processes the message as normal and will not take a checkpoint. On the other hand, if a process receives a tentative message associated with a steering transaction determined to be

inconsistent (*obsolete tentative message*), it knows that message has been or will be resent and thus throws away the present message and performs another *Receive*.

Each recovering process will re-execute forward to the transaction time that was determined to be a consistent cut by the SM; it will then reapply the steering changes. Each process transitions back into a normal state of execution once the steering changes have been reapplied and all of its message queues have been emptied.

4.2.4.4 Example

In Fig. 4.4, we show a computation that consists of four application processes. In this example, m_1 is a normal message, m_2 is an in-transit message, m_3 is a tentative message, and m_4 is either a confirmed tentative message or obsolete tentative message depending on the consistency of steering transaction.

In this example, process P_2 takes a checkpoint $C_{2,0}$ before it applies steering changes. Similarly, process P_3 takes a checkpoint $C_{3,0}$ before it applies steering changes. Upon receiving the in-transit message m_2, process P_2 logs the message into the log queue. Upon receiving the tentative message m_3, process P_1 takes a checkpoint $C_{1,0}$.

If the steering transaction is deemed consist, m_4 will be a confirmed tentative message and be processed as a normal message. The computation will just continue. If the steering transaction is deemed inconsistent, m_4 will be an obsolete tentative message and be discarded. The computation must roll back to its previous state before the steering changes. The checkpoints $C_{1,0}$, $C_{2,0}$, and $C_{3,0}$, and the current state of process P_0 together with the log queues for message m_2 comprise a consistent global system state. Process P_1, P_2, and P_3 will roll back to their checkpoints and resume execution. Re-execution is

almost same as the normal execution except that it must check the log queue first for a possible message replay before it resorts to a normal receive.

4.3 Comparison of Conservative Approach and Optimistic Approach

In this section, we discuss the perturbation and steering lag of the conservative and optimistic steering algorithms. For the steering lag, we consider three kinds of lag, the IM local lag, the IM global lag, and the SM lag. The IM local lag refers to the elapsed time between the first EOT event of a process after receiving the steering request and the successful application of the steering change at the process. The IM global lag is the elapsed time between the first EOT event of all processes after receiving the steering requests and the last successful application of the steering changes among all steered processes. The SM lag refers to the elapsed time between the time the SM sends out the steering requests and the last successful application of the steering changes among all steered processes.

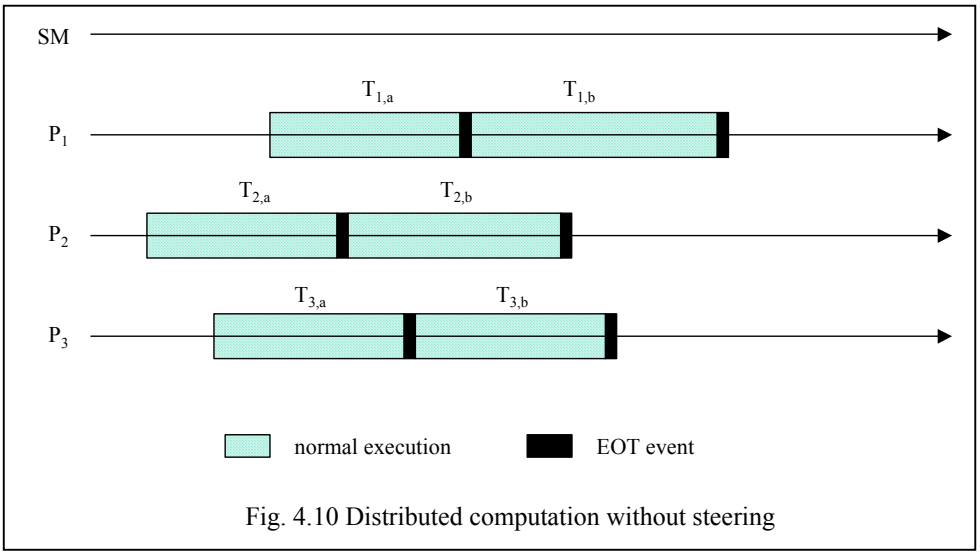


Fig. 4.10 Distributed computation without steering

To facilitate illustration of the perturbation and steering lag of the conservative and optimistic steering algorithms, we first consider a simple example. Fig.4.10 shows a distributed computation with 3 three processes involved. Local transactions $T_{1,a}$, $T_{2,a}$, and $T_{3,a}$ belong to the same global transaction T_a. Similarly, local transactions $T_{1,b}$, $T_{2,b}$, and $T_{3,b}$ belong to the same global transaction T_b. A steering transaction that involves all three processes will be consistent if the steering actions are applied all before T_a, or all after T_a and before T_b, or all after T_b. For this example, we assume that the steering command is broadcast to all involved processes and is received at roughly at the same time by each process.

4.3.1 The next steerable points are consistent

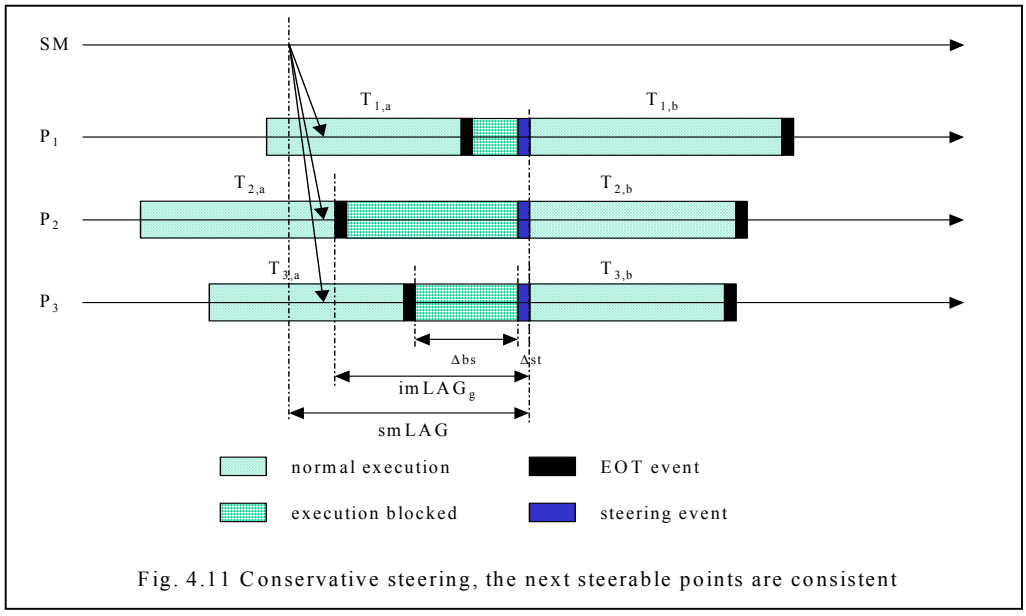


Fig. 4.11 Conservative steering, the next steerable points are consistent

When the next steerable points across steered processes are consistent, both the optimistic and conservative approaches will perform well. As seen in Fig. 4.11 and Fig. 4.12, all processes in the example execution receive the steering command during the transaction T_a. Thus, the next steerable points, which are points immediately after transaction T_a, are consistent.

