Cloning mechanisms for high level architecture based distributed simulation

Chen, Dan

2006

Chen, D. (2006). Cloning mechanisms for HLA-based distributed simulation. Doctoral thesis,
Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2469>

<https://doi.org/10.32657/10356/2469>

Nanyang Technological University

Downloaded on 26 Apr 2023 21:19:41 SGT

CLONING MECHANISMS FOR HIGH LEVEL ARCHITECTURE BASED DISTRIBUTED SIMULATION

CHEN DAN

SCHOOL OF COMPUTER ENGINEERING
NANYANG TECHNOLOGICAL UNIVERSITY

2006

CLONING MECHANISMS FOR HIGH LEVEL
ARCHITECTURE BASED DISTRIBUTED SIMULATION

Dan Chen

2006

Cloning Mechanisms for High Level Architecture Based Distributed Simulation

Chen Dan

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Doctor of Philosophy

2006

Abstract

This thesis investigates novel control mechanisms for large-scale distributed simulation based on simulation cloning. Distributed simulation cloning technology is designed to analyze alternative scenarios of a distributed simulation concurrently within the same simulation execution session. One important goal of the technology is to optimize execution by avoiding repeated computation amongst independent scenarios.

The foundation theory of distributed simulation cloning is established in this thesis. The theory defines the concept of a decision point to trigger cloning and the scenario trees which represent the concurrent scenarios. Types of cloning are classified according to the conditions under which cloning of an individual simulation component is required and the different mechanisms to replicate scenarios.

This thesis investigates the issues involved in cloning based on the High Level Architecture (HLA), an IEEE standard for distributed simulation. Alternative candidate solutions are proposed and compared from both the qualitative and quantitative point of view. A middleware approach is adopted to hide the complexity of simulation cloning.

The simulation models that form a distributed simulation are referred to as federates in the HLA and these interact using the Runtime Infrastructure (RTI). The idea of decoupling the Local RTI Component from a normal HLA federate is introduced. The decoupled federate architecture forms the basis to ensure the correct replication of federates. At the same time, it provides user transparency and reusability to legacy federate codes.

In order to ensure correctness and optimize the performance, this thesis describes a solution for managing the concurrent scenarios in the whole cloning-enabled distributed simulation. It also presents the algorithms for managing distributed simulation cloning, which ensure the state consistency of the overall simulation. It is desirable to replicate

federates only when strictly necessary and share execution amongst scenarios as much as possible. This thesis discusses two cloning mechanisms for HLA-based distributed simulations, namely the entire mechanism and the incremental mechanism.

Moreover, this thesis also discusses other important uses of the decoupled architecture. A fault-tolerant model is suggested to support runtime robustness in the HLA. The architecture can also be exploited to facilitate a Web or Grid Enabled architecture and load balancing.

Finally, an example of a simple supply-chain simulation indicates that the proposed approaches provide correct distributed simulation cloning and can effectively reduce the execution time for evaluating different scenarios of distributed simulations. The experimental results also indicate that the incremental cloning mechanism significantly surpasses entire cloning in terms of execution efficiency.

Acknowledgements

I would like to express my deepest gratitude to my project supervisor, Dr. **Stephen John Turner**. This erudite and perspicacious scholar gives me massive support in faith, with great patience, throughout the course of this research project. Under his guidance, I benefit vastly ranging from English skill to research attitude and methodology. His true heart to humanity and zealousness to research work convince me that this material world is still full of sincerity and warmth.

Special thanks to Dr. **Cai Wentong** for his unreserved contributions and ideas. Through him, I have acquired countless knowledge in this area. I am grateful to his help and guidance for nearly five years since I first came to Singapore. I have been constantly inspired by his intelligence and achievements.

Dr. **Lee Bu Sung**, the strict and knowledgeable expert, led me to the domain of distributed computing and simulation. His indoctrination will accompany me in a long way. I will never forget his generosity to me and his other students.

Dr. **Wei Junhu**, the friend and colleague, thank you so much for your great help during this research work.

Most of the time when I compose this thesis, my wife is faraway from me. However, her support is everywhere surrounding me. Everything I do here, I do it for you.

Table of Contents

Abstract.....	I
Acknowledgements	III
Table of Contents.....	IV
Table of Figures	VIII
Table of Tables.....	XI
Chapter 1 Introduction	1
1.1. Background	1
1.2. Motivation	3
1.3. Objectives and Scope.....	7
1.4. Organization of the Thesis	8
Chapter 2 Literature Review	11
2.1. High Level Architecture and Runtime Infrastructure	11
2.2. Cloning and Replication.....	16
2.2.1. Cloning in Programming Language	16
2.2.2. Data Replication in Distributed Systems	17
2.2.3. Agent Cloning in Multi-agent Systems.....	18
2.2.4. Object Replication in Parallel Object-oriented Environments.....	18
2.2.5. Fault Tolerance using Replication.....	19
2.3. Simulation Cloning	19
2.3.1. Cloning in Rare Event Simulations.....	20
2.3.2. Multitrajectory Simulations	21
2.3.3. Cloning in Simulation Software Packages.....	22
2.3.4. Parallel Simulation Cloning	22

2.3.5. Cloning of HLA-compliant Federates.....	23
2.3.6. Fault Tolerant Distributed Simulation.....	24
2.4. Summary	25
Chapter 3 Theory and Issues in Distributed Simulation Cloning.....	27
3.1. Decision Points.....	27
3.2. Active and Passive Cloning of Federates.....	28
3.3. Entire versus Incremental Cloning	28
3.3.1. Shared Clones.....	29
3.3.2. Theory and Issues in Incremental Cloning.....	30
3.4. Scenario Tree.....	33
3.5. Summary	36
Chapter 4 Alternative Solutions for Distributed Simulation Cloning ...	37
4.1. Single-federation Solution versus Multiple-federation Solution.....	37
4.2. DDM versus Non-DDM in Single-federation Solution	39
4.3. Middleware Approach	41
4.4. Benchmark Experiments and Results	43
4.4.1. Experiment Design.....	43
4.4.2. Benchmark Results and Analysis	46
4.4.3. Comparing Alternative Cloning Solutions using TSO Federates	47
4.4.4. Comparing Alternative Cloning Solutions using RO Federates.....	49
4.4.5. Comparing Alternative Cloning Solutions using Time Advancement Benchmark Federates.....	50
4.5. Summary	51
Chapter 5: A Decoupled Federate Architecture	53
5.1. Problem Statement.....	53
5.2. Virtual Federate and Physical Federate	55

5.3. Inside the Decoupled Architecture	58
5.4. Federate Cloning Procedure	63
5.5. Benchmark Experiments and Results	65
5.5.1. Experiment Design.....	65
5.5.2. Latency Benchmark Results	67
5.5.3. Time Advancement Benchmark Results	69
5.6. Summary	70
Chapter 6: Managing Scenarios.....	71
6.1. Problem Statement.....	71
6.2. Recursive Region Division Solution	74
6.3. Point Region Solution	78
6.4. Summary	81
Chapter 7: Algorithms for Distributed Simulation Cloning	83
7.1. Overview of Simulation Cloning Infrastructure.....	83
7.2. Active Simulation Cloning.....	87
7.3. Passive Simulation Cloning	93
7.4. Mapping Entities.....	96
7.5. Incremental Distributed Simulation Cloning	98
7.5.1. Illustrating Incremental Simulation Cloning	99
7.5.2. Managing Shared Clones.....	101
7.6. Summary	106
Chapter 8 Experiments and Results	107
8.1. An Application Example.....	107
8.2. Configuration of Experiments	108
8.3. Correctness of Distributed Simulation Cloning	109

8.4. Efficiency of Distributed Simulation Cloning	111
8.5. Scalability of Distributed Simulation Cloning	113
8.6. Optimizing the Cloning Procedure	115
8.7. Summary	119
Chapter 9 Exploiting the Decoupled Federate Architecture	120
9.1. Fault Tolerance	120
9.1.1. Background and Motivation	120
9.1.2. Fault-tolerant Model	122
9.2. Dealing with In-transit Events	125
9.2.1 Scheme for Recovering In-transit Events	127
9.2.2. Fossil Collection.....	128
9.3. Other Uses of the Decoupled Federated Architecture	132
9.3.1. Web or Grid Enabled Architecture.....	132
9.3.2. Load Balancing.....	135
9.4. Summary	137
Chapter 10 Conclusions and Future Work	138
10.1. Achievements	138
10.2. Future Work	143
10.3. Summary	144
References:	145
Appendix: Papers Published from this Research.....	154

Table of Figures

Figure 1.1: A simple manufacturing simulation	4
Figure 2.1: Functional view of an HLA federation	11
Figure 2.2: Overview of HLA federation and RTI.....	13
Figure 3.1. Entire Cloning vs. Incremental Cloning	29
Figure 3.2. A typical shared clone.....	30
Figure 3.3. Relationship between terms related to shared clones	33
Figure 3.4: Example of incremental cloning and scenario tree.....	34
Figure 4.1: Example of Single-federation Solution and Multiple-federation Solution	38
Figure 4.2: Example of routing space and regions.....	40
Figure 4.3: Example of middleware for cloning	42
Figure 4.4: Example of RTI++ implementation.....	42
Figure 4.5: Test bed for MF solution.....	44
Figure 4.6: Test bed for DSF & NDSF solution	44
Figure 4.7: Execution time comparison between cloning solutions using TSO federates	47
Figure 4.8: CPU Utilization comparison between cloning solutions using TSO federates.....	47
Figure 4.9: Execution time comparison between cloning solutions using RO federates	49
Figure 4.10: CPU Utilization comparison between cloning solutions using RO federates.....	49
Figure 4.11: Execution time comparison between cloning solutions using Time Advancement federates	51
Figure 4.12: CPU utilization comparison between cloning solutions using Time Advancement federates	51
Figure 5.1: Abstract model of a simulation federate	53
Figure 5.2: Decoupled federate architecture	56
Figure 5.3: Executing an RTI call in the decoupled architecture.....	58

Figure 5.4: Messaging between virtual federate and physical federate.....	60
Figure 5.5: Conveying callbacks to the virtual federate.....	61
Figure 5.6: Saving RTI states.....	64
Figure 5.7: Federate cloning	64
Figure 5.8: Architecture of benchmark experiments for decoupled federate architecture	66
Figure 5.9: Latency benchmark on decoupled federate vs. normal federate with payload of 100 bytes.....	68
Figure 5.10: Latency benchmark on decoupled federate vs. normal federate with payload of 1000 bytes.....	68
Figure 5.11: Latency benchmark on decoupled federate vs. normal federate with payload of 10,000 bytes.....	69
Figure 5.12: Time Advancement benchmark on decoupled federate vs. normal federate	69
Figure 6.1: Formation of a scenario tree.....	72
Figure 6.2: An example of a scenario tree	73
Figure 6.3: Development of binary scenario tree.....	75
Figure 6.4: Binary scenario tree and scenario region code.....	75
Figure 6.5: Coding the scenario tree.....	79
Figure 6.6: Point region allocation flow	80
Figure 7.1: RTI++ and internal modules	84
Figure 7.2: Cloning Manager Module	85
Figure 7.3: Active simulation cloning	87
Figure 7.4: Replicating states for new clones	91
Figure 7.5: Coordinating federates in state replication.....	91
Figure 7.6: Passive simulation cloning	94
Figure 7.7: Mapping RTI entities	97
Figure 7.8: A distributed simulation example.....	99

Figure 7.9: Executing simulation with incremental cloning	100
Figure 7.10: Internal design of the Callback Processor.....	101
Figure 7.11: Checking sensitive events	103
Figure 7.12: Processing events in Pending-Passive-Cloning.....	105
Figure 8.1: A simple distributed supply chain simulation.....	107
Figure 8.2: Execution time for examining multiple scenarios	113
Figure 8.3: Percentage of saved execution time using entire and incremental cloning ..	113
Figure 8.4: Initial federation for the n th set of experiments for the scalability test.....	114
Figure 8.5: Execution time for examining three policies with increasing number of federates.....	115
Figure 8.6: Percentage of saved execution time with increasing number of federates ...	115
Figure 8.7: Physical federate pool approach.....	117
Figure 8.8: Experiments for studying physical federate pool approach.....	117
Figure 9.1: Fault-Tolerant model upon dynamic LRC substitution	124
Figure 9.2: Problem in dealing with in-transit events	125
Figure 9.3: Example for calculating timestamp of events to be disposed	129
Figure 9.4: Model supporting Web or Grid Enabled Architecture	134
Figure 9.5: Load balancing model for simulation model	136

Table of Tables

Table 4.1: Comparison between Single-federation and Multiple-federation Solutions	38
Table 4.2: Notations of the federate attributes	46
Table 4.3: The index of the experiments	46
Table 5.1: Configuration of experiment test bed for examining decoupled federate architecture.....	66
Table 6.1: Comparison between the recursive region division and the point region solution	81
Table 8.1: Declaration information of the federates.....	109
Table 8.2: Configuration of experiment test bed	109
Table 8.3: Experiments for verifying the correctness of cloning technology	110
Table 8.4: Experiments for studying the efficiency of cloning technology.....	112
Table 8.5: Comparison between normal and pool approach.....	118

Chapter 1: Introduction

1.1. Background

Simulation technology is commonly used to study physical systems or imagined systems [Fujimoto00]. A complex system can be modeled as a simulation program (or programs), which simulates the system's behavior on computers. The simulations can be used to perform what-if analysis on the simulated system, to predict the different results of varying key parameters/estimates or to examine a series of solutions prior to making the final implementation decision.

