NEW METHODS FOR PARALLEL DISCRETE EVENT SIMULATION

By

Gilbert (Gang) Chen

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

____
Dr. Boleslaw K. Szymanski, Thesis Adviser

____
Dr. David R. Musser, Member

____
Dr. Linda F. Wilson, Member

____
Dr. Christopher D. Carothers, Member

____
Dr. Sibylle Schupp, Member

Rensselaer Polytechnic Institute
Troy, New York

May 2003

NEW METHODS FOR PARALLEL DISCRETE EVENT SIMULATION

By

Gilbert (Gang) Chen

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. David R. Musser, Member

Dr. Linda F. Wilson, Member

Dr. Christopher D. Carothers, Member

Dr. Sibylle Schupp, Member

Rensselaer Polytechnic Institute
Troy, New York

May 2003

© Copyright 2003
by
Gilbert (Gang) Chen
All Rights Reserved

CONTENTS

LIST OF FIGURES . . . vii

LIST OF TABLES . . . ix

1. INTRODUCTION: SIMULATION AND BEYOND . . . 1
   - 1.1 Background . . . 1
     - 1.1.1 The Process of Simulating . . . 2
     - 1.1.2 Discrete Event Simulation . . . 3
     - 1.1.3 Problem Statement . . . 4
   - 1.2 Component-Based Simulation . . . 6
     - 1.2.1 Component-Based Software Development . . . 6
     - 1.2.2 Component Model for Simulation . . . 7
   - 1.3 Parallel Discrete Event Simulation . . . 8
   - 1.4 Research Contributions . . . 10
2. COMPONENT-BASED SIMULATION: MAXIMIZING REUSABILITY . . 11
   - 2.1 Related Work . . . 11
     - 2.1.1 CORBA . . . 11
     - 2.1.2 HLA . . . 13
     - 2.1.3 Problems with CORBA and HLA . . . 14
   - 2.2 Traditional Simulation World-Views . . . 16
     - 2.2.1 Event Scheduling . . . 16
     - 2.2.2 Activity Scanning . . . 17
     - 2.2.3 Process Interaction . . . 18
     - 2.2.4 Logical Process . . . 18
   - 2.3 Component-Oriented World-View . . . 20
   - 2.4 Component-Port Model . . . 23
   - 2.5 Component Classification . . . 24
     - 2.5.1 Time-Independent Components . . . 25
     - 2.5.2 Time-Aware Components . . . 25
     - 2.5.3 Autonomous Components . . . 26
     - 2.5.4 Simulation Engine . . . 27
     - 2.5.5 Time Properties . . . 28

2.6 Port Classification 29
2.7 Comparison with CORBA and HLA 32
2.8 Conclusion 33

3. COST: A COMPONENT-ORIENTED DISCRETE EVENT SIMULATOR 34
3.1 Design and Implementation 35
3.1.1 Functor 36
3.1.2 Inport and Outport Classes 37
3.1.3 Simulated Time and Port Index 39
3.2 Timer 41
3.3 Simulation of an M/M/1 System 41
3.3.1 Data Type 42
3.3.2 Source 42
3.3.3 FCFS Server 44
3.3.4 Sink 47
3.3.5 Constructing the Simulation 48
3.3.6 Running the Simulation 49
3.4 Reusability in COST 50
3.5 Conclusion 50

4. COMPONENT-ORIENTED SIMULATION ARCHITECTURE: TOWARD INTEROPERABILITY AND INTERCHANGEABILITY 52
4.1 Interoperability and Interchangeability 52
4.2 Simulation Platform 53
4.3 Design of A CORSA Prototype 55
4.3.1 Two Levels of Reuse 55
4.3.2 Component Interface Description Language 56
4.4 Implementation Issues 56
4.4.1 Class Hierarchy of Components 56
4.4.2 Functor 57
4.5 PCS Simulation Using CORSA 59
4.5.1 Two Reusable Components 59
4.5.2 Experimental Results 61
4.6 Conclusion 62

5. LOOKBACK: LOOKING INTO THE PAST IN SEARCH FOR PARALLELISM . . . 63
5.1 Introduction . . . 63
5.2 Lookback-Based Protocol . . . 64
5.3 LB-GVT and LB-EIT . . . 69
5.3.1 LB-EIT Protocol with Deadlock Recovery . . . 72
5.4 A Retrospective View . . . 75
5.4.1 Out-of-Timestamp Order Execution . . . 75
5.4.2 Local Rollback . . . 77
5.5 Lookback and Lookahead . . . 79
5.6 Lookback and Super-Criticality . . . 83
5.7 Four Types of Lookback . . . 87
5.8 Finding Lookback . . . 90
5.9 Lookback-Based Optimization and Lazy Cancellation . . . 91
5.10 Conclusion . . . 93
6. CLOSED QUEUING NETWORK SIMULATION: FIRST APPLICATION OF LOOKBACK-BASED PROTOCOLS . . . 95
6.1 Closed Queuing Network . . . 95
6.2 Determining Lookback in CQN Simulation . . . 96
6.3 Lookback-Enabled FCFS Server . . . 98
6.3.1 Handling Arrivals . . . 98
6.3.2 Handling Departures . . . 100
6.4 Dealing with Bounded Queues . . . 102
6.4.1 Initial Position . . . 103
6.4.2 Lazy Evaluation . . . 106
6.5 Experimental Results . . . 107
6.6 Conclusion . . . 109
7. A HYBRID LOOKBACK-BASED PROTOCOL: MIXING LOOKBACK-ENABLED COMPONENTS WITH SEQUENTIAL COMPONENTS . . . 110
7.1 Boundary Components . . . 111
7.2 Guard Events . . . 112
7.3 Comparing Hybrid Lookback-Based Protocol with Conservative Protocol . . . 113

7.4 CQN Simulation 114
7.5 UDP Network Simulation 115
7.6 Conclusion 117

8. PCS SIMULATION: FOUR TYPES OF LOOKBACK 120
8.1 Exploiting Lookback in PCS Network Simulation 120
8.1.1 Direct Strong Lookback 120
8.1.2 Universal Strong Lookback 122
8.1.3 Direct Weak Lookback 123
8.1.4 Universal Weak Lookback 124
8.2 Performance Evaluation 126
8.3 Conclusion 129

9. CONCLUSIONS AND FUTURE DIRECTIONS 131
9.1 Opportunities for PDES 132
9.2 Future Directions 133

LIST OF FIGURES

1.1 Four Steps in a Simulation Process . . . 2

3.1 An M/M/1 System . . . 42

4.1 The CORSA Simulation Platform . . . 54

4.2 The Class Hierarchy for the CORSA Simulation Platform . . . 57

4.3 Functor Objects Used in the Implementation . . . 58

4.4 Efficiency of Different Linking Approaches . . . 61

5.1 Deadlock in the LB-EIT protocol . . . 71

5.2 Eager Delivery versus Lazy Delivery . . . 80

5.3 Supercritical Events in Lookback-Based Protocols . . . 85

5.4 Four Types of Lookback . . . 88

5.5 Relations of Four Types of Lookback and Lookahead . . . 89

6.1 A CQN with 3 Switches and 9 FCFS Servers . . . 95

6.2 A Counter Example of Hypothesis 1 . . . 102

6.3 A Straggler Can Only Increase the Initial Position of Another Packet Entering the Service . . . 104

6.4 Performance of Lookback-Based Protocols for the CQN simulation. . . . 108

6.5 Eager and Lazy Evaluation of Initial Positions . . . 109

7.1 Performance of COST and Three Parallel Protocols (4 CPUs) on CQN Simulation . . . 115

7.2 Speedup of Three Protocols Relative to COST on CQN Simulation (4 CPUs) . . . 116

7.3 Speedup of Three Protocols Relative to Their Sequential Execution on CQN Simulation (4 CPUs) . . . 117

7.4 A UDP Network with a=2 and b=3 . . . 118

7.5 Performance of Hybrid LB-GVT and LA-EIT on UDP Network Simulation 119

7.6 Speedup of Hybrid LB-GVT and LA-EIT on UDP Network Simulation (4 CPUs) 119

8.1 Data Structure for Exploiting Direct Weak Lookback in PCS Simulation 121

8.2 The Absolute Impact Time of a Departure Event in PCS Network Simulation 122

8.3 A Straggler May Affect the Number of Free Channels By TWO 126

8.4 The Dynamic Impact Time of a Departure Event in PCS Network Simulation 127

8.5 Performance of the LB-GVT protocol with Direct Strong Lookback and Universal Strong Lookback 128

8.6 Improve the LB-GVT Protocol by Exploiting Direct Weak Lookback . . 129

8.7 Optimistic PCS Simulation with Several Optimization Techniques (Average Residence Time = 450 seconds) 130

8.8 Optimistic PCS Simulation with Several Optimization Techniques (Average Residence Time = 50 seconds) 130

LIST OF TABLES

2.1 Properties of the Simulated Time 29

2.2 Typical LBTS values of Ports 31

5.1 All Five Possible Cases of Rollbacks and Anti-messages 89

ABSTRACT

This thesis makes two contributions in the area of Parallel Discrete Event Simulation. Firstly, a component-oriented simulation world-view is proposed in which a simulation is viewed as a composition of components. The proposed component-port model enforces such a world-view by classifying components into time-independent, time-aware, and autonomous components, and by classifying ports into serial, virtual, and lookback ports.

Secondly, a component property named *lookback* is identified that enables a new class of PDES synchronization protocols. Lookback has proven to be more commonly present that lookahead, the basis for traditional conservative PDES synchronization protocols. Moreover, lookback-based protocols are shown to be capable of completing the simulation earlier than the limit imposed by the critical times of events, which is an insurmountable bound for traditional conservative protocols and optimistic protocols without optimization. Furthermore, it has been demonstrated that lookback can also be exploited in optimistic simulation to reduce the number of rollbacks and anti-messages.

ACKNOWLEDGEMENT

Throughout my PhD study, what I am most proud of is not the discovery of lookback, but the fruitful collaboration with Dr. Boleslaw Szymanski that led to the discovery. Discussion with him is invariably challenging, exciting and productive. Many times after I presented him with the problem, he could quickly point out the direction to proceed, which always turned out to be the right direction. Without his insightful guidance, lookback would have been studied in a completely different and far less successful way and the theory of lookback would not have been as complete and profound as it is now. His help, not only academic but in other forms as well, guided me through the arduous PhD study.

I am also greatly indebted to Dr. Christopher Carothers. It was him who first introduced me to the interesting field of Parallel Discrete Event Simulation. His course on PDES helped me establish a deeper understanding of the field, which made the discovery of lookback possible. I am also grateful to Dr. David Musser and Dr. Sibylle Schupp. Many programming techniques I learned from them became quite useful in doing my own research. I am thankful to Dr. Linda Wilson who provided me with the precious research opportunity to work on simulation projects.

CHAPTER 1
INTRODUCTION: SIMULATION AND BEYOND

Simulation is the technology that uses special devices, in most cases digital computers, to replicate the behavior of the system under investigation. A simulation can, if constructed correctly, represent the real system with a high degree of fidelity. The dynamic characteristics of the real system can then be inferred from the results produced by the simulation.

We are particularly interested in Discrete Event Simulation, for it is a very effective technique in practice. The underlying discrete event model is fairly simple, yet it is applicable to a wide variety of physical systems. We will start the discussion by presenting some fundamentals of simulation. After that, we will introduce two research directions aimed at improving the effectiveness and applicability of discrete event simulation.

1.1 Background

Prior to the emergence of simulation technology, two other methods were widely used to study the behavior of real systems. They are analytical and experimental methods [54]. An analytical method requires the establishment of an abstract model that is mathematically solvable, otherwise it would be useless. In the real world, however, such an abstract model is difficult to obtain for many complex systems. The experimental approach involves direct manipulation of the real system or a simplified reproduction of the real system. Its weakness is that such manipulation is often costly, sometimes even infeasible if, for instance, the system to be studied does not exist yet.

Simulation seems to be a hybrid of analytical and experimental methods in the sense that it bears some resemblance to both. Like the analytical method, it requires an abstract model, often referred to as a simulation model. Nevertheless, this model is only required to be executable in discrete event simulation, or numerically tractable in continuous simulation, both of which are looser conditions. As

a result simulation has a much broader range of applications than the analytical method does. Similar to the experimental method, simulation studies the system behavior by carrying out experiments, which are now conducted on computers. Unlike the experimental method, the cost of reproducing the real system is low, since it consumes only computer resources. Combining the advantages of the other two, the simulation method is playing an important role in today's scientific research.

1.1.1 The Process of Simulating

A typical simulation process often consists of four steps:

- formalization of a conceptual model,
- translation of the conceptual model into an executable program,
- model execution, and
- validation of the model with the real system.

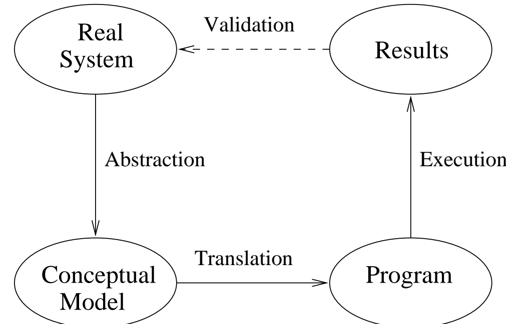

Figure 1.1: Four Steps in a Simulation Process

Formalizing a conceptual model is an important step. The model builder must understand the objectives of simulation, and select and modify basic assumptions of the characteristics of the real system. Once these assumptions have been established, the detailed specification of the conceptual model can be formulated. Translation from the conceptual model to an executable program often involves the choice of a computer language to describe the model. One can use either a general-purpose programming language, or a special simulation language, or even a simulation software

package. Validation of the model is normally done by comparing the simulation output with output generated by the real system or an analytical model.

The applicability of simulation is limited by two factors: the complexity of the simulation model and the available computing power. The extent to which the simulation model represents the real system depends on the level of model details. Generally speaking, the more details the abstract model contains, the more accurate result the simulation will produce, and the more computation power will be required. Sometimes, an over-sophisticated model demands more resources than those available. For instance, a simulation program may need a huge amount of memory that is beyond the capacity of affordable computers, or it may take so long that it literally becomes impracticable to execute. Hence, there often is a tradeoff between the accuracy and the level of model details to avoid such problems.

1.1.2 Discrete Event Simulation

There are roughly two categories of simulations [40]. In discrete event simulation, the state variables of the simulation model change only at a countable number of points in the simulated time, whereas in continuous simulation they can change continuously. The decision whether to use a discrete or continuous model for a particular system depends on the objectives of the simulation study. Continuous models are often described by ordinary or partial differential equations or other mathematical equations. Simulations based on such models are close to the pure mathematical method. The computer is only used as a means to solve the mathematical equations numerically. Discrete event models, on the other hand, involve less mathematics and often tend to focus on the more primitive principles underlying the system to be studied.

Because almost all simulations are carried out on digital computers, it is necessary even for continuous simulation to discretize the states before any computation can be performed. In this sense, continuous simulation can be seen as a special case of discrete event simulation in which one step of computation occurring at an instant of the simulated time can be abstracted as an event. Consequently, modeling methodologies developed for discrete event simulation may be also applicable to

continuous simulation. However, there is a fundamental difference between them. In discrete event simulation, events are always treated as atomic, since modeling the changes of system states as events is relatively simple. On the contrary, in continuous simulation, due to the computation complexity, events are usually characterized by complicated algorithms and course granularity. The consequence is, parallelization of continuous simulation can be accomplished by executing a single event concurrently on multiprocessors, while in parallel discrete event simulation different events are executed concurrently on multiprocessors.

The simplest discrete event simulation algorithm works as follows:

```
while(simulation end time has not been reached)
{
    find the earliest event e in the future event list;
    advance the simulation clock to the timestamp of event e;
    execute event e;
}
```

This algorithm reflects the event scheduling simulation world-view. It has many variants, differing by the policy of choosing the next event to be executed. Other simulation world-views also exist, but event scheduling is the most efficient and the easiest to implement with general-purpose programming languages, due to the way it handles simulated time. This algorithm simply associates each executable statement with a timestamp, therefore a simulation model is merely a number of executable statements sorted in timestamp order. Other simulation world-views, as we will see in next chapter, often require language constructs involving complicated simulation semantics.

1.1.3 Problem Statement

Simulations often tend to be interdisciplinary efforts. The modeling process is more likely to be performed by scientists who are familiar with the systems to be modeled. Input data generation and output analysis are often done by mathematicians. Two tasks are left for computer scientists: to make the modeling process easy and quick and to make the simulation fast.

The discrete event simulation algorithm does not have much room for algorithmic optimization because of its relative simplicity. The execution speed of simulation can be improved by simply using a faster computer. When the performance on the fastest uniprocessor is no longer satisfactory, parallel execution becomes a necessity. The parallelization of discrete event simulation, as we will explain later, is by no means a trivial problem.

Moreover, due to the greatly increasing computer speed in recent years and the employment of parallel simulation, the gap between the size of the simulation model that the state-of-the-art computer can handle and that people can effectively build has become more apparent. Simulations are hard to program because programmers need to take the time semantics into consideration. Simulation programs are hard to debug because a procedure might be executed millions of times, and each time with different arguments and within a different context. More fundamentally, the two widely used world-views, event scheduling and process interaction, do not scale well for large-scale simulations. The modeling methodology has become an obstacle that severely limits the use of simulation technology outside the research community.

We believe that a component-based approach is a natural and intuitive way to the development of large-scale simulations. We also believe that component-based modeling methodology provides a solid basis for the parallelization of simulations. The boundaries between components may imply a good partitioning criterion to map components to different processors. With proper design, the intra-component interaction is much more intensive than the inter-component interaction, so this type of parallelization can reduce the inter-processor communication significantly.

In the next two sections, we will review the current status of the research in component-based simulation and parallel discrete event simulation, respectively. Both approaches are aimed at improving the efficiency of the simulation method, though with different meanings of the word “efficiency.” Component-based simulation is focused on making the modeling process smoother and quicker, while the purpose of parallel discrete event simulation is to make simulations more feasible by speeding up the simulation execution. We will conclude this chapter by briefly mentioning our contributions in these two aspects.

1.2 Component-Based Simulation

From a computer scientist's point of view, building a simulation is nothing more than programming. It is helpful to look at the field of software engineering when we are studying simulation modeling methodologies.

1.2.1 Component-Based Software Development

Producing solid, bug-free code has long been a dream in the software engineering community. Many technologies have been proposed to improve software quality, such as management approaches, formal specification, and systematic testing [47], but none of these ideas suffice to meet the need for high-quality software. Hence, software methodologists have turned their attention to Component-Based Software Development (CBSD). CBSD builds software systems by composing existing software elements. This approach is able to increase the size and complexity of software systems that software engineers can handle, while dramatically reducing the cost and time of the development.

It is difficult to give a precise definition of what is a component. There are many definitions but none is generally accepted. One definition lists three properties [7]:

- A component is an opaque implementation of functionality;
- A component is subject to third-party composition;
- A component is conformant with a component model.

The first criterion emphasizes that the clients of software components should not need to rely upon implementation details, and that software components should be protected from disclosure as intellectual property. The second criterion states the purpose of the software components. The last criterion specifies that software components should interact with each other according to a coordination standard.

Another more informal definition suggests that a software component is a program element with the following properties [48]:

- The element may be used by other program elements;

- The client programs that use the element and their authors do not need to be known to the element's authors.

The first property excludes programs meant only for use by humans. The second property excludes the simple case of a module that is used by other modules but without the fundamental requirement of general reusability, such as a subroutine in a traditional program.

Historically, the origin of the concept of software component is hard to trace. Some people think that it derives from the idea of "divide and conquer," while others believe it is based on the concept of modular programming [19]. Still others point out that there is a close link between the component-based approach and object-oriented programming [44].

Neither the fact that component is hard to define nor the fact that its origin remains unclear prevents the widespread use of the component-based approach. In fact, many state-of-the-art commercial software development environments support the drag-and-drop technique, which is a visual form of the component-based approach.

1.2.2 Component Model for Simulation

Similar to what has happened in software engineering, the simulation community has realized that the component-based approach allows a simulation model to be built more easily and more quickly by partitioning it into smaller parts than when the model is treated monolithically. Its particular effectiveness in simulation arises from the fact that many complex systems are heterogeneous and divisible. They can be partitioned into a number of parts that interact with each other. The interaction between parts is relatively simple and therefore easy to describe. Modeling these individual parts usually requires specific knowledge in different disciplines, which makes it extremely difficult to model them as a whole.

Another advantage of the component-based approach is that it significantly improves the reusability of simulation models. Existing models can be easily integrated to form a larger simulation, with or without new ones that are built from scratch. This is beneficial because the task of model verification and validation can

be at least partially avoided. From a practical point of view, simulations that have been running for a long time and proven to be practically reliable are very valuable. New simulation design paradigms tend to emphasize reusability in order to reduce the cost and the development time of the implementation.

Still, there is a problem in the current study of the component-based approach. The need for components is merely motivated by practical necessity. Lack of a generally accepted component model leads to diverse component concepts and definitions. As long as the system can be constructed according to a modular and composable paradigm, it will be advertised as component-oriented. Looking into the simulation community, this problem is more severe, since simulated time management imposes new questions. How does a component with the notion of simulated time differ from a non-simulation component? How do we express the simulated time semantic in the component-oriented architecture? Can we connect two components that exhibit different behaviors with respect to time? If yes, how? We will try to answer these questions in this thesis.

1.3 Parallel Discrete Event Simulation

Researchers have long realized that Parallel Discrete Event Simulation is an effective approach to simulating large-scale complex systems. Research on PDES has been going on for more than twenty years. The main difficulty in this area is to achieve high efficiency of parallel execution while preserving the causality order between events for the simulation carried out on multiple processors. The logical process paradigm [28], widely used in the PDES community, assures that no causality errors will occur if each logical process adheres to the local causality constraint, i.e., if each logical process executes its events in non-decreasing timestamp order. Therefore, to preserve the causality order, it is sufficient, though not always necessary, that each logical process finds and executes the earliest future event.

The advent of PDES was marked by the invention of conservative protocols, the first of which was the null message protocol, or so-called Chandy/Misra/Bryant protocol [16, 12], developed back in 1979. In most cases, conservative protocols require each logical process to broadcast to its neighbors, in the form of null mes-

sages, a lower bound on the timestamp of events that it will send to other logical processes. This bound is often called Earliest Output Time (EOT) [36]. By listening to the null messages from all neighbors, each logical process can determine the lowest timestamp in any message it will receive later, or Earliest Input Time (EIT). If this timestamp is greater than or equal to the timestamp of the earliest event in its local event list, the process is sure that this earliest event can be processed without violating the causality constraint. Otherwise, the logical process has to block until this condition is met (i.e., the message in transit carrying the event with the timestamp smaller or equal to EIT is received and placed on the local event list).

In 1985, Jefferson proposed a dramatically different synchronization paradigm called Time Warp [35], the first optimistic protocol. In the Time Warp and other optimistic protocols, every logical process optimistically assumes that the earliest local event is safe to process. Hence, logical processes are allowed to aggressively process events in its local event list, and send to other logical processes new messages produced during the event execution. To preserve the correctness of the simulation, an arrival event with a timestamp smaller than the local simulated time will trigger a causality error. In such a case, all processed events with a larger timestamp must be rolled back, and anti-messages must be sent to other logical processes to counteract those messages sent during the erroneous computation. Ironically, although they are called optimistic, these protocols are actually quite pessimistic, in the sense that they must save every change made to the state in order to recover from the erroneous computation, because they assume that every operation is unsafe and subject to a rollback. The Global Virtual Time (GVT) [35] gives a lower bound on the timestamp of the earliest event that a logical process may receive. Therefore, any event processed earlier than the GVT is regarded as a committed event because it will never be rolled back. For such events, the logical process can reclaim the memory used to store the associated state (or state changes if incremental state saving is used).

Researchers in the PDES community meet once a year at the PADS (Parallel and Distributed Simulation) conference. This year, 2002, the conference committee claimed that “the synchronization problem, however, seems largely to have been

solved." Consequently, the theme of the conference will be extended to including general simulation techniques not necessarily parallel and distributed.

Is the synchronization problem really solved? Our answer is definitively "no." Our recent work shows that the PDES synchronization has not been completely understood by researchers, and there exist other solutions.

1.4 Research Contributions

This thesis presents contributions that address the aforementioned two existing problems in the field of discrete event simulation: the component model for simulation and synchronization for parallel execution.

First, a component-port model has been developed that differs from many existing component models. This component-port model emphasizes the independence between components, which is the key to the solution to issues such as reusability, interoperability, and interchangeability. Furthermore, a component classification and a port classification allows for a natural extension of such a component-port model to simulation.

Second, a new property existing in many simulation models, called **lookback**, has been discovered. Lookback is defined as the ability to change the past, as opposed to lookahead, the ability to predict the future, which has been known to be the key to the performance of conservative protocols. Lookback, contradicting the common opinion that the PDES synchronization is a solved problem, enables a new class of synchronization protocols that lies exactly in between conservative protocols and optimistic protocols. Moreover, we proved that lookback is always no smaller than lookahead, and lookback-based protocols can circumvent the supercritical speedup limit imposed on lookahead-based protocols. For optimistic simulations, lookback is capable of reducing the frequency of rollback and the number of anti-messages while incurring little overhead.

CHAPTER 2
COMPONENT-BASED SIMULATION: MAXIMIZING REUSABILITY

In this chapter, we focus on the goal of an efficient programming methodology for simulation. First, two existing component-based approaches are examined. After the discussion of several traditional simulation world-views, we present the component-oriented simulation world-view, in which a new component-port model plays an important role. Component classification and port classification make this component-port model well suited for the simulation domain. More importantly, these two classifications provide a theoretical basis for composability in simulation: the component classification groups components into different categories by their time behaviors, while the port classification determines whether a set of components can be connected together and, if they can, how they are connected.

2.1 Related Work

Many component-based approaches have been developed and put into practical use. We limit our discussion to two that are widely recognized: CORBA for general software engineering and HLA designed exclusively for simulation.

2.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) [3] is a general-purpose component-based approach. Developed by the Object Management Group, CORBA is targeted at building object-oriented, distributed applications by integrating software components. A legacy application can be reused by wrapping it with a standard interface.

CORBA can be seen as an improvement over the Remote Procedure Call (RPC) [10]. RPC was introduced as an attempt to make some features of a system transparent to most programmers. It hides the low level details of network-based message passing by generating functions that a client can call to request some service

from a different machine. Calling an RPC function is just as simple as calling any ordinary function, thus making the network transparent to programmers. CORBA follows this direction, but it uses an object-oriented approach.

The basic building block in CORBA is the CORBA object [59]. It consists of data and methods that may be invoked by other objects. An object's public interface is defined with the CORBA Interface Definition Language (IDL). A client simply uses this interface to invoke methods—it has no information regarding the location, the platform, or implementation details of the server object. The Object Request Broker (ORB) acts as a broker between the client and the server object, and hides the details of the server object from the client. All messages between the client and the server object must go through the ORB.

CORBA is a language independent approach. Clients and servers may be implemented in any supported language and still be able to work together. Currently CORBA supports C, C++, Smalltalk, Ada, Cobol and Java. The IDL compiler translates the CORBA interface into stub and skeleton code in the supported languages.

Although usually referred to as a component-based approach, the CORBA component model [4] was just recently completed. It is an extension of the CORBA object model in that it emphasizes composability, i.e., enabling rapid application assembly from off-the-shelf, pre-packaged software objects. A separate specification effort accompanying the CORBA component model is the CORBA scripting language [2]. It can be used to assemble existing components, to configure, administrate and connect components, as well as used for fast prototyping and automated testing.

The abstract interface is at the heart of the CORBA technology. It can be viewed as a generalization of object-oriented programming and uses the same key concept: the implementation of the object should be hidden from the clients that use that object. Such information hiding increases reusability because clients make no assumptions about how the object was implemented. With an object-oriented programming language, however, it is assumed that the client should always use the same language in which the object was implemented. CORBA goes one step

further. An object programmed in one language may be used by a client that is using a different language, and neither language needs to be object-oriented. In this way, existing applications can be easily reused regardless of the language in which they were built.

2.1.2 HLA

Developed by the US Department of Defense, the HLA (High Level Architecture) [1, 21] provides a software architecture for integration of a wide variety of simulations. A major design goal of the HLA was to provide a modeling mechanism for reusing existing simulations so that the cost and time required to create new ones can be dramatically decreased. The Department of Defense (DoD) has mandated that the HLA be used across all classes of simulations within the DoD. In addition, the HLA has been adopted as the standard for distributed simulation by the Object Modeling Group, and has already been accepted as IEEE standard 1516.

An HLA component is not restricted to a simulation. It can be a manned simulator, a supporting utility or even an interface to a live player or instrumented facility. In the HLA terminology, components are referred to as federates. A collection of federates linked together during a simulation run is referred to as a federation. Each federate must provide an interface through which messages will be passed and received. Federates do not communicate directly with one another but only with the Run-Time Infrastructure (RTI), which acts as a communication bus.

The HLA standard is composed of three parts [1]:

- HLA rules are design principles and constraints on HLA-compliant federates (simulations) and the entire federation. The benefit for the developer of introducing the rules is that they summarize the way the HLA is intended to be used.

- HLA object models deal with description of the critical aspects of simulations and federates which are shared across a federation. For this purpose, the HLA defines a standard object model template to document object models. There are two types of object models: the HLA Federation Object Model (FOM) and the HLA Simulation Object Model (SOM). The HLA FOM describes the

set of objects, attributes and interactions which are shared across a federation. The HLA SOM describes the simulation (federate) in terms of the types of objects, attributes and interactions it can offer to future federations.

- An HLA interface specification describes the runtime services provided by the RTI to federates and by federates to the RTI. There are six classes of services: federation management, declaration management, object management, ownership management, time management, and data distribution management.