In the case of conservative steering, depicted in Fig. 4.11, each process blocks its execution after reaching the end of transaction and until it receives a steering-change command. Upon receiving the steering-change command, it will apply the steering change and execute forward. As seen in Fig. 4.11, processes P_2 and P_3 reach the steerable points earlier than process P_1. P_2 and P_3 are blocked until P_1 reaches the steerable point, sends its ready message to the SM, and the SM receives all three ready messages and issues a steering-change command to all steered processes. Upon receiving this steering-change command, all three processes apply their steering changes and continue their normal execution. In this case, the perturbation induced by steering for each process includes the elapsed time when it is blocked for synchronization and the time to perform the steering change, denoted as Δ_{bs} and Δ_{st} respectively. Therefore, the local perturbation of each process is $\Delta_{bs} + \Delta_{st}$.. In this case, the IM local lag is the same as the local perturbation. The IM global lag is almost the same as the maximal IM local lag because all three processes apply their steering changes at almost the same time.

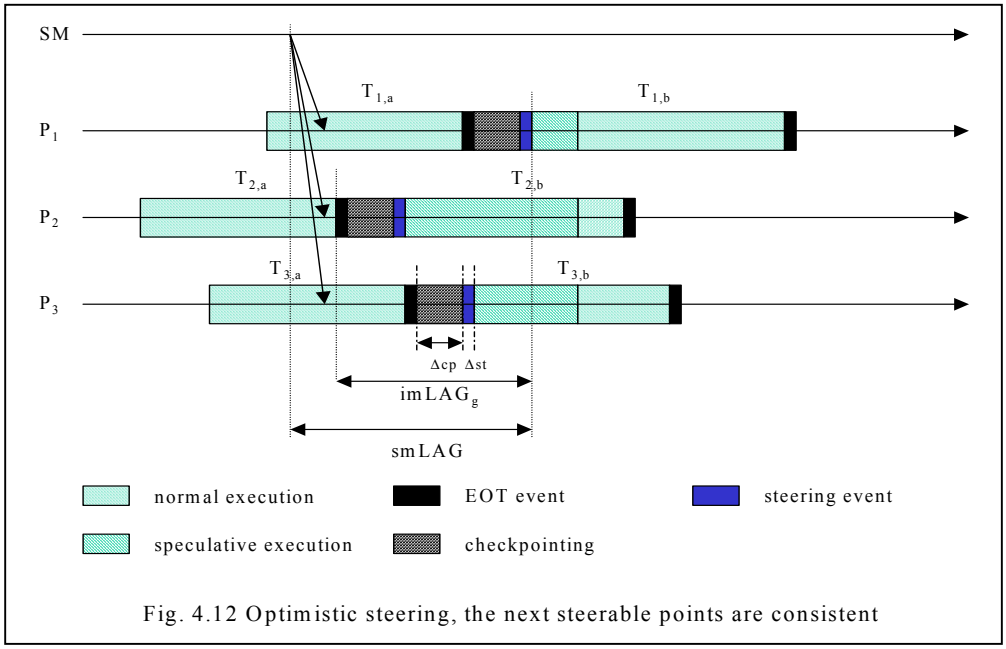


Fig. 4.12 Optimistic steering, the next steerable points are consistent

In optimistic steering, depicted in Fig. 4.12, a steered process takes a checkpoint, applies the steering change, and starts the speculative execution after it reaches the steerable point. Later, when all steering events have been received by the SM, the SM then can determine that the steering transaction is consistent and broadcast an OK message to all involved processes. Upon receiving the OK message, a process then clears its checkpoint and message logs, and accepts the speculative execution and continues its normal execution. In this case, the perturbation induced by steering for each process includes the time to take the local checkpoint (denoted as Δ_{cp}) and the time to perform the steering change. Therefore, the perturbation is $\Delta_{cp} + \Delta_{st..}$ Similarly, the IM local lag is the same as the local perturbation, and the IM local lags between different processes are almost the same. However, the IM global lag is much larger than the maximal IM local lag because processes are not well synchronized.

In summary, in the case of consistent steering, for the conservative steering approach, the overhead is mainly dependent on how well the computation is synchronized. If the computation is well synchronized, the average blocking time for each process Δ_{bs} will be very low. For the optimistic steering approach, the overhead is mainly dependent on the time for checkpointing. For both approaches, the SM lag is bigger than the IM global lag. The average difference between the SM lag and the IM global lag is dependent on the transaction size. The larger the transaction size, the larger the difference.

4.3.2 The next steerable points are inconsistent

As seen in Fig. 4.13 and Fig. 4.14, process P_1 and P_3 in this example receive the steering commands during transaction T_a, while process P2 receives the steering command after it has begun transaction T_b. Thus, the next steerable points are inconsistent.

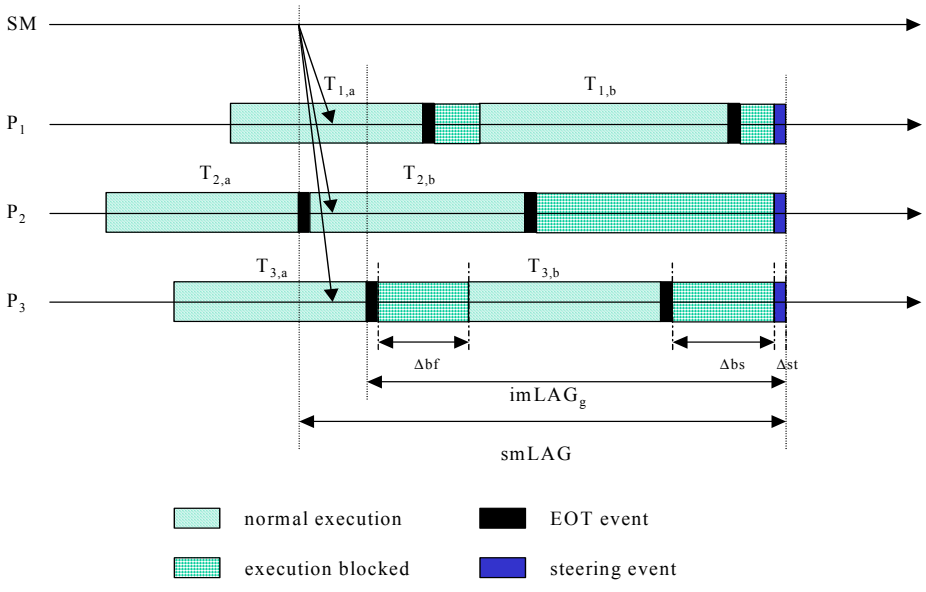


Fig. 4.13 Conservative steering, the next steerable points are inconsistent

In conservative steering, see Fig. 4.13, processes P_1 and P_3 block their executions after they reach the end of transaction of T_a. When process P_2 executes forward, it must send messages to process P_1 or P_3, or receive messages from P_1 or P_3 because the current transaction involves both P_1 and P_3. Then, processes P_1 and P_3 will become active and execute forward. Later, when they reach the end of transaction T_b, they will again block to wait for the steering-change command. In this case, the perturbation induced by steering for each process includes the elapsed time when it is first blocked (denoted as

Δ_{bf}), the elapsed time when it is blocked for synchronization, and the time for performing the steering change. Therefore, the total perturbation is $\Delta_{bf} + \Delta_{bs} + \Delta_{st}$. For process P_2, the local lag is the same as the local perturbation. However, for Process P_1 and P_3, the local lags are the local perturbations plus the local transaction size.