Sometimes it is unmanageable, expensive and/or risky to construct a real system for analysis. For example, people want to explore the eruption process of an active volcano. Astronomers attempt to discover the evolution of the Universe. Researchers may have to choose to use simulation to emulate the activities of these systems. Simulation technology is safer, more feasible and flexible, and possibly much less costly in practice. Simulation can overcome the inconvenience incurred by the span in time and geographic distribution. Days or even years of activity can be executed in a very short time in simulation executions. Thus, simulation can provide prediction to support decision-making. Furthermore, the external environment is unlikely to impact the execution of simulations while it often limits the operation of real systems.

Nowadays, simulation technology has a wide application spectrum from training to gaming, from scientific research to business operation and from military to civilian purposes. It is involved in a variety of human and natural activities. Battlefield simulation environments over networked computers have already replaced the sand table for plotting military operations. Flight simulators can help in training pilots safely and efficiently

from the Boeing series to Longbow Apache. Networked multimedia together with simulation facilitates the creation of new exciting automated learning environments [Rocchetti01]. Many PC games like Command&Conquer construct an amazing virtual world for many people [EA04]. As for scientific research, algorithms and prototypes are often developed based on simulation models. In practice, system designers often exploit simulations to verify the proposed designs, thus they can perform their work conveniently and efficiently [Cunning99]. Industry and business collaborations also benefit from simulation for decision support and productivity enhancement [Mchany99]. Simulation can help companies to facilitate the design, evaluation and optimization of their daily operations.

There are different types of simulations, such as continuous simulation and discrete event simulation, parallel and distributed simulation. In continuous simulation, states change continuously over simulation time, but in discrete event simulation, system states change at discrete points in time [Fujimoto00]. The events represent the system activities or changes of system state. An associated timestamp denotes the simulation time at which an event occurs. For example, in a supply-chain simulation [Gan00], the arrival of a customer order can be modeled as an event.

The simulation of a complex system can be divided into smaller components to form a parallel simulation. Each component, known as a Logical Process (LP), models a subset of the system. Parallel simulation concurrently executes LPs over multiple processors to reduce execution time.

Distributed simulation is an important technology that enables a simulation program to be executed on a distributed computer system. A large-scale simulation may be constructed by linking together existing simulation models at possibly different locations.

For example, distributed simulation technology meets the pressing need of supply-chain simulation, as a supply-chain often involves multiple companies across enterprise boundaries [Gan00]. Each of the companies participating in a supply-chain simulation may already have their existing simulation models, which can be developed independently based on heterogeneous platforms [Turner01]. Distributed simulation technology based on a common standard can facilitate the interoperability amongst these distributed simulation models.

Distributed simulation has been the focus of the U.S. defense industry for over a decade. These defense-related efforts originated with SIMNET [Pope91] and evolved into the Distributed Interactive Simulation (DIS) protocol initiative [DIS94][IEEE1278]. To meet the demand for standardization on a higher abstraction level, the Defense Modeling & Simulation Office (DMSO) sponsored the definition and development of the High Level Architecture (HLA) for modeling and simulation (M&S). The HLA defines an architecture for reuse and interoperation of simulations. The HLA supports component-based simulation development; the components are referred to as simulation federates [Kuhl99]. Thus, a set of simulation federates, possibly developed independently, can be linked together to form a large federation. The HLA has been approved as an open standard through the Institute of Electrical and Electronic Engineers (IEEE) [IEEE1516]. The Runtime Infrastructure (RTI) is the software to support an HLA-compliant distributed simulation.

1.2. Motivation for Simulation Cloning

In a normal simulation, the output reports one single set of results per simulation run. In order to optimize the simulated system or perform “what-if” analysis, one should repeat the simulation multiple times to examine alternative scenarios, decision policies and

strategies using different decision rules and parameters¹. Therefore, the simulation analyst may choose some best solutions based on several possible results. Basically it is a time-consuming and onerous task in which a lot of computation is repeated unnecessarily. In the context of a large-scale distributed simulation, considering the complexity and distribution of the individual simulation models and the number of federates, it is costly to reconfigure and re-execute the overall distributed simulation again and again.

Figure 1.1: A simple manufacturing simulation

During the simulation, a federate will meet some points (decision points) at which there occurs a critical change of system states, and it is faced with different choices to proceed. For example, in a simple manufacturing simulation illustrated in Figure 1.1, a batch machine is processing a lot while some lots are waiting in a queue. In the figure, part (A) presents the manufacturing process and part (B) indicates the advancing of the simulation. When the current job finishes, the machine status becomes “idle” from “busy”. If it is found that the queue length has become greater than the alert level, one needs to use some special dispatching rule to process the waiting lots. The time and cost to be consumed may vary for different rules. Instead of simulating all the three possibilities from the start, it is possible to clone two more executions to examine the rules of interest concurrently once the condition “Queue_Length_Greater_Alert_Level” is met. The point where the queue length reaches the alert level and the machine becomes idle is one decision point in this simulation.

¹ A single run is not sufficient to understand the behavior of stochastic simulations. Cloning can be used to initiate several runs of a simulation, but with different values of the random variables.

At a predefined decision point, cloning of a federate may be triggered at runtime, following which the federate can replicate itself into multiple copies (clones) to explore each possibility. Each clone explores one particular path together with its partner clones spawned from the other federates in the original scenario. This provides the user with the opportunity to evaluate multiple alternative results concurrently using the same simulation models within a single session.

An individual clone should only interact with its partner federates to form an independent scenario to report a unique set of results. Thus, an additional mechanism is required to partition the interactions amongst the scenarios. Partitioning ensures that clones do not perform incorrect simulations due to events being received from other scenarios. It provides transparency to the cloned federate about the existence of other scenarios. Moreover, to minimize the overhead incurred by creating more and more scenarios, we need to reduce bandwidth requirements by only transmitting interactions or attribute updates where necessary. To support cloning, there needs to be a dynamic event “routing” and “filtering” mechanism.

There may exist identical execution of federates in different scenarios. Having the capability to allow individual federates to simultaneously operate in multiple scenarios can further reduce unnecessary computing and accelerate the overall simulation execution. This poses some challenging complications, for example, sharing federates in different scenarios needs accurate control to achieve correctness and efficiency.

One of the key benefits of HLA-based simulation is reusability [Dahmann98]. It means the simulation models can be reused in different simulation scenarios and applications. To provide reusability of existing simulations, it is also necessary to hide the complexity of the above operations. In an HLA-based simulation, this means that we need to maintain

the standard HLA interface to the user's programs while providing extended functionalities. As our design targets users who may have their own existing complex simulation models, we have the additional aim to provide reusability and transparency while enabling simulation cloning. Providing easy utilization and deployment is another major concern.

Reliability is a crucial issue for a distributed simulation cloning mechanism, which includes correctness of simulation execution, stability of simulation processes and fault tolerance. However, it is challenging work to ensure correct and efficient simulation cloning especially in a distributed environment. For example, it needs state saving and replication [Franks97] at both the simulation model level and the RTI level, rather than simply starting another federate instance. A distributed control mechanism is needed to coordinate the federates within the same distributed simulation session. The cloning needs to guarantee consistency of state for all the participating federates. This is important to keep "fidelity" of a cloned scenario to the physical system that the distributed simulation represents. The cloning approach needs to ensure that a clone has the same states as the original federate at the decision point. The solution should keep the correct semantics of each scenario, that is, each scenario should report simulation results identical to those that the same set of federates would achieve in a normal independent federation execution. Furthermore, this research intends to produce an RTI independent and generic solution to distributed simulation cloning.

A further consideration is that simulation federates running at different locations are liable to failure. In a distributed simulation, the failure of one federate can lead to the crash of the overall simulation execution. The risk of such failure increases with the number of federates inside one single federation. Thus, we need to investigate the fault

tolerance issue in simulation cloning and apply it to distributed simulation technology. It is feasible to use the replication mechanisms designed for cloning, for example state saving and recovery, to achieve fault tolerance. If one scenario encounters failure, the scenario can be resumed or the remaining concurrent scenarios are still able to continue execution. Failure isolation provides a solution to deal with failure at individual components, while performing distributed computing. It is difficult to achieve these properties in existing federates while providing reusability and transparency.

1.3. Objectives of the Research

The project aims to develop novel control mechanisms for large-scale distributed simulations based on the HLA. We establish an infrastructure incorporating the distributed simulation cloning technology. The infrastructure serves as a decision support tool much more powerful and flexible than traditional “linear” simulations by enabling analysts to carry out prompt “what-if” analysis.

The research work described in this thesis covers the investigation and development of an enabling technology for distributed simulation cloning and related issues. The following objectives and scope are identified:

- To establish the foundation theory of distributed simulation cloning. This project pioneers the domain of distributed simulation cloning, and provides the theoretical basis to identify the research issues. The thesis is the first to define cloning and related concepts systematically in the distributed simulation domain.
- To investigate and design an efficient, reliable mechanism to support distributed simulation cloning. Alternative solutions are designed and compared based on reliability, efficiency, complexity in management and other specifications.

- To develop a generic approach to cloning federates at runtime. It is very difficult to have a generic state manipulation mechanism for any existing federate, as these can be developed independently and freely. The correctness of replicating a running federate significantly depends on the RTI software. The approach aims to overcome these difficulties by providing federate cloning mechanisms that are generic and platform independent.
- To develop a mechanism for managing concurrent scenarios in a cloning-enabled distributed simulation. To ensure correctness and efficiency, solutions are needed for managing concurrent scenarios and identifying clones and scenarios.
- To provide user transparency and reusability to existing simulation models. It is an important and challenging work to keep intact the distributed simulation models while enabling cloning and other technologies.
- To design alternative simulation cloning mechanisms, including entire cloning and incremental cloning. An incremental cloning mechanism ensures a federate requires cloning only when strictly necessary, and enables sharing identical execution amongst concurrent scenarios as much as possible. The cloning mechanisms should produce the same outputs to those obtained from traditional distributed simulations.

1.4. Organization of the Thesis

Including the introduction chapter, this thesis contains ten chapters in total. Chapter 2 introduces the basics of HLA and RTI services. This chapter also introduces the work related to simulation cloning.

Chapter 3 introduces the foundation theory of distributed simulation cloning. Basic concepts of the technology are defined and critical research issues are identified. This

chapter outlines various types of federate cloning and scenario representation in distributed simulations.