In the HLA, reusability is understood in a much broader sense than the common notion of reusability in the software engineering community [52]. Not only the simulation itself but also all objects programmed in the simulation should be reusable. Runtime information of all objects within a simulation is collected by a federate wrapper and available to the RTI. In this way, a simulation becomes a federate. Communication between federates is implemented with a publishing and subscription scheme and must go through the RTI. A federate may subscribe to a specific object class and, as a result, the RTI will notify it whenever a new object of this class is discovered. The subscriber federate may also request receiving updates to the subscribed object whenever its attributes are changed. The publishing federate will then notify the RTI whenever an object's attribute value changes. The changes will be forwarded to the subscriber federates. In this way, simulations developed completely independently can interoperate with each other in a new simulation environment.

2.1.3 Problems with CORBA and HLA

One of the problems of CORBA is that it does not permit direct communication between objects, since the CORBA ORB hides the location of the object. A client can request services from an object by its identity, without knowing whether the object resides on the local machine or on the remote machine. However, if the client and the object happen to reside on the same machine, this communication paradigm incurs unnecessary overhead, because it must still go through the ORB. Thus, CORBA is only suited for application-level integration, where coarse granularity can amortize the communication overhead.

CORBA has been used to integrate simulations. Since it is targeted at general applications, it has no specific infrastructure to support simulation modeling. As a result, integration of simulations using CORBA is not easy. Time management, which has been implemented in the HLA, must be fully taken care of by the modeler.

Another disadvantage of CORBA when applied to simulation is the problem with the underlying Client/Sever paradigm. A client behaves differently from a server object, and only the client can initialize the communication. However, components of a simulation should be treated equally. They can either invoke other components or be invoked by others. The asymmetry in CORBA further complicates simulation modeling.

In contrast to CORBA, the HLA was specifically designed for the purpose of simulation. The HLA primarily addresses the problem of interoperability at the object level. However, it accomplishes this goal by ignoring the other factors that may impede its applicability. For instance, the HLA designers largely neglected the issue of efficiency. The publishing and subscription scheme defined by the HLA prevents peer federates from communicating efficiently with each other. The peer-to-peer communication is done by placing the information messages on the RTI which are then received by other federates listening on the communication bus. This is somehow analogous to a hardware bus that is a communication link between devices that implement a well-defined interface. The hardware bus reduces the cost by sharing a single set of wires among a number of devices.

A software bus certainly enables the same degree of composability. New components can be easily added without any modification to the existing system. However, there is a major difference between a software bus and a hardware bus. A hardware bus supports multicast communication without loss of efficiency due to its inherent parallelism. A software bus, however, cannot implement multicast efficiently unless it is directly built on top of special hardware. For example, the TCP/IP protocol, the primary communication method in distributed computation, supports only end-to-end communication. Multicast can be mimicked but it is not as efficient as in a hardware bus.

Thus, the software bus approach runs into difficulty whenever a publishing

federate notifies the RTI that it has changed the value of an object's attributes. This update has to be multicast to all subscriber federates that have subscribed to the object class. Not all update information is needed by all subscriber federates, but there is no way to single out those federates that are not interested in a particular update. They still have to receive the unnecessary data, and throw them away afterwards.

Besides, the primary drawback of the hardware bus is that it may become a bottleneck when the number of connected devices increases. The software bus inherits the same problem, which means that the HLA is not a scalable solution to simulation modeling.

Having recognized that neither CORBA nor HLA is a perfect approach for simulation modeling, we started to develop a new approach that enables component-based simulation. We realized that a clearly defined component model is at the center of component-based simulation. However, in order to take a component view of simulation, we must first turn to the classical simulation world-views, since a component-based approach sees a simulation as a composition of components and therefore, as we shall see, deviates from all classical world-views.

2.2 Traditional Simulation World-Views

Traditional simulation world-views, such as *Event Scheduling*, *Activity Scanning*, and *Process Interaction*, do not emphasize the composability of simulation models [20]. When adopting these world-views, modelers tend to model the real system as a whole, which inevitably limits the reusability of the simulation model.

A world-view is used by the simulation modeler to describe the dynamic structure, which defines how the system state changes over time. Basically, there are three classical world-views: *event scheduling*, *activity scanning*, and *process interaction* [55, 20].

2.2.1 Event Scheduling

In the event scheduling world-view, a modeler defines events that can occur during the course of the simulation. An event may change the state of the system.

In addition, an event can schedule other events to occur at later times, or cancel other events that have been previously scheduled but have not yet occurred. All these actions are performed by a predefined procedure associated with that event, which is often referred to as an event handler.

The actions that occur at the same simulated time can be described by either a single event or a set of simultaneous events. The former, called the monolithic event approach, is able to achieve higher efficiency but it also makes it difficult to modify the simulation program when the model is changed. On the contrary, the latter, referred to as the composite event approach, can minimize modification if the partition has been conducted appropriately. However, it incurs event processing overhead, because more events need to be inserted into and removed from the event list.

2.2.2 Activity Scanning

In the activity scanning world-view, a modeler also defines events with their causal relationships and their impact on the system state. There is an additional type of events, called *contingent events*, which occur when some condition is met. They cannot be directly scheduled since they depend on the current system state. The events that are bound to happen are therefore called *determined events*. Contingent events often simulate the beginning of some activity and schedule another determined event that simulates the end of that activity.

The simulation strategy based on the activity scanning world-view differs from that of event scheduling in that an extra phase must be introduced where the conditions associated with every contingent event must be evaluated to determine if they are met. For this reason, the activity scanning world-view is sometimes referred to as a three-phase world-view. It may seem to be more complicated and less efficient than event scheduling. However, this approach pays off when the system being modeled is subject to frequent future changes. With careful design, the system change can be incorporated by only adding extra determined events and contingent events. Obviously, this is much more desirable than the monolithic event approach.

2.2.3 Process Interaction

In the process interaction world-view, a modeler defines a set of *processes*, each being a chronologically sequenced set of events and activities. Every process is associated with a procedure, which usually includes instructions for suspending execution of the procedure for an interval of time, and instructions executed when a condition becomes true. As a result, one essential distinction between process interaction and event scheduling is that execution of an event handler is instantaneous, while the execution of a process may span an interval of simulated time, or even a whole simulation run.

The process interaction world-view is often said to be the combination of activity scanning and event scheduling because reactivation of a process waiting for a condition to be true always requires periodic checking of that condition, much like the activity scanning strategy.

2.2.4 Logical Process

None of the above world-views supports the notion of constructing the simulation by composition of well-defined components. They all tend to model the system as a whole, irrespective of the natural boundaries dividing different parts. For instance, when using the event scheduling world-view to simulate an M/M/1 system, the modeler must define an event that happens when a job leaves the server. The procedure that handles this event should retrieve the first job in the queue (if there is any), put it in the server, and schedule another event of the same type that will occur after an interval of time equal to the service time needed by the new job. This modeling methodology gives no consideration to the fact that the queue and the server are two distinct parts, and that changing the state of both in a single event procedure violates the criterion of encapsulation.

The notion of *logical process* was introduced by Fujimoto [28]. Taking a logical process view, the simulation is seen as being composed of a set of logical processes that interact with each other by exchanging timestamped messages. Each logical process contains a portion of the system state, as well as a local clock that denotes how far the process has progressed. Shared variables are forbidden in the logical

process paradigm, but as Fujimoto pointed out, the exclusion of shared variables may or may not be burdensome, depending on the applications and the necessity of parallelization.

Fujimoto did not specify the world-view used to describe logical processes. The reason becomes clear if we look at the definition of world-views. The simulation world-view is used to describe the *dynamic structure* of the simulation model, which defines how the system state changes over time. By contrast, the *static structure* defines the possible states of the physical system [20, 37]. According to its definition, the logical process paradigm partitions the system state variables into a set of disjoint states. Therefore, it only deals with the static structure.

The orthogonality between world-views and the logical process paradigm leads to two kinds of logical process implementations, either based on event scheduling or process interaction. Parallel simulation frameworks using event scheduling include Georgia Time Warp [29], and ROSS [13], while Maisie [9] and TeD [53] are based on process interaction. There is no logical process implementation that uses the activity scanning world-view. Due to the absence of shared variables, the conditions used by the activity scanning and process interaction world-view now depend only on the content of received messages. As a result, the activity scanning regresses to event scheduling, and the only significant difference between process interaction and event scheduling is that process interaction allows the use of a *wait* statement, which suspends the logical process for a certain amount of time.

The logical process paradigm has become so popular that many papers on PDES use it to help clarify the discussion and almost every parallel simulator is built according to this paradigm. Its success may be attributed to the need to partition the simulation program in order to carry out parallel execution, but we believe that the real reason is its similarity to the component-based approach: the complexity of developing a parallel simulation model is drastically reduced with the logical process paradigm.

However, the sender/receiver relationship in the logical process paradigm is still not well defined. This incompleteness leads to various implementations. For instance, in some simulators, such as Maisie and Parsec, a logical process is required

to explicitly specify the destination of each sent message. The shortcoming of this approach is that the interconnection topology of the logical processes becomes embedded into the code of the logical process, which prevents the process from being reused in other simulations. In others simulators like Georgia Time Warp and ROSS, logical processes are not allowed to directly communicate with each other. A process can only schedule an event for other processes. This incurs extra communication overhead for synchronous messages that are sent and received at the same simulated time.

We identify the sender/receiver relationship as a key issue in the component-based approach for simulation. It is required that the development of a component should be completely isolated from the context in which the component will be used. This independence requirement implies that prior to the simulation the component knows neither the source of the messages it will receive nor the destination of messages it will send. In the next section we will present a component-oriented world-view that adheres to this requirement.

2.3 Component-Oriented World-View

In a component-oriented world-view a simulation is viewed as a composition of a number of components. Each component contains a set of parameters, which must be initialized during a configuration phase. After the parameter initialization, the components of the same type may exhibit different behavior, if their parameters are assigned different values.

The most distinctive feature of the component-oriented world-view is that it utilizes *ports* to send and received messages. Ports act as a communication abstraction. There are two types of ports, *inports* and *outports*. To send out a message, the component simply places the message in the desired outport. Upon arrival of a message, the component must process the message immediately.

Besides parameterization, another task of the configuration phase is to connect components by linking their matched port. Ports are typed. Each inport or outport can pass on only a certain type of messages. We say that an inport and an outport are matched if they are of the same type. An inport and an outport

can be connected together only if they are matched. An inport can be connected with multiple outports, and similarly multiple inports can be connected with one outport.

The event scheduling world-view is used to describe the behavior of components. Every component contains an event handler that will be invoked when the component is activated, which happens upon arrival of two types of events. One are incoming messages at an inport. They are called external events, because such messages are always caused by an event in another component. Besides, the second type of events are internal events scheduled by the component itself.

The sender/receiver relationship in the component-oriented world-view is different from that in the logical process paradigm. As discussed earlier, when building a logical process, the modeler must explicitly specify the destination of an outgoing message, and there is no way to know which other processes may send messages to this process. When ports are used as a communication abstraction in the component-oriented world-view, however, the decision of choosing the destination of a message placed on an outport can be postponed until the configuration phase is performed. It also becomes possible to determine the set of components that will be the senders to a particular component as those that are connecting their outports to the inport of that component. This idea is referred to as *delayed binding* by Zeigler in his DEVS framework [67].

Achieving a degree of sender/receiver independence might be viable in the logical process paradigm by specifying the destination of an outgoing message as a parameter, as in Maisie/Parsec [9, 39]. This approach runs into difficulties, however, when a message must be sent to multiple processes. A quick solution is to use a variable size parameter array instead of a single parameter. Specifying such a parameter array as the destination of a message means that the message is sent to every process that belongs to the set denoted by the parameter array. Indeed, this solution is equivalent to delayed binding, because connecting an inport with an outport can be viewed as adding an element to an imaginary destination parameter array associated with the inport.

Interestingly, the sender/receiver relationship is more dynamic and more im-

plicit in the HLA. A simulation may publish its objects to the RTI, and when another simulation subscribes to one of these objects, the two simulations implicitly become a pair of sender and receiver. However, some communication overhead is incurred when one packet is dynamically routed from one simulation to another. The fact that the sender/receiver relationship can be determined before execution and that it is not necessary to be hard-coded into the simulation model dictates a configuration phase, which many existing simulators lack.

Actually, similar world-views have already been adopted by existing simulators or simulation languages. TeD, for instance, is a simulation language designed to facilitate modeling of telecommunication networks [53]. A TeD simulation is defined as dynamic interactions of *entities* and their compositions. *Entities* are connected to and interact with each other via *channels*. Another example is the BONeS (Block-Oriented Network Simulator) [60], where components are called *blocks*. In both of them, the interface consists only of parameters and ports (or channels). The characteristics of component in terms of simulated time are not reflected in the interface. Therefore, information such as whether or not components can schedule or receive events, and more generally, how components handle the simulated time, is determined only by the implementation. This limitation, as we shall see in the next section, imposes serious problems when these approaches are used to connect a wide variety of components. Both TeD and BONeS, for instance, lack the ability to link existing simulations.

The idea of configuration phases is also not new. The Equational Programming Language (EPL) [65], for instance, uses *configurations* which define interconnections between ports of different processes to allow the programmer to reuse the same EPL programs in different computations.

To summarize, in the component-oriented world-view a simulation is built by composing a collection of components. Each component contains parameters, inports and outports, all of which are specified in the component interface. Components place outgoing messages on outports and receive incoming messages from inports. Before the execution, each component must undergo a configuration phase, in which two operations are performed. First, all parameters are initialized. Second,

the corresponding pairs of inports and outports are connected.

2.4 Component-Port Model

The component-oriented world-view sees a simulation as being composed of a set of components. It takes a divide-and-conquer approach to partition the whole simulation into a number of smaller simulation tasks, which are modeled by each component individually. The immediate benefit of doing so is that the complexity is significantly reduced. Each component is now a smaller unit whose internal logic is much simpler than that of the whole simulation. With this approach, reusability can also be achieved if the components are designed in such a way that context-relevant information is not embedded into the code of the components.

The component-port model ensures that each component is independent of the simulation in which it will be used. Components exchange messages with each other only via inports or outports. The receiver of a message is not determined until the configuration phase. Messages placed on the outport during the simulation will then be delivered to connected inport(s). Similarly, the component responds to messages arriving at an inport, regardless of the sender, thus, any component is able to accept messages at the inport sent by any other component. These features of inports and outports as well as of the configuration phase are the source of the independence between the components and the simulations, and such independence leads to maximum reusability.

The term component has been widely used without elaboration. We define a component as an object with an interface that enables it to be combined with other objects. The interface, either explicit or implicit, prescribes in what kind of interaction the components can be involved with other components. The interface alone does not distinguish a component from an object, however. The real requirement for our components is that *all interaction between components must be specified in the interface.* This is not the case in most object-oriented languages. In C++, for instance, it is taken for granted that an object can directly call a method of other objects. Such method calls are not reflected in the interface of the caller, causing problems for the object integrator who must look into the implementation

details of an object to derive an object's dependency on others. Even if this is possible, such method calls are not fully reconfigurable, for they are embedded in the implementation code.

From this point of view, many existing approaches like CORBA, DCOM, and JavaBeans, are not truly component-based because most of them allow an object to directly call a function of another object. Although the function to be called exists in the callee's interface, it is not specified in the caller's interface. Consequently, the interaction between objects is not fully captured by the interface. The more serious problem is that this kind of binding does not produce composable objects. The dependency is now buried in the code of the caller object and would remain fixed unless the code can be modified.

Outports are the solution to this problem. Inports are equivalent to functions. They prescribe what functionalities a component can provide. Outports, in contrast, prescribe what functionalities a component may require from others. By delaying their connection until the configuration phase, the binding becomes more flexible. For instance, a component integrator may try to link the outport of a component with the inport of several different other components in order to obtain optimal results.

However, a problem still exists regarding how to handle the simulated time. Should time be ignored in the basic model as in CORBA, or should the time be built-in as in the HLA? The former is unrealistic, for our intention was to develop a component-based approach for simulation. The latter is also problematic because different components exhibit different time behaviors. The component classification is devised to resolve this issue.

2.5 Component Classification

Components can be classified into three types with respect to the way they handle the time semantics: time-independent, time-aware and autonomous, named also Type I, Type II and Type III respectively.

2.5.1 Time-Independent Components

A Type I component does not have the notion of simulated time. It is passive in the sense that it never generates messages without first having received a message. A component, when processing a message received from other components, may generate new messages that are required to have the same timestamp as the incoming message that triggered it. Yet, the component itself is unaware of the time semantics. Neither does it know whether it is running as a part of a simulation program or a part of a non-simulation program. For this reason, a time-independent component is said to be time-unaware.

The interface of a time-independent component is composed of parameters, inports and outports. Inports receive messages while outports send out messages. An example of time-independent components is the FCFS (First-Come-First-Serve) queue without a server, whose interface is given below. The FCFS queue has an inport *in* that accepts arriving items, which are then stored in an internal queue according the order they are received. Whenever a trigger message arrives at the inport *next* requesting the next item, the first item in the queue will be sent to the outport *out*.

```
component FIFO
{
    param int capacity;
    inport in(ITEM item);
    inport next();
    outport out(ITEM item);
}
```

2.5.2 Time-Aware Components

Type II components are time-aware components. They cannot advance the simulated time themselves but they can request a time advance via a special object called a *timer*. Timers provide a mechanism for components to schedule and receive events. To schedule a future event, a timer is set with a time increment representing the difference between the current simulated time and the timestamp of the event.

As soon as the specified simulated time increment elapses, the component where the timer resides will be activated and then forced to process the timer event.

A server is a simple time-aware component, as shown below. The function of a server is to simulate processing of an item for a certain amount of time. When there is an item arriving at the inport *in*, the server stores the item in an internal buffer and sets the delay duration by writing the duration value to the timer *wait*. The delay duration is randomly chosen from an exponential distribution with a mean value of *average_delay*. The timer *wait* will be activated by the simulation engine when the specified delay elapses. The item is then released from the internal buffer and passed to the outport *out*. At the same time, an empty message is delivered to the outport *next* to notify other components (possibly an FCFS queue) that are waiting for the service completion.

```
component Server
{
    param double average_delay;
    inport in(ITEM item);
    outport out(ITEM item);
    outport next();
    timer wait;
}
```

2.5.3 Autonomous Components

Type III components are named autonomous components because they maintain their own simulation clock themselves. They do not have any timers. Instead, they contain a *clock*, which indicates the simulated time throughout the simulation. In a conservative simulation, these components can receive only a message that will not cause any causality errors. The easiest way to guarantee it is to ensure that each received message has a timestamp larger than the value of the simulation clock.

All stand-alone simulations can be viewed as autonomous components without ports. For a simulation to be linkable, it should have outports that are used by the simulation to produce data available to other components and/or inports that

require additional information (often dynamic) during execution. For example, a Lyme disease simulation [24, 64] can be constructed as a composition of two components, one simulating mice and the other simulating ticks. Ticks are immobile but they can be carried around and dropped by mice. The Mice component therefore needs to access the tick density at the location where an event related to mice activities has occurred.

The following shows the interface of a Mice component. When a mouse is bitten, the component will send a message to the outport *tickbite*, indicating the location where the bite occurs. The number of ticks biting the mouse will be returned in the argument *tick* since it is passed by reference. Similarly, when a mouse drops ticks that it carries, the component will send a message to the outport *tickdrop*, indicating the location of the mouse and the number of dropped ticks.

```
component Mice
{
    param int width, height;
    outport tickbite(int x, int y, TICK& tick);
    outport tickdrop(int x, int y, TICK tick);
    clock c;
}
```

2.5.4 Simulation Engine

Simulation engines play an essential role in the component-based simulation. Components are always coupled together by a simulation engine. The simulation engine is responsible for parameterization and interconnection of components. In the configuration phase, it sets up channels that directly connect matched ports. During the runtime, the simulation engine must also keep track of all activities on timers for Type II components, and clocks for Type III components. When a timer is scheduled to be active in a future simulated time, the simulation engine inserts a corresponding event into an event queue. It repeatedly removes events one by one, usually in increasing timestamp order, from the event queue and then activates the corresponding timer by invoking the event handler of the component to which the

timer belongs.

A simulation engine could be built as either a Type II component or a Type III component. A Type II simulation engine should contain a master timer which must always be synchronized with the next local event in the simulation engine. The engine executes only one event every time the master timer is activated, and then sets the activation time of the timer to the timestamp of its next local event before returning control to the higher-level simulation engine.

The identification of simulation engines clarifies the role of the simulation developers. A simulation program can be divided into two parts: one modeling the behavior of the real system and the other simply making the model executable. In this sense, a simulation engine has nothing to do with modeling of the real system, since it is only concerned with the underlying implementation issues. Hence, the details about the implementation of the simulation engine should be hidden from the model builders who usually have no specialized knowledge on simulation. On the contrary, simulation models which could be of Type I or Type II include both the code that represents the real system and the code that allows components to exchange data with the simulation engine. The latter is the sole responsibility of the simulator builders but should be transparent to the model builders.

2.5.5 Time Properties

A close examination reveals that components impose various requirements on the timestamp of the simulated time (Table 2.1). Type I components require *copyable* timestamps, because they must copy the timestamp of a message arriving in an inport to outgoing messages which can only occur at the simulated time of the received message. Type II components require *incrementable* timestamps, because when they write a delay duration to a timer, they implicitly schedule an event whose timestamp is equal to the current simulated time plus the specified delay. Type III components naturally ask for *comparable* timestamps for the purpose of deciding causality. For sequential Type III components, timestamps are *totally-ordered* while for parallel and distributed Type II component they should be *partially-ordered.* We will explain why Type I and II components may require the property of being

comparable after we present the idea of lookback.

There are still two other properties to be considered. A component modeling a time-variant system may require the property of being *readable*. Type III components also needs *writable* timestamps for the purpose of modifying the simulation clock. Specifically, sequential and conservative Type III components require *monotonically writable* timestamps.

Component	Copyable	Incrementable	Comparable	Readable	Writable
Type I	Yes	No	Optional	Optional	No
Type II	Yes	Yes	Optional	Optional	No
Sequential Type III	Yes	Yes	Totally	Optional	Monotonically
Parallel Type III	Yes	Yes	Partially	Optional	Yes

Table 2.1: Properties of the Simulated Time

In practice, the simulated time is usually implemented as floating-point numbers, which possess all properties discussed above.

2.6 Port Classification

In the component-port model, all messages exchanged between components via ports are timestamped (Type I components may ignore the timestamp). This imposes no problem for sequential simulations. However, in parallel and distributed simulation where time is no longer totally-ordered, a component is selective in what messages it can receive. A serial component can only receive messages with a timestamp no smaller than its current simulated time. One the other hand, if a component was designed to handle out-of-timestamp order messages (stragglers), there are no reasons not to permit this. Furthermore, optimistic simulations require components to be able to receive anti-messages, which are to annihilate erroneous positive messages. Since ports are the carriers to send and receive messages, their connections can be checked in order to guarantee consistent message delivery.

To accomplish this goal, we observed that ports can be divided into three categories:

serial ports that send and receive only positive messages with the timestamps larger than the current simulated time of the component,

virtual ports that may send and receive positive messages and anti-messages, and

lookback ports that may send and receive positive messages with arbitrary timestamps (including those smaller than the current simulated time of the component).

The distinction between the first two port types has been well known since the invention of optimistic simulations. The third type was made possible by our discovery of lookback, yet lookback ports have actually existed in optimistic simulation engines even before the concept of lookback was invented. Optimistic simulation engines are able to accept and handle a straggler by translating it into a set of anti-messages whose corresponding positive messages are affected by this straggler. Traditional optimistic components that reside in such a simulation engine only observe the arrival of anti-messages, not the existence of stragglers. The explicit identification of lookback ports leads to the feasibility of directly exposing components to stragglers, which requires Type I and II components to be capable of comparing timestamps.

Different types of ports may coexist in the same component. For example, imagine an FCFS server with a lookback inport. It deals with stragglers without difficulty, because such stragglers can be correctly processed by inserting them into the internal list sorted in the timestamp order. The outport of the FCFS server, however, is always a serial outport. The nature of the FCFS server ensures that it never outputs events in out-of-timestamp order. This outport can then be connected to a serial inport of another component, which would only receive timestamp ordered messages.

Tentatively we propose two criteria regulating how ports can be connected together.

Criterion 1. *For an outport to be connected with an inport, the type of an outport must be the same as or more primitive than the type of inport.*

We say that serial ports are more primitive than virtual ports and lookback ports, however virtual ports and lookback ports cannot be compared. This criterion is almost self-explanatory, because otherwise the inport would have difficulties in processing the messages sent by the outport. The opposite is not true, however. It is possible, for instance, to link a virtual inport with a serial outport. In such a case, the virtual inport will never receive an anti-message.

The type information provides only a necessary but not sufficient condition of the correctness of the connection between two ports. We must also guarantee that the message passed to an inport of a component is acceptable to the component. To ensure such a guarantee, a lower bound on timestamp of future events that can pass through, which is referred to as *LBTS*,[1] is defined for each port. The LBTS of an inport is the timestamp of the earliest event received from this inport that the component can receive and successfully process. The LBTS of an outport is a guarantee made by the component to not deliver to this outport any messages with a smaller timestamp than the LBTS. Table 2.2 lists typical LBTS values of ports that belong to different types of components, where T is the current simulated time of the component.

Component Type	LBTS of Inports	LBTS of Outports
Type II	T	T
Conservative Type III	T	$T + LOOKAHEAD$
Optimistic Type III	GVT	GVT

Table 2.2: Typical LBTS values of Ports

Criterion 2. *An inport can never receive a message earlier than its LBTS.*

While the first criterion can be performed in the configuration phase prior to simulation, the second criterion has to be enforced during the execution of the simulation. For a connected pair of an inport and an outport, if the LBTS of the outport is greater than the LBTS of the inport, the connection has to be temporarily broken until the constraint holds again.

[1]LBTS is in fact a function with the current simulated time as its only argument. For simplicity we omit the argument here.

2.7 Comparison with CORBA and HLA

CORBA can be said of being a component-based approach in which only time-independent components are involved. As mentioned above, the CORBA object model makes no use of outports, which is a serious flaw.

The main problem with the HLA, from our point of view, is that it treats all components uniformly. As made evident by the classifications of components and ports, a simulation model is characterized by its component type and port types. It is difficult, if ever possible, to directly connect simulation models that are not compatible with each other in terms of component type and port type. The attempt of the HLA to link together components without considering their types results in a centralized software bus that is cumbersome to implement and inefficient to execute.

Our component-based approach relies on typed simulation engines to couple components together. There are no predefined or built-in simulation engines. Rather, the simulation engine is treated as a component too, either of Type II or of Type III, and it can only link components of the type for which it was designed. A new type of components may require a new simulation engine, but once the simulation engine has been designed it could be used to link any components of the same type. More importantly, since a simulation engine is also a component, it could be linked with other components by a higher-level simulation engine. This hierarchical structure of linkage may avoid the bottleneck problem caused by the communication bus in the HLA.

With the component and port classification, it is feasible to unify the modeling process of a single simulation and the linking process in which a multi-simulation is built from several simulations. For instance, an M/M/1 simulation is composed of three components: a source component, an FCFS server and a sink component. Similarly, a forest fire simulation may consist of a local weather simulation, a fire model, a sensor network and so on. The process of building such a forest fire simulation should be no different from that of the M/M/1 simulation, if we are to develop a truly scalable modeling process. With computing power increasing steadily, the scale of simulations that can to be built grows equally steadily. A huge multi-simulation

in today's standard that needs to run on state-of-the-art supercomputers may fit into a single processor in the near future, thus becoming readily subject to further integration. With the help of the component and port classification, we can apply the same modeling process to simulations with different scales, by using different simulation engines.

2.8 Conclusion

We have proposed a component-oriented world-view for simulation, by refining the sender/receiver relationship in the logical process paradigm. A component-port model was then devised to support such a world-view. The component and port classification naturally extends the model to the simulation domain.

CHAPTER 3
COST: A COMPONENT-ORIENTED DISCRETE EVENT SIMULATOR

In this chapter, we describe a design of a sequential discrete event simulator, named COST (Component-Oriented Simulation Toolkit), based on the approach proposed in the previous chapter. According to the idea of hierarchical modeling and of the component classification, there should not be too much difference between a single simulator and a simulation integration architecture, except that the former deals with Type I and II components rather than Type III components.

However, the actual reason for designing yet another sequential discrete event simulator is that none of existing ones enable perfect component-based simulation. The main design purpose of COST, therefore, is to follow the component-oriented world-view detailed in last chapter, in order to maximize the reusability and independence of simulation models without losing efficiency.

Simply put, a simulation written with COST is built by configuring and connecting a number of components, either off-the-shelf or fully customized. Components interact with each other only via inports and outports, thus the development of a component becomes completely independent of others. The component-port model underlying COST makes it easy to construct simulation components from scratch. Implemented in C++, COST also features a wide use of templates to facilitate language-level reuse.

From a practical point of view, a good simulator must possess two essential features. First, it must support reusable models. A model written for one simulation should be able to be effortlessly embedded into other simulations that require the same kind of a model. Second, the model should be easy to be built from scratch. Interestingly, we observe that most existing simulators emphasize only one feature. Most commercial simulators provide a reusable model library, often accompanied with a quite friendly graphical user interface, but adding new models to the library is often painful. On the other hand, most freely available simulators follow a bottom-

up approach: writing models from scratch is straightforward, but the reusability of models is severely limited.

COST attempts to address these two problems simultaneously. The key to the solution is the component-oriented worldview as well as the underlying component-port model.