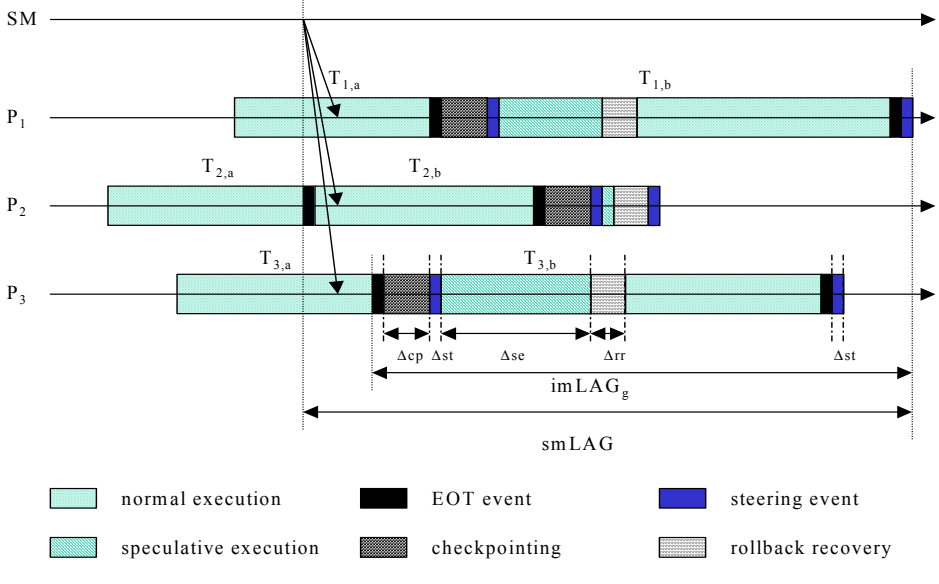


Fig. 4.14 Optimistic steering, the next steerable points are inconsistent

In optimistic steering, see Fig. 4.14, a steered process takes a checkpoint, applies the steering change, and starts the speculative execution after it reaches the steerable point. Later, when all steering events have been received by the SM, the SM then can determine that the steering transaction is inconsistent and broadcast a **rollback** message to all involved processes. Upon receiving the rollback message, a process will roll back to its previous state, thus canceling the steering change and its speculative execution. Then, the process executes forward and applies the steering change at the consistent point. In this case, the perturbation induced by steering for each process includes the time for

taking the local checkpoint (denoted as Δ_{cp}), twice the time for performing the steering change, the elapsed time for speculative execution (denoted as Δ_{se}), and the time for rollback / recovery (denoted as Δ_{rr}). Therefore, the total perturbation is $\Delta_{cp} + 2 * \Delta_{st} + \Delta_{se} + \Delta_{rr}$. Similarly, for process P_2, the local lag is the same as the local perturbation. However, for Process P_1 and P_3, the local lags are the local perturbations plus the local transaction size

Note that when rollback occurs in optimistic steering, the conservative steering approach would also have required blocking for a time almost equal to that spent on wasted computation.

4.4 Experiments

This section presents experiments performed to evaluate performance of the two consistent steering approaches in term of perturbation and lag. We ran a set of tests encompassing five variables that may affect our system overhead: communication pattern, transaction size, number of processes, number of steered processes, steering status, and size of checkpoints. Three basic types of distributed computation communication patterns were tested: one in which all processes communicate during each transaction and synchronize at the end of each transaction (Global); one in which processes communicate pairwise during each transaction and once in every 10 transactions all processes communicate and synchronize (Mixed); finally, one in which processes perform only pairwise communication and no global communication or synchronization takes place (Pairwise). For each communication type, we considered short transactions in which each process had a .25 second computation and large transactions in which each process had a 2.5 second computation. For the optimistic

steering approach, we also varied the size of checkpoints, 10KB (Small), 100KB(Medium), and 1MB (Large).

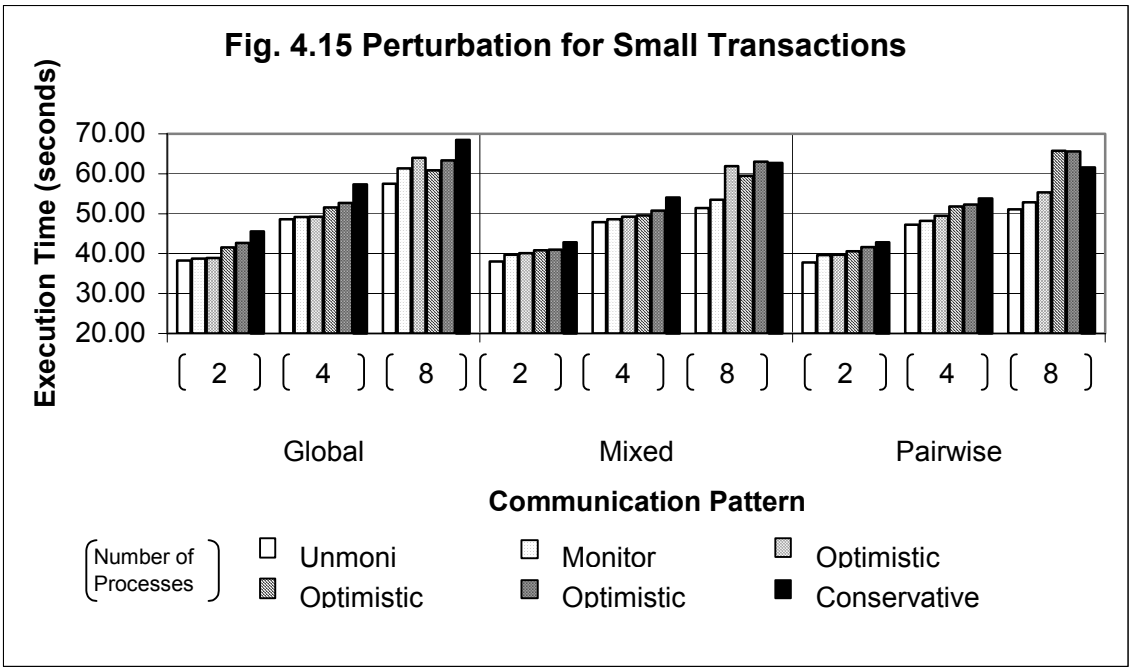
All tests were run on a cluster of 4 Pentium II 450 Mhz workstations with 128 MB of RAM and running RedHat Linux 7.2. Within Pathfinder, there exist two extra control processes, one for the SM and one for a daemon process responsible for establishing all UI and SM communication channels. Only the application and control processes were executed on the workstations during the tests, and the GUI was run on a separate server, to avoid workstation load imbalance.