Chapter 4 discusses the issues involved in cloning distributed simulations based on the HLA as well as proposing tentative solutions. Alternative solutions are compared from both the qualitative and quantitative point of view. A middleware approach is suggested to hide the implementation complexity and provide reusability and user transparency to existing simulation federates. To measure the trade-off between complexity and efficiency, this chapter introduces a series of experiments to benchmark various solutions at the RTI level.

Chapter 5 introduces a novel decoupled federate architecture to enable federate cloning at runtime. The decoupled architecture ensures the correct replication of federates and facilitates fault tolerance at the RTI level. This chapter gives preliminary benchmark experiments to study the extra overhead incurred by the decoupled federate architecture against the normal federate.

Chapter 6 describes the method of using Data Distribution Management (DDM) to partition concurrent scenarios in a cloning-enabled distributed simulation. Two candidate solutions are introduced for managing scenarios and identifying each clone and scenario. This section details the design of these two solutions and analyzes their advantages and drawbacks. This chapter also discusses how the partitioning mechanism is implemented using a middleware method to address reusability issues.

Chapter 7 first describes the design and functionalities of the infrastructure enabling distributed simulation cloning. Secondly, this chapter details the algorithms of simulation cloning at decision points including state saving and replication, the method of coordinating and synchronizing clones etc. Thirdly, an incremental cloning mechanism is

covered in detail, which is designed to replicate only those federates whose states will be affected. This mechanism employs an event checking algorithm for sharing federates in multiple scenarios, and supports correct HLA semantics. This chapter also summarizes the entire cloning mechanism, which make clones of all federates at the same time.

The performance of our mechanisms is examined in chapter 8, and their correctness is established as well. Experiments have been carried out to compare the execution time of entire cloning and incremental cloning enabled federates using an example of a simple supply-chain simulation. Experimental results indicate that the proposed cloning technology provides correct distributed simulation cloning and can significantly reduce the execution time for evaluating different scenarios of distributed simulations. Moreover the incremental cloning mechanism significantly surpasses entire cloning in terms of execution efficiency.

Chapter 9 discusses the exploitation of the decoupled federate architecture used in the distributed simulation cloning. A fault-tolerant model is proposed to provide runtime robustness. A Web/Grid enabled architecture is introduced to support the use of HLA in a Web/Grid environment. The advantages of the decoupled architecture in providing load balancing are also discussed.

Chapter 10 evaluates and concludes the work that has been accomplished. This chapter also summarizes issues for future research.

Chapter 2: Literature Review

This chapter first introduces the basic concepts of HLA and RTI. The HLA provides the underlying infrastructure for the mechanisms developed in this study. The second section describes existing replication techniques used in software engineering and distributed systems. The third section presents previous research work on cloning in simulation. Some specific features and requirements for the proposed distributed simulation cloning mechanisms are also presented.

2.1. High Level Architecture and Runtime Infrastructure

Figure 2.1: Functional view of an HLA federation

The Defense Modeling and Simulation Office (DMSO) sponsored and developed the HLA standard [DMSO95]. The DMSO provides a full-time focal point for information concerning Department of Defence (DoD) modeling and simulation (M&S) activities. The HLA defines a software architecture for modeling and simulation. The HLA is designed to provide reuse and interoperability of simulation components. The simulation components are referred to as federates. A simulation federation can be created to achieve some specific objective by combining simulation federates. The HLA supports

component-based simulation development in this way [Dahmann98]. A functional view of an HLA federation is illustrated in Figure 2.1.

The HLA federation is a collection of federates (observers, live participants, simulations etc.) interacting with each other for a common purpose, for example war-gaming. These federates interact with each other with the support of the Runtime Infrastructure (RTI) and the use of a common **Federation Object Model (FOM)**.

In the formal definition, the HLA standard comprises four main components: **HLA Rules**, **Object Model Template (OMT)**, **Interface Specification** [IEEE1516] and **Federation Development and Execution Process (FEDEP)** [IEEE1516.3]. The HLA rules define the principles of HLA in terms of responsibilities that federates and federations must uphold. Each federation has a FOM, which is a common object model for the data exchanged between federates in a federation. The OMT defines the meta-model for all FOMs [Kuhl99]. The HLA Interface Specification identifies the RTI services available to each federate and the functions each federate must provide to the federation. The FEDEP mainly defines a process framework for federation developers including the necessary activities to build HLA federations.

As defined in the OMT, an “**object**” represents an entity with a distinct identity created in a federate, which belongs to a certain “**class**” and defines a set of named data called “**attributes**”. An “**interaction**” is a collection of data sent at one time through the RTI to other federates, with accompanying “**parameters**”, representing some occurrence in a federate. Objects and interactions are identified by unique handles assigned by the RTI. In the context of an HLA-compliant simulation, events can be updating attributes of an object or sending an interaction. Any federation model can be written in terms of objects and interactions [Kuhl99], in which (1) any occurrence can be modeled as an interaction, (2) if an entity is to be modeled such that it has persistent state, the entity should be represented as an object.

The HLA is an architecture defining the rules and interface, whereas the Runtime Infrastructure (RTI) is the software conforming to the HLA standard, that is used to support a federation execution. Figure 2.2 gives an overview of an HLA federation and the RTI [Kuhl99]. The RTI provides a set of services to the federates for data interchange and synchronization in a coordinated fashion. The RTI services are provided to each federate through its Local RTI Component (LRC) [RTING02]. The RTI can be viewed as a distributed operating system providing services to support interoperable simulations executing in distributed computing environments [Fujimoto04]. The HLA exhibits three features [Kuhl99]: a layered architecture, data abstraction architecture and event-based architecture. The HLA architecture separates the simulation model from the infrastructure functions. Thus, all behavior specific to a given simulation model is in the federate, and the infrastructure contains functions generic to simulation interoperability. In a distributed federation, the LRC contains the network functions needed to facilitate distribution. The data abstraction architecture frees federates from retaining references to other federates. The event-based architecture provides implicit invocations to call back all receiving federates for an event and does not require the sending federate to know the receivers.

Figure 2.2: Overview of HLA federation and RTI

A total of six service categories are defined in the specification, namely Federation Management, Declaration Management, Object Management, Ownership Management, Data Distribution Management and Time Management [RTING02]. The definition and functionalities of these RTI services are given as follows:

Federation management services provide methods such as creating federations, joining federates to federations, observing federation-wide synchronization points, effecting federation-wide saves and restores, resigning federates from federations, and destroying federations.

Declaration management services include publication, subscription, and supporting control functions. Using services defined in this category, federates declare object class attributes or interactions (predefined in the OMT) that they intend to publish (produce) or subscribe (consume).

Object management services include instance registration and instance updates for the object producers. On the other hand, it provides instance discovery and reflection for the object consumers. It also supports methods for sending and receiving interactions, controlling instance updates based on consumer demand, and other miscellaneous functions.

Ownership management services can be used to transfer ownership of instance attributes among federates. The ability to transfer ownership is intended to support the cooperative modeling of a given object instance across a federation. The services provided support both push and pull mechanisms for ownership transfer.

Data distribution management (DDM) services are employed by the data producers and consumers to assert properties of their data or to specify their data requirements respectively based on specified regions. The RTI then distributes the data from the producers to the consumers based on the matches between the properties and the requirements. This management controls the efficient routing of class attributes and interactions via the RTI.

Time management is concerned with the mechanisms for controlling the advancement of each federate along the federation time axis. The services in this group synchronize event delivery amongst federates. Time advances are coordinated by the RTI with object

management services so that information is delivered to federates in a causally correct and ordered fashion. Time regulating federates may send time stamp ordered (TSO) events while time constrained federates are able to receive TSO events in time order. Each regulating federate must specify a “lookahead” value, and ensure that it will not generate any TSO event earlier than its current time plus lookahead [RTING02]. A federate can be both time regulating and time constrained, either of them, or neither. Each federate’s LRC maintains two internal event queues, i.e. a time stamp (TSO) queue and a FIFO receive queue. When TSO events arrive at a time constrained federate, the LRC buffers these events in the TSO queue [RTING02]. After requesting a time advance, the federate is passed all events in the TSO queue with timestamp less than or equal to the federate's granted time. As for the events without timestamp, known as Receive Order (RO) events, the LRC places them in the receive queue in the order in which they arrive. Information in the receive queue is immediately available to the federate.

The RTI services are available as a library (C++ or Java) to the federate developers. Within the RTI library, the class RTIAmbassador [RTING02] bundles the services provided by the RTI. A federate may invoke operations on the interface to request a service (federate-initiated service) from the RTI. The FederateAmbassador [RTING02] is a pure virtual class that identifies the “callback” functions each federate is obliged to provide (RTI-initiated) service. The federate developers need to implement the FederateAmbassador. The callback functions provide a mechanism for the RTI to invoke operations and communicate back to the federate.

The HLA specification leaves the RTI implementation details to the RTI implementers while defining a standard interface. Nowadays there are various RTI software available to the developers, including both commercial and academic implementations. These include the RTI Next Generation (RTI-NG) implementation sponsored by the DMSO [DMSO03], the pRTI developed by Pitch AB [Pitch04], the Federated Simulations Development Kit

developed by the Georgia Tech PADS research group [FDK04], etc. The research work studied in this thesis is based on the DMSO RTI-NG software, however the implementation need not be specific to this software. The standard HLA interface enables the use of different RTI implementations for the work discussed in this thesis.

2.2. Cloning and Replication

Cloning and replication have been widely used in software engineering and distributed applications for various objectives, such as improving performance, enhancing system reliability and scalability etc. In this section, the terms “cloning” and “replication” are interchangeable with respect to the existing work, while in chapter 3 a distinction is made between them in our theory of distributed simulation cloning.

2.2.1. Cloning in Programming Languages

Methods of procedure cloning have been studied in the domain of programming languages, such as those presented by Cooper et al and Vahid [Cooper93] [Vahid99]. Their approaches aim at optimizing code generation for the compiler, by creating clones of procedure bodies. By carefully partitioning the calls, the compiler ensures that each clone inherits an environment that allows for better code optimization.

Object cloning was discussed in [Plevyak95], and is used for eliminating parametric polymorphism and minimizing code duplication to overcome some typical inferior performance of object-oriented programs. For instance, polymorphic functions do not know the exact types of the data on which they operate, and must use indirection to operate on them. Plevyak proposed specializing a polymorphic class into a set of monomorphic classes (clones of the corresponding class) to improve efficiency.

2.2.2. Data Replication in Distributed Systems

Data replication techniques play an important role in distributed systems. For the concern of performance, data are often replicated to facilitate fast local access. Scalability is about how to address the performance bottleneck in scaling a system. Data replication is also used as a scaling technique by placing data copies close to the process using them. Moreover, data replication helps to improve the reliability of a distributed system in the sense that the system continues working when some replicas crash and also provides protection against corrupted data [Tanenbaum02].

Data replication has been widely applied in distributed database systems for improving both performance and reliability. In geographically distributed database systems, the access of remote data is expensive. Data are often fully or partially replicated to avoid remote access for read transactions [Ciciani90]. Full replication requires all data objects being replicated to all sites so that each site holds a complete copy of the distributed database. This extreme case of replication has been recognized not to be the optimal configuration for many applications. Partial replication can consist of replicating (1) all data objects to some sites, (2) some data objects to all sites and (3) some data objects to some sites [Nicola00]. The several partial replication approaches are suggested to balance the overhead in keeping data consistency and the benefit gained in access efficiency. In addition, data replication has also been exploited to control concurrency in distributed database systems [Ciciani90].

The general principles and issues studied in data replication are also used as guidelines in designing and developing our distributed simulation cloning technology. For example, we need to ensure the scalability of our cloning-enabled systems and keep state consistency without sacrificing execution efficiency.

2.2.3. Agent Cloning in Multi-agent Systems

Multi-agent systems (MASs) are increasingly used when a problem domain is particularly complex, large, or unpredictable [Sycara98]. Basically, in MASs an agent has incomplete information or capabilities for addressing the whole problem, and multiple agents interact and collaborate to solve the problem. Shehory et al use agent cloning as a comprehensive approach to the performance bottlenecks in MASs. Their approach allows agents to make clones, transfer tasks or merge with other agents [Shehory98].