3.1 Design and Implementation

The first issue of implementing the aforementioned simulation component model is the choice of the implementation language. Discrete event simulators can be roughly divided into two groups: those based on a special simulation language, such as GPSS and SIMSCRIPT [40] and those based on a general programming language, such as SIMPACK [26] and SIMKIT [31]. Simulation languages contain abundant semantics designed for simulation, but require a steep learning curve. General programming languages are more familiar to programmers, but lack the essential simulation constructs.

We chose C++ as the implementation language for two reasons. First, general programming languages always have good compiler support, and their execution speed is generally faster after optimization. Second, language-level reusability is a factor as important as component-level reusability, and C++ is one of the few languages that support code reuse well. With STL [6, 49], C++ programs can easily achieve high efficiency while maintaining a high level of code reuse, and these properties match our design goal. However, with C++ we ran into a problem. As mentioned in last section, inports are equivalent to functions, so it is natural to define them as member functions of the component. But how can we represent outports? The C++ language standard requires that the address of an object must be provided when the member function is being called. This requirement conflicts with the requirement that component development should be completely independent. The classical solution for such a problem is a functor, which is the generalization of the function pointer.

3.1.1 Functor

A functor, or a function object, is an object "that can be called in the same way that a function is" [6, 49]. A functor class overloads the function-call operator, *operator()*. For instance, the following is declaration of a *Functor* class that takes exactly one function argument.

```
template <class T>
class Functor
{
public:
    typedef T data_t;
    struct event_t : public CEvent
    {
        T data;
    };
    virtual ~Functor(){};
    virtual return_t operator() (const T&)
    {
        return false;
    };
};
```

The data type associated with the functor can be obtained by using *Functor::data_t*. *event_t* defines the type of the event, derived from the base event type *CEvent*, that can be used to store the data if necessary.

The class *Functor* mainly serves as the base class of *MFunctor*, a helper class that wraps a member function. In C++, a member function of a class always takes an implicit parameter *this*, which is a pointer to the object upon which the member function will be invoked. As a result, two member functions that belong to different classes but take the same explicit parameters are treated as functions of different types. In the component level, however, they should be viewed as interchangeable. A *MFunctor* declared below can hide the class type as well as the implicit parameter *this*.

```
template <class C, class T>
class MFunctor : public Functor <T>
{
public:
    typedef C class_t;
    typedef return_t (C::*funct_t)(const T&);

    MFunctor(C* _obj, funct_t _f)
        :obj(_obj), f(_f)
        {
        };
    return_t operator() (const T& t)
        {
            return (obj->*f)(t);
        }
private:
    C* obj;
    funct_t f;
};
```

With these two classes, *Functor* and *MFunctor*, it is now straightforward to implement inports and outports. An inport can be simply an instance of the *MFunctor* class. Since an outport does not know the component(s) to which it will be connected, it can be represented as a pointer to a functor. When connecting an inport to an outport implemented in this way, the address of the *MFunctor* object corresponding to the inport is assigned to the functor pointer corresponding to the outport, because the class *MFunctor* is derived from the *Functor* class. When the outport is invoked, the *operator()* of the *MFunctor* class will be called instead, because it is declared as virtual.

3.1.2 Inport and Outport Classes

The above method of implementing inports and outports directly on top of two functor classes should work well, but there are some practical considerations. For instance, a port should have a name for the purpose of the debugging, as well as some status flags to indicate whether or not it has been set up correctly. Moreover, one to multiple connections would make topology generation more convenient. It is easy to connect an inport to multiple outports by passing its address to each of them, but when connecting an outport to multiple inports, the outport must store the addresses of all connected inports. All such reasons are the main motivation for having separate *inport* and *outport* classes on top of functor classes.

The *outport* class is declared to be a class with a template parameter that is the type of the events that can be handled by the outport. The function *Setup* gives the port a string name. The function *Write* is invoked by the component to output a message. *ConnectTo* connects an inport to the outport. *typeii* is the base of all customized component classes.

```
template <class T> class outport
{
public:
    void Setup(typeii* c, const char* name);
    bool Write(T& t);
    void ConnectTo(inport<T>& port);
private:
    std::vector<Functor<T>*> inports;
};
```

Similarly, the inport class takes one template parameter that is the type of the function argument. It must be bound to a member function of a component, therefore the type of the component is passed as the template parameter of the member template function *Setup*, as shown below.

```
template <class T> class inport
{
```

```
public:
    template <class C>
    void Setup(C* c,MFunctor<C,T>::funct_t f,
                 const char* name);
    bool Write(T& t) { return (*ft)(t); };
private:
    Functor<T>* ft;
};
```

Since the type of the member function bound to the inport must be passed to the *Setup* function, we need to find a way to construct this type from two template parameters, C and T. Fortunately, this type is declared publicly in the class *MFunctor*$<C,T>$ as *funct_t*.

3.1.3 Simulated Time and Port Index

Until now, functors in COST take only one function argument, which is the message exchanged between components. In practice, two more arguments are necessary. First, all the components in COST are time-aware components (a time-independent component can be viewed as a special time-aware component without timers), so messages should be timestamped. Hence, an extra argument is needed to denote the simulated time at which the message is generated. Another extra argument is necessary for arrays of inports, which are convenient if a number of inports are of the same type. All elements in an inport array share the member function bound to them, while associated with different indices. Thus, it is necessary to have an extra argument to distinguish between them by their indices. For an inport that does not belong to any inport array, the index is by default zero.

After having introduced two extra arguments, two versions of *operator()* functions are needed, differing in whether or not the index argument is required. If it is not provided, the corresponding argument passed to the wrapped member function will be retrieved from the member variable f in *MFunctor*. The modified declaration of *Functor* and *MFunctor* is shown below:

```
template <class T> class Functor
```

```
{
public:
    typedef T data_t;
    struct event_t : public CEvent
    {
        T data;
    };
    virtual ~Functor(){};
    virtual return_t operator() (const T&,simtime_t)
    {
        return false;
    };
    virtual return_t operator() (const T&,simtime_t,unsigned int)
    {
        return false;
    };
};

template <class C, class T>
class MFunctor : public Functor <T>
{
public:
    typedef C class_t;
    typedef return_t (C::*funct_t)(const T&, simtime_t, int );

    MFunctor(C* _obj, funct_t _f, unsigned int _i)
        :obj(_obj), f(_f), index(_i)
    {
    };
    return_t operator() (const T& t1, simtime_t time)
    {
```

```
     return (obj->*f)(t1,time,static_cast<int>(index));
    }
    return_t oeprator() (const T& t1, simtime_t time,unsigned int i)
    {
        return (obj->*f)(t1,time,static_cast<int>(i));
    }
private:
    C* obj;
    funct_t f;
    unsigned int index;
};
```

3.2 Timer

A timer can be seen as a mix of an inport and an outport. When it is used to schedule an event, it acts as an outport, except that the message is sent to the simulation engine. When the scheduled event time is reached, the timer is activated and receives the event that it scheduled earlier. As a result, a timer must be bound to a functor, like *inports*.

A timer object is actually an array of timers, each of which is identified with a unique integer index, as in the inport arrays. The *timer* class has two methods: *Set* to schedule an event and *Cancel* to cancel an event.

```
template <class T> class timer
{
public:
    void Setup(typeii*, mem_functor<C>::funct_t, const char* name);
    void Set(T t, double time, int index=0);
    void Cancel(int index=0);
private:
    Functor<T> *f;
};
```

So far, we have described techniques for implementing the component-port model in C++. It should be noted that all these implementation details are transparent to users. Users do not need to have advanced knowledge of C++ templates in order to program simulations in COST.

3.3 Simulation of an M/M/1 System

To illustrate the modeling process with COST, we will describe in detail how to build an M/M/1 simulation. In an M/M/1 system, packets arrive according to a Poisson distribution and compete for the service in an FCFS (First-Come-First-Served) manner. The service time is also drawn from a Poisson distribution. In practice, M/M/1 systems are useful because many complex systems can be abstracted as compositions of simple M/M/1 systems. M/M/1 systems also have an accurate mathematical solution with respect to the arrival rate and the service rate, which makes them well suited for validating simulation results.

An M/M/1 system built in COST is composed of three components, namely, *source*, *FCFS server*, and *sink*, as shown in Figure 3.1. Packets are generated by *source*, queued and served by *FCFS server*, and dispatched from *sink*.

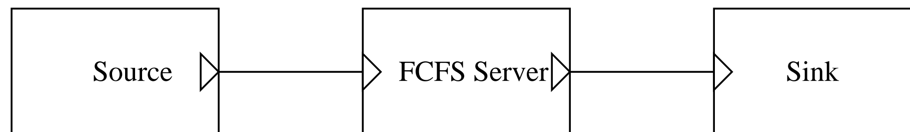

Figure 3.1: An M/M/1 System

3.3.1 Data Type

A new data type, *Packet*, is defined to represent packets that flow through the M/M/1 system. To measure the time spent in the *FCFS* component for each packet, a field *arrival_time* records the arrival time of a packet at the *FCFS* component.

```
struct Packet {
    double arrival_time;
};
```

3.3.2 Source

The *source* component creates packets at a rate specified by a random interval. It contains an outport of type *Packet* and a timer for scheduling the time to deliver the next packet. It is derived from the class *typeii*, the base class of all COST components. Since events used to schedule delivery need not contain any information, the timer *wait* is declared as of type *trigger*, a predefined class without any fields.

```
class Source : public typeii
{
public:
    double interval;
    outport < Packet > out;
    timer < trigger>  wait;

    void Setup(const char*);
    void Start();

private:
    bool Create(const trigger&, simtime_t, int index);
};
```

All COST components must provide a *Setup* function in which the *Setup* function of every port and timer must be called. The *Setup* function of the base class *typeii* must be invoked first.

```
void Source::Setup(const char* name)
{
    typeii::Setup(name);
    out.Setup(this,"out");
    wait.Setup(this, &Source::Create,"timer");
}
```

The *Start* function, invoked the moment the simulation gets started, i.e., at the simulated time zero, is where a component can perform initialization of variables and

schedule initial events using the *Set* method declared in the *timer* class. *Exponential* is another method declared by *typeii* to generate a Poisson distribution.

```
void Source::Start()
{
    m_seq_number=0;
    wait.Set(Exponential(interval));
}
```

The *Create* function is bound to the timer *wait*, so it is invoked every time the timer becomes activated. Its tasks include scheduling the event representing the next packet to be generated and delivering the current packet to the outport. Finally, it returns a true value, which is required for all member functions that are bound to inports or timers. A true value indicates that the function has finished successfully.

```
bool Source::Create(const trigger&,simtime_t time,int)
{
    wait.Set(time+Exponential(interval) );
    Packet packet;
    packet.arrival_time = time;
    out.Write(packet,time);
    return true;
}
```

3.3.3 FCFS Server

The *FCFS* component is declared as a template class with a template parameter that has the type of packets that the FCFS server can hold. By instantiating it with different packet types, the FCFS component is capable of holding packets of any type. It could have been designed particularly for packets of type *Packet*, but that would prevent it from being used in a different simulation for any types other than *Packet*. This exemplifies the great benefit of using C++ template.

The FCFS component contains an inport and an outport, to receive and send packets, as well as a timer to simulate the service of packets. A public member variable *service_time* specifies the average service time each packet will require. There are three private member variables: *m_busy* reflects the status of the server; *m_queue* stores the packets waiting to be serviced; *in_service* is the packet that is currently being serviced.

```
template < class DATATYPE >
class FCFS : public typeii
{
public:
    void Setup(const char*);
    void Start(){m_busy=false;}

    double service_time;
    inport < DATATYPE > in;
    outport < DATATYPE > out;
    timer <trigger> wait;
private:
    bool m_busy;
    std::deque<DATATYPE> m_queue;
    DATATYPE in_service;

    bool Arrive(const DATATYPE& packet, simtime_t, int index);
    bool Depart(const trigger&, simtime_t, int index);
};
```

Again, the *Setup* function sets up every port and timer. We can also perform some variable initialization here.

```
template < class DATATYPE >
void FCFS <DATATYPE>::Setup(const char * name)
{
```

```
    typeii::Setup(name);
    in.Setup(this,&FCFS<DATATYPE>::Arrive,"in");
    out.Setup(this,"out");
    wait.Setup(this,&FCFS<DATATYPE>::Depart,"next");
    m_busy=false;
}
```

The *Arrive* function is called when a packet arrives. Notice that packets are passed by reference to avoid variable copying overhead. The keyword *const* prevents the packet from being modified accidentally in the function.

The value of *m_busy* denotes whether or not the server is busy serving another packet. If it is not, the arriving packet is put into service and a service time is generated randomly. If it is, this packet is simply stored into the internal queue.

```
template < class DATATYPE >
bool FCFS<DATATYPE>::Arrive(const DATATYPE& packet,
                            simtime_t time, int)
{
    if (!m_busy)
    {
        in_service=packet;
        wait.Set(time + Exponential(service_time));
        m_busy=true;
    }
    else
        m_queue.push_back(packet);
    return true;
}
```

The *Depart* function is called when the timer *wait* becomes activated. It outputs the current packet in service, and then checks if there are any other packets waiting in the queue.

```
template < class DATATYPE >
bool FCFS <DATATYPE> :: Depart(const trigger&, simtime_t time, int)
{
    out.Write(in_service,time);
    if (m_queue.size()>0)
    {
        in_service=m_queue.front();
        m_queue.pop_front();
        wait.Set(time + Exponential(service_time));
    }
    else
        m_busy=false;
    return true;
}
```

3.3.4 Sink

In the *sink* component, we collect waiting times that each packet spent in the FCFS server. It only has one inport. The two private member variables are used to record the cumulative delay time and the number of packets received, respectively. *Start* is called when the simulation begins, and *Stop*, in which we print out the result, is called when the simulation reaches the preset ending time.

```
class Sink : public typeii
{
private:
    double m_total;
    int m_number;
public:
    inport< Packet > in;

    void Start()
    {
```

```
        m_total=0.0;
        m_number=0;
    }
    void Setup(const char* name)
    {
        typeii::Setup(name);
        in.Setup(this,&Sink::Arrive,"in");
    }
    void Stop()
    {
        printf("Average delay is: %f (%d packets) \n",
                m_total/m_number, m_number);
    }
private:
    bool Arrive(const Packet& packet, simtime_t time,int )
    {
        m_total+=timem_packet.arrival_time;
        m_number++;
        return true;
    }
};
```

3.3.5 Constructing the Simulation

The simulation class is derived from the *CostSystem* class. Components are instantiated as private member variables. Two public member variables are two simulation parameters that determine the arrival rate and the service rate.

```
class MM1 : public CostSystem
{
public:
    void Setup(const char*);
    double interval;
```

```
    double service_time;
private:
    Source source;
    FCFS <Packet> server;
    Sink sink;
};
```

The simulation has a *Setup* function too. It first maps component parameters to simulation parameters, and then invokes the *Setup* function of every component. After that, it connects pairs of inports and outports. Finally, the *Setup* function of the base class is invoked.

```
void MM1::Setup(const char*name)
{
    source.interval=interval;
    server.service_time=service_time;
    source.Setup("source");
    server.Setup("server");
    sink.Setup("sink");

    Connect(source.out,server.in);
    Connect(server.out,sink.in);
    CostSystem::Setup(name);
}
```

3.3.6 Running the Simulation

To run the M/M/1 simulation, first we need to instantiate an M/M/1 simulation object, and then choose the parameters. *StopTime* is a default parameter indicating the ending time of the simulation. The *Setup* function must be invoked before the *Run* function, which starts the simulation.

```
int main(int argc, char* argv[])
{
```

```
    MM1 mm1;
    mm1.interval=1;
    mm1.service_time=0.5;
    mm1.StopTime=1000000.0;
    mm1.Setup("mm1");
    mm1.Run();
    return 0;
}
```

3.4 Reusability in COST

COST has been used for other, far more complex, simulations, like queuing networks, computer networks and PCS simulations. These examples can be found at <http://www.cs.rpi.edu/˜cheng3/cost>. It is targeted at simulation modelers who have beginning or intermediate knowledge of the C++ language. Once they understand the basic component-port model and its supporting classes, it is fairly easy for them to write models with COST, and, more importantly, to take the component-based world-view to model the system to be simulated.

Although some simulators, like CSIM [58] , may simulate an M/M/1 system in perhaps tens of lines of code, COST does not necessarily imply longer code. First, we can see that a large portion of the COST code is quite mechanical and amenable to code generation. Second, COST components are highly reusable. For instance, the *FCFS* component built for the M/M/1 simulation can process any types of packets. Even the *Source* and the *Sink* components can be modified with few changes into template classes that can take any type of packets with a field *arrival_time*. Once a component repository with a wide variety of models has been developed, the modeler will be able to construct a simulation just by connecting components obtained from the component repository.

3.5 Conclusion

COST is a discrete event simulator that promotes a component-oriented modeling style. At the heart of COST is a component-port model, which is distinguished

from many existing component models by the notion of outports. The most distinct feature of COST is the component reusability. Components developed for one simulation can be effortlessly reused in other simulations. With an extensive set of library components, writing simulations in COST could be as simple as dragging a few components from the library and connecting them, as some commercial simulators do. Building COST components from scratch is also fairly easy.

The only inefficiency of COST simulations comes from the message exchanging between components, which may involve several layers of function calls and a few virtual function table lookups. However, this is rather the deficiency of the C++ language, not of the underlying component-port model, because theoretically such overhead can be eliminated during the configuration phase. Had we had a truly component-oriented language, COST would have achieved perfect efficiency.

CHAPTER 4
COMPONENT-ORIENTED SIMULATION ARCHITECTURE: TOWARD INTEROPERABILITY AND INTERCHANGEABILITY

In this chapter we first investigate two aspects of the simulation reusability: interoperability and interchangeability. Their implications on the simulation technology are discussed. Based on our previous work on component-based simulation, the Component-ORiented Simulation Architecture (CORSA) is devised to address both interoperability and interchangeability. The ideas and considerations which led to an implementation of a prototype are presented. A sequential PCS simulation has been developed using CORSA. This exercise demonstrated several advantages of the component-based approach: flexibility, extensibility, and reusability. Experimental results showed that the component-based approach is only slightly slower than the monolithic approach, whose complexity quickly grows to nearly insurmountable proportions with the growth of complexity of the simulated system.

4.1 Interoperability and Interchangeability

The importance of interoperability between simulators has been gradually realized and understood. Interoperability enables the system designer to reuse existing simulations, and/or to combine them with new ones to form large and complex simulations which seemed impossible to design even a few years ago.

The High Level Architecture (HLA)[38] is the first extensive effort to attack the problem of simulation interoperability. The publishing and subscription scheme allows all objects in an HLA-compliant simulation to be reused by other simulations.

However, simulation interoperability only represents one level of reuse. Another level of reusability has often been ignored: the interchangeability of simulation models. A simulation model is a part of a simulation. It describes the dynamic behavior of a component of the system being simulated. It differs from a simulation

in that it is not executable. For instance, an event list usually cannot be found in a simulation model. Since the real-world component described by a simulation model may exist in different systems, it is expected that the same simulation model should appear in many simulations.

This, unfortunately, is not the case at all. The best that we can achieve nowadays is the interchangeability among simulations built with the same simulator. When facing the problem of simulating a real system, the first thing coming to mind is to select a simulator that seems to be best suitable for the particular application. After making a choice, the designers start building simulations according to the standard defined by the simulator. Selecting the simulator is always difficult, and the choice more often depends on the experience of the system designer than on the features of the simulator. In the worst case, the selected simulator may be later found unsuitable for the particular application because of some factors unforeseen at the design stage. Switching the designed system to another simulator usually means that all models developed so far have to be rewritten.

Our approach tries to address both interoperability and interchangeability in a balanced way. This is made possible by the component-oriented world view and the simulation component classification proposed in the previous chapter.

4.2 Simulation Platform

Historically, operating systems, the platforms on which traditional simulations are executed, are all component-oriented. A complete operating system consists of many software packages, which can be viewed as components providing a wide variety of functionalities. Users can install packages that they need and remove those they do not. An operating system often allows a simple but efficient form of interoperability between programs. A program may invoke other programs, in the form of a shared library [42], or even in a more sophisticated model like Microsoft COM [5]. Model level reuse is made possible by efforts like STL [6, 49].

However, current operating systems do not provide sufficient support for the configuration phase required by the component-oriented world view. For example, although it is possible to parameterize a program by passing command line argu-

ments, the parameters represented by these arguments are program-specific, thus parameterization cannot be done in a unified way. Binding a function call to the address of the function resembles linking an outport with an inport, but multiple connections to one outport cannot be defined.

To overcome these difficulties, we propose a simulation platform, referred to as CORSA (Component-ORiented Simulation Architecture), based on the component-oriented world view. The component classification described in the previous chapter plays an important role in the design and implementation of this simulation platform. Since we are only concerned with sequential simulations at the prototype stage, port classification is not being considered for now.

CORSA defines a component development standard which serves as a contract between the developers of simulation engines and the developers of components. It enables a scenario in which once a CORSA-compliant simulation engine has been designed to link a certain type of components, it will be able to accept any components of this type. The term "simulator" will be synonymous with the term "simulation engine". Non-compliant simulators not designed within the CORSA domain can be wrapped and treated as Type III components but they will not be able to use standard components.

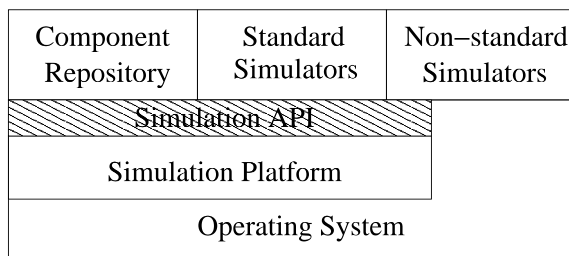

Figure 4.1: The CORSA Simulation Platform

Figure 4.1 shows the architecture of the CORSA simulation platform. Built on top of the operating system, it provides supports for standard simulators. Non-standard simulators are wrapped with interfaces which allow them to be plugged into the simulation platform. The simulation platform also contains a component repository to facilitate management of the components. The key part of the development will be a simulation API (Application Programming Interface). Besides

common functionalities required by simulation programming, like random number generators, priority queues, message passing, etc., it must allow the simulators to access the information of components and their ports, timers, and clocks.

4.3 Design of A CORSA Prototype

We have developed a prototype of the proposed simulation platform. The main purpose of this prototype is to demonstrate that both interoperability and interchangeability are achievable at the same time. One important part that is missing is the simulation API. Design of this API is by no means a trivial task: it must be based on the knowledge and experience gained from continuous efforts in the component-based simulation.

4.3.1 Two Levels of Reuse

As mentioned in the first section, interoperability enables simulation-level reuse and interchangeability enables model-level reuse. These two kinds of reuses can be achieved in two ways, depending on the format and class of components involved. Normally, Type III components are in the form of either binary libraries or source code, but Type I and Type II components exist only as source code modules. Correspondingly, two types of simulation engines are usually available: one type accepts binary libraries and the other accepts source code modules. Type I or Type II components can be compiled into binary libraries. This kind of compilation is useful when a Type I or Type II component is to be linked to Type III binary components. For example, a delay between two Type III components can be modeled as a component of Type II, therefore a delay model can be retrieved from the component repository and then compiled into a binary component.

Binary components are actually implemented as shared libraries. This is a natural choice because it allows the simulation platform to load components during runtime. A predefined component creation function is required to be implemented by every binary component. It will create a component instance upon invocation after the component library is loaded into the computer memory for execution. An existing simulation can be easily wrapped with a few functions to become a shared

library.

4.3.2 Component Interface Description Language

A CIDL (Component Interface Description Language) is defined to describe the interface of components. A CIDL file describes the name and type of every port, timer or clock in a component. This way of describing interface is similar to the CORBA IDL. However, CORBA IDL cannot be directly used to describe CORSA components, because the primitive elements are different. Moreover, CORBA only deals with Type I components according to our classification. Examples of CIDL interfaces can be found in later sections.

We have also implemented a CIDL compiler to facilitate the development of components. The CIDL compiler translates a CIDL interface into a skeleton from which the component implementation can be derived. Currently only a CIDL to C++ mapping is supported.

4.4 Implementation Issues

In this section, we introduce the class hierarchy of components used in the simulation platform prototype. The use of function objects is also described.

4.4.1 Class Hierarchy of Components

Figure 4.2 shows the class hierarchy of the prototype of the CORSA simulation platform. A *Component* class is defined as the base class of all types. It contains public member functions for parameterization and interconnection required by the component standard. In addition, it defines a *SimTime* function which returns the current simulation time.

The Type I component class is a trivially derived class of the *Component* class. It does not introduce any new member functions.

The Type II component class defines two new member functions: *SetTimer* and *CancelTimer*. The *SetTimer* function sets a timer with a specified delay. As soon as the elapsed time from the instant at which the timer was set is equal to the delay, this timer will become active. *CancelTimer* simply disables a timer.

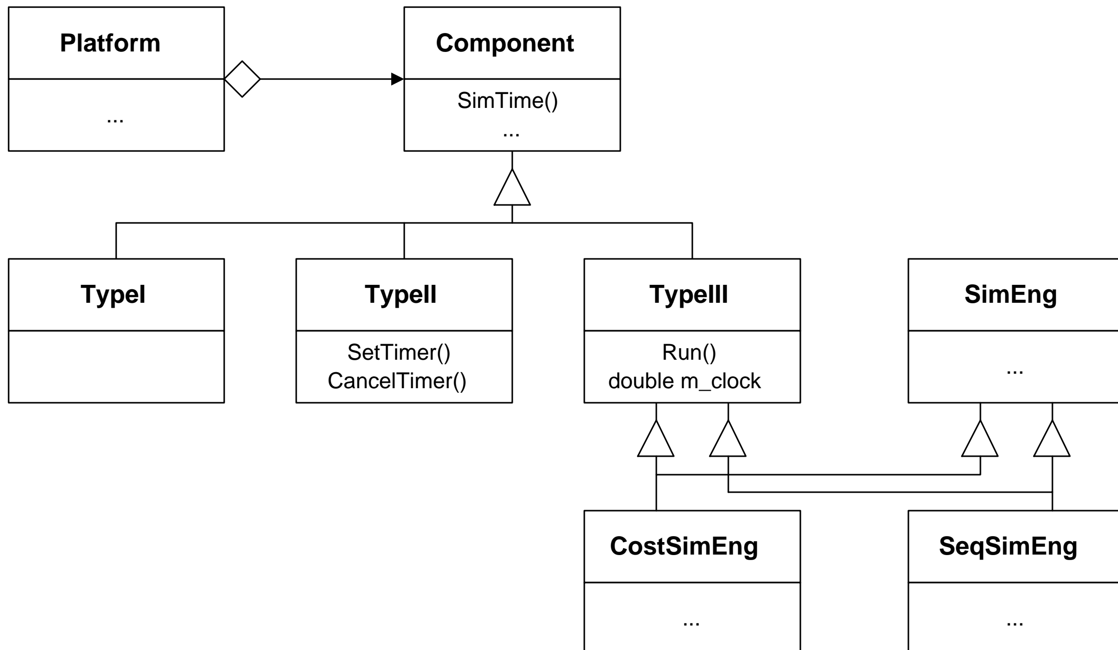

Figure 4.2: The Class Hierarchy for the CORSA Simulation Platform

Type III components are provided with a new member function *Run* because they can manage the simulation clock themselves so that they are runnable. The member variable *m_clock* contains the value of the current simulation time.

An abstract *SimEng* class is also defined in the class hierarchy. It serves as the interface for all simulation engine classes, i. e., every simulation engine class should implement the pure virtual functions declared in the *SimEng* class.

Because a simulation engine is also a component, either of Type II or of Type III, a simulation engine class is constructed using multiple-inheritance. A simulation engine that is of Type II permits hierarchical modeling methodology. We have developed two Type III sequential simulation engines: *SeqSimEng* accepts binary components and *CostSimEng* accepts source code components, both of which are modified from the COST simulator to fit into the CORSA architecture.

4.4.2 Functor

As in COST, functors are widely used in the implementations of the ports and timers in CORSA. Conceptually, it is a generalization of the function pointer [6, 49]. In C++, functors are constructed using operator overloading. Figure 4.3 shows the

class hierarchy of the function objects.

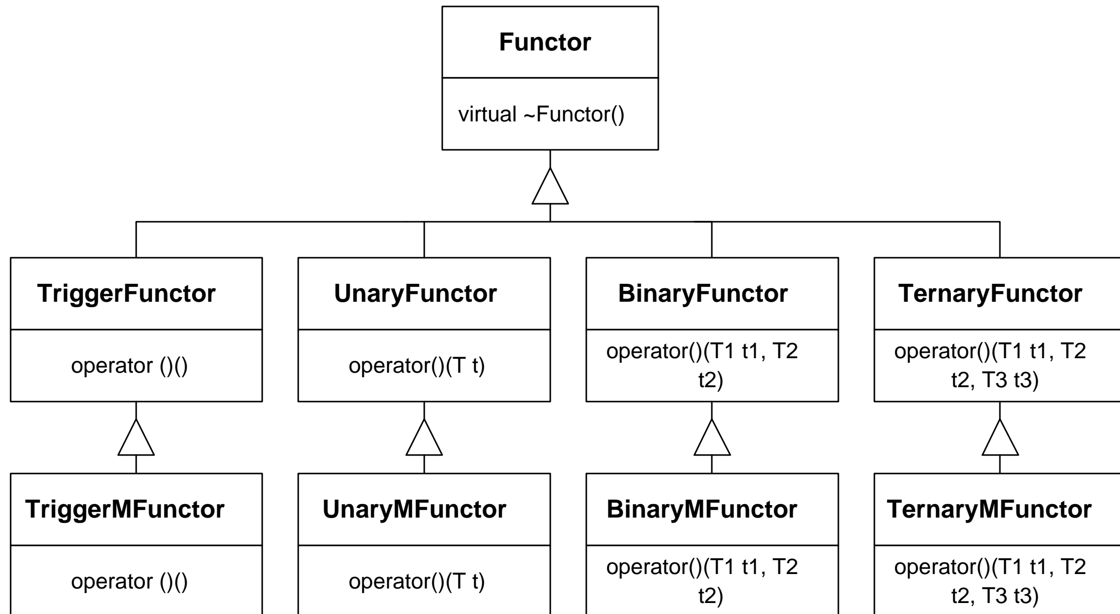

Figure 4.3: Functor Objects Used in the Implementation

A base *Functor* class is defined first. A set of functor classes are then derived, each of which takes a different number of arguments. Finally, a set of member functor classes are derived from the corresponding non-member functor classes.