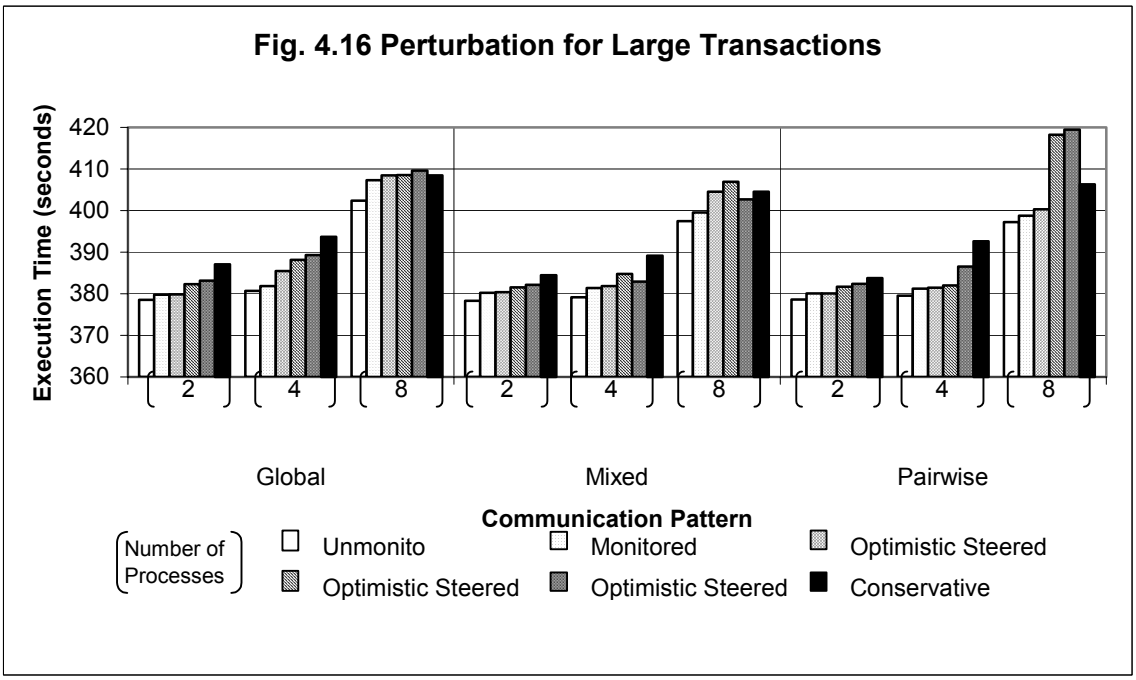
4.4.1 Perturbation

For perturbation tests, each test was run on 2, 4, and 8 processes, taking the average execution time of 5 runs of 150 transactions for the unmonitored computation, the monitored computation, the optimistic steered computation with the small checkpoint, the optimistic steered computation with the medium checkpoint, the optimistic steered computation with the large checkpoint, and the conservative steered computation. For each of the runs that included steering, 5 steering transactions took place. Except for the tests containing only 2 processes, during each run 2 steering transactions contained all the processes, 2 steering transactions contained 75% of the processes and one steering transaction contained 50% of the processes. For the 2 processes runs, each steering transaction affected all processes. This was done because steering only 1 process would guarantee consistency, and thus would introduce very little overhead and provide less meaningful performance results.

The average execution times for unmonitored, monitored, optimistic steered (10KB), optimistic steered (100KB), optimistic steered (1MB), and conservative steered test sets

with small transactions and large transactions are presented in Fig. 4.15 and Fig. 4.16, respectively.





On average for small transactions, the optimistic approach with small checkpoint size (10KB) introduced a 3.39% overhead, the optimistic approach with medium checkpoint

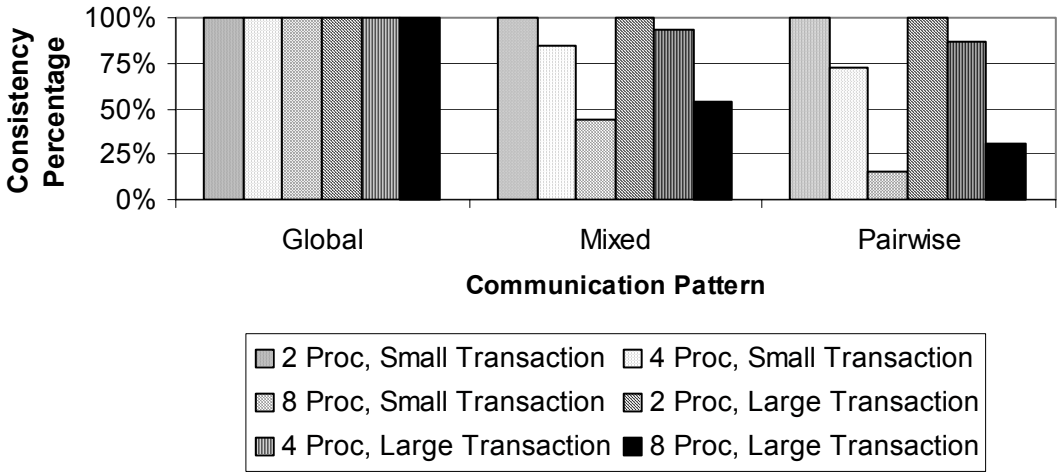
size (100KB) introduced a 6.83% overhead, and the optimistic approach with large checkpoint size (1MB) introduced a 9.27% overhead, and the conservative approach introduces a overhead 13.11%. On average for large transactions, the optimistic approach with small checkpoint size (10KB) introduced a 0.41% overhead, the optimistic approach with medium checkpoint size (100KB) introduced a 0.87% overhead, and the optimistic approach with large checkpoint size (1MB) introduced a 1.14% overhead, and the conservative approach introduces a overhead 1.77%.

For the optimistic approach, a clear trend is that the overhead increases with the size of checkpoint. However, it is still in the reasonable range. In the case of 2 and 4 processes, the overhead introduced by optimistic steering of any checkpoint size is less than the overhead introduced by conservative steering approaches. However, for 8 processes with mixed and pairwise communication pattern, the conservative steering results in a smaller overhead. The main reason is that the percentages of consistency in these cases are very low, less than 50 percent. On average, the optimistic steering approach with any checkpoint size is better than the conservative approach.

Fig. 4.17 presents the average percentage of steering transactions applied consistently on the first attempt, based on the number of application processes, size of transaction, and communication pattern. Seen in figure 4.17, steering consistency varies with the average length of time a transaction spends performing the computation, the number of processes, and the communication pattern between the processes. As we expected, the tests with only two processes were highly consistent. As the number of processes increases so does the likelihood of inconsistency. Similarly, mixed and pairwise communication patterns also increase the likelihood of steering inconsistencies, with an average consistency of

slightly above 80% and 70% respectively while the global communication patterns were 100% consistent. [MGK01] points out that steering inconsistencies can be easily visualized as a program transaction boundary being cut by a steering transaction; some steering events occurred before or during the program transaction while others took place after. With mixed and pairwise communication patterns, where faster processes or process sets are allowed to progress uninterrupted and then forced to synchronize with slower processes, this scenario becomes more likely, as can be seen in the mixed communication tests for 8 processes. Note that the high percentages of inconsistency for 8 processes are also due to the imbalance of workstation load because we had to run either 2 or 3 processes on each machine in this case.

Fig. 4.17 Percentages of Steering Transactions Consistent on First Attempt



Overall, the system provides reasonable performance for monitoring and steering, both introducing an average overhead of less than 2%. While these test cases are not all encompassing, they represent some basic communication patterns. The transaction sizes are meant to serve as benchmarks for performance testing. While the .25 second

transactions size is smaller than we expect in practice, the 2.5 second transaction represents a reasonable, though still conservative, value. Performing 5 steering transactions per run, thus steering approximately every minute in the case of the large transactions, is far more frequent than will occur in practical use. The user must have the ability to comprehend what is occurring in a computation to make informed and meaningful steering decisions. Steering is more likely to occur on the scale of once every 1000 transactions, to permit observation of meaningful patterns within the computation.