They use an agent cloning approach to deal with overload in MASs. In the context of their approach, agent overload means either an agent is not able to process current tasks due to its limited capability or machine overloads. The first case does not imply machine overload, and therefore they suggest cloning the agent locally to let clones take over part of the tasks. For the second case, their approach will create and activate clones of agents at a remote machine. Through agent cloning, load balancing can be achieved and the performance of an overloaded MAS can be improved.

Agent cloning aims at optimizing the resource usage of the whole MAS. It should be noted that the approach requires an agent to reason about its current and future load. Furthermore, their design and implementation are built upon the RETSINA infrastructure, which is specially designed for intelligent network agents [Sycara96].

2.2.4. Object Replication in Parallel Object-oriented Environments

Concurrent and parallel object-oriented systems have been designed to allow the developers to manage the high complexity of parallel programming and provide modularity and code reuse. The capability of supporting load balancing is one of the major factors affecting the performance of these systems. Jie et al developed a Parallel Object-oriented Environment for Multi-computer Systems (POEMS) to dynamically

balance the processing load based on object replication, thus the performance of parallel object-oriented applications can be improved [Jie01].

Their approach achieves dynamic load balancing by migrating objects. They designed a Parallel Object Replication model (POR) to support task parallelism by distributing the replicas of POR objects in different nodes. Their model only requires migrating the data rather than the entire objects, hence the overheads of migrating objects are reduced considerably.

2.2.5. Fault Tolerance using Replication

Fault-tolerant techniques often employ redundant/backup components to achieve system robustness. Birman used backup to achieve fault tolerance in building reliable network applications [Birman96]. In [Elnozahy93], fault-tolerance was enabled in a distributed system using rollback-recovery and process replication. In rollback-recovery, processes periodically save their states on stable storage during failure free operation, and the states can be loaded to restore consistent execution. Process replication is used to provide high availability of servers with replicas of server programs running in multiple machines. Principally these methods take advantage of replication to ensure reliability of distributed applications. Replication consumes extra resources and requires synchronization of the replicas to maintain consistency [Damani98].

2.3. Simulation Cloning

Simulation users often execute multiple “replications” of the same simulation to obtain outputs that meet required confidence levels [Law99]. In later chapters, the term replication is slightly different from that used in traditional simulations, and does not merely mean repeating multiple runs of an application or simulation. In this section, cloning and replication denote making replicas of simulation processes, threads or data in

the middle of execution. We give a survey on existing techniques exploiting cloning/replication in simulations.

2.3.1. Cloning in Rare Event Simulations

The performance of computer and communication systems is commonly characterized by the occurrence of rare events, e.g. the probability of cell loss in asynchronous transfer mode (ATM) switches is typically less than 10^{-9} . Since straightforward simulation of rare events often takes an extremely long execution time, Rare Event Simulations are often used to evaluate the performance of this kind of system [Görg01].

Glasserman et al adopted the cloning approach to improve the efficiency and effectiveness of rare event simulations [Glasserman96]. In the standard simulation of a stochastic process, a lot of time is spent in the region of state space faraway from the rare set of interest, from where the chance of entering the rare set is extremely low. In order to tackle this problem, their approach splits multiple identical copies of the process to get more chances for rare events to occur when the state space is close to the rare set. Thus, within a given amount of execution time, the approach increases the number of times of hitting the desired rare set. Their approach has a common point with ours in the sense of using cloning to improve execution efficiency. However, our approach is significantly different from theirs. First of all, their approach makes multiple identical copies of the stochastic process while our approach makes clones of a federate with each clone exploring different execution paths. More importantly, our approach aims at distributed simulation cloning, and their approach does not concern distributed simulation at all.

Addie proposed an approach to emphasize the execution paths of interest in quantum simulations by simulation cloning and killing off the uninteresting paths [Addie01]. His approach is intended to increase the speed and accuracy of estimation of rare events. It uses threads to simulate quantum stochastic processes, and makes multiple clones of selected threads or terminates clones in the middle of a simulation following dedicated

rules. The clones evolve independently, thus execution efficiency is gained compared with repeating simulations using multiple parameters. His idea also aims to avoid unnecessary or undesired repetition of executions amongst different execution paths.

2.3.2. Multitrajectory Simulations

Gilmer, Jr. and Sullivan developed a multitrajectory simulation technique to obtain better understanding of the possible outcome set of stochastic simulation [Gilmer98, 99]. In conventional stochastic simulation, each replication gives only one outcome. In contrast, the multitrajectory simulation generates multiple outcomes when a random event occurs, with each outcome constituting a trajectory maintaining its own states cloned from the original execution. Their technique has advantages over traditional stochastic simulation, and requires much fewer runs to generate an equivalent quality histogram. Multitrajectory simulation technology is also adopted to support recursive simulation, which means a simulation recursively uses the outputs of another instance of the same simulation, thus improving the quality of decision making [Gilmer00].

In [Gilmer99], the same authors briefly describe the issue of multitrajectory routes and trajectory management. Their discussion approximates to the rudiments of our scenario management mechanism (see section 3.4 and chapter 6). However, their cloning technique often requires the modeler to take care of state cloning and initiating alternative trajectories when building the models, which becomes even more difficult when the stochastic event is embedded in the functional code [Gilmer98]. Hence, there are limitations to the application of this technique to existing simulation models. Furthermore, their cloning approach does not perform well compared with normal runs when dealing with small scenarios.

2.3.3. Cloning in Simulation Software Packages

Some simulation software packages also incorporate cloning functionality. For example, the Simulation Language with Extensibility (SLX) supports cloning of transactions in its simulation language [Henriksen97, 98]. SIMPHONY also provides methods for cloning entities in construction simulation [AbouRizk00]. Developers may take advantage of the functionalities provided by those packages to customize their simulation cloning mechanisms.

2.3.4. Parallel Simulation Cloning

Hybinette and Fujimoto were the first to employ simulation cloning technology as a concurrent evaluation mechanism in the context of parallel simulation [Hybinette97, 01, 02]. Their work provides a thorough study of parallel simulation cloning technology. The motivation for this technique was to develop a cloning mechanism that supports an efficient, simple, and effective way to evaluate and compare alternate scenarios. The method was targeted for parallel discrete event simulators that provide the simulation application developer with a logical process (LP) execution model. Through their work, the basic theory of parallel simulation cloning has been established.

Parallel simulation cloning improves performance of simulation execution in two ways: (1) cloned simulations share the same execution path before the point that cloning happens, and (2) the idea of a virtual logical process avoids repeating unnecessary computation among clones and aims at further sharing computations after the decision point.

Cloning of an LP may occur at predefined decision points so the original LP and the clones execute along different execution paths in parallel, thus the execution path before cloning is shared. In designing the cloning mechanism, they proposed replicating LPs incrementally rather than copying the whole simulation at once as it is likely that only

some of the LPs are different amongst execution paths. They designed an LP as a combination of a virtual LP and an underlying physical LP. The physical LPs realize the real parallel simulation system and maintain the state space and physical messages. In contrast, a virtual LP only maps to the corresponding physical LP and interacts with other virtual LPs via its physical LP. When cloning a simulation, their approach only requires replicating the physical LPs that are different in the alternative scenarios. For identical LPs, only the virtual LPs need to be replicated (which is cheap) and multiple replicas share a single physical LP. Besides process cloning, their cloning mechanism also includes message cloning. A message is cloned when a shared physical LP sends a message to LPs that are not shared.

Some concepts and issues from parallel simulation cloning are also examined in our research, such as decision point, execution paths and evaluation of alternative paths etc. Their work gives us a viable guideline in identifying some basic research issues.

Their cloning mechanism, which relies on the Georgia Tech Time Warp simulation executive [Das94], is implemented as a Clone-Sim software package. The package requires the simulation executive to support copying of LPs as well as dynamic creation, allocation and initialization of LPs [Hybinette01]. Hence, their approach does not imply a generic LP cloning mechanism. Furthermore, their approach is aimed at the parallel simulation paradigm, and it seems that there exists a gap to directly apply their approaches to HLA-based distributed simulation. For example, their design does not concern maintaining state consistency of LPs in the distributed environment on cloning, which is crucial in distributed simulation cloning.

2.3.5. Cloning of HLA-compliant Federates

Schulze et al extended the simulation cloning technology to HLA-compliant simulations [Schulze99]. They introduced a cloning approach to extend the flexibility of system

composition to run-time, and gave a proposal for cloning federates and the federation at runtime [Schulze99, 00].

Internal cloning and external cloning techniques were suggested to clone the federates. Internal cloning means replicating federates and letting the replicas participate in the original federation. This technique requires the federates to distinguish the messages from clones. It is also suitable for federates with only passive behaviors, such as a pure “observer” federate which does not generate any event, or in other words does not affect the states of any other federate during the simulation. The external cloning approach makes replicas of federates so that they operate in a different federation. Forecast functionality is provided by this approach via the parallel scenario management of different time axes.

They describe the problems of cloning federates in terms of dealing with cloning all relevant elements of the simulation model, such as object instances and other internal states of the model. Their work proves the feasibility of applying cloning in HLA-based simulation. However, other critical issues are not clearly addressed, such as how to ensure the correctness of the interaction between the clones (and the federate being cloned) and other related federates. Their cloning design intensively relies on the SLX simulation package, so that their approach lacks generality when applied to other simulation models [Henriksen97, 98]. The detailed federate replication algorithms and the performance of the proposed approaches are not given.

2.3.6. Fault Tolerant Distributed Simulation

Distributed simulations are liable to encounter failure, and normally users have to restart the entire simulation session from the scratch. Obviously, this results in considerable waste of time and computation. Approaches for fault-tolerant distributed simulation have been studied to address this problem.

State manipulation methods such as checkpoint and migration are often exploited in existing fault tolerant approaches for distributed simulations. In [Damani98], a rollback based optimistic fault tolerance scheme is integrated with an optimistic distributed simulation scheme. The scheme models a failure as a straggler event to optimize implementation efforts. This kind of approach has some common drawbacks from the developers' perspective:

They require simulation components to have the capability of saving system states and recovering the saved states at the model level in the case of failure. The approach does not apply in the case that existing simulation models are not designed to support state manipulation or process/thread replication.

Berchtold and Hezel proposed the Replica Federate approach for fault-tolerant HLA-based simulation [Berchtold01]. Their approach produces multiple identical instances of one single federate, and failures can be detected and recovered upon the outputs of those identical instances. However, redundancy is liable to result in lower system performance. Furthermore, extra federate replicas in a distributed simulation increase the probability of overall system failure incurred by an LRC failure.

2.4. Summary

In this chapter, we introduce the basics on HLA and its supporting software, i.e. the RTI. The distributed simulation cloning technology of our study is based on this architecture. Related work on the major research issues in this study, i.e. replication and cloning in various computing technologies, distributed systems and simulations, is described and briefly analyzed.

Our research is significantly different from the above research in the following aspects:

- We aim to enable distributed simulation cloning technology based on the High Level Architecture. This is not addressed by the above research except Schulze et al's work.

- In our project, simulation cloning means the replication of the entire federate at runtime.
- The proposed cloning technology supports evaluation of concurrent scenarios of a distributed simulation. Automatic scenario partitioning/managing and dynamic execution/computation sharing will be enabled. Our technology is designed to mask the complexity involved in the above efforts.
- As our design targets potential academic or industrial users who may have their own complex simulation models, we have the additional aim to support reusability and transparency while enabling simulation cloning.
- We intend to provide a generic cloning infrastructure to existing distributed simulations, which is independent of the RTI software and the implementation of the simulation model.
- The technology is designed to offer analysts the flexibility to specify the conditions/rules according to which distributed simulation cloning can be initiated automatically.
- The technology ensures the correctness of the synchronization between federates in cloning-enabled simulations.