The advantage of using functors is that type checking for ports becomes possible. For instance, when we connect an inport with an outport we need to make sure that they are matched. That is, both of them should take the same number of argument, and arguments in the same position should be of the same type. Without the use of functors, type checking of two arbitrary ports is impossible because two ports may reside in two shared libraries compiled separately, thus the compiler cannot detect the type mismatch. With functors, the pointer to a function object is first upcast to a pointer to a *Functor* object. This pointer is then passed to the port in another shared library. To decode this pointer to its original type, the C++ operator *dynamic_cast* is used. Decoding the pointer to any other type will result in a null pointer, which can be detected by the component at runtime.

4.5 PCS Simulation Using CORSA

A PCS (Personal Communication Service) network [14, 11] is composed of a number of geographically distributed radio base stations. The PCS network simulation involves two object types: *cell* and *portable*. The portables, carried by users in the coverage area (or cell) of a base station can use the channels assigned to that station. The number of channels allocated to each cell may be smaller than the number of portables simultaneously in the cell, so whoever makes the request first, acquires the available channels. A channel can be occupied by a portable for a random call time and then released. A block occurs when a portable requests a channel while all channels in the cell have been allocated to other portables. When a portable moves from one cell to another during a phone call a hand-off is said to occur. In this case the portable releases the occupied channel to the old cell and then attempts to get a channel from the new cell.

4.5.1 Two Reusable Components

In this PCS simulation experiment, we adopt a partition scheme different from those normally used for parallel simulation [14, 11]. A *cell* component simulates all the cells, that is, it processes all the requests of channel allocation and releases. A *portable* component simulates all the portables. This partition scheme is difficult to be parallelized, but is well-suited for the purpose of demonstrating the advantages of the component-based simulation and of estimating the overhead of inter-component communication.

The interface of the *cell* component, which is a Type I component, is given below. The parameters *width* and *height* define the number of cells in the horizontal and vertical direction, respectively. The parameter *channels_per_cell* specifies the number of available channels that each cell initially possesses. The inport *GetChannel* is activated when a portable needs to make a phone call. The *cell* component then checks if there is any channel currently available in the cell specified by the argument *index*. The availability is returned by the argument *available* that is passed by reference. The inport *ReleaseChannel* is activated when a portable releases a channel that it has been using.

```
component cell
{
    param int width,height,channels_per_cell;
    inport GetChannel(int index, bool& available);
    inport ReleaseChannel(int index);
}
```

The *portable* component is a Type II component. The parameter *portables_per_cell* states the number of portables in each cell at the beginning of the simulation. The parameters *next_call_mean* and *next_move_mean* define the average time between two consecutive calls, and two consecutive moves, respectively. The parameter *call_time_mean* indicates the average duration time of a call. All time durations are drawn from exponential distributions. Two outports, *GetChannel* and *ReleaseChannel*, are used to request and release channels. The *portable* component also contains a timer, which is set with the smallest timestamp of all future events occurring to this portable component. When this timer is activated, the *portable* component knows that it is its turn to process the next event.

```
component portable
{
    param int width,height,portables_per_cell;
    param double next_call_mean,next_move_mean,call_time_mean;
    outport GetChannel(int index, bool& available);
    outport ReleaseChannel(int index);
    timer next;
}
```

There are two advantages to modeling PCS networks in this way. First, this model separates the cell component from the portable component, making it fairly easy to modify the cell component when a different channel allocation policy is adopted. Second, the simulation constructed in this way is fully extensible. For example, some of the portables may be carried by moving vehicles, and we want to

simulate the behavior of vehicles as well. We can simply create a new *vehicle* component, and link it with the cell component. The resulting three components (cell, portable, and vehicle) will work correctly with each other without any additional considerations for their synchronization!

4.5.2 Experimental Results

A series of experiments have been conducted to test the performance of the component-based approach. Figure 4.4 shows the simulation speeds in these experiments. Two simulation engines mentioned earlier were tested. For each simulation engine, two experiments were performed. First, the simulation was constructed using one cell component and one portable component. Second, one cell component and two portable components were used, with each portable component simulating half of the workload. The number of events processed in one simulation run remains the same.

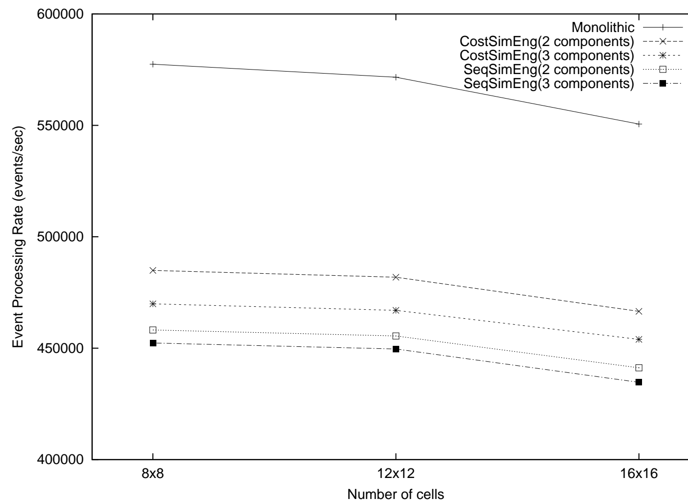

Figure 4.4: Efficiency of Different Linking Approaches

It is no surprise that the monolithic approach is the fastest. In general, the component-based approach is approximately 20% slower than the monolithic approach for the PCS simulation. It is worth mentioning that the PCS model is fine-grained: the amount of computation required to process each event is relatively small. Hence, the overhead of inter-component communication becomes more significant. For coarse-granularity models, the inter-component communication would

impact the overhead and overall performance to a lesser degree.

We can also see that the simulation engine *CostSimEng* is faster than *SeqSimEng*. This is explainable because the former integrates source code components so it is more tightly-coupled. Dividing the workload into two portable components only slightly slows down the execution speed.

4.6 Conclusion

The Component-ORiented Simulation Architecture, CORSA, presented in this chapter, supports both interoperability and interchangeability. We have built a CORSA prototype and conducted a set of PCS network simulations consisting of two components. The same two components that simulate cells and portables can be used in two different simulation engines. Experimental results show that the component-based approach incurs insignificant inter-component communication overhead, while supporting reusability, composability and extensibility.

However, the prototype that we presented here is far from being the full-fledged simulation platform that we have envisioned. The design of a key part of the simulation platform, the simulation API, has been left out in this prototype. The component development standard, which serves as a contract between the simulator developers and the model builders, is still informal and subject to future changes.

CHAPTER 5
LOOKBACK: LOOKING INTO THE PAST IN SEARCH FOR PARALLELISM

The previous three chapters are mainly concerned with designing new modeling methodologies that simplify and accelerate development of simulations. From now on, we turn to the other aspect of the simulation technology, which is to speed up discrete event simulations by exploiting parallelism on multiprocessors.

5.1 Introduction

Researchers have long realized that Parallel Discrete Event Simulation (PDES) is an effective approach to simulating large-scale complex systems. Research on PDES has been going on for more than 20 years. The main difficulty in this area is to achieve high efficiency of parallel execution while preserving the causality order between concurrent events on different processors. The logical process paradigm, widely used in the PDES community, assures that no causality errors will occur if each component adheres to the local causality constraint [28], i.e., if each component executes its events in non-decreasing timestamp order. Therefore, to preserve the causality order, it is sufficient, though not always necessary, that each component finds and executes the future event with the smallest timestamp.

Research on PDES has been largely dominated by the studies of conservative [12, 16] and optimistic protocols [35], and of their performance. Unfortunately, both types of protocols have their strengths and weaknesses. Efficiency of conservative protocols in parallel execution is limited by the amount of lookahead in individual components, which is equal to the difference between its Earliest Output Time (EOT) and its current simulated time. EOTs are known exactly only during run-time, and in many real-world applications deriving bounds for this difference, and consequently lookahead, is difficult. Additionally, the null messages required carrying the EOT values which are used to collaboratively advance the simulation clock in conservative protocols often incur significant overhead.

Optimistic protocols appear to be superior to conservative protocols since they do not depend on lookahead and null messages, and thus can be applied to more applications. However, state saving usually requires storing and accessing large amounts of memory. This disadvantage becomes more serious if we consider the relatively slow improvement in the memory access speed within the current VLSI technology. The handling of anti-messages also complicates the simulation model development. As Nicol and Liu have pointed out [50], optimistic models may exhibit unexpected behavior caused by inconsistent messages resulting from rollback inconsistencies and stale states.

We show that a quantity named *lookback* can provide yet another source of parallelism both for conservative and optimistic protocols [17]. Exploitation of lookback can reduce the number of rollbacks in optimistic simulations, because a component may, in some cases, process an out-of-timestamp order event without rolling back other events. Lookback also supports a new classes of protocols, referred to as *lookback-based protocols*, some of which may no longer depend on lookahead. The most important theoretical consequence is that the execution time of lookback-based protocols may be lower than the bound given by the critical times of events, which is an insurmountable limit for lookahead-based conservative protocols and optimistic protocols without optimization.

5.2 Lookback-Based Protocol

Informally, lookback is defined as the ability of a component to execute out-of-timestamp order events, which are often referred to as *stragglers*, without impacting states of other components. We first consider semantics of the simulated time, then give the formal definition of lookback and other related terms.

Let's consider an instant, t, of a simulation execution. Assume also that at this execution time, the simulation is at the simulated time $T(t)$ that is defined by the timestamp of the currently or most recently executed event. In sequential and conservative parallel simulations, the time T is monotonic. Still, in general, if events with the same timestamp are not excluded, the simulated time T does not uniquely define the state of the simulation. Yet, because we deal with discrete

event simulation, the simulation state changes discretely, so the execution time is too detailed an index of the state of the simulated system. Hence, we propose the following convention.

The extended simulated time, or e-time in short, denoted by τ, is a pair (T, n), where T is the simulated time (equal to the timestamp of the last event fully executed), and n is a counter of the number of extended events that the simulation execution has processed with a timestamp smaller or equal to T. Extended events include all simulation events and global execution events, such as computation of the lower bound of future timestamp or reception of a message. By definition, the e-time τ uniquely defines the state of a simulation.

Two orders can be defined on e-times:

- Execution order: $\tau_1 = (T_1, n_1) >_e \tau_2 = (T_2, n_2)$ iff $n_1 > n_2$ or $n_1 = n_2 \& T_1 > T_2$. By definition of e-time,
- Simulated time order : $\tau_1 = (T_1, n_1) >_s \tau_2 = (T_2, n_2)$ iff $T_1 > T_2$

In the following context when two e-times are being compared, the execution order is used by default. If the simulated time order is to be taken, the notation of simulated time will be used, such as T, T_1, T_2.

Now we can introduce two auxiliary functions, *pred* and *maxT*, defined as follows. We can define $pred(\tau)$ as $\tau' < \tau$ such that for every $\tau'' < \tau$ we have $\tau'' \leq \tau'$, or more formally:

$$pred(\tau) = max(\{\tau' : \tau' < \tau\})$$

The function *maxT* returns the maximum simulated time that has been advanced to between two e-times:

$$maxT(\tau_1 = (T_1, n_1), \tau_2 = (T_2, n_2)) = \begin{cases} T_1, \text{ if } \tau_2 \leq \tau_1 \\ max(maxT(\tau_1, pred(\tau_2)), T_2), \text{ otherwise} \end{cases}$$

Similarly as in distributed systems, we will consider for each component local and global (externally observable) events. In our simulation, the only global events

are inter-component messages. Hence, lookback and other related terms can be briefly defined as:

Definition 5.1. • *Consider a component at e-time $\tau = (T, n)$. If this component can execute correctly, without changing any external events that it has produced, any number of stragglers with timestamp between $T - L$ and T, at any e-time $\tau_1 > \tau$ and such that $maxT(\tau, \tau_1) \leq T$, then the time window $[T - L, T]$ is said to be the* lookback window *of the component at time T.*

- *The lower end of the lookback window, $T - L$, is referred to as the* virtual lookback time, *or* vlt.

- *The procedure used to process events falling into the lookback window is called the* lookback procedure.

- Lookback *is defined as the size of the lookback window.*

- *Finally, for any event e that is to be processed at the given simulated time T, $LB(e, \tau)$ denotes the function that gives the value that the virtual lookback time would have if the event e were executed at e-time τ.*

As an example, let us assume that the current e-time of a component is $\tau = (15, 5)$ when a straggler with a timestamp of 10 comes. The new e-time after the straggler has been processed by the lookback procedure is $\tau_1 = (10, 6)$. According to the definition of the $maxT$ function, $maxT(\tau, \tau_1)$ is 15, since 15 is the maximum simulated time the component has reached. Therefore, the lookback at τ is at least $15 - 10 = 5$.

The $LB(e, \tau)$ function for an event e at τ has an interesting property given by the following theorem.

Theorem 5.1.

$$LB(e, \tau) \leq ts(e, \tau)$$

where $ts(e, \tau)$ denotes the timestamp of event e.

Proof. Suppose at e-time τ the event e is a straggler and $LB(e, \tau) > ts(e, \tau)$. After e has been processed, the vlt would be $LB(e, \tau)$ and the e-time would be τ', according

to the definition. Now consider another straggler e'. If $ts(e', \tau') \geq ts(e, \tau)$, it is possible that $ts(e', \tau') < LB(e, \tau) = vlt$, which implies that the second straggler e' cannot be correctly processed by the lookback procedure, contradicting the definition of lookback. Therefore by contradiction $LB(e, \tau) \leq ts(e, \tau)$ for any e. Both e and e' can now be processed if the one with smaller timestamp is processed first. □

To avoid any causality errors or anti-messages, each component must maintain a virtual lookback time that is less than or equal to the timestamp of any future event. Under such circumstances, any received event can be successfully processed by either the regular event handler or the lookback procedure. Therefore, we define the lookback constraint as follows:

Definition 5.2. *A component obeys the* lookback constraint *if and only if after processing each event, the virtual lookback time is smaller or equal to the minimum timestamp of all unprocessed events of this component (including those events that will arrive at this component later). The minimum timestamp of all unprocessed events at wall time τ is denoted as $LBTS(\tau)$ (Lower Bound on TimeStamp). Expressed mathematically, the lookback constraint is*

$$LB(e, \tau) \leq LBTS(\tau)$$

for every event e to be processed.

The lookback constraint is basically a relaxation of the local causality constraint given by Fujimoto [1]. Adherence to the lookback constraint naturally leads to a new kind of synchronization protocol for PDES. We give the description of the protocol in its most general form below.

Let $U(\tau)$ be the set of unprocessed events on the given component at e-time τ, and $S(\tau)$ be a subset of $U(\tau)$ defined by the designer of the protocol. Finally, let $E(\tau)$ be a subset of the set of all events in $S(\tau)$ that satisfy the lookback constraint at e-time TT. Note that subset $S(\tau)$ is introduced to limit the events for which the lookback constraint is checked, for example subset $S(\tau)$ can be defined as all events that have the smallest timestamp in $U(\tau)$. Likewise, $E(\tau)$ is defined as a subset because selection of the first eligible event in $S(\tau)$ may lead to the most efficient

implementation. With this notation, the lookback protocol can be described as follows:

```
while (the simulation termination condition is not met)
    construct E by checking the lookback constraint
        for all events in S;
    while (E is nonempty and the stop condition is not met)
        remove and process an event from E;
    end while
    wait until there are received messages and process them;
    recompute LBTS;
end while
```

This general lookback-based protocol first constructs a subset E consisting of events all of which are eligible for execution, therefore E is called the *eligible event set*. It then repeatedly removes and executes every event in E, until the stop condition is true. The stop condition can be avoided if E is so chosen that execution of an event in the set will not affect the lookback of any other events in the set.

Theoretically, using all unprocessed eligible events and the exact value of $LBTS$ makes the protocol deadlock free. However, the efficiency of creating the eligible event set E is important for overall efficiency of simulation because this operation is frequently invoked. $LBTS$ is also costly to evaluate exactly, and often only the estimation of its lower bound, can be maintained throughout the simulation. For practical reasons we consider variants of this general protocol in which only a single event is in set E. One way to do it is to select only the event with the smallest timestamp among all events currently in the local event list. Such a solution reduces the amount of processing since the function $LB(e, \tau)$ is only invoked on the earliest event.

For completeness we should point out that there is another possibility that the local event list can be sorted by the virtual lookback time of events and the event with the smallest virtual lookback time can be selected as the only member of S. Maintaining the local event list in this way will work well for simulations whose

virtual lookback time is independent of the simulated time, where the earliest event may not be eligible for execution, because this event may not be the globally earliest one, while the event with the smallest virtual lookback time is more likely to be executable.

5.3 LB-GVT and LB-EIT

We have implemented two variants of the generic lookback-based protocol, in which the earliest event is selected as the only member of E. The algorithm is given below. These two variants only differ in the estimations of the LBTS: one is using GVT (Global Virtual Time) and the other is based on EIT (Earliest Input Time). We abbreviate them LB-GVT and LB-EIT, respectively.

```
while (the termination condition is not met)
    locate the earliest local event e;
    if LB(e,T) <= GVT(or EIT)
        remove e from the local event list and process it;
    else
        recompute GVT(or EIT) and/or receive messages;
    end if
end while
```

If no event is eligible for execution, the recomputation of GVT (or EIT) may provide a more accurate value of LBTS, or other events previously in transit may be received by a component and then become eligible for execution.

The LB-GVT protocol uses the GVT as an estimate of the LBTS for each component. However, the GVT would be equal to the LBTS only in those components that are destinations of the earliest event(s). In other components, the GVT is merely a rough estimate of the LBTS. In this sense, the interconnection topology between components is not taken into consideration by the GVT computation. Still, we can prove that LB-GVT is deadlock-free if the GVT computation meets the following requirement.

Theorem 5.2. *LB-GVT is deadlock-free, if GVT, when updated, is equal to the smallest timestamp of all unprocessed events in the simulation.*

Proof. We will show that the event with the smallest timestamp in the entire simulation is always eligible to be processed. There might be multiple earliest events with the same timestamp. For simplicity denote an arbitrary one as e. If this event e is still in transit, it will eventually be received by the destination component because the event queue will be scanned when there are no eligible events. After having being received, the event e will become the earliest local event and be chosen by the component to see if it is eligible for execution. Assume it is not, in which case the GVT computation will be invoked, and by the condition given in the theorem $GVT = ts(e, \tau)$. Since $LB(e, \tau) \leq ts(e, \tau)$ by Theorem 5.1, we have

$$LB(e, \tau) \leq ts(e, \tau) = GVT$$

implying that the event e is always safe to be processed without violating the lookback constraint. ☐

On the other hand, the LB-EIT protocol requires that each component maintain its own estimation of the EIT based on all neighbor's EOTs, thus helping to improve the accuracy of the estimation. Similar to the Null Message protocol, however, LB-EIT is prone to deadlock, when a cycle of components is formed where every component assumes the earliest event will come from others.

Let us look at a simple example to see how deadlock occurs in the LB-EIT protocol. As shown in Figure 5.1, there are two components, A and B, exchanging events with each other. Their current simulated times are 4 and 5, respectively. For simplicity, assume a constant lookback of 3 for both components. Each component also has other future events in the event list, with much larger timestamps. If there is no lookahead in both components, their EOTs are 4 and 5. The EIT of A is 5, and the EIT of B is 4. The EIT of 5 in component A prevents the future event at time 9 from being processed, because the execution of the event at time 9 would change the virtual lookback time to 6, but component A conservatively assumes that component B may send it an event with timestamp less than 6. For component A

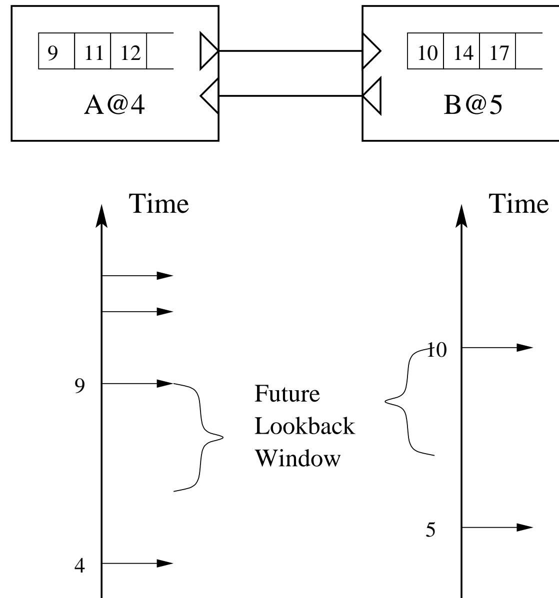

Figure 5.1: Deadlock in the LB-EIT protocol

to be able to process this event, component B must inform component A about its progress beyond the simulated time 6 to enable component A to update its EIT. But component B cannot do so since it has no event to process either (event execution may change the EIT in other components). A dependency cycle is formed and the simulation cannot progress.

This situation is exactly the same as what happens with the null message protocol when no knowledge of lookahead is utilized. A cycle of components, in this case components A and B, whose EITs depend on each other, contains no events in their lookback windows, while no future events in their event lists can be processed. There is no global mechanism to inform them of this deadlock situation. Consequently, every component is assuming that the earliest event will come from other component.

We conclude that the LB-EIT protocol still requires the knowledge of lookahead to break up potential deadlocks. In the example given above, suppose each component has lookahead of 2. Now the EOT of B becomes 7, which is also the EIT of A. The event at time 9 becomes eligible for execution. The deadlock can be broken even if the lookahead is only 1. By several rounds of sending and receiving null messages, these two components can gradually increase their EITs so that the

events at 9 and 10 eventually become eligible for processing.

5.3.1 LB-EIT Protocol with Deadlock Recovery

We have also devised a general deadlock resolution algorithm for the LB-EIT protocol that does not rely on lookback or lookahead. We start by listing two assumptions that we made in designing the algorithm.

1. Connections between components are First-In-First-Out reliable channels, so any new message flushes all preceding messages in the same channel.

2. Components poll periodically channels connecting them to their predecessors (all components that produce messages whose destination is the component in question). Hence, a component knows its predecessors and successors (i.e., components to which this component is a predecessor) and collectively the predecessor relation defines a dependence graph. The necessary condition for any deadlock to occur is that there is a cycle in this dependence graph.

Since many processors may start the algorithm simultaneously, messages from different instances of the algorithm need to be disambiguated. This can be easily accomplished by adding the id of the initiating node in all messages, labels and marks produced by the algorithm. For brevity, we omit the detailed description how this could be achieved.

Let s denote the node initiating the following algorithm.

1 Construct a spanning tree rooted at node s:

1.1 Initialize queue of unprocessed nodes Q with node s in it and set $label(s) = 0$.

1.2 for each node n in Q do until Q is empty:

1.2.1 for each edge (n, m) send $label(n)$ to m

1.2.2 read message (M_i, T_i) from all k successor nodes and set $p = \sum M_i$. If $p = 0$ then (node n is a leaf) select the only incoming edge (m, n) marked 1 and send the time of next local event t_n to node m, then read the value of global minimum time t_g from m

1.3 for each node m in the graph, if a message with a label from the predecessor n is received: If m is unlabeled, then set $label(m) = 1 + received_label$ and set $mark(n, m) = 1$, else set $label(m) = 0$ and send $(0, 0)$ message to n.

2 Compute minimum time for all events in the subsystem: The construction (step 1) stops at leaves of the spanning tree (step 1.2.2 with $p = 0$), and the components embark upon computing the minimum next event:

2.1 for each node n that finished reading started in 1.2.2 (with $p > 0$) select the smaller of $mint_i; 0 < i < p + 1$ and t_n, the next local event time. If node n is different than s, then select the only edge (m, n) marked 1 and send the computed value to node m, then read the value of global minimum time t_g from m.

3 Send the minimum time to all participating nodes. When the starting node s computes the minimum, the step 2 stops and the node s can send the time of such established next event to all nodes reachable from s (using edges marked 1).

3.1 for each edge (s, m) marked 1, send the value t_g, computed in node s, to m.

3.2 for each node n that received t_g and for each edge (n, m) marked 1 send the value t_g to node m, then compare it with the next local event time, and if they are equal, execute the event.

The computed minimum includes also future event times at nodes that may not be in a cycle with node s. To eliminate such nodes, the inverse spanning tree rooted in s can be established by applying step 1 to reversed edges. Only the nodes inversely reachable would then contribute their times to the minimum (others may report the total simulation time T_{sim}, for example). Yet, such an algorithm would be less efficient, because it would require effectively running the first two steps of the algorithm twice (the step 2 is needed to make sure that the step 1 on reverse edges

completed execution at all nodes). Rarely the nodes dependent on s will not be in a cycle with s, so rarely this modification would be beneficial. Moreover, if indeed the minimum time found by the algorithm above will not resolve the deadlock (so the found time is the event outside the deadlock cycle), just repeating the algorithm will yield a new value.

Clearly each edge is inspected only three times (first forward to send the node label to the successor, then backward to send the mark and the locally minimum time to the predecessor and finally to send the global minimum to the successor), so the number of messages and steps is less than three times the number of edges. Moreover, the algorithm proceeds in parallel on all processors assigned to components. As a result, parallel complexity is three times the length of the longest unique path originating in node s.

Every node reachable from node s is labeled and a unique path is built with edges marked 1 to each, so these edges form a spanning tree of the dependence graph. Consequently, each node belongs to a path of edges marked 1 originating from s, and therefore its next local event time will be considered for the minimum next event time. Finally, sending messages along each edge flushes all messages in the edge, so the local queues on each node contain all known future events in the subsystem created by the dependence graph. If there is an event with timestamp smaller than the selected minimum, this event will not be sent to the nodes in the spanning tree, so it will not cause a rollback. Finally, synchronization is implicit. Only when all outgoing edges marked 1 reported results, the node sends its minimum computation further.

It should be noted that once a participating process receives an event that enables it to execute, it may send the time of this event as its t_n and the algorithm will terminate correctly. Similarly, by always selecting to continue the algorithm initiated by the processor with the smallest id, only one of many initiated algorithms will continue, if special termination markers are chosen and sent when needed. To avoid deadlock in such case, reading of the global minimum time in steps 1.2.2 and 1.3 needs to be done together with the polling of the predecessor messages, so the new events can be received by every participating node.

5.4 A Retrospective View

The most perfect lookback might be that of linear systems, whose output can be described as a linear function of input. We demonstrated that the superposition property of a linear system can be used to process a straggler without a rollback [18]. Instead, a new copy of the current system is initialized with all state variables set to zero. This newly created copy is run with the straggler as the input from the timestamp of the straggler to the current simulated time. Finally, the correct state information is created by summing up the state variables in the original system with those in the new copy. The interesting fact is that we are able to correct the values of the state variables at the current simulated time without ever recreating their values at the timestamp of the straggler.

After the invention of conservative and optimistic protocols, a major part of research in PDES was aimed at improving their performance by a wide variety of optimization techniques. From a historical point of view, the idea of lookback unifies two trends in the PDES research.

5.4.1 Out-of-Timestamp Order Execution

In the PDES community, it has been noticed by many researchers that in some special circumstances events can be processed in out-of-timestamp order, for instance, when events are independent of each other.

Query events, proposed by Sokol [61], represent the simplest form of independent events. They do not change the internal state of the component, and therefore can be safely processed even if the local simulated time is larger than their timestamp (i.e., when they are stragglers), by simply reading the state history, instead of rolling back the entire component. Other stragglers, which Sokol called side-effecting events, require rollbacks.

Rollback relaxation is a similar technique [66]. Logical processes can be classified into memoried processes and memoryless processes. In the former, the output messages are computed as a function of both input messages and internal state variables, while in the later only the input messages are used. Consequently, in memoryless processes rollback relaxation is able to process stragglers without state-

saving.

Weak Causality [56] is a formal notion defining the event dependency which simplifies the detection of independent events. Traditionally, causality is defined in terms of the happened-before relation. The Weak Causality relation is based on the conflicts created by events that operate on the same data, rather than on the timestamps of events.

Leong and Agrawal [41] have proposed a semantics-based optimistic protocol that exploits the semantics of messages to avoid certain types of rollback. For example, the *commutativity* relation determines if two events can be executed out of order, while the *invalidated-by* relation determines whether a processed event should be rolled back due to the arrival of another event with a smaller timestamp. However, even for sets, the data structure for which relations between different types of message were discussed in their paper, this protocol is too complex. Their work cannot be easily extended to other general simulation models.

Most previous attempts trying to exploit the independence between events have one thing in common: upon arrival of a straggler, they first check, by some means, whether the straggler can be safely handled. If it can, the *same* procedure that processes regular messages is invoked. Otherwise, a rollback-recovery procedure is invoked to recover from the causality error. The event independence is used only to reduce the number of rollbacks.

Lookback-based protocols handle stragglers in a different way. The component first determines whether a straggler can be safely processed according to the notion of lookback. A *different* procedure, the lookback procedure, is invoked to process stragglers whose timestamp is within the lookback window. The lookback procedure processes stragglers in such a way that the resulting state of the component is always causally correct. Not only does our approach reduce the number of rollbacks in optimistic protocols, it may also completely eliminate rollbacks under the lookback constraint which guarantees that all stragglers will be correctly processed by the lookback procedure.