The study of global perturbation in terms of execution time alone is not sufficient to gain full insight into the characteristics of the optimistic and conservative approaches. Next, we will discuss the average steering lag for each steering transaction and define the tradeoff between the optimistic approach and the conservative approach.

4.4.2 Steering Lags

Lag proved to be both more difficult and more interesting to measure. We consider three kinds of lag: the IM local lag, the IM global lag, and the SM lag. Since the IM local lags of different processes may be quite different, we consider maximal IM local lag, average IM local lag, and minimal IM local lag. As described earlier, the main difference between the IM lag and SM lag is the elapsed time for the steering request message to reach the IM from the SM and the elapsed time between the arrival of steering request message and the occurrence of the first EOT event. This difference is the same for both the optimistic approach and the conservative approach. Since the main purpose of this study is to find the components of lag affected by the choice of steering algorithms, we will mainly focus on the comparison of IM lags. However, we will also discuss the relationship between IM local lag, IM global lag, and SM lag. As discussed in section

4.3, the steering lags vary greatly depending on whether or not the next steerable points are consistent. To gain insight into steering lag, we separate the cases when steering transactions are consistent and inconsistent.

For the steering lags tests, each test was run on 2, 4, and 8 processes. For each run with 150 program transactions, 15 steering transactions took place. For 2 processes, each steering transaction contains 2 processes. For 4 processes, during each run 5 steering transactions contained all the processes, and 10 steering transactions contained 2 processes. For 8 processes, during each run 5 steering transactions contains all the processes, 5 steering transactions contains 4 processes, and 5 steering transaction contains 2 processes.

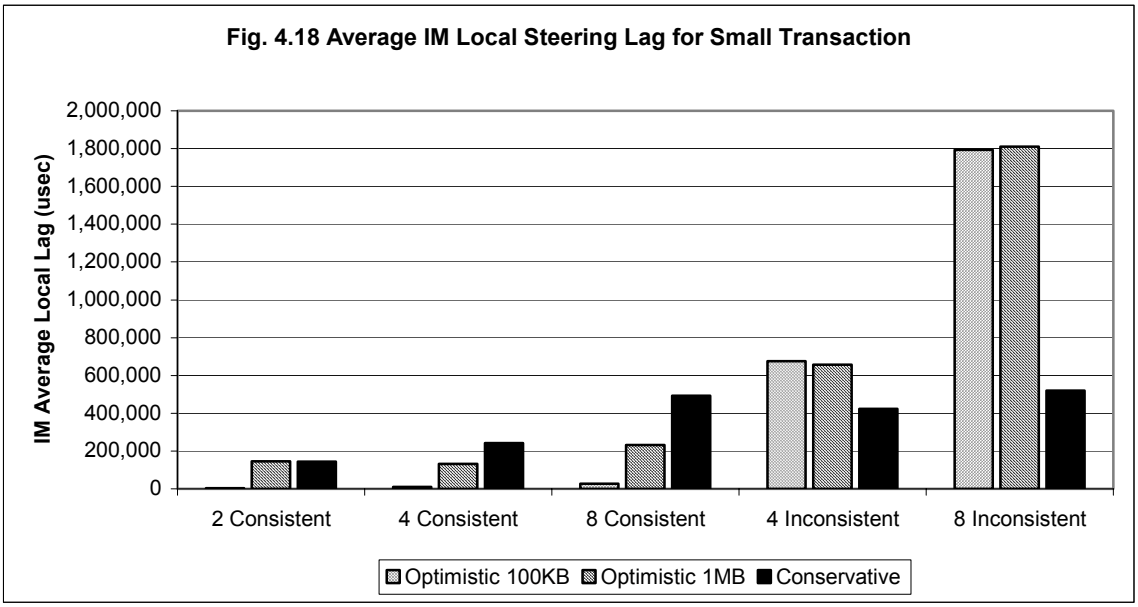
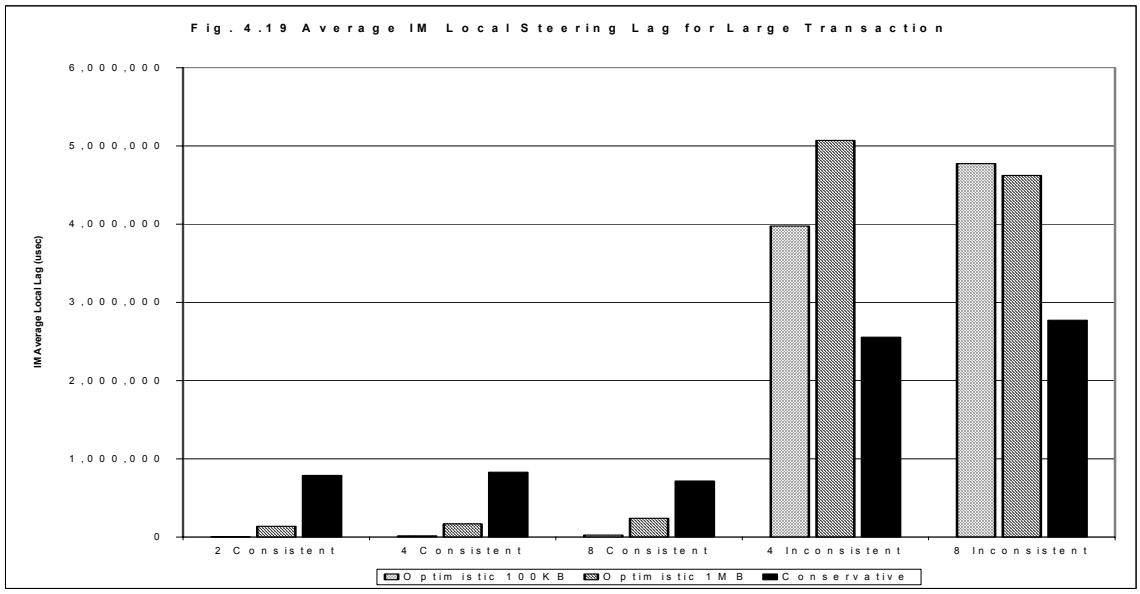
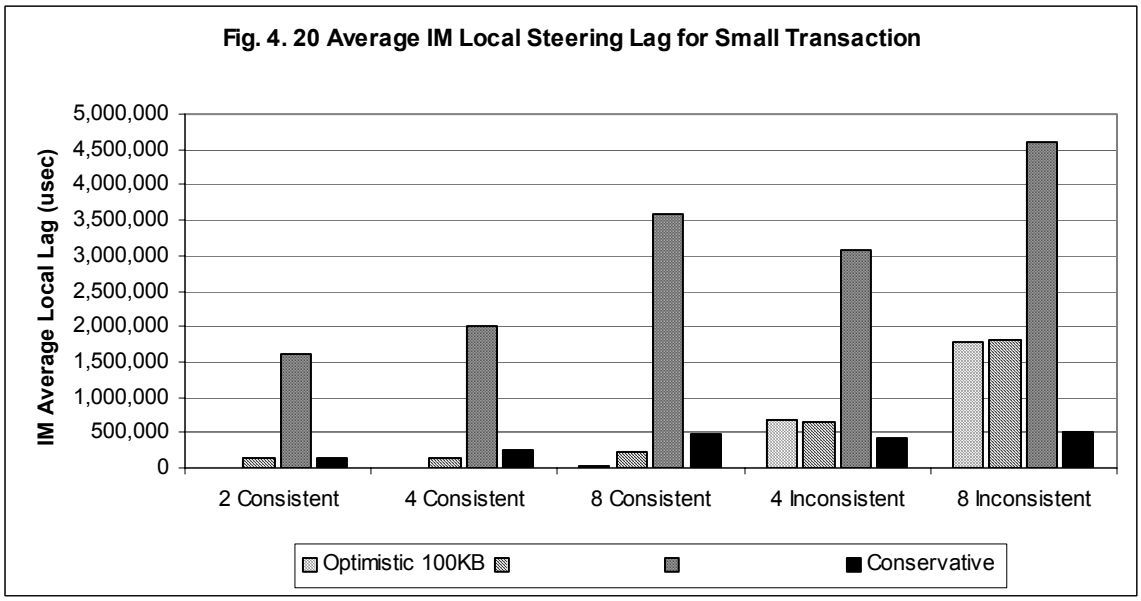