The proposed solution must guarantee the state consistency of distributed federates. Our technology needs to ensure the scenarios created by cloning have an identical execution to the traditional approach. Designers of replicated systems have to make tradeoffs between consistency, performance, and availability [Yu02], and at the same time this technology needs to minimize the extra overhead incurred by cloning. We need to explore the mechanisms of managing a cloning-enabled distributed simulation at both the scenario level and clone (federate) level.

Chapter 3: Theory and Issues in Distributed Simulation Cloning

This chapter presents the theory of distributed simulation cloning and the critical research issues involved. It defines basic concepts and notations that are used in the technology. Different types of federate cloning and scenario cloning in distributed simulations are identified and classified.

3.1. Decision Points

As discussed in section 1.2, during the execution of a simulation, a federate may face different choices to perform alternative actions. The federate is cloned at such decision points according to some predefined triggering conditions or rules. A **decision point** represents the location in the execution path where the states of the system start to diverge in a cloning-enabled simulation. “Cloning” differs from simple “replication” in the sense that clones of the original federate execute in different “paths” rather than simply repeat the same executions, even though the computation of clones are identical at the decision point. From the decision point onwards, a simulation spawns multiple execution paths to exploit alternative scenarios concurrently.

From the user’s point of view, decision points can be specified either at the modeling stage or at run-time. A user may predefine the cloning triggers for a decision point, which comprise the cloning conditions, alternative actions and properties of the trigger. The properties of a trigger describe how it is defined: this may be based on (1) the values of an object instance, (2) the status of the model’s internal states, (3) a certain simulation time, or (4) dependency on other triggers. At the modeling stage the user can specify the conditions of the decision points, and the alternative actions for the decision points. During run-time, the user can insert decision points dynamically and can even specify

alternative execution paths at a decision point interactively. The cloning trigger will initiate cloning when necessary.

Any occurrence of cloning has impact on the existing scenarios and other federates in a distributed simulation. Section 3.2 will classify the simulation cloning from an individual federate's point of view, whereas section 3.3 will define different cloning mechanisms from a scenario's perspective.

3.2. Active and Passive Cloning of Federates

When a federate reaches a decision point, it makes clones on its own initiative. This federate is said to perform **active cloning**. Each clone of the federate executes a separate scenario. An active cloning results in the creation of new scenarios.

In distributed simulations, there are multiple federates interoperating with each other. When one federate splits into different executions, the partners who interact with this federate may have to spawn clones to perform proper interaction even though the partners have not yet met a decision point. Those partners are said to perform **passive cloning**. Generally speaking, the clones generated in an active cloning have separate initial states while those created in a passive cloning have identical initial states. Only the active cloning leads to the creation of new scenarios, and this induces passive cloning in some other federates.

3.3. Entire versus Incremental Cloning

A simple approach to keep the correctness of the simulation is to clone the whole simulation whenever any federate reaches a decision point, i.e. **entire cloning**. Each cloned simulation consists of a separate set of federates that report results independently. One can see that the scalability of distributed simulation is a challenge in this case. Another approach replicates the simulation incrementally, i.e. only those federates whose

states will alter at a decision point need to be cloned, other federates will remain intact. This alternative approach is referred to as **incremental cloning**. Such an incremental cloning approach shares computation between federates in alternative scenarios, and provides a more efficient and scalable method to clone the distributed simulation.

Figure 3.1: Entire Cloning vs. Incremental Cloning

3.3.1. Shared Clones

Figure 3.1 illustrates the different effects of an active cloning using entire cloning versus incremental cloning. For those federates whose states are not affected, the incremental cloning mechanism allows them to operate in the new scenarios in addition to the original one as **shared federates** (clones), such as *Fed A[0]* and *Fed X[0]* in the right half of Figure 3.1. A shared clone may subsequently perform cloning passively during the execution of the simulation on demand of its partners.

Normal clones are those clones that operate in a single scenario (e.g. *Fed B[i]* in Figure 3.1); this term is used in this thesis to distinguish them from the shared clones. The clones created from the same root federate are referred to as **sibling clones**. The federate being cloned is referred to as the **parent federate** (clone) of the clones directly replicated

from it. Those federates that interact within the same scenario are known as **partner federates**¹.

A clone needs to inherit the same RTI objects from the parent federate. We name the object instances registered by the original federates prior to cloning as **original object instances** whereas we use **image object instances** to denote those object instances re-registered by the clones in the state recovery procedure.

3.3.2. Theory and Issues in Incremental Cloning

Figure 3.2: A typical shared clone

The incremental cloning mechanism enables a shared clone to execute in multiple scenarios as long as it keeps receiving identical events from corresponding federates in all scenarios in which it participates. This design aims to avoid repeating identical computation amongst scenarios as much as possible. The shared clone persists in this mode until the condition for triggering passive cloning is met.

A typical shared clone is shown in Figure 3.2. The shared clone (SC) executes in n concurrent scenarios, and those scenarios are said to be SC's **related scenarios** (written as $RELASCEN = \{relaScen[i] | i = 1, 2, \dots, n\}$). Let $X = \{x[i] | i = 1, 2, \dots, n\}$ denote the

¹ In our design, each federate (clone) is associated with a scenario ID, from which partnership can be distinguished amongst different federates (clones).

set of sibling clones that are created from the same simulation federate x , with $x[i]$ operating in $relaScen[i]$. SC may receive events from $x[i]$ and generate events for each related scenario. It is unnecessary to perform extra checking on the events generated by SC , as those events must be identical in any scenario. However, the events received by SC have to be checked.

Definition 1 (Sensitive Update) If an object instance $ObjX$ registered by federate x has been discovered by SC , then SC treats $ObjX$ and its image objects (see section 3.3.1) as a set of sensitive object instances. Obviously, the object class to which $ObjX$ belongs must be published by x and subscribed by SC . Let $inEv[i]$ represent an update of $ObjX$ (or its image objects) issued by any $x[i] \in X$, then $inEv[i]$ is defined as a **sensitive update** for the shared clone SC .

Definition 2 (Sensitive Interaction) Any interaction class published by x and subscribed by the shared clone SC is regarded as a sensitive interaction class. Let $inEv[i]$ represent an interaction of any sensitive interaction class sent by any $x[i] \in X$, then $inEv[i]$ is defined as a **sensitive interaction** for the shared clone SC .

A **sensitive event** is defined as a sensitive update or interaction. A shared clone may present non-sensitive events straightforwardly to its simulation model without extra checking, whereas it has to check each sensitive event before conveying it to the simulation model. A non-sensitive event can be an event sent by another shared clone executing in all related scenarios of the receiver. A sensitive event needs to be compared with corresponding counterpart events. In each round of event comparison, the first received sensitive event is referred to as the **target event** by subsequent counterpart events.

Definition 3 (Comparable Updates) Any two sensitive updates for a shared clone are **comparable** to each other only when following conditions are satisfied:

- *They carry equivalent timestamps*
- *They are updates of two individual image objects (or an original object and one of its image objects) representing the same original object*

Definition 4 (Comparable Interactions) Any two sensitive interactions are **comparable** only when following conditions are satisfied:

- *They carry equivalent timestamps*
- *They belong to the same sensitive interaction class*
- *They originate from two individual sibling clones*

A shared clone should not compare received interactions that are not sent by sibling clones even they belong to the same interaction class. According to definition 3 and the definition of original and image object instance, it is obvious that comparable events must originate from sibling clones.

Definition 5 (Identical Events) Comparable events are called **identical** if they have the same associated attributes/parameters and the values of all attributes/parameters are identical.

Comparable events need to be checked to verify whether they are identical or not. If a shared clone detects any two comparable events are not identical, the shared clone has to perform passive cloning to handle this situation. On the other hand, the shared clone may remain intact if:

- *All received comparable events are identical*
- *The shared clone receives comparable events from all the sibling clones in the related scenarios before it is granted a simulation time greater than (or equal to) the target event's timestamp*

If the either condition is not met, it means that the shared clone has obtained different behaviors from related scenarios and requires passive cloning. As a consequence, the federate previously shared and its clones created in this passive cloning each operate as a normal clone in only one individual scenario (at least until the next decision point). Figure 3.3 depicts the relationship among the terms defined in this section.

Figure 3.3: Relationship between terms related to shared clones

3.4. Scenario Tree

In the context of distributed simulation cloning, each clone (federate) is an individual entity whereas each scenario is a dynamic group, which involves a changing combination of member clones. Each scenario reports simulation results independently; it is the basic unit in our consideration and discussion. Only active cloning can drive the creation of new scenarios. We utilize a tree data structure to represent the relationship and development of the scenarios.

Figure 3.4 gives an example of a cloning-enabled simulation in which *Fed A[0]* operates as an event publisher and *Fed B[0]* and *Fed C[0]* exchange events. Part (A) illustrates the details of the overall cloning procedure. Part (B) gives an abstraction of the scenario tree, in which each **parent** node (full circle) represents an occurrence of active cloning. The **leaf** nodes (empty circle) stand for active scenarios at the current simulation time. An active cloning results in spawning new scenarios, this is reflected in the figure as parent nodes who have multiple children. Each scenario is marked as $S[i]$ ($i = 0, 1, 2, \dots$). The scenario tree grows along the simulation time axis as follows:

Figure 3.4: Example of incremental cloning and scenario tree

- At simulation time 0 , there exists a single scenario $S[0]$. When simulation time is advanced to T_1 , *Fed B* meets a decision point and performs active cloning, splitting into clones $B[0]$ and $B[1]$. A new branch $S[1]$ is created in the scenario tree at this point. An event generated by $B[0]$ is named as event X while an event from $B[1]$ is called event X' . *Fed C[0]* keeps intact and this will continue as long as events X and

X' remain the same. *Fed C[0]* operates as a shared clone for the duration between time T_1 and time T_2 .

- At simulation time T_2 , an event X deviates from event X' , this incurs a passive cloning of *Fed C[0]* and results in the birth of clone *C[1]*. This passive cloning does not trigger any change in the scenario tree.
- At simulation time T_3 , *Fed C[1]* performs an active cloning, spawning off clone *C[2]*. A new scenario is created and marked as *S[2]* in the scenario tree.

A combinatorial explosion of scenarios in distributed simulation cloning may occur in some situations. The number of possible scenarios is determined by (1) the number of active cloning federates, (2) the times those federates perform active cloning and (3) the candidate choices that each decision point represents. Human intervention may reduce the combinatorial explosion, but it is difficult to reach a general solution [Agarwal05].

In practice, it is unlikely that such a combinatorial explosion will occur. For example in a supply chain simulation, one company may wish to examine its own decision strategies concurrently. It is unlikely that one company could manipulate its partners' internal decision policies. So in this case, there will be only one or a few federates that perform active cloning while the remaining federates perform passive cloning at the request of the active cloning federates.

From the above discussion we observe that different scenarios may have common member clones. The relationship amongst scenarios and clones can be complex and highly dynamic. There is a need for an identification and partitioning mechanism to manage the concurrent scenarios in the distributed simulation cloning procedure. This requires us to:

- Represent the relationship amongst scenarios
- Identify each scenario and clone to enable control

- Partition the event messages belonging to different scenarios
- Support the sharing of clones between scenarios
- Provide reusability to user federates while enabling the former functionalities

In chapter 6, we will address the above problems by presenting a recursive region division solution and a point region solution to manage the scenarios using Data Distribution Management (DDM).

3.5. Summary

This chapter gives the basic theory and identifies issues involved in distributed simulation cloning. Different types of cloning are introduced, such as active and passive cloning, total and incremental cloning. Key terms are defined, which include decision point, normal clone and shared clone, partner federate and sibling clones. Although some concepts, e.g. decision point, have been identified in [Hybinette97] for parallel simulation cloning, our thesis is the first to provide a systematic definition of simulation cloning at both the federate and distributed simulation level.

This chapter also describes the theory of incremental cloning. Sensitive events and related concepts are explained. Similar ideas for incremental cloning have subsequently been adopted in the parallel simulation domain [Hybinette04].