5.4.2 Local Rollback

The efficiency of handling stragglers by the lookback procedure depends largely on the complexity of repairing the damages caused by out-of-timestamp order execution. It is intuitively clear that any changes made to the local state of a component can be recovered by the component by some means, for instance, state-saving. However, sent messages can only be cancelled by other components. Hence, the largest possible lookback arises if every processed event did not send out messages that could be subject to any changes made by any stragglers. The following definition specifies a universal lookback procedure that is able to correctly process stragglers within such a largest lookback window.

Definition 5.3. *A* universal lookback procedure *requires every component to save a list of processed events and every change made to the state by each event. Upon arrival of a straggler, it rolls back all processed events with a timestamp larger than the timestamp of the straggler, in decreasing timestamp order. Then, it processes the straggler, and finally re-executes the events that have been rolled back, in increasing timestamp order.*

Clearly, a lookback-based protocol adopting the universal lookback procedure belongs to the class of local rollback mechanisms [25]. Anti-messages are strictly avoided but local rollbacks are still necessary, suggesting that lookback-based protocols allow for aggressiveness, but not risk [57]. Two such protocols previously known, SRADS with local rollback [25] and Breathing Time Buckets [63], require components to work collaboratively, by exchanging event information, to avoid sending any erroneous messages. In lookback-based protocols, however, each component can determine alone whether a message can be sent out (by allowing or disallowing execution of the corresponding event).

In SRADS, components are periodically synchronized by *poll events*, which are events sent by a reader (a component that may receive a message) to a writer (a component that may send a message to another component). After sending out a poll event to a writer, a reader must block until the writer answers the poll. With local rollback, the reader can aggressively process messages received from other writers if a poll has not been answered by the writer. However, the results of the

event processing are not sent immediately to other components until the reader has sufficient information to determine that new messages received from writers would not invalidate those results (for example, if the simulation clocks of all other writers are greater than the timestamps of the messages that were used in computing those results).

The Breathing Time Buckets algorithm is based on a quantity called the *event horizon*. The algorithm requires all components to synchronize periodically. Event horizon is defined as the minimum timestamp increment of messages generated in one synchronization cycle. Components are allowed to advance to the next cycle boundary determined by their own local event horizon, but must be synchronized by the global event horizon, which is the minimum of local event horizons of all components. As a result, each component aggressively processes events with a timestamp greater than the local event horizon but less than the global event horizon, which may be subject to rollback, while sending only messages within the global event horizon.

The original Time Warp protocol allows a component to remove and execute the earliest event in the local event list repeatedly, until either the local event list becomes empty or a causality error is detected. The idea of local rollback is to limit the optimism, so that spreading the erroneous computation to other components can be avoided. Determining whether or not a message generated during the event execution can be affected by stragglers is a difficult problem, and both SRADS with local rollback and Breathing Time Buckets require components to acquire enough information from others before they can make such a decision, thus incurring the overhead of information exchanging.

The universal lookback procedure deals with the worst case of event dependence. It is possible that no state, or only a small portion of the entire state needs to be saved to enable a component to process the stragglers, as is the case of the Closed Queuing Network simulation that we will discuss in the next chapter. In some models, when the lookback procedure is used to reduce the number of rollbacks in optimistic simulations, only an extra variable might be needed to store either the value of the virtual lookback time or the size of the lookback window.

5.5 Lookback and Lookahead

We have introduced the concept of lookback and sketched two synchronization algorithms for PDES that rely upon the existence of lookback. The key question regarding the usefulness of these two synchronization algorithms is, how common is lookback in real world simulations? We try to answer this question by showing an interesting relation between lookback and lookahead.

Strangely enough, lookahead, informally known as the ability to predict the future, has not had a consistent definition. In Fujimoto's definition [28], a component is said to contain a lookahead of L at simulated time T if it can schedule events with timestamp at least $T + L$. This definition does not consider the case in which previously scheduled messages may not have left the component. For instance, it is possible that the next outgoing message that was scheduled earlier contains a timestamp of $T + L/2$. The lookahead in such a case is $L/2$, not L.

Another definition, given by Jha and Bagrodia [36], defines lookahead as the difference between the EOT and EIT. This definition does not take into consideration local events. If a component is in the middle of processing a local event, the lookahead should be the difference between the EOT and the timestamp of the local event being processed (which is the current simulated time). Therefore, a more accurate definition of lookahead at time T is the difference between the EOT and the current simulation time. An equivalent definition, more like Fujimoto's, is that a lookahead of L at T means that at T or later, the component cannot *send out* messages with timestamp smaller than $T + L$.

To understand the relation between lookback and lookahead, we must first turn our attention to two different mechanisms for delivering inter-component events (or messages). The distinction between *eager delivery* and *lazy delivery* was motivated by Fujimoto's notion of eager and lazy server [27]. In eager delivery, a message is sent out immediately when it is generated. Such a message usually contains a timestamp larger than the current simulated time of the sender. In lazy delivery, a message is not sent out until the simulation clock reaches the timestamp of the message. Therefore the timestamp of any outgoing message is always equal to the current simulated time of the sender.

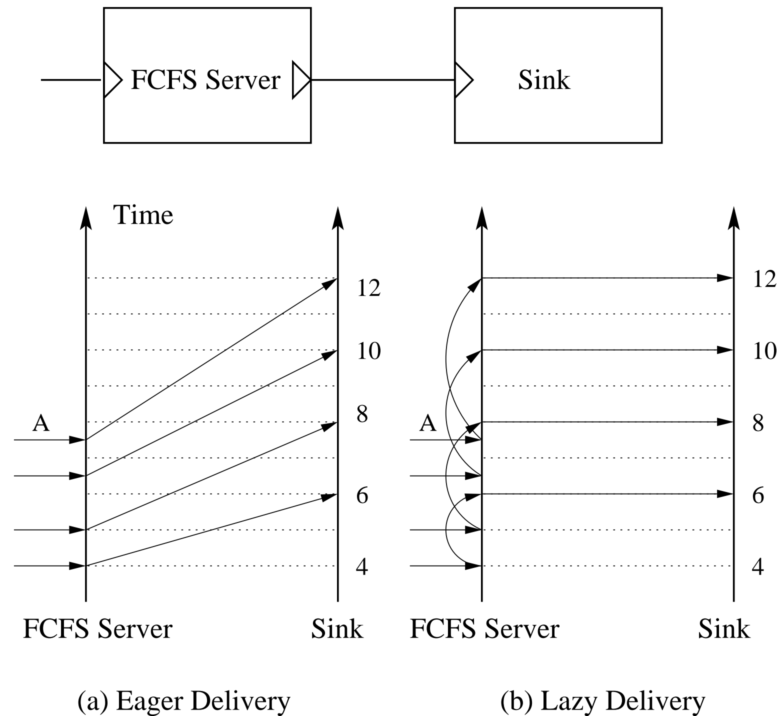

Figure 5.2: Eager Delivery versus Lazy Delivery

In Figure 5.2, we consider an example of an FCFS server using these two delivering schemes. Suppose a sequence of packets arrive at times 4, 5, 6.5 and 7.5. The FCFS server has a constant service time of 2 time units for all packets. With eager delivery, the departure event for each packet can be scheduled and sent to the sink as soon as the packet arrives, because of the nature of the FCFS server. Here, the lookahead is exploited to the maximum degree. For instance, when the packet A arrives at time 7.5, it can be immediately calculated that it will depart at time 12, so the lookahead is 12-7.5=4.5. On the other hand, lookback does not exist, because at the moment of the event departing from the server, the effect of the processed event becomes permanent and cannot be recovered.

In lazy delivery, the departure event is not immediately sent to the sink. Instead, it is withheld in the server's event list until the current simulated time is equal to its timestamp. This delay gives the component a chance to retract a departing event in case a new arrival event with a timestamp smaller than the arrival timestamp of the departing packet is received later. In such a case, the previously scheduled departure time should be modified. Lookback is the largest if the

scheduled departure event is sent out as late as possible. Therefore, lookback-based protocols prefer lazy delivery.

How large is the lookback in an FCFS server? We can see that when the packet A leaves the server at time 12, the virtual lookback time should be changed to the arrival time of the packet A, which is 7.5. Indeed, any packet arriving at a time later than 7.5 can be simply inserted into the waiting queue. However, a packet arriving earlier than 7.5 would force a causality error, because it is this packet that must leave the server at simulated time 12, not packet A. The lookback window size is therefore 12-7.5=4.5. Interestingly, the lookback under lazy delivery is the same as the lookahead under eager delivery in this case.

Is this a coincidence? No, it rather implies that there is a deep connection between lookback and lookahead, which is revealed by the following theorems.

Theorem 5.3. *If under eager delivery, a simulation contains a lookahead of L at simulated time T after event execution, then under lazy delivery this simulation contains a lookback of at least L at time $T + L$ before sending out any messages timestamped with $T + L$.*

Proof. Under eager delivery, a component is said to contain a lookahead of L at simulated time T if it can only send out new events with timestamp of at least $T + L$. Consider this simulation under lazy delivery using a lookback-based protocol. Suppose at simulated time $T + L$ a straggler arrives with a timestamp within $[T, T + L)$. We can perform the universal lookback procedure, in which all events with a timestamp larger than that of the straggler are first rolled back. After the straggler has been processed, these affected events are reprocessed. Consider the messages generated during the first execution of these events (before the arrival of the straggler). All these messages must contain a timestamp greater than or equal to $T + L$, because they are produced by events later than T and the lookahead is L at time T. Therefore, they could not have been delivered to other components, for lazy delivery is adopted. Consequently, when these affected events are being reprocessed no anti-messages will be necessary. The universal lookback procedure can always be performed successfully, so the lookback is at least L at time $T + L$ before delivering the messages timestamped with $T + L$. ☐

Now we know that lookback is at least as common as lookahead. Whenever lookahead exists, the same amount of lookback is also available. Can lookback exist without lookahead? The following theorem answers this question positively.

Theorem 5.4. *Lookback may exist even when lookahead is zero.*

Proof. We give a simple example to support this claim. Assume a zero delay component whose only function is to copy any received message from an inport to an outport. Apparently there is no lookahead in the component, because messages can arrive at any time and leave immediately. The lookback, however, is infinite. The component can accept messages with any timestamp no matter what is its current simulated time. □

Theorem 5.3 and 5.4 together suggest that lookback is always larger than or equal to lookahead, but when are they equal? In other words, under what circumstances does a certain amount of lookback translate into the same amount of lookahead? The following theorem gives the condition under which lookback and lookahead will be equal.

Theorem 5.5. *Assume a component has a lookback of L at simulated time T, under lazy delivery, and there is an outgoing message timestamped with time T. If the underlying simulation model guarantees that all other messages sent out later will contain a larger timestamp than T, then at simulated time $T - L$ this component has a lookahead of L.*

Proof. Denote by t the timestamp of the event e that schedules the outgoing message at time T. By definition of lookback at time T, $t \leq T - L$, otherwise another event e' with timestamp $t' > T - L$ but $t' < t$ could affect the event e, and subsequently the outgoing message at time T, which violates the assumption that the lookback is L. Let us consider the same component at the simulation time $T - L$ under eager delivery. Since $t \leq T - L$, the event at time t must have already been processed, with the outgoing message scheduled. Since no messages can be sent in the time window $[T - L, T)$, it is apparent that $EOT = T$ at time $T - L$, i.e., the next output message is at T, therefore the lookahead at time $T - L$ is exactly L. □

Theorem 5.5 explains why in the FCFS server model the lookback is exactly the same as the lookahead. On the other hand, lookback and lookahead do not always exclude each other. With lazy delivery, a component can still inform others of its EOT without sending the actual event, as stated by the theorem below.

One primary source of lookahead is the opaque period, during which no message will be sent [43]. It is in essence promises made by a component not to send messages to neighbors within a certain time window. We now show that at least the same amount of lookback exists with an opaque period.

Theorem 5.6. *Under lazy delivery, if a component can predict an* opaque period *of L at the current simulated time T, then the lookahead at time T is equal to L and the lookback at time $T + L$ is at least L.*

Proof. By definition, lookahead at time T is L, because the EOT is at least $T + L$. Also, when the simulation clock advances to $T+L$ under lazy delivery, the component can, in the worst case using the universal lookback procedure, execute any events received within the time $[T, T + L)$, as described in the proof of Theorem 5.3. Thus, the opaque period L becomes part of the lookback window. □

Theorem 5.6 establishes that an opaque period also implies at least the same amount of lookback. This may establish a new mechanism to exploit both lookback and lookahead at the same time.

To summarize, to some extent, lookback looks like a dual of lookahead: lookahead is the ability to predict the future, while the lookback is the ability to change the past. At first glance they seem to be completely unrelated. Nevertheless, when we are able to advance the simulation clock more aggressively, the future becomes the past. The relations between lookahead and lookback defined by the above three theorems therefore become easy to understand.

5.6 Lookback and Super-Criticality

Are lookback-based protocols, namely LB-GVT and LB-EIT, conservative or optimistic? Lookback procedures make them more or less optimistic, but since there exist components in which no state-saving is required to exploit lookback, we argue

that the optimism of lookback-based protocols varies with different components. At least for components requiring no state-saving, lookback-based synchronization can be convincingly characterized as conservative.

There is a widespread belief that conservative protocols cannot beat the bound imposed by the critical times of events [34]. The earliest possible completion time of an event in these protocols is determined by the completion time of two other events: one is its *predecessor*, the previous event in the same component, and the other is its *antecedent*, the event that scheduled it. An event can start execution when both its antecedent and predecessor has been completed. Thus, The completion time of an event is equal to the maximum completion time of antecedent and predecessor plus its execution time. The critical time of an event is then recursively defined as the critical time of its predecessor or antecedent, whichever is larger. Startup events that are scheduled when simulation starts have a critical time equal to its execution time.

The critical time of an event gives a lower bound on its completion time in traditional conservative protocols. However, we will show that this is not true for lookback-based protocols. For convenience of discussion, we denote the starting time of an event e by $start(e)$, the completion time by $complete(e)$, and the critical time by $critical(e)$.

There are two cases in which lookback-based protocols can produce super-critical speedup.

An example of the first case is depicted in Figure 5.3(a), in which we assume that the event A executes first. Later, another event B is received with a timestamp smaller than that of A but greater than the virtual lookback time. The event B can then be processed safely by the lookback procedure. For this case, assume that these two events are independent of each other. Thus, when processing the event B, the lookback procedure does not need to correct the component state reached after the execution of the event A. Apparently, the event A completes even before the event B starts execution. For simplicity, assume that the event B is on the critical

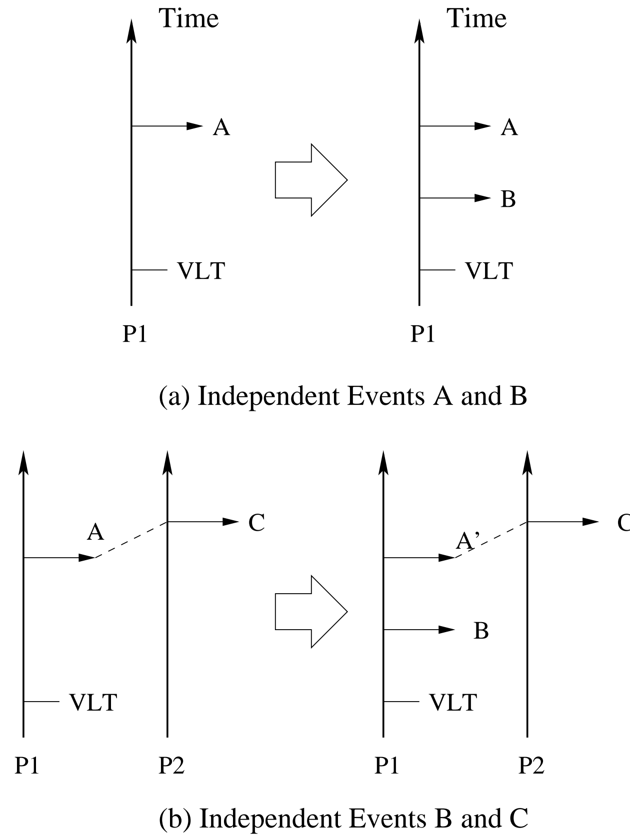

Figure 5.3: Supercritical Events in Lookback-Based Protocols

path, i.e., $complete(B) = critical(B)$, thus we have

$$complete(A) < start(B) < complete(B) = critical(B) < critical(A)$$

or

$$complete(A) < critical(A)$$

which is the sufficient condition for super-criticality given in [62].

In the other case, depicted by Figure 5.3(b), the two events A and B are no longer independent. Rather, after processing the straggler B, we have to repair the incorrect state produced by the first execution of the event A. Therefore $complete(A) > complete(B)$. But the event C created by the first execution of the event A is not affected. We say that the event C is independent of the event B for

this reason. Again, if we assume that the event A is on the critical path, then

$$complete(C) < complete(B) < complete(A) = critical(A) < critical(C)$$

or

$$complete(C) < critical(C)$$

which proves the event C is a super-critical event.

Jefferson and Reiher have proven that all conservative mechanisms are bound by the critical times of events [34] but under the assumption that all correct conservative simulation mechanisms must employ elementary scheduling. In it, for all pairs of committed events e and e', whenever e is either the predecessor of e' or the antecedent of e', then $complete(e) = start(e')$. The authors claimed that otherwise the simulation might be incorrect. For convenience, we reproduce the relevant part of their proof here:

> Either e is the predecessor or the antecedent of e'. If e is the predecessor of e', then the last instruction of e might create a side-effect that affects the process state for event e'. If e' is started before e finished, then it cannot execute in the context of the exact state produced by e; the last instruction of e might produce a state change upon which e' depends. Hence e' might execute incorrectly.
>
> Likewise, if e is the antecedent of e', then any mechanism that would start to execute e' before finishing e must in effect be "guessing" that e' will be scheduled at the end of event e, and also "guessing" what the parameters from e to e' would be. (Recall our assumption that e can only send the event message to schedule e' at the very end of e's execution.) Since those guesses might be wrong, the simulation might be incorrect.

In the first part, Jefferson and Reiher did not consider the feasibility of mechanisms that can ensure that stragglers can always be executed correctly. The lookback-based protocol is such a mechanism. It guarantees that once an event is about to execute, all predecessors of this event, if there are any, can also be pro-

cessed correctly. If it cannot make such a commitment, the event processing has to be delayed.

As to the second part of the proof, one has to realize that guessing is not the only mechanism that allows the component to start executing an event before finishing its antecedent. The failure to consider other possibilities results from the traditional approach, as pointed out in [33], which always views events as atomic. As in the example of Figure 5.3(b), the event A is executed two times, one by the regular event procedure and the other by the lookback procedure, and the event C, produced by the first execution of the event A, is guaranteed to be correct by the notion of the lookback (one simple example could be the event triggered by the arrival of event A on component P1, which is not changed by earlier or later arrivals of other events on component P1). If it were not, the event A would fail to be processed in the first execution and would delay its first completion, because no sent out event can be rolled back.

5.7 Four Types of Lookback

A closer study of lookback reveals that there are some other types of lookback different from the one that is discussed above upon which lookback-based protocols depend. We could, for instance, define lookback as the ability of a component to process only a *limited* number of stragglers, instead of any number of stragglers, within the lookback windows. This kind of lookback, as we will see, cannot serve as the basis of the lookback-based protocols. However, it is still useful in optimistic simulations as an optimization over optimistic protocols, since subsequent stragglers can resort to the rollback and recovery procedure. We name this type of lookback as *weak lookback*, as oppose to *strong lookback* that we have been discussing.

Moreover, we could classify lookback according to whether or not rollbacks are prohibited. The type of lookback in lookback-based protocols is naturally *universal*, since rollbacks are allowed in the universal lookback procedure. If we define lookback in a stricter way to avoid any rollbacks, such lookback would be *direct*.

These two classifications are orthogonal, so we now have four types of lookback illustrated by Figure 5.4.

		What is avoided when stragglers are successfully processed?	
		No rollbacks	No anti–messages
Number of stragglers within lookback window that can be processed correctly	Unlimited	Direct Strong Lookback	Universal Strong Lookback
	Limited	Direct Weak Lookback	Universal Weak Lookback

Figure 5.4: Four Types of Lookback

By definition, a weak type of lookback always contains the corresponding strong lookback. The same is true for direct and universal lookback. Therefore, universal strong lookback always includes direct strong lookback, direct weak lookback always includes direct strong lookback, etc. These containing relations are depicted by arrows in Figure 5.4.

The relation between universal strong lookback and direct weak lookback bears some elaboration. It is not readily clear whether one encloses the other, but from the definition it can be derived that they at least have an intersection which is direct strong lookback.

Let us consider whether rollbacks and anti-messages can occur during the lookback procedure for each type of lookback. For rollbacks, there are three possibilities: no rollbacks, some rollbacks, and always rollbacks, denoted by NR, SR, and AR respectively. For anti-messages, there are only two possibilities, no anti-messages and some anti-messages, or NA and SA, because anti-messages are not always necessary if no messages have ever been produced during the event execution. Of the 6 combinations of these possibilities, only 5 are possible. 'NR,SA' which stands for 'No Rollbacks and Some Anti-messages' is contradictory, because anti-messages may produce rollbacks in recipient components. Let's denote these 5 cases as 1,2,3,4,5

respectively as shown in Table 5.1.

Case	Rollbacks	Anti-messages
1	No Rollbacks	No Anti-messages
2	Some Rollbacks	No Anti-messages
3	Always Rollbacks	No Anti-messages
4	Some Rollbacks	Some Anti-messages
5	Always Rollbacks	Some Anti-messages

Table 5.1: All Five Possible Cases of Rollbacks and Anti-messages

It is now clear that with universal strong lookback case 1,2, and 3 could happen, while with direct weak lookback case 1,2, and 4 are possible. This explains why there is no containing relation between them. Direct strong lookback corresponds to case 1, while for universal weak lookback, the weakest form of the four types, all five cases could occur. From these we can deduce their relations, illustrated by Figure 5.5.

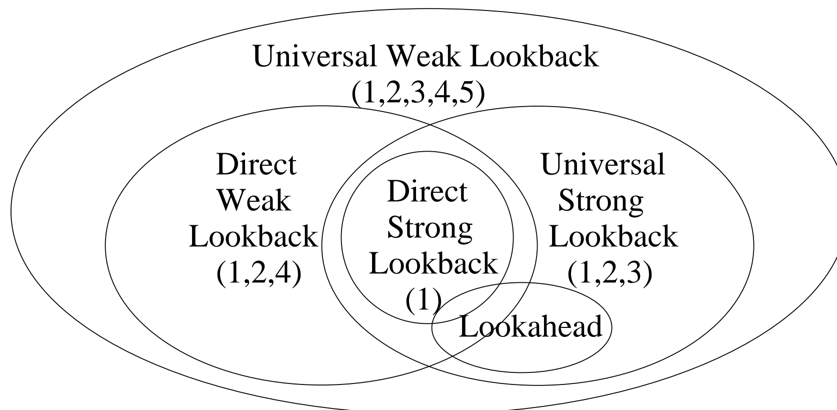

Figure 5.5: Relations of Four Types of Lookback and Lookahead

The lookback discussed previously, which enables lookback-based protocols, is actually universal strong lookback. Weak lookback does not support lookback-based protocols, because the component has no knowledge of how many stragglers would come and therefore some of the stragglers may violate the lookback constraint. For weak lookback, Theorem 5.1 no longer holds and $LB(e, \tau)$ might be larger than the timestamp of the straggler e. In such a case, the execution of the earliest event in the entire simulation would increase the virtual lookback time to a value larger than the minimum timestamp of other unprocessed events, thus making the violation of

the lookback constraint inevitable. Subsequent stragglers would have to resort to the rollback and recovery procedure, which means weak lookback can be exploited in optimistic simulation.

In fact, the lookback in the FCFS server belongs to a even stronger type: *constant lookback*. Constant lookback may be changed by a timestamp-ordered event, but not a straggler. Formally, $LB(e, \tau = (T, n)) \leq T$, if $ts(e, \tau) < T$. It is still unclear at this time whether the explicit identification of constant lookback has any particular significance, so for now we only consider strong and weak lookback.

5.8 Finding Lookback

How do we know whether or not lookback exists, and, if it does, which type it belongs to and how large it is?

First, let us see how to identify universal lookback. It is not unreasonable to believe that any change made to local state variables can be aggressively repaired by some means. Because shared variables are usually excluded in the PDES simulation, the universal lookback would be infinite if no messages have been sent out during the event execution. If the event execution produces messages, the universal lookback may not necessarily be zero. It actually depends on a property of the message being sent out.

We define the *impact time* of a message as the upper bound on the timestamp of any stragglers that can change or cancel the message. We further distinguish *absolute impact time* from *dynamic impact time*. The former cannot be changed by the arrival of a straggler, so any number of stragglers with a timestamp larger than the absolute impact time of a message cannot affect the messages. The latter is subject to change after a straggler has been processed, so only a limited number of stragglers can be correctly handled. One can easily deduce that the absolute impact time implies universal strong lookback while the dynamic impact time implies universal weak lookback. In both cases, if a straggler comes with a timestamp smaller than the impact time of the message, this straggler cannot be processed by the lookback procedure, because it may change the message. As a result, an anti-message is required in order to cancel out the old message, which is explicitly prohibited by

the lookback-based protocols but allowed in optimistic protocols.

If the event execution produces exactly one message, the impact time of the message gives the largest possible lookback. If the event execution produces more than one message, the maximum amount of available lookback is equal to the current simulated time minus the maximum of the impact times of all messages.

It is now relatively easy to distinguish direct lookback from universal lookback. If the lookback procedure can directly process stragglers by simply modifying the local state of the component, the lookback is certainly direct. On the other hand, if the lookback procedure has to work in a rollback and recovery manner then the lookback must be universal.

5.9 Lookback-Based Optimization and Lazy Cancellation

Based on the concept of different types of lookback, we can design three techniques to avoid unnecessary rollbacks and anti-messages in optimistic simulation. The first technique is based on direct lookback. When a straggler is found, its timestamp is compared against the value of the virtual lookback time. If it is larger than the latter, it is known for sure that no rollbacks are required, and hence no anti-messages are needed.

If the straggler has a timestamp smaller than the virtual lookback time, we have to resort to the rollback and recovery procedure. When we are rolling back processed events that are later than the straggler, we can compare the impact time of those messages generated by processed events with the timestamp of the straggler. If the impact time of a message is smaller than the timestamp of the straggler, it cannot be affected by the straggler, and therefore it is unnecessary to cancel it by sending out a corresponding anti-message. The only difference between the absolute impact time and dynamic impact time is that the dynamic impact time must be reset as each straggler is processed. As we discussed earlier, often the reset value after one straggler is equal to absolute impact time.

Lazy cancellation [30] is the technique of avoiding unnecessary anti-messages if the previously delivered messages have not been affected by a straggler, by comparing the new messages generated by the reprocessing of the rolled-back event and

the old messages generated during the first execution of the same event. The importance of lazy cancellation is that it is one of the techniques that enable optimistic simulations to circumvent the execution time limit imposed by the critical times of events [34]. Practically, it has been shown to be 15% faster on average than aggressive cancellation (i.e., optimistic simulation without lazy cancellation).

It is still unclear, however, in what applications lazy cancellation would perform better than aggressive cancellation. Fujimoto suggests that lazy cancellation automatically exploits lookahead, so simulations with good lookahead would be more amenable to lazy cancellation. With the establishment of the notion of lookback, we can now claim that *what is automatically exploited by lazy cancellation is actually lookback*.

This is because when a component contains a certain amount of lookback, it may be able to process at least some stragglers without sending anti-messages. That is, if there are any messages generated in the execution of events later than a straggler, these messages cannot be affected by the straggler, otherwise anti-messages would be absolutely necessary. When applying lazy cancellation to this component, it is known for sure that the new messages generated by the reprocessing of the rolled-back event should be the same as the old ones. Therefore, lazy cancellation benefits from the existence of lookback. This, however, does not invalidate the claim that lazy cancellation automatically exploits lookahead, because according to Theorem 5.3 lookahead is contained within universal strong lookback.

It is interesting to compare lazy cancellation and the optimization technique based on lookback. Both are trying to suppress unnecessary anti-messages. However, lazy cancellation avoids unnecessary anti-messages strictly, while with lookback it could still happen that the component might cancel the original message with a corresponding anti-message and then send the same message again. As an example, consider the function $y = f(x) = x^2$, where y is the output message and x represents the state of a component. Suppose a straggler s arrives which changes x from 3 to -3. With lazy cancellation, the anti-message is avoided because the new value of y happens to be the same as the old one. However, any optimization technique based on lookback cannot distinguish s from another s' that changes x, say, from 3 to -2.

Neither is it able to compare the new x with the old x, because it is not allowed to look into the content of the message. Therefore, unless other techniques based on a more generalized concept of lookback, such as conditional weak lookback, are adopted, with weak lookback the component has to send the anti-message for y, even if y remains unchanged. We conclude that lookback is incapable of capturing dynamic independence among events.

We believe, however, that such dynamic independence happens rarely in practice. The more frequent cases are where lazy cancellation and lookback have the same effect. In these cases, lookback would be definitely superior to lazy cancellation. Direct lookback determines whether a rollback caused by a straggler is necessary by comparing the timestamp of the straggler with the virtual lookback time of the component. Universal lookback determines whether or not a message has been affected by a straggler by comparing the timestamp of the straggler with the impact time of the message. In either case only comparison of a pair of floating point numbers is involved and the delay of extra computation is almost negligible. Lazy cancellation, however, determines the validity of a message only after the message is re-generated, not only delaying the propagation of the anti-message if the message needs to be cancelled, but also incurring significant overhead if the message is represented with complex data structures.