Fig. 4.18 and Fig. 4.19 show the average local lags of the optimistic approach with medium checkpoint size (100KB), the optimistic approach with large checkpoint size (1MB), and the conservative approach for small transactions and large transactions, respectively. For the optimistic approach, the steering lag grows with the checkpoint size. When steering transactions are consistent on the first attempt, the steering lag of the

optimistic approach with both medium and large checkpoint size is smaller than for the conservative approach. When steering transactions are inconsistent on the first attempt, the steering lag of both approaches increases dramatically and the steering lag of the optimistic approach is much larger than the steering lag of the conservative approach.





When steering transactions are consistent on the first attempt, the IM steering lag depends largely on the overhead of checkpointing. When the size of the checkpoint is too large (i.e. equal to 10MB), the conservative approach will always win, even when the first attempt is consistent, as seen in Fig. 4.20.

Fig. 4.21 IM Lags and SM Lags for Small Transaction, 8 Processes

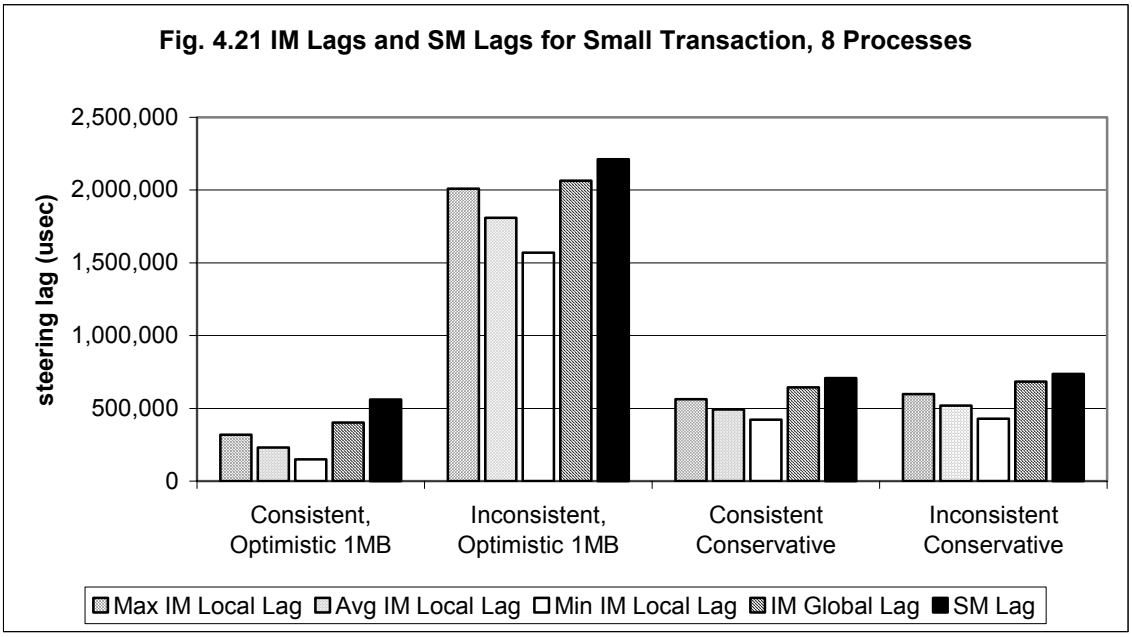


Fig. 4.22 IM Lags and SM Lags for Large Transaction, 8 Processes

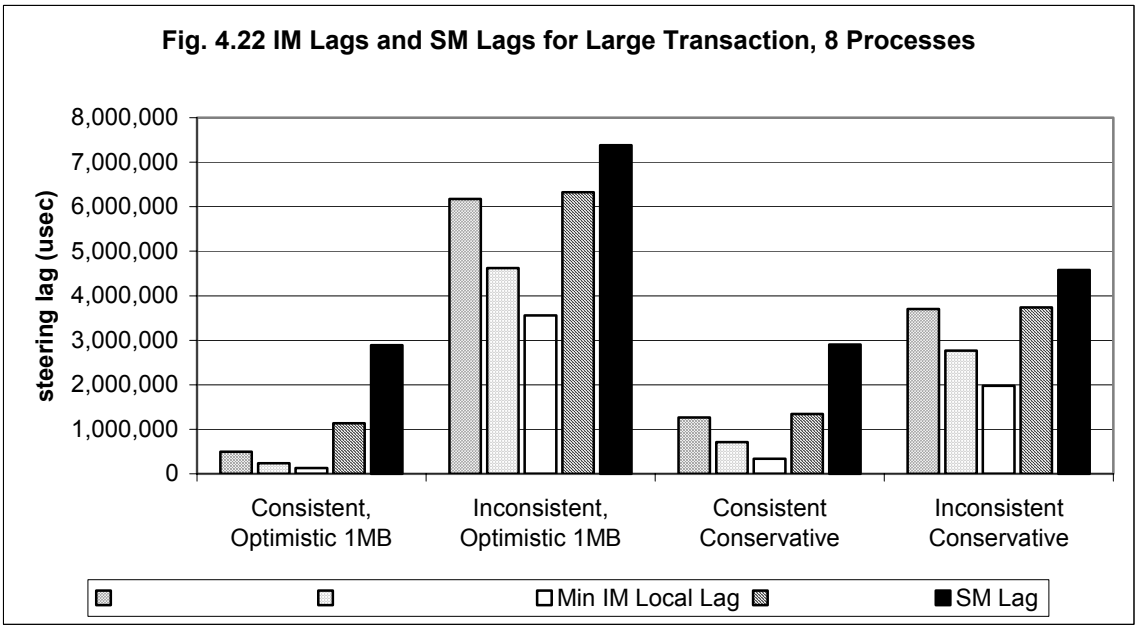


Fig. 4.21 and Fig. 4.22 show the relationship among IM local lag, IM global lag and SM lag for small transactions and large transactions, respectively. For the optimistic approach, the difference between IM global lag and IM local lag increases dramatically when the transaction size is increased. However, for the conservative approach, the IM global lag and the maximal IM local lag is almost the same for both small transactions and large transactions because all steered processes are synchronized at the steerable points and apply their steering changes at roughly the same time. For both the optimistic and conservative approaches, the difference between IM global lag and SM lag increases with the transaction size. This difference is roughly about half of the transaction size.