A scenario tree is designed to represent the evolvement of concurrent scenarios resulting from cloning and the dynamic relationship amongst them. This chapter clearly describes the interdependencies between scenarios, clones, RTI objects and events which is missing in previous approaches to cloning of HLA-based simulations.

Chapter 4: Alternative Solutions for Cloning in HLA-based Distributed Simulation

Simulation cloning is designed to satisfy the requirement of examining alternative scenarios concurrently. In this chapter, alternative solutions are proposed and compared from both the qualitative and quantitative points of view. In terms of federation organization, candidate solutions can be classified as either single-federation or multiple-federation. In order to guarantee the correctness and optimize the performance of the whole cloning-enabled distributed simulation, the single-federation solution requires an additional mechanism to isolate the interactions amongst alternative executions. Data Distribution Management (DDM) is one of the candidate approaches. To measure the trade-off between complexity and efficiency, we also introduce a series of experiments to benchmark various solutions at the RTI level.

4.1. Single-federation Solution versus Multiple-federation Solution

When a federate is cloned, we can create multiple federations to meet the demand of executing alternative scenarios or generate new federates to operate in the original federation without intervening in the execution of any other scenario. This thesis uses Multiple-federation Solution (**MF**) to denote the former design, and Single-federation Solution (**SF**) to denote the latter one.

Figure 4.1 depicts the cloning of a simulation using both solutions. Federates inside the dashed rectangle represent the clones originating from a common ancestor. Following the cloning action that is triggered at the decision point, both original federates (*A* and *B*) duplicate themselves to form two different scenarios. By applying the SF solution, the new federates *Fed A[1]* and *B[1]* participate in the original federation *RTI[0]* and there is no need for an additional federation to support the new scenario (labeled“1”). By

applying the MF solution, *Fed A[1]* and *B[1]* form another federation *RTI[1]* to facilitate another scenario (labeled “1”).

Figure 4.1: Example of Single-federation Solution and Multiple-federation Solution

Table 4.1: Comparison between Single-federation and Multiple-federation Solutions

Issues	Single-Federation	Multiple-Federation
Interaction	Additional mechanism is needed to deal with unnecessary event-crossing amongst concurrent scenarios	Interaction between clones in various scenarios is isolated by default
Synchronization	Unnecessary Synchronization amongst clones in different scenarios is inevitable	Synchronization amongst clones in different scenarios is eliminated
Complexity of Sharing	Clone sharing is available in a single federation	Clone sharing is difficult amongst federations
Robustness	If an RTI instance crashes, the simulation will fail	Multiple RTI instances for one user federation are maintained, thus one RTI instance crash will not result in failure of the whole simulation
Management of Clones	Management of clones is easier inside the same federation	Management of clones crossing multiple federations is indirect

The two solutions mentioned above involve different research issues and problems, especially at the RTI level. Table 4.1 gives a comparison showing the advantages and disadvantages of both solutions from different viewpoints. To make a trade-off amongst these issues is difficult. As RTI does not provide destination-specific delivery in its Object Management services, it is mandatory that the Single-Federation solution requires an additional mechanism to isolate interactions amongst clones in different scenarios.

4.2. DDM versus Non-DDM in Single-Federation Solution

To isolate separate scenarios, a straightforward approach is to filter events at the receiver side. Each event is attached with the exclusive identity of the scenario of the sender. The receivers discard those events from other scenarios and merely reflect those belonging to the same scenario. A minimal effort is required to enable this filtering in addition to the standard RTI services.

It is also possible to use Data Distribution Management (DDM) services to partition scenarios in the overall cloning-enabled distributed simulation. In general, the purpose of DDM services is to reduce the transmission and receipt of irrelevant data by the federates. DDM services are employed by the federates, which can be interpreted as data producers and consumers, to assert properties of their data or to specify their data requirements respectively based on specified regions. The RTI then distributes the data from the producers to the consumers based on the match between the properties and the requirements. DDM controls the efficient routing of class attributes and interactions via the RTI.

Routing spaces are a collection of dimensions that represent coordinate axes of the federation problem space with a bounded range [Morse01]. A region defines a multidimensional sub-space in the routing space by defining the lower bound and upper bound on each dimension of the routing space. Figure 4.2 illustrates an example of

routing space and regions, in which two dimensions “X” and “Y” define a routing space. Three regions R1~3 are indicated as the large rectangles in the figure. R1 overlaps with R2 in area O1 and R2 overlaps with R3 in area O2. Obviously, there is no overlap between R1 and R3.

Figure 4.2: Example of routing space and regions

In DDM, data producers and consumers specify their data properties and data requirements by providing update regions and subscription regions. Data connection will be established between a pair of federates only when an update region and a subscription region overlap. With this property, DDM seems to be a natural candidate for developing the mechanism to restrict the interaction amongst clones to within the same scenario.

The RTI offers a region modification method to enable the dynamic change of sub-space without creating a different region. The change takes effect immediately after notifying the RTI. This feature enables the adjusting of a clone’s “characteristic” region at runtime, thus dynamic routing and filtering of events from one clone to different scenario combinations can be realized.

The RTI specification leaves the DDM implementation details to the RTI implementers. The current DMSO RTI-NG supports a number of DDM data filtering strategies, one of which is the **StaticGridPartitioned** strategy. This strategy partitions individual spaces into a grid in which each grid cell is assigned a separate reliable and best-effort channel

[Hyett02]. Wise use of the StaticGridPartitioned strategy can increase performance through sender-side filtering.

By assigning a scenario-specific region to one set of clones, the interactions will automatically be confined to this scenario. However, this incurs some extra overhead for managing the regions and increases the complexity of implementation. It is not necessary to introduce the DDM mechanism to the multiple-federation solution, as the communication traffic has already been confined within each federation. Thus, there are three candidate solutions for cloning HLA-based distributed simulations: Multiple-federation Solution (**MF**), DDM Single-federation Solution (**DSF**) and Non-DDM Single-federation Solution (**NDSF**). In order to evaluate these solutions, another important criterion is their efficiency. The trade-off between efficiency and complexity is a main concern of the distributed system designer. Section 4.4 will introduce the benchmark experiments to measure the overall performance of the three solutions in terms of execution time.

4.3. Middleware Approach

No matter what kind of solution is chosen to perform simulation cloning, a critical principle is the reusability of the user's program. Our solutions should minimize the modification to the user's existing code even though it is difficult to eliminate all extra complexity when enabling simulation cloning. We propose a middleware approach to hide the complexity as indicated in Figure 4.3.

We extend the standard RTI to RTI++ to encapsulate cloning operations directly related to the RTI while presenting the standard RTI interface to the user. The user code still uses the standard RTI interface while the enhanced functionality remains transparent to the user. The middleware sits between the user's code and the real RTI, and contains a library for cloning management and the RTI++.

Figure 4.3: Example of middleware for cloning

```

Get region associated with current clone federate

    currentRegion = scenManger ->getCloneRegion();

Override standard RTI functions with DDM enabled services

For example:
RTIambassadorPlus::subscribeObjectClassAttributes (theClass, attributeList,...){
    (RTI::RTIambassador*this)->subscribeObjectClassAttributesWithRegion(theClass,
    currentRegion, attributeList,...);
}
...

```

Figure 4.4: Example of RTI++ implementation

As mentioned previously, any of the solutions discussed should work as an underlying control mechanism with transparency to federate codes. Hiding the implementation details while enabling the management of scenarios can maximize the reusability of the user's existing simulation model. Our RTI++ software is built to encapsulate the cloning-related modules while maintaining the same RTI interface to the calling federate code. Figure 4.4 gives an example of pseudo codes for implementing the above inside the middleware.

4.4. Benchmark Experiments and Results

The primary objective of the experiments is to provide some criteria on complexity and efficiency to help us decide what kind of cloning method we should adopt: namely MF, DSF and NDSF as discussed.

We study the complexity and efficiency using three factors: (1) **Lower Bound Time Stamp (LBTS) computation** for time advance [RTING02], (2) the **interaction latency** between federates and (3) **the load of the federate processes**. The LBTS computation determines the speed at which federates can advance simulation time. The interaction latency means the communication overhead for delivering events amongst federates. The load of the federate processes is a measure of the computational resources required for executing a simulation. The experiments explore how these crucial factors impact the overall performance. The experiments report the execution times of adopting alternative solutions in terms of the efficiency of the LBTS computation and the interaction latency. They also measure the CPU utilization of each node to study the load of federate processes under different circumstances.

4.4.1. Experiment Design

The experiments use three PCs in total (PC 1, 2 and 3 in Figure 4.5 and 4.6), in which PC 2 executes the RTIEXEC and FEDEX processes [RTING02]. The federates that run at each independent PC are enclosed in a dashed rectangle. In our case *Fed A[i]* and *Fed B[i]* ($i \geq 1$) occupy PC 1 and PC 3 respectively. The PCs are interlinked via an EtherFast 100 5-port Workgroup Switch, which forms an isolated subnet to avoid fluctuation incurred by additional network traffic. The PCs' configuration is as follows:

- Intel 1700 MHz Pentium IV
- 256 Mbytes of RAM

- Windows 2000 Professional
- DMSO RTI NG 1.3 V4

Figure 4.5: Test bed for MF solution

Figure 4.6: Test bed for DSF & NDSF solution

The experiments emulate the simulation cloning process by increasing the number of identical federates. In Figure 4.5 and 4.6, *Fed A[1]* and *B[1]* form a pair of initial federate

partners, which represent the federates to be cloned. $Fed A[i]$ and $B[i](i>1)$ denote the i th clones of the two original federates, respectively. The federates are tailored based on the DMSO standard benchmarking programs [DMSO03].

As indicated in Figure 4.5, each pair of $Fed A[i]$ and $B[i]$ comprises an exclusive federation, which is denoted as $RTI[i]$. We employ this set of scenarios to perform the benchmarking experiments for the MF Solution. In Figure 4.6, all federates form one single federation, accordingly we use this set of scenarios to measure the performance of NDSF and DSF solutions.

Each federate is time constrained and time regulating, or neither. In one run, each federate updates an attribute instance and receives an acknowledgement from its partner (from $Fed A[i]$ to $Fed B[i]$, and vice versa) for 10,000 times with a payload of 150 bytes. A federate merely reflects the events with identical ID to itself. In other words, $Fed A[i]$ will discard any events not generated by $Fed B[i]$, and vice versa. In TSO mode, federates advance federate time from 0 to 10,000 with timestep = 1 and lookahead = 1. To investigate the efficiency for time synchronization amongst federates, we also examine the execution time for the standard Time Advancement benchmarking federates using both the MF and SF solutions.

Each federate can also be set as DDM enabled and Non-DDM. An exclusive ID is shared between $Fed A[i]$ and $Fed B[i]$. In DDM enabled mode, each pair of federates has an associated region, which is pair-specific and non-overlapping to any other region. As previously mentioned, the **StaticGridPartitioned** strategy partitions individual spaces into a grid in which each grid cell is assigned a separate reliable and best-effort channel [Hyett02]. The size of the grid is specified by a parameter *NumPartitionsPerDimension* (*NPPD*). To optimize the utilization of communication channels, we split the full dimension [MIN_EXTENT, MAX_EXTENT) evenly into the same number of segments as *NPPD*, which is set to the number of scenarios (federate pairs). The middle point of

each segment will be defined as one region. We set the region associated to the k th pair of federates as:

$$\left\lceil \frac{(2k-1)(\text{MAX_EXTENT} - \text{MIN_EXTENT})}{2 \times \text{NPPD}} + \text{MIN_EXTENT} \right\rceil, \\ \frac{(2k-1)(\text{MAX_EXTENT} - \text{MIN_EXTENT})}{2 \times \text{NPPD}} + \text{MIN_EXTENT} + 1).$$

In these experiments, all federates subscribe and publish the same object class, and associate the designated region to their subscription and updates.