5.10 Conclusion

Lookback-based protocols provide a new technique to exploit intra-component parallelism in simulation models. Surprisingly, the property of lookback, the ability to change the past, is closely related to the property of lookahead, the ability to predict the future. We have shown that in simulation models strong universal lookback is more common than lookahead. Furthermore, these two notions are complementary and can be exploited at the same time.

It is of theoretical importance that lookback-based protocols allow conservative simulation to circumvent the critical path execution time limit. However, the lookback-based protocols are still not a general solution to PDES problems. Lookback does not exist in all simulation models, and even if it does exist, the lookback

procedure may not be efficient. The utilization of lookback also complicates the simulation modeling, because out-of-timestamp execution has to be taken into consideration.

Despite of these two problems, we believe that our discovery of lookback points to a new direction for research in PDES. It contradicts the prevailing belief that the PDES synchronization is a solved problem. The classification of four types of lookback further clarifies and completes the theory. Nevertheless, much still needs to be done. The feasibility and the performance of lookback-based protocols and of optimistic protocols with lookback-enabled optimizations on a wide variety of simulation models have yet to be established by extensive experiments. More efforts must be expended to understand the semantics of the simulated time, which may lead to still more interesting solutions to the PDES synchronization problem.

CHAPTER 6
CLOSED QUEUING NETWORK SIMULATION: FIRST APPLICATION OF LOOKBACK-BASED PROTOCOLS

From now on, we will turn our attention to applications of lookback-based protocols in parallel simulation. The theory of lookback has been thoroughly discussed, yet the efficiency of lookback-based protocols remains to be tested for real-world applications. In this chapter we detail our effort to exploit lookback in queuing network simulation.

6.1 Closed Queuing Network

Queuing network simulations have been extensively studied by many PDES researchers. There are many varieties of queuing networks, differing in interconnections between components and many specific parameters of components, such as the mean of the service time, the random distribution of the service time, the scheduling policy, etc. We chose the particular one that was described in [36, 46], which is known as Closed Queuing Network, or CQN.

A CQN consists of a configurable number of switches and FCFS (First-Come-First-Served) servers (Figure 6.1). Each packet traveling through one of the FCFS server tandems will eventually arrive at a switch. The switch will then dispatch the packet to one of the tandems randomly.

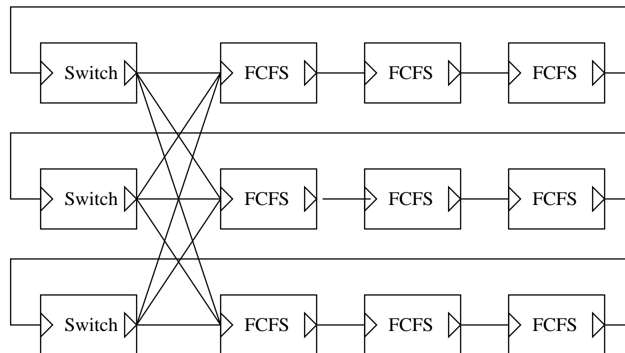

Figure 6.1: A CQN with 3 Switches and 9 FCFS Servers

CQNs are readily amenable to conservative protocols because lookahead usually comes from the FCFS servers where packets have to be delayed for random service times. Denote the arrival time, the departure time and the service time of the ith packet P_i by A_i, D_i and S_i respectively. We have

$$D_i = \begin{cases} A_i + S_i & \text{if the server is available at } A_i \\ D_{i-1} + S_i & \text{otherwise} \end{cases}$$

The departure time can be determined as soon as the packet arrives. If eager scheduling is employed, the departure event will be immediately generated and sent out. The lookahead, which is equal to $D_i - A_i$, is at least S_i. Nicol noticed that lookahead can be augmented if the service time of the next packet can be pre-sampled [51]. In such a case, the departure time of the $i+1$st packet is at least $D_i + S_{i+1}$, so lookahead becomes $D_i - A_i + S_{i+1}$. The service time must be independent of packets if the service time is to be sampled ahead of time. This, however, may not always be true in practice. A more general case is the one in which a lower bound on the next service time can be obtained, and the lookahead is now $D_i - A_i + min\{S_{i+1}\}$.

Bagrodia and Liao [8] proposed a technique to reduce the rollback distance for optimistic CQN simulation. They distinguished between the receive time of a packet and the time at which a packet can be served. When a straggler packet arrives, the logical process needs to be rolled back only to the earliest time this message can be served. This approach bears some resemblance to lookback-based protocols, for both algorithms stem from the observation that rollback to the timestamp of the straggler is unnecessary. However, lookback-based protocols can totally eliminate stragglers by deliberately delaying the execution of events that would raise the virtual lookback time above the minimum timestamp of any stragglers.

6.2 Determining Lookback in CQN Simulation

Before applying lookback-based protocols to the CQN simulation, we want to make sure that every component in the CQN contains a substantial amount of lookback. This is, however, not a necessary condition for the applicability of lookback-

based protocols. Even in a simulation where lookback is zero in every component, they still work, because the earliest event in the entire simulation is always guaranteed to be eligible for execution. Nevertheless, without adequate lookback, the performance of these protocols could be fairly disappointing.

Let us first look at the switch component only. The function of a switch is to generate a uniformly distributed random integer to determine the destination of a packet. Normally a switch keeps a private random number generator which is invoked when a packet arrives. A problem arises with this solution when packets can arrive in out-of-timestamp order. The statistical property of the destination distribution remains unaffected, if the random numbers are still generated in the same way. However, the parallel simulation becomes no longer *repeatable*, meaning two simulation runs with exactly the same set of parameters may produce totally different results. This is because out-of-timestamp order events are produced when different processors advance the simulated time unevenly, therefore they depend only on the runtime behaviors of the simulation, which is completely uncontrollable.

A simple solution to preserve repeatability is to let the switch obtain a random number from a random number generator carried by each packet. This solution is similar to the one proposed in breadth-first rollback [22, 23], in which the rollbacks are localized to each entity, and therefore for repeatability, they also maintain their own random number generators. It would not be a burden in terms of memory usage because a random number generator in fact does not occupy much space. For instance, a Linear Congruential Generator requires as few as 8 bytes. Packets passing through the switch now become completely independent of each other.

Nevertheless, the solution of distributing random number generators across components is unnecessary in this particular CQN simulation. We noticed an interesting fact that a switch may never receive an out-of-timestamp order packet, because a switch receives packets from only one FCFS server. The simulation would not have been correct had a switch received a straggler. Therefore, the switch component needs no change in lookback-based CQN simulation.

As to the FCFS server, we have mentioned in the previous chapter that in it lookahead under eager delivery and lookback under lazy delivery are always the

same. With lazy delivery, the virtual lookback time of the FCFS server at any time is always equal to the arrival time of the last packet leaving the server. Any received packet with a timestamp smaller than that of the current packet in service must preempt the later. All those stragglers with a timestamp greater than or equal to the virtual lookback time can be correctly inserted into the waiting queue at positions decided by their timestamps. For packet P_i at its departure time D_i, the virtual lookback time is A_i, resulting in a lookback of $D_i - A_i$ equal to the lookahead under eager scheduling given previously. If pre-sampling is employed, at time $D_i + min\{S_{i+1}\}$, which is the earliest time next packet can complete the service, the virtual lookback time is still A_i. Therefore at this instant the lookback is $D_i - A_i + min\{S_{i+1}\}$, still the same as the lookahead.

Apparently, packets with timestamp smaller than the virtual lookback time will trigger a causality error. But such errors are preventable by adhering to the lookback constraint: when a packet is about to leave the server, its arrival time must be checked against the LBTS. If the arrival time is greater than the LBTS, it means that another packet with a smaller timestamp may arrive later. The packet cannot be sent out, and the departure event is returned back to the local event list. Two cases can happen afterwards. Either the LBTS is advanced by invoking a new round of LBTS computation, making the same event eligible for execution according to the lookback constraint, or a new event may be received with a smaller timestamp satisfying the lookback constraint.

6.3 Lookback-Enabled FCFS Server

In Chapter 4 we have seen an FCFS Server implemented in COST. It contains one inport, one outport and one timer. The inport is bound to a member function *Arrive*, which means whenever a packet arrives at the inport the function *Arrive* will be activated. The timer is bound to a member function *Depart*, which is the event handler for departing packets.

6.3.1 Handling Arrivals

The sequential algorithm for handling the arriving packets is given below.

```
if the server is free
    schedule the departure event;
    mark the arriving packet as the in-service packet;
    set the server to busy;
else
    put the packet at the tail of the internal queue;
end if
return true;
```

When the packet can arrive in out-of-timestamp order, things become much more complicated. First we must check if the packet is a straggler by comparing its timestamp with the current simulated time. If it is not a straggler the algorithm remains the same. Otherwise there are only two ways to handle the packet. Either the packet must be marked as the current in-service packet, or it must be inserted into the internal queue at a position determined by its timestamp. If the server is free, the first case must be chosen. If it is not, the current in-service packet may have a larger timestamp, so the arriving packet can preempt the current in-service packet, cancel the old departure event and schedule a new one. Only when neither of these two conditions is true will the packet go to the internal queue.

```
if the arriving packet is not a straggler
    if the server is free
        schedule the departure event;
        mark the arriving packet as the in-service packet;
        set the server to busy;
    else
        put the packet at the tail of the internal queue;
    end if
else
    if the server is free or the current in-service packet has
    a timestamp larger than the newly arriving one
        if the server is not free
```

```
            cancel the departure event associated with the
            current in-service packet;
        end if
        compute the departure time for the arriving packet;
        schedule the corresponding departure event;
        mark the arriving packet as the in-service packet;
        set the server to busy;
    else
        insert the arriving packet into the internal queue
        according to its timestamp;
    end if
end if
return true;
```

How to compute the departure time is worth elaboration. In the case of a straggler packet, it could happen that the last in-service packet with a larger departure time than the timestamp of the straggler has already left the server, and the service starting time should be the last departure time instead of the timestamp of the straggler. Consequently, the correct formula to calculate the departure time should be

$$D_i = max\{A_i, D_{i-1}\} + S_i$$

It can be verified that this is in accordance with the other formula for calculating D_i given earlier. The implication is that we must keep track of the departure time of last packet. This can be done by writing the value of the departure time to a local variable when a packet is about to leave.

6.3.2 Handling Departures

In the sequential case, the event handler to process the departure event is quite simple:

```
send out the in-service packet via the outport;
if the internal queue is not empty
```

```
        mark the head packet as the in-service packet;
        compute the departure time for the arriving packet;
        schedule the departure event;
    else
        set the server to free;
    end
    return true;
```

With stragglers, one significant difference is that before delivering the in-service packet to the outport, its arrival time must be checked against the LBTS. If the arrival time is smaller than the LBTS, the event handler returns immediately with a false value, indicating that the departure event cannot be successfully processed.

```
    if the in-service packet arrived earlier than the LBTS
        return false;
    end if
    send out the in-service packet via the outport;
    record the departure time;
    if the internal queue is not empty
        mark the head packet as the in-service packet;
        compute the departure time for the in-service packet;
        schedule the departure event;
    else
        set the server to free;
    end
    return true;
```

A departure event could be a straggler. This occurs when a packet straggler schedules a departure event earlier than the current simulated time. Therefore, when computing the departure time for the new in-service packet, the starting service time must be the maximum of its arrival time and the departure time of the packet just departing (or the timestamp of the current departure event).

6.4 Dealing with Bounded Queues

So far the correctness of the lookback-enabled FCFS server is based on an assumption that the internal queue is unbounded so no packet would be dropped. When stragglers are allowed, the decision whether or not a packet should be dropped due to a full queue is extremely difficult to make.

To see how complicated the situation is, let us look at a seemingly plausible hypothesis:

Hypothesis 1. *If a packet arrives at a time when the internal queue is already full, then this packet is destined to be discarded, no matter how many packets with a smaller timestamp will be received later.*

The intuition behind this hypothesis is that a straggler packet can only make the internal queue more crowded. It would have been very useful if it were true because the decision of dropping a packet could then be made immediately once the packet arrives when the internal queue is full. However, it is not sustainable, due to the fact that a straggler packet may leave earlier than the current simulated time. Figure 6.2 depicts such an example.

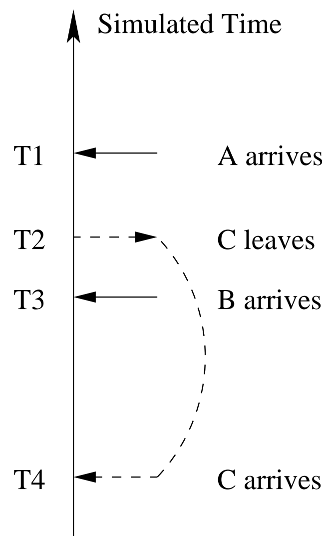

Figure 6.2: A Counter Example of Hypothesis 1

In the example we assume the size of the internal queue is 1, meaning any packets other than the in-service packet have to be discarded. At $T1$ Packet A

comes after packet B, so it has to be dropped if there are no stragglers. When a straggler C arrives at $T4$, not only does it force packet B to be dropped, but also it leaves early at $T2$ to allow packet A to occupy the empty server!

6.4.1 Initial Position

The implication of the above example is that the decision of dropping a packet cannot be made at the time the packet arrives. A straightforward solution is to keep track of *initial positions*, which can be used to help make the dropping decision. The initial position of a packet is defined as the number of packets in the queue at the arrival time of the packet if all packets were received in the timestamp order. The sufficient and necessary condition for a packet to be discarded is to have an initial position greater than or equal to the size of queue.

At what time should we make the dropping decision? We only have two choices left: either at the time the packet starts receiving service or at the time the packet leaves the server. The latter seems feasible, because a packet only leaves when it cannot be affected by any stragglers, so its initial position in the queue will not change any more. However, it is not a good choice for two reasons. First, it is a waste of cpu time to schedule and process the departure event of the to-be-dropped packet. We should try to avoid scheduling these events. The second reason will be presented when we discuss the possibility of mixing up lookback-enabled components and sequential components.

Can we really make the dropping decision when the packet enters the service and becomes the in-service packet? One may think this is impossible because at this time the initial position of the packet is still subject to changes caused by stragglers, and by the disproof of Hypothesis 1 whether or not a packet should be discarded is never certain. Fortunately, we can assure that the case depicted in Hypothesis 1 would never happen when the packet is entering the service.

Theorem 6.1. *At the time a packet is to be marked as the in-service packet, if its initial position is greater than or equal to the size of queue, then it is destined to be discarded.*

Proof. As shown in Figure 6.3, suppose the current simulated time is $T2$ and the

packet A with an arrival time of $T3$ is about to start the service. It is allowed to do so because the packet B which arrived at $T5$ has just completed its service (its service starting time is irrelevant to discussion here). There should be no other arrival events between $T3$ and $T5$ at this time, because otherwise it would be such an event that enters the service at $T2$. Any straggler that is received later must have a timestamp no smaller than $T5$, because otherwise packet B would not be allowed to leave and consequently packet A would not have any chance to start the service (in the departure event handler the lookback constraint is always checked first). If any straggler occurs, say, packet C, with a timestamp between $T3$ and $T5$, its departure time must be greater than $T2$, because it arrives later than packet B. Therefore, no departure events can be added to the window $[T3, T5]$ after the simulated time reached $T2$, which means the initial position of packet A can only be increased once it enters the service. □

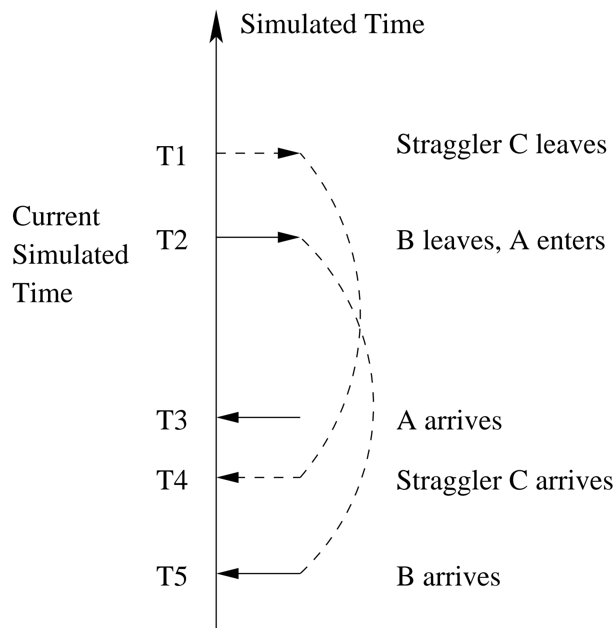

Figure 6.3: A Straggler Can Only Increase the Initial Position of Another Packet Entering the Service

Notice that according to Theorem 6.1 we are sure that a packet can be dropped if it has an initial position that is too large, but we are uncertain whether or not the a packet with a small initial position can survive the entire service time. It is

possible that a number of stragglers (sometimes one is enough) preempt a packet that is already in service and increase its initial position forcing it to be dropped.

Now the rest of the task is to calculate the initial position for every packet. For this purpose we must maintain a departure time list that records the times at which packets leave the *server* (if the in-service packet is viewed as not occupying a place in the queue, the departure time list would contain the time each packet leaves the *queue* and becomes the in-service packet).

Since there are only two types of events for an FCFS component, and each event could be either a straggler or not, we need to consider these four cases as well as an additional one in which a packet is dropped.

- The initial position of an arriving non-straggler packet is simply the number of packets currently in the queue. Such arrival will not affect the initial position of any other packets.

- When calculating the initial position of an arriving straggler packet, those packets that have already left but whose departure time is greater than the timestamp of the straggler must be considered. It can be shown that these packets would have been in the queue at the time the straggler arrives if event arrivals obeyed timestamp order (because their departure times are greater than the timestamp of the straggler). The number of these packets can be obtained by scanning through the departure time list. The initial position of a straggler packet is then the number of these packets plus the number of packets currently in the queue with a smaller timestamp. A straggler packet would increase by one the initial position of every packet with a larger timestamp.

- A non-straggler departure event does not change the initial position of any other packets.

- A straggler departure event, triggered by a straggler packet that leaves earlier than the current simulated time, decreases by one the initial position of every packet in the queue with a larger timestamp.

- A dropped packet decreases by one the initial position of every packet in the queue with a larger timestamp.

6.4.2 Lazy Evaluation

The algorithm to maintain the initial positions for all received packets is not only complicated but also inefficient. The worse case happens when a packet to be sent into service is found out having to be dropped. The entire queue must be scanned in order for initial positions of all the packets currently in the queue to be decreased by one. Apparently, FCFS components based on this algorithm can only be efficient to the cases where packets are rarely dropped.

A better design originates from the fact that initial positions can be derived recursively. Denoting by IP_i the initial position of the ith packet, we have:

$$IP_i = \begin{cases} 0, & \text{if the server is free at } A_i \\ IP_{i-1} + 1 - LP[A_{i-1}, A_i], & \text{otherwise} \end{cases}$$

where $LP[A_{i-1}, A_i]$ is the number of packets leaving between $[A_{i-1}, A_i]$.

The useful observation is that IP_i can be completely determined according to the above formula when the ith packet is entering the service. At this time the $i-1$th packet has already left, so D_{i-1} is known. Whether the service is free at A_i can be determined by comparing D_{i-1} with A_i. If $A_i \geq D_{i-1}$, the ith packet arrives after the last one left, so the server is free and the packet can be immediately put into service. Otherwise, $D_{i-1} > A_i$, and IP_i must be calculated from IP_{i-1} and the number of packets leaving between A_{i-1} and A_i. We only need to make sure the no straggler can leave within the window $[A_{i-1}, A_i]$. Any stragglers cannot arrive before A_{i-1}, because the $i-1$th packet has already left. Any straggler arriving after A_{i-1} must leave at a time later than D_{i-1}, which is greater than A_i. Hence, after the ith packet enters the service, $LP[A_{i-1}, A_i]$, the number of packets leaving between A_{i-1} and A_i is fixed and cannot be affected by any stragglers.

This idea reflects the general guideline of *lazy evaluation* in designing lookback-enabled component: *everything should be computed as late as possible.* The previous implementation, referred to as *eager evaluation*, attempts to calculate the initial

position as soon as the packet just arrives. Obviously, the calculation of the initial position at the arrival time is wasteful, for it will much likely be changed by the arrival of a straggler. At the time the packet enters the service, it is much less likely that a straggler would occur, so it is meaningful to carry out the computation now. Notice that the result of computation should be regarded as merely an estimation, since it may still be affected by a straggler, though with a much smaller probability. This estimation is only true when no straggler occurs. The dropping decision based on the estimation, however, is guaranteed to be always correct: we would never drop a packet that should not be dropped.

6.5 Experimental Results

We tested the performance of the two lookback-based protocols, LB-GVT and LB-EIT, on the CQN simulation [2]. We assumed infinite queue capacity, but for LB-GVT we also included the dropping control algorithm based on eager evaluation described above (no packets are actually dropped to keep number of packets constant in the simulation). All experiments were conducted on a 500Mhz Quad Pentium III machine. Four CPUs were used in the parallel execution. Implicit heap queue was used to maintain the simulation event list.

Figure 6.4 shows the event processing rates, measured in the number of events processed per second, of lookback-based protocols on the CQN with different configurations. The X axis represents the configuration of the network, denoted by three numbers in the format of $(a, b \times c)$, where a is the number of packets initially assigned to each FCFS server, b is the number of servers in each tandem, and c is the number of tandems (which is equal to the number of switches). We can see that the LB-EIT protocol performs relatively poorly; it achieves satisfactory speedup only with greater density of packets. This result is the same as those obtained for simulations with lookahead-based protocols [36], because low event activities require large number of null messages to advance the simulated time. We also see that the

[2]This set of data were obtained with the very first implementation of lookback-based protocols. Since then, we have rewritten the program several times, making some optimizations which resulted in higher processing rates. However, we did not implement the LB-EIT protocol in the newer versions due to its inferior performance.

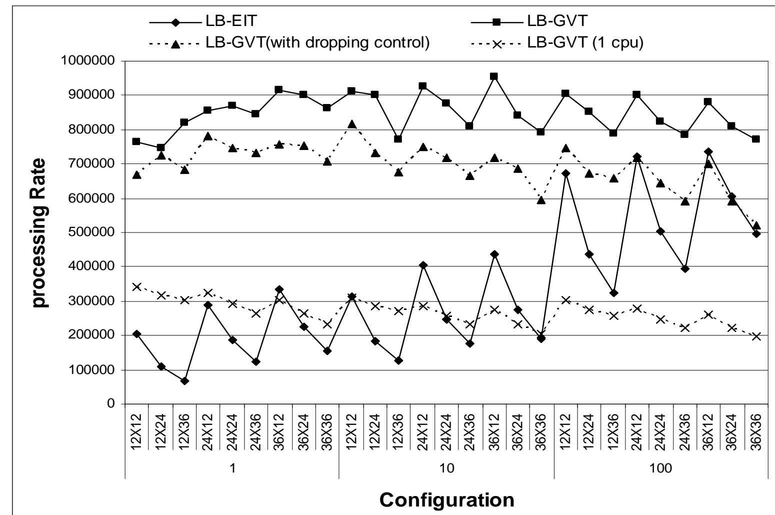

Figure 6.4: Performance of Lookback-Based Protocols for the CQN simulation.

number of switches has a more noticeable impact on the performance of LB-EIT. With more switches, there are more FCFS servers connected to one switch, thus a switch must send more null messages to the FCFS servers, making the LB-EIT protocol less efficient.

The performance of LB-GVT with dropping control also drops slightly when the packet density is high. This is to be expected because as the queue and the departure time list become larger, the cost of maintaining the initial positions for packets rises.

To see the effects of calculating the initial positions at different times, we made two changes in the simulation. The capacity of the FCFS queue was set equal to the number of packets initially in the queue, and new packets are created by a Poisson process on each queue continuously throughout the simulation. Figure 6.5 shows their performances with the LB-GVT protocol.

As mentioned before, the inefficiency of the eager evaluation comes from in-service packets that have to be discarded, for these packets affect all other packets

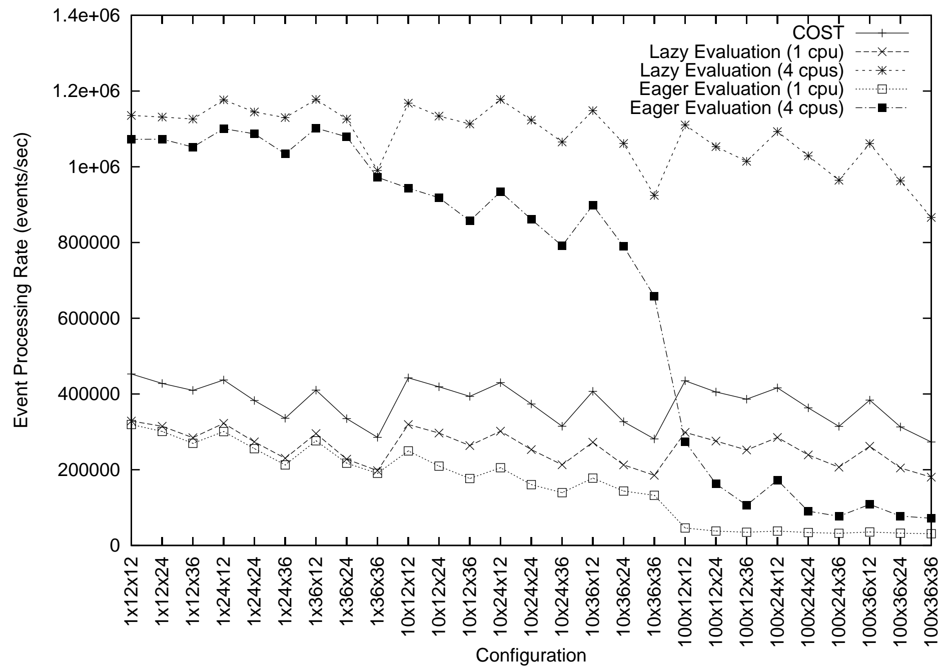

Figure 6.5: Eager and Lazy Evaluation of Initial Positions

currently in the server. The effect becomes apparent when each server can hold a large number of packets.

6.6 Conclusion

The CQN simulation was our first attempt to apply lookback-based protocols. Initial performance results were promising. On four processors we were able to obtain nearly linear speedup. This was the very first evidence that lookback-based protocols did work.

However, a head-to-head comparison with lookahead-based protocols is still missing. We realized that such a comparison should be absolutely unbiased. It is therefore more appropriate to ask other researchers who have more extensive experiences on traditional conservative protocols to conduct comparative studies. On the other hand, in the next chapter we will see more positive empirical evidences which will justify our claim that lookback-based protocols outperform lookahead-based protocols

CHAPTER 7
A HYBRID LOOKBACK-BASED PROTOCOL: MIXING LOOKBACK-ENABLED COMPONENTS WITH SEQUENTIAL COMPONENTS

The only disadvantage of lookback-based protocols is their associated modeling complexity. This was made evident by the lengthy discussion in the previous chapter about the implementation of the lookback-enabled FCFS server with finite buffer, which is a trivial problem if events are received in timestamp order.

The most effective technique to alleviate the modeling complexity is certainly the component-based approach previously discussed which allows components developed form one simulation to be reused in others. For example, a reusable FCFS server needs only to be written once and used everywhere. The modeler constructing a simulation that is composed of FCFS servers would not have to worry about the details inside the FCFS server.

Based on the component-based approach, we derived two other techniques which address the problem of modeling with lookback-based protocols. The first one is a component testing method in which the outputs of two identical components running in different modes are compared. One component runs in sequential mode, while the other in lookback mode. The same sequence of events are fed to them but events for the lookback component are scrambled so that they arrive out-of-timestamp order in the lookback component. By comparing their output events one can easily determine if the lookback component has been correctly implemented.

The second technique, a hybrid lookback-based protocol that is capable of mixing lookback-enabled components with sequential components, is the main topic of this chapter.

7.1 Boundary Components

To design the hybrid lookback-based protocol, we must understand how stragglers are created. If all time-aware components on the same processor share a single simulation clock, it is clear that only those components that reside on the processor boundary can receive stragglers from other processors. Thus, a natural idea is to 'absorb' all stragglers in the boundary components so that other intra-processor components will never observe any stragglers.

It turns out that lookahead must exist in the boundary components, in order for the intra-processor components not to receive stragglers. Consider an event e in an intra-processor component with a timestamp T'. Suppose the current simulated time of this processor is T and $T' > T$. Since all components in the same processor share the same simulation clock, all boundary components must be at the same simulated time T. If they send out in the future any messages with timestamp less than T', the event e could be affected so its execution cannot proceed. Therefore, the sufficient but not necessary condition for event e with timestamp T' to be safely executable is that all boundary components must guarantee that all messages they will send later must have a timestamp greater than or equal to T'. By definition of lookahead, this means that every boundary component must have a lookahead of at least $T' - T$.

Another requirement is that a boundary component can never output a message smaller than the current simulated time. Suppose a boundary component at simulated time T outputs a message with timestamp T'' such that $T'' < T$. It is possible that a previously processed event e in an intra-processor component contains a timestamp T''' satisfying $T'' < T''' < T$. Apparently the event e may be affected by the message at T'', which becomes a straggler in the intra-processor component.

This requirement rules out the choice of making the dropping decision when a packet is departing from the FCFS server. If the departing packet is to be dropped after the calculation of its initial position, the next packet may have an earlier departure time, given that the service time is dependent on the packet itself. The next packet would then leave at a timestamp earlier than the current simulated time, resulting in a potential straggler for other components.