Fig. 4.23 Lags in Different Communication Patterns and Large Transactions

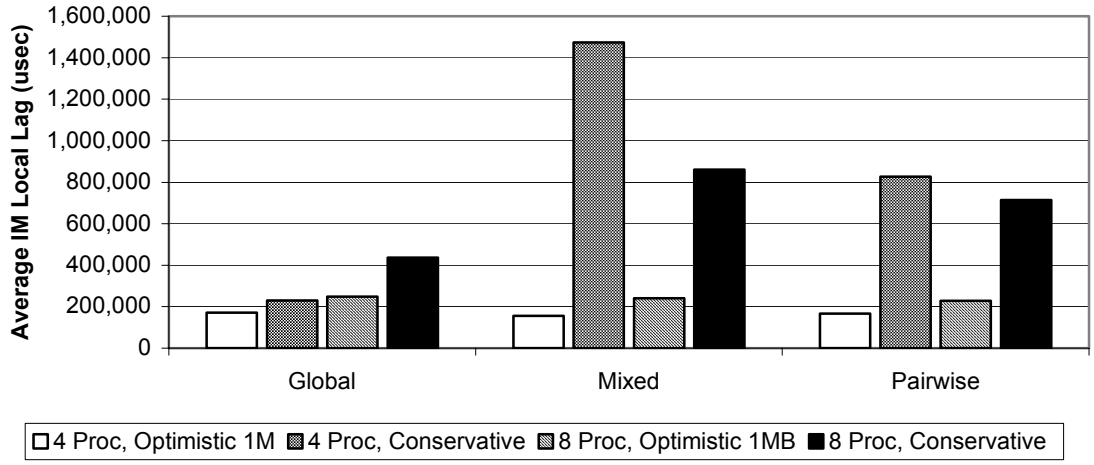


Fig. 4.23 shows another interesting trend. For optimistic steering, the steering lag is roughly the same among three communication patterns. However, for conservative steering, the steering lag of the mixed communication and the pairwise communication patterns are more than double that of the global communication pattern. This is because the processes are not well synchronized with the mixed and pairwise communication

patterns and the steering lag for the conservative approach depends largely on the synchronization overhead.

In summary, when steering transactions are consistent on the first attempt, the steering lag of the optimistic approach is mainly due to the checkpointing, and grows with the size of checkpoint; the steering lag of the conservative approach is mainly due to processes synchronization, fast processes have to wait for slow processes before steering changes are applied. When steering transactions are inconsistent on the first attempt, the steering lags of both approaches increase dramatically and the steering lag of optimistic approach is much larger than the steering lag of the conservative approach. Tradeoffs between the optimistic approach and the conservative approach depend on the percentages of consistency on the first attempt, the size of checkpoint, and I/O performance. When the percentages of consistency on the first attempt are large enough (i.e. more than 80 percent) and the size of checkpoint is not too large (i.e. less than or equal to 1MB), the optimistic approach will win; otherwise, the conservative approach will win. However, we believe that the overhead of checkpointing can be dramatically reduced if our tests can be done in a system with high performance I/O.

CHAPTER 5

CONCLUSIONS

In this chapter, we summarize the main contributions of this research and outline the future directions.

5.1 Summary

Consistent interactive steering of distributed computations is a very interesting but challenging problem. Most computational steering systems do not address this problem at all. Those that do address this problem do so in a restricted manner. For example, CUMULVS requires a computation to have a single main loop, and VASE depends on the shared global memory architecture. This work presented here is novel in that it solves the consistency problem in a more general way.

We have developed two approaches for consistent steering of distributed computations: the optimistic approach and the conservative approach. Algorithms for both the conservative steering approach and the optimistic steering approach have been designed, implemented and integrated into the Pathfinder system. The performance of conservative and optimistic steering approaches have been evaluated in term of perturbation and lag. Tradeoffs between the optimistic approach and the conservative approach depend on the percentages of consistency on the first attempt, the size of checkpoint, and I/O performance. Our experiments show that when the percentages of consistency on the first attempt are large enough (i.e. more than 80 percent) and the size of checkpoint is not too large (i.e. less than or equal to 1MB), the optimistic approach will win; otherwise, the conservative approach will win. However, we believe that the

boundary of checkpoint size can be dramatically increased if our tests can be done in a system with the high performance I/O.

Furthermore, we introduce a novel transaction-based computation model. In our monitoring and steering system, the overall computation is abstracted to an interleaving of atomic state changes involving one or more processes – by analogy with databases, we call such state transitions transactions. This abstraction not only gives users a simple and high-level view of distributed computation, but also simplifies reasoning consistency problem by reducing the amount of information to be handled.

5.2 Future Work

Future efforts of this research will include the following three parts: nested transactions, collaborative steering by multiple users, and real world applications.

One problem with transactions is that the granularity of transaction has a big effect on the performance of steering algorithms; large transaction size may result large steering lag while small transaction size may result large monitoring perturbation as the state information is collected at the end of transaction. We believe that a nested transaction-based computational model will solve this problem. By allowing users to zoom in and out to different levels of nesting, we can achieve a good balance between perturbation and lag.

Collaborative steering of a computation by multiple users is another very interesting and challenging problem. We expect that when the user requests do not conflict, i.e. they steer different processes, the steering request by different users should permit concurrent application. Otherwise, the conflicting steering changes must be applied in a certain order.

Interactive steering of computations is a useful approach for users working with long-running, complex computations with numerous parameters affecting both solutions quality and execution performance, such as grand challenge problems in structural biology, computational chemistry, computational physics, and large-scale simulations. In the future, we will seek to apply the algorithms, techniques, and systems that we have developed in interactive steering to such kind of problems. The application of such interactive steering techniques to those grand challenge problems will be both challenging and rewarding.

REFERENCES

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, Reading, MA, 1987.

[BG80] P. Bernstein and N. Goodman. "Timestamp-based algorithms for concurrency control in distributed database systems," *In VLDB*, Montreal, Canada, October 1980.

[BSS87] J. Bona, P. Souganidis, and W. Strauss, "Stability and instability of solitary waves of Korteweg-de Vries type," *Proc. R. Soc. Lond*. A 411, pp. 395-412, 1987.

[CM86] K. Chandy and J. Misra, "How processes learn," *Distributed Computing*, 1, pp 40-52, 1986.

[CL85] K. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, 3(1): 272-314, Feb. 1985.

[CL94] R. Cypher and E. Leu, "Semantics of blocking and nonblocking send and receive primitives," Proc. IEEE Int. Conf. Parallel Processing, pp 729-735, August 1994.

[CS89a] K.M. Chandy and R. Sherman, "The conditional event approach to distributed simulation," *in Proceedings of the SCS Multiconference on Distributed Simulation*, 93-99, Mar 1989.

[CS89b] K.M. Chandy and R. Sherman, "Space, time, and simulation," *in Proceedings of the SCS Multiconference on Distributed Simulation*, 53-57, Mar 1989.

[DS80] Dijkstra and B.P. Scholten, "Termination Detections for Diffusing Computations," *Information Processing Letters*, 1980, 11(1): 1-4.

[EGLT76] K. Eswaran, J. Gray, R. Lorie, and I. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, 19(11):624-633, November 1976.

[FGL82] M. Fischer, N. Griffeth, and N. Lynch, "Global states in a distributed system," IEEE Trans. Software Engineering, 8(3), pp 198-202, 1982.

[Fid88] C. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," *Australian Computer Science Communications*, 10(1): 56-66, Feb. 1988.