4.4.2. Benchmark Results and Analysis

Some abbreviated notations are used to denote the properties of the federates, as listed in Table 4.2. In order to investigate the factors that impact the performance of alternative solutions, several series of experiments are designed as indexed in Table 4.3. The notations are the same as in the previous discussion. Combining the time features and solutions together, we perform 8 series of experiments in total. In each series of experiments, the number of federates increases by 1 pair each time from 5 pairs to 14 pairs.

Table 4.2: Notations of the federate attributes

Notations	Meaning
TSO	Federates are Time Regulating /Time Constrained and use Time Stamp Order update and reflection
RO	Federates use Receive Order update and reflection only
SYN	Standard DMSO time advancement benchmarking application
MF	Experiment in Multiple-Federation mode
DSF	Experiment in Single-Federation mode using DDM
NDSF	Experiment in Single-Federation mode without using DDM

Table 4.3: The index of the experiments

	TSO	RO	SYN
NDSF	Experiment 1: TSO-NDSF	Experiment 4: RO-NDSF	Experiment 7: SYN-SF
DSF	Experiment 2: TSO-DSF	Experiment 5: RO-DSF	
MF	Experiment 3: TSO-MF	Experiment 6: RO-MF	Experiment 8: SYN-MF

4.4.3. Comparing Alternative Cloning Solutions using TSO Federates

Figure 4.7: Execution time comparison between cloning solutions using TSO federates

Figure 4.8: CPU Utilization comparison between cloning solutions using TSO federates

Figure 4.7 reports the execution time of TSO federates using the three different solutions. The execution time of the TSO-NDSF scenarios has an obvious increase when the number of federate pairs reaches 7. When no DDM services are used, the execution time increases sharply with more and more TSO federates joining the same federation. The topmost value (~2800 seconds, 14 pairs) is over 5 times the start value (~500 seconds, 5 pairs). On the contrary, the execution times of both the TSO-DSF and TSO-MF scenarios stay at a relative stable level, about 500 seconds, in spite of the increase in the number of participating federates.

In single-federation mode, all federates interact with each other through the same RTI, the computation load of LBTS increases with the number of federates. In the TSO-NDSF scenarios, of more importance is that each federate not only receives useful events from its partner federate but also has to filter out some useless messages from all other federates as they belong to other scenarios. The overall communication traffic through the RTI is proportional to $C_n^2 = \frac{n(n-1)}{2}$, where n is the number of federates. Only $\frac{1}{(n-1)}$ of the total incoming events make sense to one particular federate pair. Other unnecessary communication and reflection increase the overhead in the RTI dramatically. The DDM services strictly confine the interaction to the pair of federates with a common region. This optimization results in the significantly improved performance as indicated in the curve “TSO-DSF”.

In multiple-federation mode, a federate interacts with its partner through an exclusive RTI. Also the LBTS computations will take place between a pair of federates independently. These positive factors lead to the much better performance compared with the TSO-NDSF scenarios.

CPU utilization percentage reports the processor activity in the computer. This counter sums the average non-idle time of all processors during the sample interval and divides it by the number of processors. The CPU utilization results in Figure 4.8 indicate that the TSO-NDSF scenarios consume much more system resource than the other two solutions. However, the TSO-DSF and TSO-MF scenarios also have an uptrend in terms of CPU utilization. The CPU utilization of the TSO-NDSF scenarios reaches about 90% after the number of federate pairs exceeds 7. These experiments are a combination of complex executions including LBTS and TSO events receiving and reflecting. We attempt to isolate these two factors in the following experiments.

4.4.4. Comparing Alternative Cloning Solutions using RO Federates

Figure 4.9: Execution time comparison between cloning solutions using RO federates

Figure 4.10: CPU Utilization comparison between cloning solutions using RO federates

In order to further investigate the computational complexity in these solutions, we disable the time feature of federates and reapply the three solutions to them. Execution time and CPU utilization are presented in Figure 4.9 and Figure 4.10.

Similar to the previous experiments, the execution time of RO-NDSF scenarios increases rapidly with the number of federates. The peak execution time (~1200 seconds, 14 pairs) is about six times the start value (~200 seconds, 5 pairs). The execution time of RO-NDSF scenarios always has a greater value than that of RO-DSF and RO-MF scenarios. The RO-DSF and RO-MF scenarios have execution times that fluctuate slightly from 100 seconds to 200 seconds. From the discussion of the TSO scenarios, the extra

communication and reflection lower the performance of RO-NDSF solutions significantly.

Figure 4.10 also shows that the RO-NDSF scenarios consume much more system resource than the other two solutions. The CPU utilization of the RO-NDSF scenarios jumps to 100% after the number of federate pairs exceeds 6. The RO-DSF and RO-MF scenarios have a very low CPU utilization (less than 10%) until there are more than 12 pairs of federates.

4.4.5. Comparing Alternative Cloning Solutions using Time Advancement Benchmark Federates

In order to have a better understanding of how another factor, the LBTS calculation, impacts the performance, we re-examine the SF and MF solutions by introducing the standard Time Advancement Benchmark Federates [DMSO03]. Execution time and CPU utilizations of both solutions are shown in Figure 4.11 and Figure 4.12. As DDM does not intervene in the synchronization amongst federates, this series of benchmarks ignores the DDM mechanism.

We can conclude that the LBTS calculation does not make a significant difference to the execution time of either the MF or SF scenarios with an increase in the number of federates. The RO-NDSF and TSO-NDSF scenarios show a fast increasing execution time. It means that the reduction of interaction is the key to the optimization of the overall system performance in the simulation as long as the number of federates stays in a reasonable range.

Figure 4.11: Execution time comparison between cloning solutions using Time Advancement federates

Figure 4.12: CPU utilization comparison between cloning solutions using Time Advancement federates

4.5. Summary

This chapter proposes alternative solutions for distributed simulation cloning, namely the Single-Federation Solution (with and without DDM) and the Multiple-Federation Solution. The alternative candidate solutions are compared in terms of robustness, complexity and efficiency.

In the context of the Multiple-Federation (MF) solution, multiple federations operate in parallel to explore alternative scenarios. Thus, the MF solution surpasses in robustness

and federate synchronization. If a failure occurs in a scenario, the failure will not affect the remaining scenarios as they are in different federations. However, scenario management becomes more complex. In particular, the requirement of dynamic clone sharing by incremental cloning is extremely difficult in the MF solution as the shared clones need to operate in different federations. The Single-Federation solution (SF) thus has an advantage in cloning control and clone sharing.

The benchmark results indicate that in general MF exhibits much better performance than NDSF. The interaction among federates is the main factor that impacts the execution speed. In the case where there is a high data exchange, the performance can be improved dramatically by applying DDM to NDSF. The calculation of LBTS makes no significant difference in both the MF and SF solutions. The results show that DSF performs as well as MF in terms of efficiency. Considering the reduction of implementation complexity and convenience for scenario management, the Single-Federation solution applying DDM (DSF) is chosen for cloning distributed simulations in this current study.

Chapter 5: A Decoupled Federate Architecture

This chapter introduces the idea of decoupling the Local RTI Component from a normal HLA federate, to give a Decoupled Federate Architecture. This architecture forms the basis for enabling federate cloning at runtime.

5.1. Problem Statement

Figure 5.1: Abstract model of a simulation federate

A normal simulation federate can be viewed as an integrated program consisting of a simulation model and Local RTI Component (LRC), as shown in Figure 5.1. The simulation model executes the representation of the system of interest, whereas the LRC services it by interacting and synchronizing with other federates. In a sense, the simulation model performs local computing while the LRC carries out distributed computing for the model.

Cloning of a federate occurs at a decision point to enable different candidate actions to be performed. “Cloning” implies that the new clones of one particular federate should initially have the same features and states as the original federate both at the RTI level and at the simulation model level. This is to ensure the consistency of simulation state. For example, at the RTI level, clones must have subscribed to the same object classes and registered the same object instances etc. At the simulation model level, the clones should

have the same program structure, data structures, objects and variables; all these program entities should have identical states. Immediately after the cloning, the clones will be given some particular parameters or routines to execute in different paths.

One possible solution is to introduce a state saving and replication mechanism to the simulation federates, allowing the simulation federate to store snapshots of all the system states. When cloning occurs, new federate instances are started and initialized with stored states. However, users model their simulations in a totally free manner. It is unlikely that a generic state saving and replication mechanism can be provided that will be suitable for any simulation federate. Besides, as the LRC is not designed to be replicated, direct cloning of a federate can lead to unpredictable and uncontrollable failure at the RTI level.

Even given such a mechanism, it is unlikely that all simulation developers will use the same standard package to model their simulations. Without the ability to customize the user's simulation code, it is almost impossible to make snapshots of all system states of any federate. Furthermore, the principle of reusing existing federate code increases the difficulty of this task. On the other hand, the standard HLA specification makes it relatively easy to intercept the system states at the RTI level. Using a middleware approach, one may save and replicate the RTI states while enabling transparency. Thus we can see that the simulation model and the LRC have very different characteristics. Therefore, it suggests a distinction should be made between these two modules for cloning a federate.

Decoupling the LRC from the simulation model has further advantages, such as providing a way of integrating simulation packages or legacy simulations into an HLA-based distributed simulation. Straßburger et al presented different solutions including a gateway program approach to adapt models built using Commercial-off-the-shelf (COTS) simulation packages to the HLA [Straßburger98]. Under this approach, a simulation application consists of a COTS-based model and a gateway program that couples the

model with the RTI. Thus, previously standalone models may interoperate with each other using the HLA specification with user transparency maintained. Similarly, McLean and Riddick designed a Distributed Manufacturing Adapter to integrate legacy simulations with the HLA/RTI [McLean00]. This design aims to simplify the integration while gaining the capabilities of HLA-based distributed simulations.

Morse et al proposed an Extensible Modeling and Simulation Framework (XMSF) to allow various simulations to take advantage of Web-based technologies [Morse03]. They developed a web-enabled RTI implementation to provide support for HLA compliant simulations to communicate with the RTI through SOAP and BEEP. Several obvious advantages are identified such as: platform crossing, language crossing and interoperability of federates (for example, breaking through the constraints of firewalls etc).

Different from the above work, this project focuses on using a decoupled federate architecture to enable simulation cloning, while facilitating reusability, simulation independency and user transparency. The design also needs to ensure execution efficiency. The technology may also be used to provide a solution to other issues such as: fault tolerance, supporting a Grid/Web enabled architecture and load balancing, as discussed in chapter 9.

5.2 Virtual Federate and Physical Federate

In the context of the decoupled architecture, a federate's simulation model is decoupled from the Local RTI Component. A virtual federate is built up with the same code as the original federate. As HLA only defines the standard interface of RTI services, we are able to substitute the original RTI software with our customized RTI++ library without altering the semantics of RTI services (see chapter 4). Figure 5.2(B) gives the abstract model of the virtual federate. Compared with the original federate model illustrated in

Figure 5.1, the only difference is in the module below the RTI interface, which remains transparent to the simulation user.

Figure 5.2: Decoupled federate architecture

A physical federate is specially designed as shown in Figure 5.2(A). The physical federate associates itself with a real LRC. Physical federates interact with each other via a common RTI. Both virtual federates and physical federates operate as independent processes. Reliable external communication channels, such as Inter-Process Communication (IPC), Socket, SOAP or other out-of-band communication mechanisms, bridge the two entities into a single federate executive [Stevens98]. To ease discussion, this paper uses the abbreviation “**ExtComm**” to denote those alternative communication channels and assumes messages are delivered / received strictly in FIFO manner.

All the components inside the dashed rectangle form a **Middleware** module between the simulation model and the real RTI. Within the virtual federate, the **newRTIAmb** contains customized libraries while presenting the standard RTI services and related helpers to the simulation model. This module can be designed to contain all other management modules for the objectives mentioned previously. The **fedAmb** serves as a common callback to the user federate, which is freely designed by the user and independent of the decoupled approach. The **newRTIAmb** handles the user's RTI service calls by converting the method together with the associated parameters into ExtComm messages via the **Messaging Protocol**. The protocol mainly defines a mapping between an ExtComm message and the RTI method it represents. For example, an RTI_UPDATE message indicates that the virtual federate has invoked the RTI method *updateAttributeValues()*. The protocol can also be extended for other purposes, such as manipulating the physical federate. The ExtComm conveys these messages immediately to the physical federate for processing in a FIFO manner.