7.2 Guard Events

Guard events are scheduled by boundary components to set a lower bound on the timestamp of events they will send. The simulation engine handles the guard events in the same way as regular events, so these two types of events can share the same event list. When the simulation engine sees a guard event, it dispatches the guard event to the boundary component that scheduled it. The component may or may not schedule another guard event with same or greater timestamp, depending on the estimated minimum timestamp of future events.

We can see immediately that the inclusion of guard events demands no changes on the lookback-based protocols. A regular event will be chosen only if it is the earliest event in the processor, therefore at this time there cannot be any guard events with smaller timestamp. For convenience, we refer to the LB-GVT protocol that allows for guard events as the hybrid lookback-based protocol, or the Hybrid LB-GVT protocol. The LB-EIT protocol is not considered because of its relatively poor performance.

However, there is a slight difference between guard events and regular events. The GVT computation should not take guard events into account. If this were the case, the hybrid lookback-based protocol would be deadlock-prone with a zero lookahead boundary component, because the guard event with the smallest timestamp would be chosen to execute which will in turn schedule another guard event with the same timestamp, resulting in an infinite chain of guard events. On the other hand, if the GVT is computed from the timestamps of regular events, it can be easily shown that the earliest regular event is always eligible for execution, because all guard events must have a timestamp equal to or greater than the GVT calculated in this way.

Guard events may not be always necessary. They become unnecessary when a boundary component has a regular future event which is known to be the earliest among all future events. In this case, the guard event must have a timestamp greater than that of the regular event. It can only be reached after the regular event has been processed. Therefore it is not necessary to keep the guard event while the regular event is still in the queue, and the guard event will be scheduled only after

the regular event has been processed.

7.3 Comparing Hybrid Lookback-Based Protocol with Conservative Protocol

In the FCFS server detailed in the last chapter, a guard event is needed when there are no packets in the server or all packets in the server have an arrival time greater than the GVT. It is needed in the latter case because another arriving packet preempting the current in-service packet may leave earlier than the current in-service packet. The guard event is not needed when the current in-service packet arrived before the GVT, since the departure event of such a packet cannot be affected by a straggler and therefore is the earliest among all future events. The timestamp of the guard event, if necessary, is calculated according to the following formula:

$$GT_i = max\{last\ departure\ time, GVT\} + min\{S\}$$

where $min\{S\}$ represents the minimum service time of any packets.

For comparison, we developed a similar lookahead-based protocol, referred to as LA-EIT, which exploits only the lookahead on the boundary components. Here the set of boundary components are different from those defined for the hybrid lookback-based protocol, for they are now components that may send messages to other processors, rather than components that may receive messages from other processors.

The LA-EIT protocol works as follows. Denote EOT_{ij} as the Earliest Output Time of messages any boundary components residing on the ith processor may send to the jth processor. The Earliest Input Time of the ith processor can be calculated as

$$EIT_i = min_{all\ j}\{EOT_{ji}\} = min_{all\ j}\{CLOCK_j + LAj, i\}$$

where $CLOCK_j$ is the current simulated time of the jth processor, and LAj, i is the lookahead in the boundary components between the jth and ith processor.

After the EIT is known, each processor can then execute safe events with a

timestamp smaller than the EIT. Notice this protocol is prone to deadlock if there exists a cycle of zero lookahead, as in the traditional Chandy/Misra/Bryant protocol.

Despite of the increased modeling complexity, the hybrid LB-GVT protocol has several advantages over the LA-EIT protocol :

- Guard events incur less overhead because they are scheduled and received within the same processor, whereas null messages must be sent from one processor to another.
- Guard events disappear automatically when lookahead becomes larger than the difference between the GVT and the current simulated time, whereas in the LA-EIT protocol null messages are indispensable for advancing the simulated time.
- The GVT computation is less sensitive to topology. The EIT computation requires a large number of null messages with high component connectivity.
- The hybrid LB-GVT is deadlock free even if the lookahead is zero.

7.4 CQN Simulation

Figure 7.1 shows the performance of LB-GVT, hybrid LB-GVT, and LA-EIT running on 4 processors, and of COST which is our sequential simulation tool described in Chapter 3. The LB-GVT protocol is the most stable, for it is able to exploit lookback intra-processor components when there are few packets in each server. The hybrid LB-GVT and LA-EIT protocols work better when there are abundant packets, mainly due to the fact that intra-processor components can be more efficiently implemented as sequential component. Overall, the hybrid LB-GVT protocol consistently outperforms the LA-EIT protocol.

Figure 7.2 and 7.3 show the speedup of these three protocols on 4 processors, with respect to the COST and their sequential execution respectively. The speedup of the LA-EIT protocol relative to its sequential execution is the worst. This is mainly due to the topology of the CQN network. When a packet arrives at a switch, the switch dispatches one null message for each processor to inform the changes on EOTs, even if only one processor is the destination of the packet.

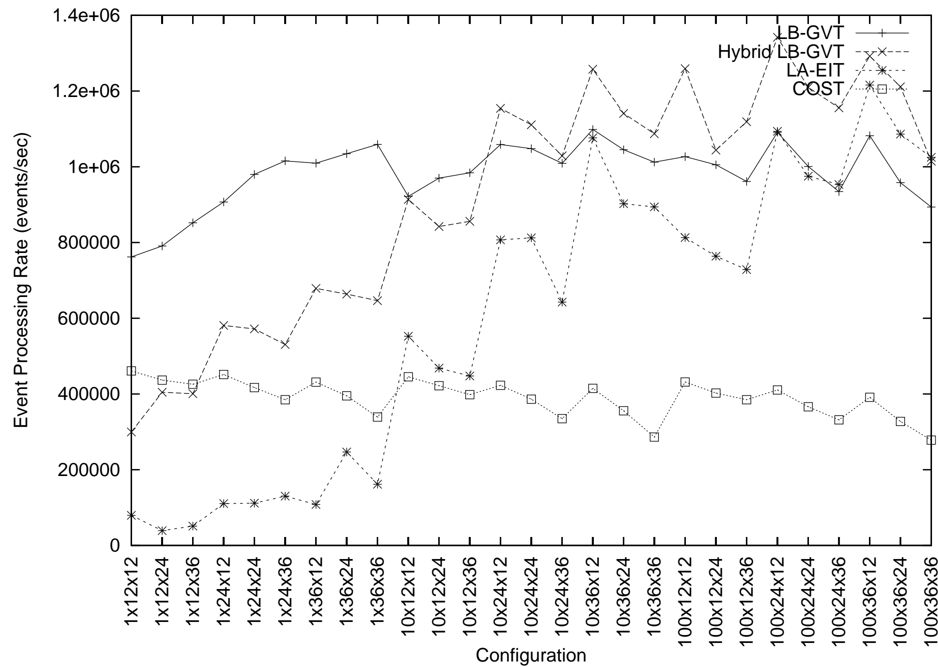

Figure 7.1: Performance of COST and Three Parallel Protocols (4 CPUs) on CQN Simulation

7.5 UDP Network Simulation

We also implemented a UDP (User Datagram Protocol) network simulation based on the Hybrid LB-GVT and the LA-EIT protocol. A UDP network consists of four types of components: routers, switches, nodes, and links. We selected a topology in which a fixed number of nodes are connected to a switch via bidirectional LAN links, and the same number of switches are connected to a router via bidirectional LAN links. Routers are connected as a ring via unidirectional WAN links. The size of the network is determined by two parameters (a, b), where a is the number of nodes connected to a switch (also the number of switches connected to a router) and b is the number of routers. Figure 7.4 shows a $(2, 3)$ UDP network, while in the experiments, the size is $(6, 12)$.

A node repeatedly generated packets at exponentially distributed intervals. The destination of a packet is selected randomly. A parameter controls the probability of a packet being forwarded to a different LAN or a different WAN. Since UDP is unreliable, links are allowed to drop packets if their internal queues are full. The capacity of the queue is equal to the product of the delay and the bandwidth

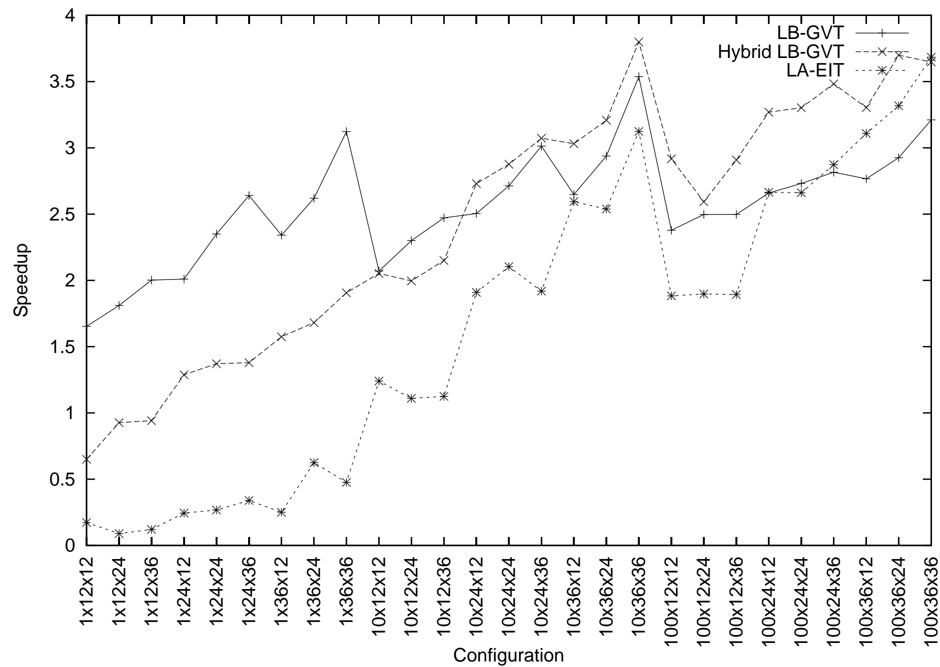

Figure 7.2: Speedup of Three Protocols Relative to COST on CQN Simulation (4 CPUs)

of the link. The larger the probability of a packet being assigned a destination node in a different WAN, the more congested the WAN links will be, and the more likely packets will be lost.

Figure 7.5 shows the performance of the hybrid LB-GVT and the LA-EIT protocol on the UDP network. In both protocols, we only exploit lookahead or lookback in the WAN links that connect routers. All node, switch and router components are run in sequential mode, so coding of them is straightforward. Again, the hybrid LB-GVT protocol performs better than the LA-EIT protocol.

In studying the speedup curves depicted in Figure 7.6, one can observe an interesting phenomenon that cannot be found in the results of the CQN experiments: the speedup of the LA-EIT protocol with respect to its sequential implementation is higher than that of hybrid LB-GVT. It means that for the UDP simulation the LA-EIT has better scalability; it is only the overhead of generating and handling of null messages that makes it less efficient. The higher speedup, indeed, results more from the property of the link, less from the simple ring connection between routers. A packet going through a link may at most trigger one null message. However, in

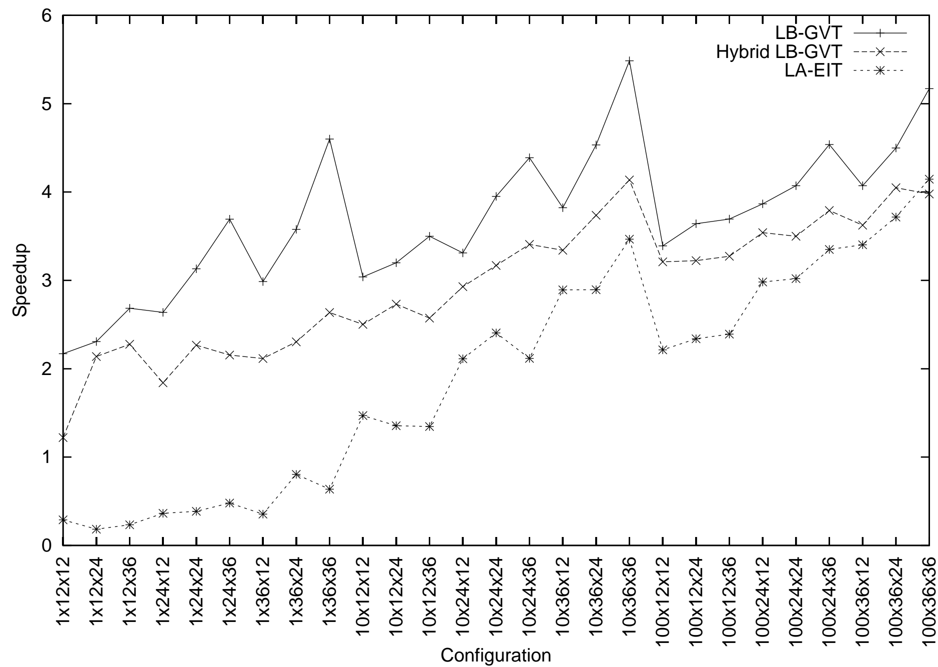

Figure 7.3: Speedup of Three Protocols Relative to Their Sequential Execution on CQN Simulation (4 CPUs)

the CQN simulation, a packet arriving at a switch may trigger one null message for each processor, regardless of the fact that it can be forwarded to only one processor.

7.6 Conclusion

In the chapter we proposed a hybrid lookback-based protocol that is able to mix up lookback-enabled components and sequential components. By only exploiting lookback in the boundary components, it relieves the burden of modeling dramatically. With a lookback-enabled link component, we can confidently claim that the hybrid lookback-based protocol can support realistic network simulations, such as those normally performed by NS.

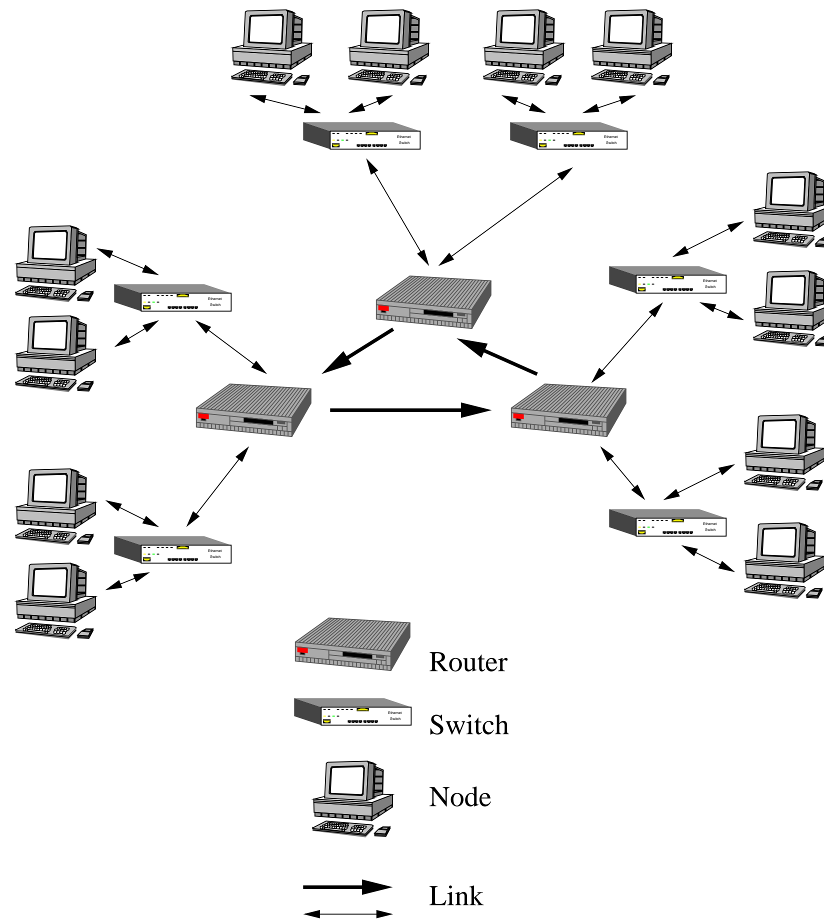

Figure 7.4: A UDP Network with a=2 and b=3

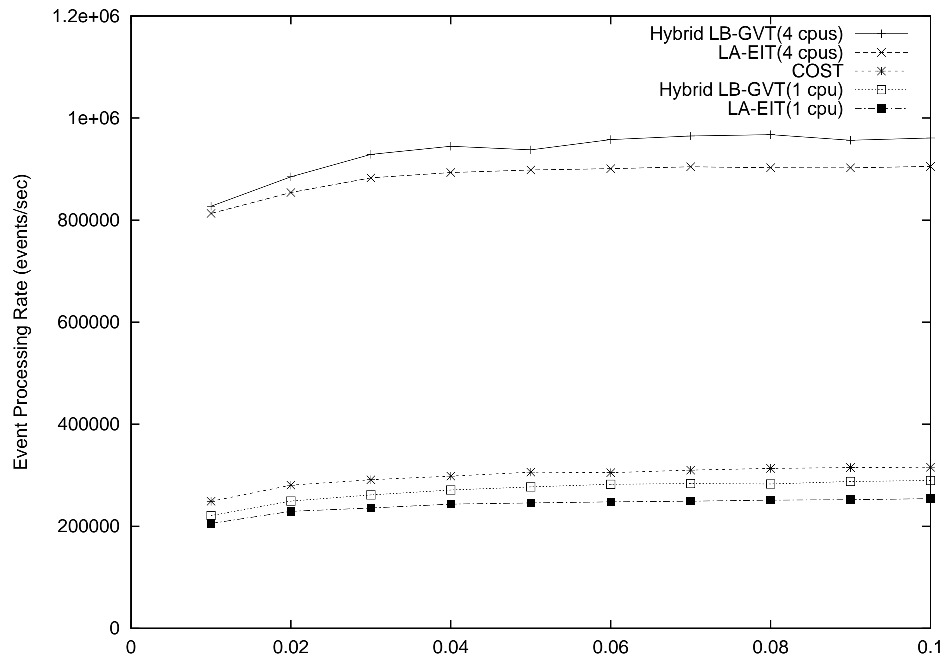

Figure 7.5: Performance of Hybrid LB-GVT and LA-EIT on UDP Network Simulation

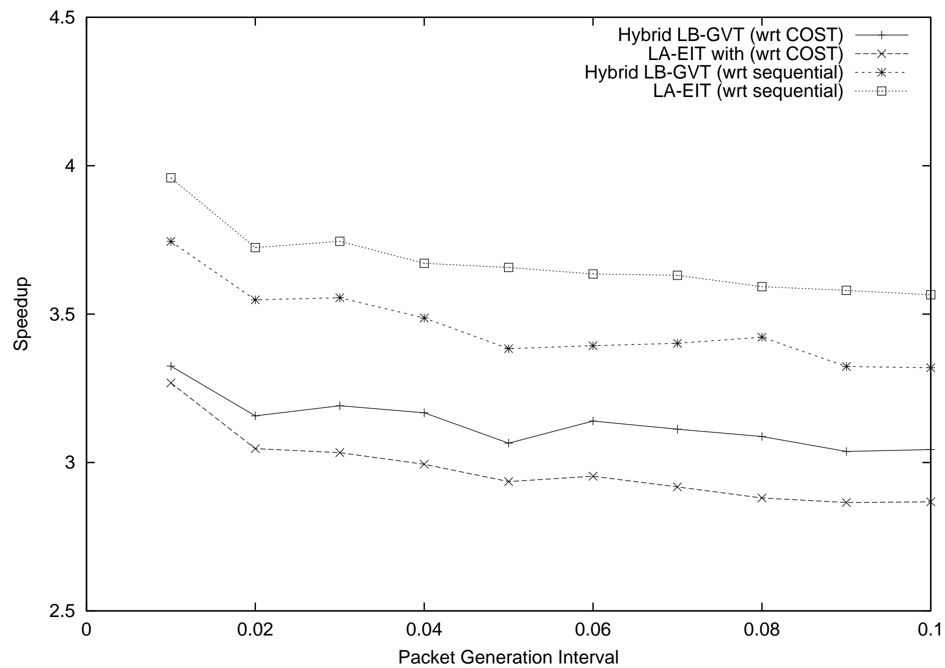

Figure 7.6: Speedup of Hybrid LB-GVT and LA-EIT on UDP Network Simulation (4 CPUs)

CHAPTER 8
PCS SIMULATION: FOUR TYPES OF LOOKBACK

In this chapter we show that all four types of lookback exist in the PCS network simulation, exploitable by either lookback-based protocols or optimistic protocols.

8.1 Exploiting Lookback in PCS Network Simulation

A classical subject of the PDES research, the PCS network simulation has attracted many researchers [14, 11, 45, 32], because it actually represents a class of more general systems where various mobile objects roaming around a geographically divided domain compete for limited amount of resources in each location. Many real systems can be abstracted in this way, so the study of PCS simulation may have influential consequences on all these systems. These systems cannot be authentically simulated by traditional lookahead-based conservative protocols, for usually the time an object spends in a location is drawn from an exponential distribution with no minimum value, which leads to in zero lookahead.

We now show that all four types of lookback exist in PCS simulation.

8.1.1 Direct Strong Lookback

The requirement for the existence of direct strong lookback is that the component must be able to process stragglers without rollbacks while maintaining the lookback constraint.

In the PCS simulation, a straggler is always a new portable moving in from a neighboring cell that has a smaller current simulated time. If this portable is not in the middle of a call, and it does not make any new call before the simulated time of the current cell, it obviously will not affect any other processed events. One the other hand, if it is in the middle of a call when moving in, or it attempts to make a new call at a time earlier than the current simulated time, it may affect some of the processed events.

Direct processing of stragglers can be implemented in the following way (Fig-

ure 8.1). Each cell maintains two lists. One is the portable list and the other the processed event list. Each event contains four fields: the timestamp, the type of the event, the pointer to the portable in which the event occurred, the number of the available channels before the event is processed. The event type could be GET or RELEASE.

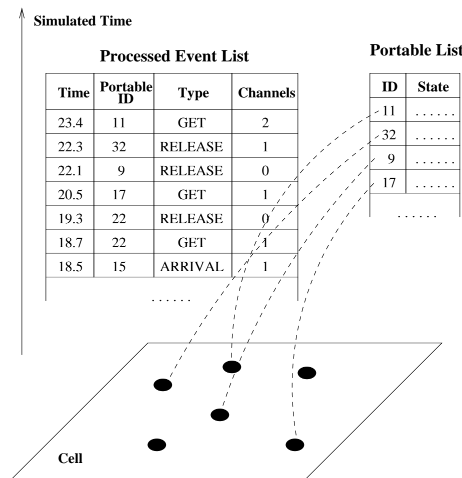

Figure 8.1: Data Structure for Exploiting Direct Weak Lookback in PCS Simulation

The last field is essential: it allows a cell to access the history of channel allocation and releases by scanning through the event list. To process a straggler, a new event is inserted into the event list, and other events with a larger timestamp are modified one by one in increasing timestamp order. Notice a change on the number of available channels may or may not invalidate the event. Only a GET event with the number of free channels changing from 0 to 1 or from 1 to 0 will become invalid. In such a case, all subsequent events occurring on the same portable must be removed from the event list, and this event must be reprocessed in order to generate correct subsequent events.

The question of determining the amount of strong lookback still remains, because it is the strong lookback upon which lookback-based protocols depends. We have to look at the absolute impact time of outgoing messages. An outgoing message in the PCS network simulation is created for a portable that wants to leave the current cell, as shown in Figure 8.2.

The absolute impact time of such a message is the last time the portable attempts to get a channel, because it is the latest point at which the portable requires any information from the cell, and any activities happening after that point would not affect the outgoing messages. Notice a RELEASE event cannot be affected by a straggler. Therefore, the timestamp of the last GET event gives the maximum amount of strong lookback possible.

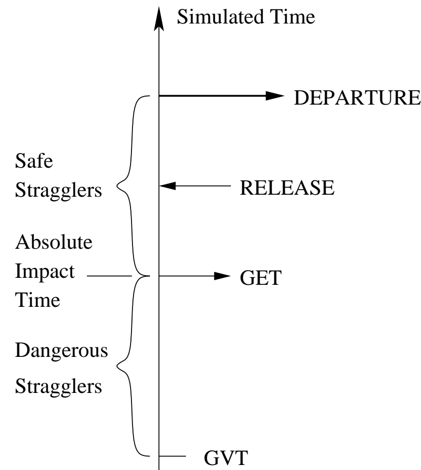

Figure 8.2: The Absolute Impact Time of a Departure Event in PCS Network Simulation

To exploit strong lookback in the LB-GVT protocol, a test is made each time before a portable leaves to see whether or not the absolute impact time is smaller than the GVT. If it is, the portable can be delivered to other cells without any problem. Otherwise, the event must be suspended until this condition is met.

8.1.2 Universal Strong Lookback

We can see that the above algorithm based direct strong lookback is quite complicated. Although the complete event history directly gives us the freedom of processing stragglers at any time, its maintenance is really a burden. We can rewrite the algorithm in a rollback and recovery style, whose structure appears to be clearer and simpler.

The processed event list is still required. However, the last field which stores the number of free channels in the event structure is no longer necessary. Instead, each cell keeps an integral variable which contains the number of available channels

at the current simulated time. When a straggler arrives, all events with a timestamp larger than that of the straggler are undone one by one in decreasing timestamp order, for the purpose to deduce the number of free channels at the time that the straggler is bound to occur. The straggler is then processed based on this number and all rolled-back events are reprocessed in increasing timestamp order.

Such a rollback and recovery method incurs redundant operations. Not all events with a larger timestamp than that of the straggler will be affected by the straggler, because when a portable gets a channel there might be plenty of them available, thus the arrival of a straggler would not matter. Keeping track of the number of available channels after each event execution would prevent unaffected events from being rolled back, but it also makes the implementation much more complicated.

8.1.3 Direct Weak Lookback

To exploit the direct weak lookback in the PCS network simulation, we define the virtual lookback time of a cell as the last simulated time at which a portable released a channel at a time when there are no free channels. As soon as a portable gets the last available channel, we set the weak lookback to zero (which means that the virtual lookback time is always equal to the current simulation time of the cell afterwards). Lookback becomes non-zero as soon as a portable release a channel, and the new virtual lookback time is the release time. Theorem 8.1 proves that a straggler that falls into the lookback window cannot affect any processed events in the lookback window. Therefore, a portable arriving from other cells in the simulation past but within the lookback window can be granted a channel, without rolling back any processed events. To preserve the correctness, we must also set the lookback to zero after processing a straggler, since a second straggler cannot always be successfully processed without affecting other events. This also confirms that the lookback is weak. It is apparent that the exploitation of direct weak lookback would definitely reduce the rollback frequency for an optimistic PCS simulation.

Theorem 8.1. *If the lookback window begins with a RELEASE event that increases the number of free channels to 1 and ends with the first GET event that decreases the*

number of free channels to 0, any single straggler arriving in the lookback window would not affect any processed event in the lookback window except the ending GET event.

Proof. Only GET events can be affected by a GET straggler. The condition for this happening is that the number of free channels before the GET event must be exactly 1. The only event that satisfies this condition in the lookback window is the ending GET event. The number of free channels before any other GET events must be at least 2, otherwise such an event would have been the ending GET event. The arrival of a GET straggler only decrease the number of free channels before these GET events by 1, so they can still obtain a channel and therefore remain unaffected.

In the case of a RELEASE straggler, there is no event in the lookback window that can be affected. A RELEASE straggler only affects a GET event that failed to get channel, which only occurs when the number of free channels is 0. However, at any instant in the lookback window the number of free channels is at least 1, because otherwise the GET event that makes this number 0 would have become the ending GET event. Therefore, A RELEASE straggler affects no event either. □

Surprisingly, one can make use of the direct weak lookback in the lookback-based protocol that is based on universal strong lookback. The idea is almost the same as in optimistic simulations. When a straggler arrives we first check to see if it falls into the lookback window. If it is, we simply process it by guaranteeing it a channel. Besides, we must set the lookback to zero and decrease the number of free channels by one. It is unnecessary to roll back any other events because they cannot be affected. If the straggler has a timestamp smaller than the virtual lookback time, we still have to resort to the rollback and recovery procedure.

8.1.4 Universal Weak Lookback

In optimistic simulation, we can take advantage of the existence of direct weak lookback to reduce rollback frequency, and the existence of universal strong lookback to avoid unnecessary anti-messages. In addition, we can exploit universal weak lookback with the notion of dynamic impact time to further reduce the number of unnecessary anti-messages.

We have known that the absolute impact time of an outgoing message in the PCS simulation is equal to the timestamp of the last GET event. By definition the dynamic impact time of the message is at least the same as the absolute impact time. Intuitively, if there were more than one channel available when the portable that is currently leaving grabbed its channel, its dynamic impact time may be smaller than the absolute impact time.

Our first guess is that the dynamic impact time should be the second last time in which the leaving portable requested a free channel. It seems that a single straggler can decrease the number of free channels at a later time by at most one:

Hypothesis 2. *A straggler can decrease the number of free channels at any later time by at most one.*

However, we can construct a counterexample to this hypothesis, as illustrated in Figure 8.3. The numbers along the simulated time axis in the figure represent the number of free channels before and after each event. Because the portable D got the only channel released by the portable E, the event at $T6$ is dependent on the event at $T8$. When a straggler affects the event at $T8$, it also changes the event at $T6$. As a result, both D and E reschedule GET events at $T2$ and $T3$ which obtain the two free channels released earlier by the portable B and C.

In fact, the correct dynamic impact time should be the timestamp of the latest RELEASE event that changes the number of free channels from 0 to 1, as shown by Figure 8.4. The RELEASE event may have occurred to the same or a different portable.

The window from the RELEASE event to the GET event in the figure is part of the lookback window given by Theorem 8.1. As a result, a straggler with a timestamp smaller than the absolute impact time but larger than the dynamic impact time will not affect any events in the lookback window except the ending GET event. The GET event shown in the Figure 8.4 is still in the lookback window starting from the RELEASE event, and it cannot be the ending GET event because after its execution there is still one free channel. Therefore, it cannot be affected by the straggler and the outgoing departure message remains unchanged.