[Fuj89] R. Fujimoto, "Time Warp on a Shared Memory Multiprocessor," *Transactions of the Society for Computer Simulation* 6(3): 211-239, July 1989.

[Fuj90] R. M. Fujimoto, "Parallel discrete event simulation," *Communication of the ACM*, 33(10): 30-53, October 1990.

[FW78] B. Fornberg, and G. Whitham, "A numerical and theoretical study of certain nonlinear wave phenomena," *Phil. Trans. R. Soc. Lond*. A 289, pp. 373-404, 1978.

[FZ90] J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints," *in Proceeding, 10th Int. Conference on Distributed Computing Systems,* Paris, France, 134-141, May 1990.

[GFS93] K. Ghosh, R. Fujimoto and K. Schwan, "A testbed for optimistic execution of real-time simulations," *in Proceedings of the IEEE Workshop on Parallel and Distributed Real-Time Systems*, Apr 1993.

[GKM02] J. Guo, E. Kraemer and D.W. Miller, "Consistency Detection in a Transaction-Based Interactive Steering System", *The 21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, in submission.

[GKP96] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "CUMULVS: Providing fault tolerance, visualization, and steering of parallel applications," *SIAM*, Aug. 1996.

[GR92] J. Gray and A. Reuter, "Transaction Processing: Techniques and Concepts," Morgan Kaufmann, San Mateo, CA, 1992.

[GT02] Jinhua Guo and Thiab Taha, "Parallel Implementation of the Split-step Fourier Method and the Pseudospectral Method For Solving Higher KdV Equation," In submission, 2002.

[Har00] D. Hart, "Supporting Exploratory Visualization of Distributed Computations," *Ph.D. dissertation*, Department of Computer Science, Washington University, Aug. 2000.

[HBJJ92] R. Haber, B. Bliss, D. Jablonowski, and C. Jog, "A Distributed Environment for Running Time Visualization and Application Steering in Computational Mechanics." *Computing Systems in Engineering*, 3(1-4):501-515, 1992.

[HKR97] D. Hart, E. Kraemer, and G.-C. Roman, "Interactive Visual Exploration of Distributed Computations," *in Proceedings, 11th International Parallel Processing Symposium,* Geneva, Switzerland, 121-127, Apr. 1997.

[JBBH93] D.J. Jablonowski, J.D. Bruner, B. Bliss and R.B. Haber, "VASE: The Visualization and Application Steering Environment," *in Proceeding, Super Computing*, 560-69, 1993.

[Jef85] D. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, 7(3): 404- 425, July 1985.

[JM86] D. R. Jefferson, and A. Motro, "The Time Warp mechanism for database concurrency control," *in Proceedings of the IEEE 2nd International Conference on Data Engineering*, 141-150, Feb 1986.

[KF93] S. Damodaran-Kamal and J. Francioni, "Nondeterminacy: testing and debugging in message-passing parallel program," *ACM SIGPLAN Notices*, 28(12), pp 118-128, 1993.

[KDR98] E. Kraemer, D. Hart, and G.-C. Roman, "Balancing Consistency and Lag in Transaction-Based Computational Steering," *in Proceedings, Thirty-First Annual Hawaii International Conference on System Sciences*, 137-147, Jan. 1998.

[KR81] H. Kung and J. Robinson, "Optimistic methods for concurrency control," *ACM Transactions on Database Systems*, Vol. 6, No 2, pp. 213-26, 1981.

[Ksh98] A. Kshemkalyani, "A framework for viewing atomic events in distributed computations," *Theoretical Computer Science,* 196, pp 45-70, 1998.

[Lam78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, 558-565, 1978.

[Lam86] L. Lamport, "On interprocess communication. Part 1. basic formalism, Part II: algorithms," Distributed Comput, 1, pp 77-101, 1986.

[LMW97] R. Liere, J. Mulder, and J. Wijk, "Computational Steering." *Future Generation Computer Systems,* 12(5):441-450, April 1997.

[Lyn96] N. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[Mat86] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing*, 2(3), pp 161-172, 1987.

[Mat89] F. Mattern, "Virtual Time and Global States of Distributed Systems," *Parallel and Distributed Algorithms*, North-Holland, 215-226, 1989.

[MGK01] D.W. Miller, J. Guo, E. Kraemer and Y. Xiong, "On-the-Fly Calculation and Verification of Consistent Steering Transactions," *Supercomputing Conference* (SC2001), Denver, CO.

[MI92] Y. Manabe and M. Imase, "Global Conditions in Debugging Distributed Programs", *J. Parallel and Distributed Computing*, Vol. 15, pp 62-69, May 1992.

[MRR90] B. Merrifield, S. Richardson, and J. Roberts, "Quantitative studies of discrete event simulation modeling of road traffic," *Proceedings of the SCS MultiConference on Distributed Simulation*, 22(1):188-193, 1990.

[MWL99] J. Mulder, J. van Wijk, and R. van Liere, "A Survey of Computational Steering Environments," *Future Generation Computer Systems*, 15(2):91-102, 1999.

[NJ95] R. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Transactions on Parallel and Distributed Systems*, 6(2): 165-169, Feb. 1995.

[NM92] R. Netzer and B. Miller, "Optimal tracing and replay for debugging message-passing parallel programs," *In Proceedings of Supercomputing' 92*, pp. 502-511, November 1992.

[Pla97] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," *Technical Report UT-CS-97372*, Department of Computer Science, University of Tennessee, Knoxville, TN, July 1997.

[PJ95] S. Parker and C. Johnson, "SCIRun: A Scientific Programming Environment for Computational Steering." *In Proceedings of SuperComputing '95*, 1995.

[PWJ97] S. Parker, D. Weinstein, and C. Johnson, "The SCIRun Computational Steering Software System." In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools for Scientific Computing*, pages 1-40. Birkauser Verlag AG, Switzerland, 1997.

[Ran75] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[RL97] S. Rathmayer and M. Lenke, "A Tool for On-line Visualization and Interactive Steering of Parallel HPC Applications." *In Proceedings of the 11th International Parallel Processing Symposium, IPPS 97*, pages 181-186, 1997.

[Sin93] M. Singhal, "A taxonomy of distributed mutual exclusion," J. Parallel Distributed Computing, 18(1), pp 94-101, 1993.

[SM94] R. Schwarz, F, Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing*, 7(3): 149-174, 1994.

[SY85] R. Strom and S. Yemini, “Optimistic recovery in distributed systems,” ACM Transactions on Computing Systems, 3(3), pp 204-226, 1985.

[Tah92] T. Taha, “A parallel-vector algorithm for an investigation of a self-focusing singularity of higher KdV equation,” *Fifth International Symposium on Domain Decomposition Methods for PDES*, *(D, Keyts stal, eds.) SIAM*, Philadelphia, PA, pp. 597-604, 1992.

[VKD00] H. Vuppula, E. Kraemer, and D. Hart, “Algorithms for Collection of Global Snapshots: An Empirical Evaluation,” *ISCA Conference on Parallel and Distributed Computing*, Las Vegas, NV, 197-204, Aug. 2000.

[VS95] J. Vetter and K. Schwan, “PROGRESS: A Toolkit for Interactive Program Steering,” *in Proc. 24th Ann. Conf. on Parallel Processing,* 1995.

[VS97] J. Vetter and K. Schwan, “High performance computational steering of physical simulations,” *in Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.