A second straggler with a timestamp smaller than the absolute impact time

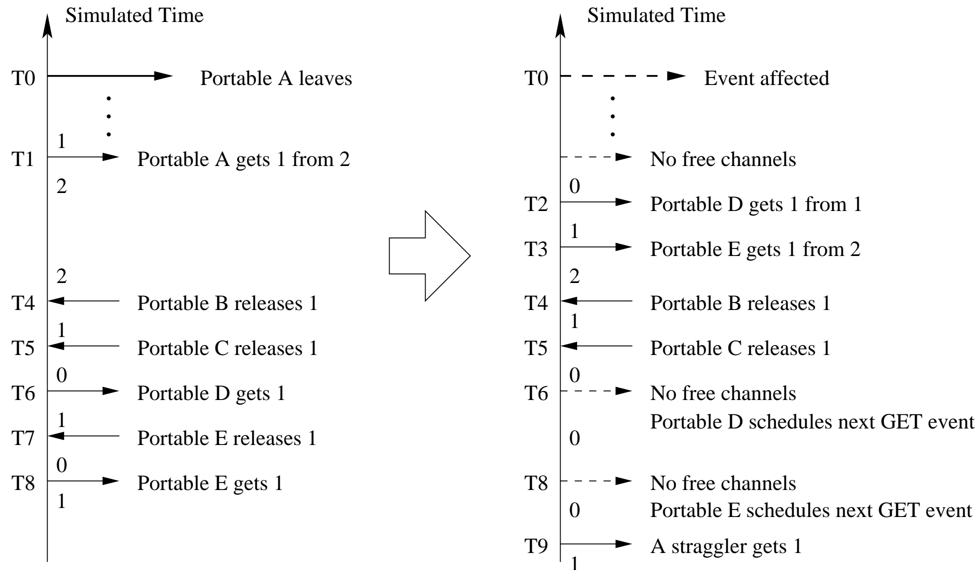

Figure 8.3: A Straggler May Affect the Number of Free Channels By TWO

but larger than the dynamic impact time cannot always be processed correctly without changing the departure message, so we must set the dynamic impact time to absolute impact time after processing the first straggler. This is the reason we call it 'dynamic'. It is also worth to note that the dynamic impact time is actually the virtual lookback time given by the direct weak lookback at the simulated time when the last GET event is being processed. This fact can be used to greatly simplify the implementation.

8.2 Performance Evaluation

Our parallel simulation platform is built as an extension of COST described in Chapter 3. We have successfully implemented two synchronization protocols: one is the lookback-based protocol LB-GVT, and the other is an optimistic protocol that utilizes Reverse Computation [15] called LBTW.

The PCS simulation that we conducted contains 256 cells. Each cell is initially assigned 16 portables. The average call time and the average idle time are 36 seconds and 18 seconds, respectively. The average time a portable stays in a cell is by default 450 seconds. The actual call time, idle time, and residence time all obey exponential distributions. All experiments run on a shared-memory computer with

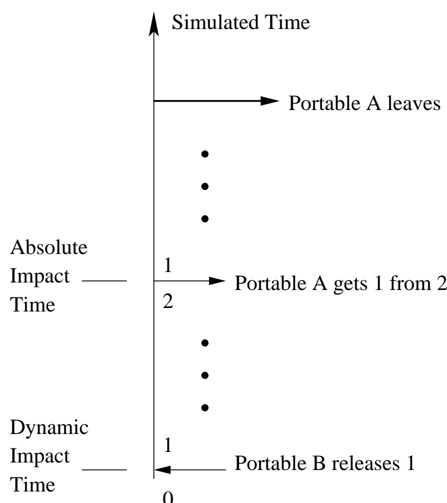

Figure 8.4: The Dynamic Impact Time of a Departure Event in PCS Network Simulation

4 Intel 500Mhz Pentium III processors. The last relevant parameter is the number of available channels in each cell. In each set of experiments, we vary this number from 2 to 16.

Figure 8.5 shows the performance of the LB-GVT protocols. Despite the fact that with universal strong lookback we may roll back processed events unaffected by a straggler, it performs better than a protocol with direct strong lookback, mainly due to a simpler design that incurs less overhead.

As we pointed out earlier, direct weak lookback can help decrease the number of rollbacks in lookback-based protocols. Figure 8.6 empirically confirmed this claim. Weak lookback is able to slightly boost the performance in all but one case where each cell possesses 10 channels. In another set of experiments, we allow a portable, when it is departing, to move to a randomly chosen cell among all cells, not only neighboring cells. Such random connections naturally increases the rollback frequency, and event process rates drop, but the performance improvement resulting from use of direct weak lookback became more significant; it increased to 3.5%.

The performance of the PCS simulation with the LBTW protocol is better than with LB-GVT protocol. This is mainly because of the different implementations of the cell model. In PCS with LB-GVT we use two events to simulate a portable (one for call activities and the other for scheduling the departure event), while with

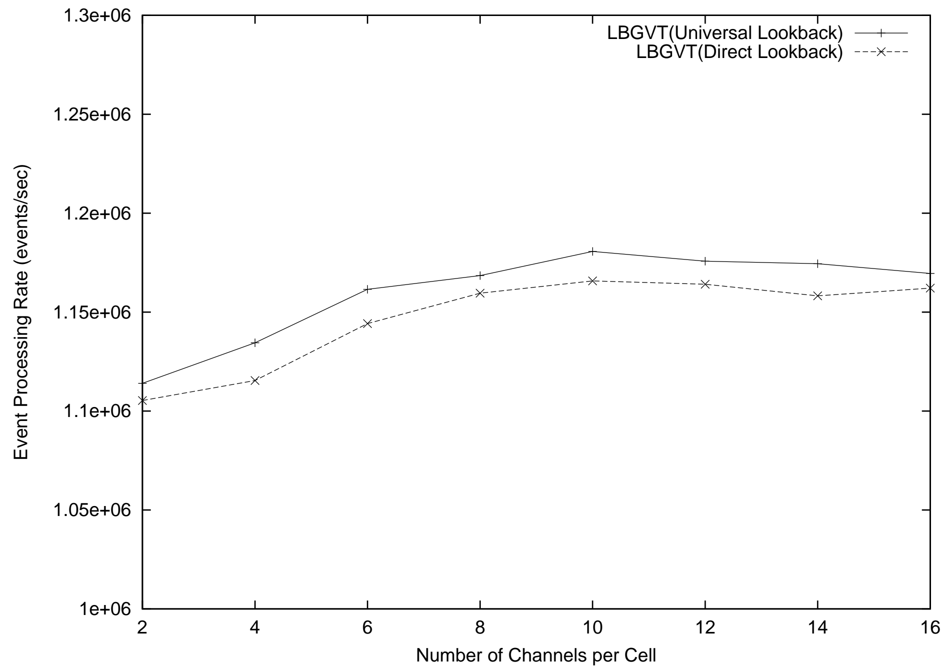

Figure 8.5: Performance of the LB-GVT protocol with Direct Strong Lookback and Universal Strong Lookback

LBTW, we use only one event.

We tested the three lookback-based optimization techniques and their combinations (random connection between cells was adopted as describe earlier). Unfortunately, only direct weak lookback showed a very small improvement (Figure 8.7). However, when we changed the average residence time from 450 to 50, we obtained positive data (Figure 8.8). Both direct weak lookback and absolute impact time can improve the performance individually, and the combination of them is even better, obtaining an average improvement of 3.7%. The inclusion of dynamic impact time seems to only drop the event processing rate due to the associated overhead. We also noticed from the data that the impact of direct weak lookback is less noticeable when there are few channels in each cell, but it continues to improve when more channels are available. This is no surprise since more free channels usually means more direct weak lookback.

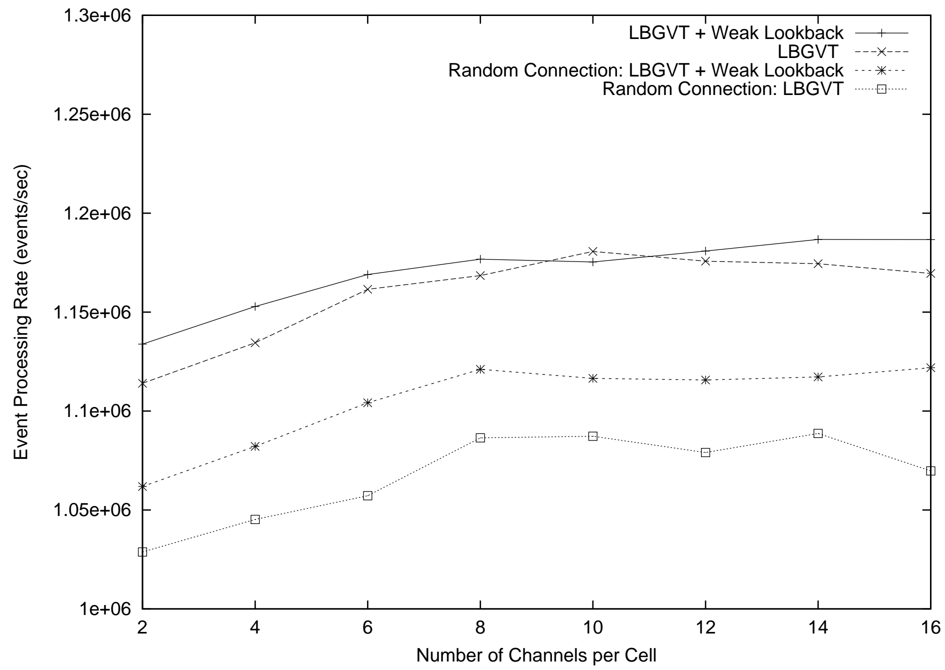

Figure 8.6: Improve the LB-GVT Protocol by Exploiting Direct Weak Lookback

8.3 Conclusion

We have shown that all four types of lookback exist in PCS simulation and discussed how to use them in either lookback-based protocols or optimistic protocols.

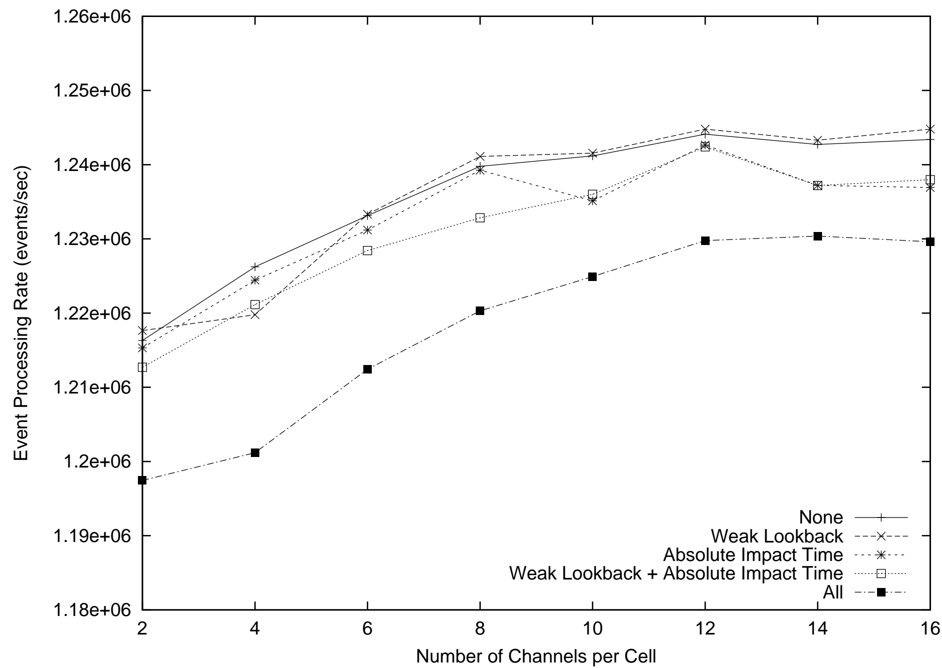

Figure 8.7: Optimistic PCS Simulation with Several Optimization Techniques (Average Residence Time = 450 seconds)

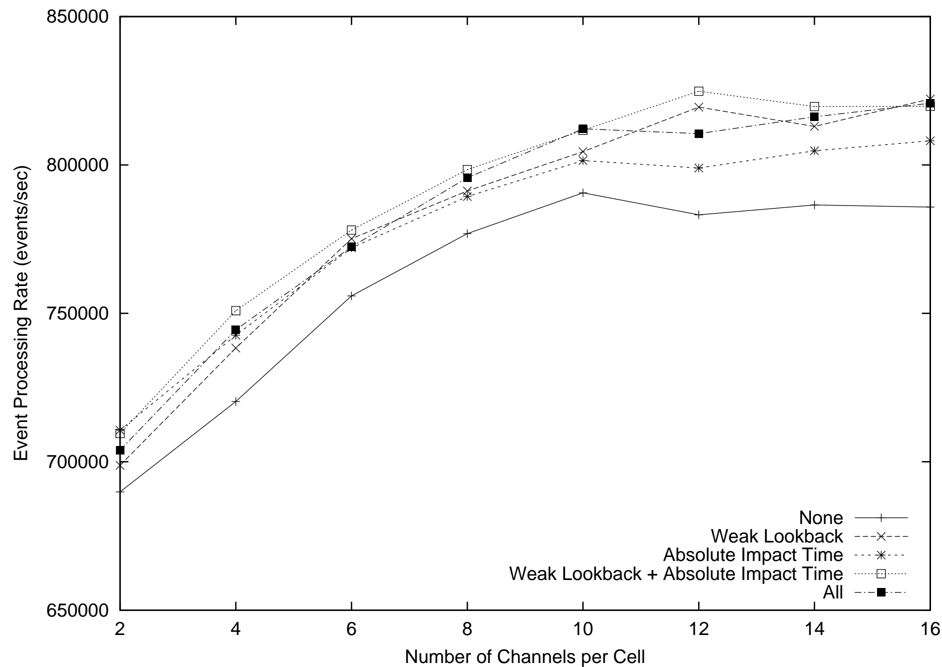

Figure 8.8: Optimistic PCS Simulation with Several Optimization Techniques (Average Residence Time = 50 seconds)

CHAPTER 9
CONCLUSIONS AND FUTURE DIRECTIONS

The following is excerpted from the statement of the panel discussion of the 16th Workshop on Parallel and Distributed Simulation held in May, 2002 (http://www.dis.uniroma1.it/~quaglia/pads2002/panel.html).

> ... The synchronization problem, however, seems largely to have been solved. Many of the recent workshops have been populated mostly with papers that are "fine-tuning" existing approaches, e.g. pushing known synchronization algorithms down into network interface cards. While applications of their synchronization algorithms are flourishing (e.g. telecommunications) these papers are presented at applications-oriented conferences (e.g. MASCOTS). PADS remains dedicated to the "core" technology/algorithm issues. As the synchronization problem becomes well-solved, the existence of PADS as annual gathering of learned researchers is threatened. Our objective in organizing the panel is to attempt to identify some challenging "core" distributed simulation technology issues that may offer the PADS research community a "life after synchronization".

The discovery of lookback proved that the study of PDES synchronization was far from complete. The implications of lookback are three-fold. For conservative simulations, the hybrid lookback-based protocol provides an approach with less implementation overhead than traditional lookahead-based protocols to link components running concurrently on different processors. This new protocol is suitable for applications where optimistic models are too complicate to develop, such as communication network simulation.

For optimistic simulations, lookback enables various optimization techniques more efficient than lazy cancellation to reduce runtime overhead by avoiding unnecessary rollbacks and anti-messages. The distinction between weak and strong

lookback, direct and universal lookback are fundamental in devising these optimization techniques: weak lookback results in less anti-messages while direct lookback reduces rollbacks. They also explain why lazy cancellation is an effective technique.

Finally, strong-lookback-based protocols fit perfectly into local rollback protocols, the third class of optimization protocols, which has been proposed but whose applications are rather restricted. Although strong-lookback-based protocols may still take a rollback and recovery style, they do not require anti-messages, which makes the implementation of the simulation engine much easier than that of the optimistic protocols.

With lookback, is the PDES synchronization a solved problem? We believe that one should be cautious to answer this question. We are more confident to claim that the PDES research is now more complete that ever before. Therefore, the focus of the study should shift to the applications of the PDES technology, to the development of more efficient modeling methodology.

The component-port model for simulation proposed in this thesis established a solid basis for an efficient, hierarchical, scalable simulation architecture. The implementation work, however, remains largely incomplete. It should be the emphasis of our research in the near future. Any other realization of PDES synchronization protocols must be carried out within this architecture.

9.1 Opportunities for PDES

Despite more than twenty years' research, the future of PDES remains uncertain and doubtful. Its real-world applications still seem to be quite limited, most of which are conducted by only PDES researchers. Recent years have seen a decrease in the number of PDES publications, and the submission and audience to the PADS conference have shrunk to such an extent that the conference committee decided to extend its scope to include sequential simulation.

We believe that PDES is still an indispensable means of conducting the simulation of complex large-scale systems, just as parallel computing is an essential part of high performance computing. The only obstacle is, the complexity required by various synchronization protocols scares away ordinary users. Although PDES

researchers have been engaged in the effort to automate the process of building parallelizable models, such techniques are still far from usable in practice. From what is learned in the field of parallel computing, one can boldly claim that automatic parallelization would eventually attain only a modest degree of success, and efficient parallelization of most PDES applications may still rely on the human beings' programming ability.

For ordinary users who wish to use the PDES technique to study the systems they are interested in, they can only hope that the complexity of PDES programming can be reduced to a minimum by the PDES research. Here, the ease of modeling is at least as important as, if not more important than, the parallel simulation speed. Therefore, in order to popularize the practice of PDES among non-PDES researchers, we should shift our focus from designing efficient algorithms to designing modeling methodologies that enable efficient simulation.

On the other hand, we should also realize that in some extreme cases, the performance of a PDES application is the only concern. Programming complexity is no longer an issue, since the problem is so fundamental that it can be solved at whatever cost it requires. Such extreme cases are not rare in real worlds. For instance, neural network simulation may be the only way to uncover the secret inside human brain. Neural networks, especially a variant referred to as pulsed neural network which recently attracted a considerable amount of interest, is inherently discrete, and therefore can be studied using the standard PDES technique. Other examples include logic simulation and security simulation.

9.2 Future Directions

The aforementioned three kinds of PDES synchronization protocols resulting from the discovery of lookback are applicable in different scenarios. The hybrid lookback-based protocol, combined with the component-based approach, is well suited for applications in which people want to spend little effort in parallelization. Sequential simulation models can be directly plugged into this protocol, while parallelism is only exploited in predefined models, such as FCFS servers or communications links. Simulations are normally executed on parallel computers with less

than ten processors, and parallel performance is not particularly sought, but should be significantly higher than sequential performance.

Both optimistic protocols and strong lookback-based protocols work well for shared memory computers. In fact, we guess that this is the reason why lookback-based optimization techniques only obtained a modest or even no improvement. Strong lookback-based protocols are more attractive that traditional optimistic protocols, because there is no need to handle anti-messages. The low communication latency between processors on shared memory computers determines that the delay of message sending would unlikely cause any catastrophic effect. Performance is more important than the modeling methodology, and in spite of the scale of the simulation, there are only a small number of underlying models. Programmers can afford to build specialized components in order to achieve a nearly ideal speedup.

For extremely large-scale simulations, optimistic protocols definitely win, but lookback-based optimization is indispensable for the ultimate success of PDES. Moreover, the impact time of a message provides a rough estimate on the likelihood of of a message being cancelled. The smaller the impact time, the less unlikely that the message will be affected by a straggler. To minimize the number of anti-messages, the impact time of an inter-processor message can be used to indicate whether to send out or to hold the message.

We now list problems that need to be solved in each of these three directions:

- The CORSA simulation architecture will play a critical role in developing a good component-based modeling methodology. The discovery of lookback completes the theoretical foundation of CORSA, and what is left undone is merely the implementation. The only problem we can foresee is the design of interface. Different from general component-based approaches, a CORSA component must expose two interfaces, one for interaction with other components, and the other for interaction with the simulation engine. How to express these two highly interrelated interfaces in order to follow the idea of a hierarchical modeling process is still unclear to us.

- So far, lookback is studied in simulations running on shared memory computers. The performance of lookback-based protocols on other platforms with sig-

nificantly larger communication latency remains unknown. Furthermore, the biggest difficulty of developing a distributed parallel simulation is the GVT algorithm. Few distributed GVT algorithms have been devised and their performance is in doubt. Beside, we wish to study the performance of distributed optimistic protocols by the means of theoretical analysis, which is even more difficult due to the lack of a rollback model.

- Identifying applications for extremely large-scale PDES simulation is time-consuming and limited by the availability of physical resources. A dilemma usually encountered by PDES researchers is that in order to apply the PDES technique to a problem in an unfamiliar discipline, it often takes lots of time to learn relevant knowledge, otherwise the effort would be infertile. Even if we can succeed in this step, we always face other adverse factors, such as no access to supercomputers, lack of interests in the simulation results.

All these problems will be the focus of my research in the near future. I believe that they all can be solved and their solutions may advance the PDES research to the next level.

BIBLIOGRAPHY

[1] High level architecture. http://hla.dmso.mil. Defense Modeling and Simulation Office.

[2] CORBA scripting language v1.0. http://www.omg.org/technology/documents/formal/corba_script_language_mapping.htm, 1997. Object Management Group.

[3] The common object request broker: Architecture and specification. ftp://ftp.omg.org/pub/docs/formal/99-10-07.pdf, 1999. Object Management Group.

[4] CORBA component v3.0. http://www.omg.org/technology/documents/formal/components.htm, 2002. Object Management Group.

[5] Tom Armstrong. *ATL Developer's Guide*. M&T Books, second edition, 2000.

[6] Matthew H. Austern. *Generic Programming and the STL*. Addison Wesley Longman, 1998.

[7] Felix Bachman et al. Technical concepts of component-based software engineering. Technical report, Carnegie Mellon Software Engineering Institute, 2000.

[8] Rajive L. Bagrodia and Wen-Toh Liao. Transparent optimizations of overheads in optimistic simulations. In *Proceedings of the 1992 Winter Simulation Conference*, pages 637–645, 1992.

[9] Rajive L. Bagrodia and Wen-Toh Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on software Engineering*, pages 225–238, April 1994.

[10] Sean Baker. *CORBA Distributed Objects Using Orbix*. ACM Press, 1997.

[11] Azzedine Boukerche et al. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 166–172, 1999.

[12] R. E. Bryant. Simulation of packet communications architecture computer systems. Technical report, Massachusetts Institute of Technology, 1979.

[13] Christopher D. Carothers, David Bauer, and Shawn Pearce. ROSS: A high-performance, low memory, modular Time Warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 53–60, 2000.

[14] Christopher D. Carothers et al. Distributed simulation of large-scale PCS networks. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 2–6, 1994.

[15] Christopher D. Carothers, Kaylan S. Perumall, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 126–135, 1999.

[16] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, TSE 5(5):440–452, 1979.

[17] Gilbert Chen and Boleslaw K. Szymanski. Lookback: A new way of exploiting parallelism in discrete event simulation. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pages 153–162, 2002.

[18] Gilbert Chen, Boleslaw K. Szymanski, and Thomas Caraco. Multiparadigm simulations in modeling spread of lyme disease. In *2000 European Simulation Multi-Conference*, 2000.

[19] Paul C. Clements. From subroutines to subsystems: Component-based software development. *The American Programmer*, November 1995.

[20] Bruce A. Cota and Robert G. Sargent. A modification of the process interaction world view. *ACM Transactions on Modeling and Computer Simulation*, pages 109–129, April 1992.

[21] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The DoD high level architecture: An update. In *Proceedings of the 1998 Winter Simulation Conference*, pages 797–804, 1998.

[22] Ewa Deelman and Boleslaw Szymanski. Breadth-first rollback in spatially explicit simulations. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 124–131, 1997.

[23] Ewa Deelman and Boleslaw Szymanski. Simulating spatially explicit problems on high performance architecture. *Journal of Parallel and Distributed Computing*, pages 446–467, 2002.

[24] Ewa Deelman, Boleslaw Szymanski, and Thomas Caraco. Simulating Lyme disease using parallel discrete event simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 1191–1198, 1996.

[25] P. Dickens and P. Reynolds. SRADS with local rollback. In *Proceedings of SCS Multiconference on Distributed Simulation*, pages 161–164, 1990.

[26] P. A. Fishwick. SIMPACK: Getting started with simulation programming in c and c++. In *Proceedings of the 1992 Winter Simulation Conference*, pages 154–162, 1992.

[27] Richard Fujimoto. Performance measurements of distributed simulation strategies. In *Proceedings of Distributed Simulation Conference*, pages 14–20, 1988.

[28] Richard M. Fujimoto. Parallel discrete event simulation. *Communication of the ACM*, pages 30–53, October 1990.

[29] Richard M. Fujimoto et al. Georgia Tech Time Warp programmer's manual for distributed network of workstations. Technical report, Georgia Institute of Technology, 1997.

[30] A. Gafni. Rollback mechanisms for optimistic distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 61–67, 1988.

[31] F. Gomes et al. SIMKIT: A high performance logical process simulation class library in c++. In *Proceedings of the 1995 Winter Simulation Conference*, pages 706–713, 1995.

[32] A. Greenberg, B. Lubachevsky, D. Nicol, and P. Wright. Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communication. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 187–194, 1994.

[33] M. A. Gunter. Understanding supercritical speedup. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 81–87, 1994.

[34] David Jefferson and Peter Reiher. Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, pages 159–168, 1991.

[35] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, pages 404–425, July 1985.

[36] Vikas Jha and Rajive L. Bagrodia. Transparent implementation of conservative algorithms in parallel simulation languages. In *Proceedings of the 1993 Winter Simulation Conference*, pages 677–686, 1993.

[37] P. J. Kiviat. *Computer Simulation Experiments with Models of Economic Systems*, chapter Simulation Languages. Wiley, 1971.

[38] Frederick Kuhl et al. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture.* Prentice Hall, 1999.

[39] UCLA Parallel Computing Laboratory. Parsec user manual. http://pcl.cs.ucla.edu/projects/parsec.

[40] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, 1982.

[41] Hong Va Leong and Divyakant Agrawal. Semantics-based time warp protocols. Technical Report TRCS93-10, Department of Computer Science, University of California, Santa Barbara, 1993.

[42] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[43] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communication of ACM*, 32(1):111–123, 1989.

[44] Michael Malak. A software component framework for HLA tools. In *1998 Fall Simulation Interoperability Workshop*, 1998.

[45] Brian A. Malloy and Albert T. Montroy. A parallel distributed simulation of a large-scale PCS network: Keeping secrets. In C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, editors, *Proceedings of the 1995 Winter Simulation Conference*, pages 571–578, 1995.

[46] Edward Mascarenhas et al. Minimum cost adaptive synchronization: Experiments with the Parasol system. *ACM Transactions on Modeling and Computer Simulation*, 8(4):401–430, 1998.

[47] Bertrand Meyer. Every little bit counts: Toward more reliable software. *Computer*, pages 131–133, November 1999.

[48] Bertrand Meyer. On to components. *Computer*, pages 139–143, January 1999.

[49] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison Wesley, 1996.

[50] David Nicol and Jason Liu. The dark side of risk (what your mother never told you about time warp). In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 188–195, 1997.

[51] David M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks. *SIGPLAN Not.*, 23(9):124–137, 1988.

[52] Ernest H. Page. The rise of web-based simulation: Implications for the high level architecture. In *1998 Winter Simulation Conference Proceedings*, pages 1663–1668, 1998.

[53] Kalyan S. Perumalla, Richard M. Fujimoto, and Andrew T. Ogielski. MetaTeD - a meta language for modeling telecommunication networks. Technical report, Georgia Institute of Technology, 1996.

[54] Michael Pidd. *Computer Simulation in Management Science*. Wiley, third edition, 1992.

[55] Michael Pidd and Ricardo A. Cassel. Taking cues from Java. *IEEE Potential*, pages 11–15, February/March 2000.

[56] Francesco Quaglia and Roberto Baldoni. Exploiting intra-object dependencies in parallel simulation. *Information Processing Letters*, 70(3):119–125, 1999.

[57] P. Reynolds, C. Weight, and J. Filder. Comparative analyses of parallel simulation protocols. In *Proceedings of the Winter Simulation Conference*, pages 671–679, 1989.

[58] H. Schwetman. CSIM: A c-based, process-oriented simulation language. In *Proceedings of the 1986 Winter Simulation Conference*, pages 387–396, 1986.

[59] Krishnan Seetharaman. The CORBA connection. *Communication of the ACM*, pages 34–36, October 1998.

[60] K. S. Shanmugan, V. S. Frost, and W. Larue. A block-oriented network simulator. *Simulation*, February 1992.

[61] Lisa M. Sokol et al. MTW: An empirical performance study. In *Proceedings of the 1991 Winter Simulation Conference*, pages 557–563, 1991.

[62] S. Srinivasan and R. F. Reynolds. Super-criticality revisited. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 130–136, 1995.

[63] J. Steinman. Breathing time warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 109–118, 1993.

[64] Boleslaw Szymanski and Thomas Caraco. Spatial analysis of vector-borne disease: A four species model. *Evolutionary Ecology*, 8(3):299–314, 1994.

[65] Boleslaw K. Szymanski. *Parallel Functional Language and Compilers*, chapter EPL–Parallel Programming with Recurrent Equations. ACM Press, 1991.

[66] P. A. Wilsey, A. C. Palaniswamy, and S. Aji. Rollback relaxation: A technique for reducing rollback costs in an optimistically synchronized simulation. In *International Conference on Simulation and Hardware Description Languages*, pages 143–148, 1994.

[67] Bernard P. Zeigler. *Object-oriented Simulation with Hierarchical, Modular Models*. Academic Press, 1990.