The physical federate converts an RTI call message generated from the virtual federate into the corresponding RTI call through its own messaging protocol layer. The **RTIAmb** module executes any RTI service initiated by the simulation model prior to passing the returned value to the ExtComm. The **phyFedAmb** serves as the callback module of the physical federate to respond to the invocation issued by the real RTI. Within the **phyFedAmb** module, the messaging protocol is employed to pack any callback method with its parameters into ExtComm messages. The ExtComm enqueues the callback message to the **Callback Processor** module at the virtual federate. Through the messaging protocol, the callback processor activates the corresponding **fedAmb** method implemented by the user. From the simulation users' perspective, a combination of one virtual federate and its corresponding physical federate operates as a simulation federate in the context of the decoupled architecture. The federate combination performs an

identical execution to a normal simulation federate with the same code as the virtual federate. In future discussion, we will explicitly use “normal federate” to refer to a traditional federate that directly interacts with the RTI. By default, in the discussion of this paper a federate contains a virtual federate module and a physical federate module.

5.3 Inside the Decoupled Architecture

As discussed above, the decoupled approach interlinks a virtual federate and the physical federate into a simulator that performs an identical simulation to the corresponding normal federate. This section gives the details of how an RTI service call is executed and the callback is invoked in the decoupled federate architecture.

Figure 5.3: Executing an RTI call in the decoupled architecture

Figure 5.3 depicts the procedure where a simulation model initiates an RTI call and waits for a return from the real RTI, using the *updateAttributeValues* method as an example. The procedure is as follows:

- The virtual federate invokes the redefined *updateAttributeValues* method.
- Inside the *updateAttributeValues* method, the *packMsg* routine extracts the data stored in the *AttributeHandleValuePairSet* (AHVPS) and packs them together with other parameters such as the associated timestamp, object instance handle and tag into an RTI_UPDATE message.
- The ExtComm enqueues the RTI_UPDATE message to the physical federate. The virtual federate switches to waiting mode for the returned message.
- Once the physical federate receives the ExtComm message, it invokes the *unpackMsg* routine to process it according to the associated type, RTI_UPDATE.
- A new AHVPS object and related parameters are recovered based on the ExtComm message and passed to the *RTI::updateAttributeValues*, which invokes the real RTI service.
- On the accomplishment of this *RTI::updateAttributeValues* call, the physical federate acknowledges the virtual federate with an ExtComm message containing the returned value.
- The *updateAttributeValues* call finishes and the data retrieved from the acknowledgement message is returned to the simulation model.

From the user's point of view, the initiation and accomplishment of an RTI call are identical to the original normal federate. The semantics of RTI services are kept intact in the decoupled approach.

The RTI software has an interface that provides flexible methods to the user for packing update data and leaves the transmission details transparent. The user can create update data of variable lengths. However, most low level communications do not offer flexible methods to handle message delivery without agreement of lengths between source and

destinations. For example, most IPC mechanisms have limitations in message size and buffer size. The **Message Queue** based on Solaris defines the maximum queue length as 4096 bytes [Stevens98]. The message sender and receiver must agree with each other on the same message length. If a fixed message size is defined for ExtComm messaging, it may incur some unnecessary overhead. A fixed large size is inefficient in transmitting small messages. On the other hand, a fixed small size increases the overhead for packing, delivering and unpacking a large number of small packets in the case of processing large messages. Therefore a simple protocol is proposed for messaging between the virtual federate and physical federate. We define a small message size (MSG_DEF) and a large message size (MSG_LG) for assembling user data into packets. A special “PREDEFINE” packet is used to notify the receiver if large or multiple packets are to be sent for a single data block. Figure 5.4 gives the messaging details based on this simple protocol.

Figure 5.4: Messaging between virtual federate and physical federate

Figure 5.5: Conveying callbacks to the virtual federate

The RTI communicates with a federate via its Federate Ambassador provided by the user [DMSO02]. In the DMSO RTI, a federate must explicitly pass control to the RTI by invoking the *tick* method. For example, the RTI delivers the Timestamp Order (TSO) events and Time Advance Granted (TAG) to a time-constrained federate in strict order of simulation time, which coordinates event interchange among time regulating and time constrained federates in a correct and causal manner. Therefore, the decoupled architecture should guarantee that (1) the Federate Ambassador at the user federate works in a callback like manner and (2) callback methods are invoked in the correct order. Figure 5.5 depicts how to realize these functionalities. To ease discussion, we assume the physical federate will get the callbacks shown in Figure 5.5. This procedure is illustrated by the following steps:

- The virtual federate invokes the routine *newRTI::tick* and the latter sends out an RTI_TICK message to the physical federate.
- The physical federate calls the real RTI *tick* according to the RTI_TICK message.
- The LRC acquires control and delivers events to the *phyfedAmb* module of the physical federate in a strict order.
- In each callback method invoked, the data sent by the RTI is enqueued to the callback ExtComm channel. The routine inside the *newRTI::tick* accesses the queue for the virtual federate.
- As long as the RTI_TICK_DONE message is not detected, the callback processor continuously processes the messages in a FIFO order while activating the corresponding method in the *fedAmb* module based on the messaging protocol.
- At the physical federate side, once the RTI finishes its current work and passes control to the physical federate, the latter returns an RTI_TICK_DONE message to the virtual federate.
- On receiving the RTI_TICK_DONE message, the virtual federate accomplishes the *newRTI::tick*, and control is returned to the caller immediately.

The real RTI starts to take charge only when the physical federate explicitly invokes *RTI::tick*. On the other hand, the *newRTI::tick* can only return when the real RTI finishes its work. As the ExtComm channels work in a FIFO manner, the order of each callback method invoked at the physical federate is identical to the sequence in which the callback processor at the virtual federate processes the data. From the user's perspective, the callback mechanism based on the decoupled approach executes the equivalent operations to the normal federate. It guarantees consistency in presenting messages from the real RTI to the simulation model and also ensures user transparency.

The decoupled architecture requires an additional ExtComm communication layer although it performs exactly the same computation as the corresponding normal federate. The external communication may incur some extra overhead. To investigate the overhead incurred by the decoupled approach, a series of benchmark experiments has been performed to compare with the normal federates. Section 5.5 reports and analyzes the experimental results in terms of event transmission latency and synchronization efficiency.

5.4. Federate Cloning Procedure

Using the decoupled approach can solve the problem caused by making copies of the LRC at runtime in the case of executing traditional federates. Cloning a federate can be achieved by replicating the virtual federate process and starting an additional physical federate instance with restored system state.

As illustrated in Figure 5.6, at runtime the middleware intercepts the invocation of each RTI service method. The interceptor logs all the RTI system states into stable storage. Some RTI states are relatively static, such as the federate identity, federation information, the published/subscribed classes and time constrained/regulating status. Other states include the registered or deleted objected instances, and granted federate time. Some event data may also need to be saved, such as sent and received interactions, updated and reflected attribute values of object instances, etc.

As soon as a federate reaches a decision point, the cloning conditions are checked to decide whether cloning is required or not. If necessary, the middleware spawns clones of the federate immediately to explore alternative execution paths. From the perspective of the federate making clones, the simulation cloning procedure can be described as follows (see Figure 5.7):

Figure 5.6: Saving RTI states

Figure 5.7: Federate cloning

- **Copying simulation model.** The cloning manager within the middleware makes the specified number of clones of the virtual federate (simulation model).
- **Initiating physical federates.** The cloning manager initiates an individual physical federate for each clone of the virtual federate and hooks up the two new processes into a whole.
- **Replicating states.** A clone's physical federate is initialized with the stored system states from the parent federate, after which a new clone of the original federate is formed.

The distributed simulation cloning algorithms will be described in detail in chapter 7.

5.5. Benchmark Experiments and Results

The decoupled architecture separates the simulation model and the LRC into independent processes. In contrast, in a normal federate these operate within the same memory space, which may be superior in terms of execution efficiency. In order to investigate the overhead incurred in the proposed architecture due to decoupling, we perform a series of benchmark experiments to compare the decoupled federate with a normal federate.

The experiments study the scalability by emulating the simulation cloning procedure based on the decoupled architecture using the IPC *Message Queue* [Stevens98] as the external communication to bridge the virtual and physical federate. Through these experiments, we also find out the overhead of adopting the *Message Queue* as the external communication backbone. The performance is compared in terms of latency and time advancement calculation. Latency is reported as the one-way event transmission time between one pair of federates. The time advancement performance is represented as the time advance grant rate.

5.5.1. Experiment Design

The experiments use three computers in total (two workstations and one server), in which the server executes the RTIEXEC and FEDEX processes (see Figure 5.8). The federates that run at one independent workstation are enclosed in a dashed rectangle. In our case *Fed A[i]* and *Fed B[i]* ($i \geq 1$) occupy workstation 1 and workstation 2 respectively. The computers are interlinked via a 100Mbps-based backbone. Table 5.1 gives the configuration of the test bed.

The experiments study the scalability by increasing the number of identical federates. As shown in Figure 5.8, *Fed A[1]* and *B[1]* form a pair of initial federate partners, which represent the federates to be cloned in the original scenario. *Fed A[i]* and *B[i]* ($i > 1$) stand for the i th clones of the two initial federates respectively and form an independent

scenario. The architecture is used through all the benchmarks experiments and for both normal federates and decoupled federates.

Figure 5.8: Architecture of benchmark experiments for decoupled federate architecture

Table 5.1: Configuration of experiment test bed for examining decoupled federate architecture

Specification	Computers		
	Workstation1	Workstation2	Server1
Operating System	Sun Solaris OS 5.8	Sun Solaris OS 5.8	Sun Solaris OS 5.8
CPU	Sparcv9 CPU, at 900 MHz	Sparcv9 CPU, at 900 MHz	Sparcv9 CPU * 6, at 248 MHz
RAM	1024M	1024M	2048M
Compiler	GCC 2.95.3	GCC 2.95.3	GCC 2.95.3
Underlying RTI	DMSO NG 1.3 V6	DMSO NG 1.3 V6	DMSO NG 1.3 V6
Processes running on	Fed A[i]	Fed B[i]	RTIEXEC & FEDEXE

A DDM based approach is used to partition concurrent scenarios (see chapters 4 and 6). For the latency benchmark, each pair of federates have an exclusive point region associated to any event being exchanged. The federates are neither time regulating nor time constrained. In one run, each federate updates an attribute instance and waits for an acknowledgement from its partner (from *Fed A[i]* to *Fed B[i]*, and vice versa) for 5,000 times with a **payload of 100, 1k and 10k bytes**. The time interval in the ping-pong procedure will be averaged and divided by 2 to give the latency in **milliseconds**. A

federate merely reflects the events with identical region to itself. In other words, *Fed A[i]* only exchanges events with *Fed B[i]*.

As for the time advancement benchmark, all federates are time regulating and time constrained. Each federate has lookahead 1.0 and advances the federate time from 0.0 to 5,000.0 with timestep 1.0 using *timeAdvanceRequest* [DMSO02]. The results report the rate that the RTI issues *timeAdvanceGranted* (TAGs/Second).

5.5.2. Latency Benchmark Results

The latency benchmark experiments report the latency with three different payload sizes. From Figure 5.9 to Figure 5.11, we can see that no matter whether the payload size is small or large, the latency increases steadily with the number of federates. The increment becomes obvious when the number of federates exceeds 4 pairs (8 federates in total). As indicated in Figure 5.9 and Figure 5.10, when the payload is not greater than 1000 bytes, the latency varies from about 10 milliseconds for one pair of federates to about 30 milliseconds for 7 pairs of federates. The decoupled federate and normal federate show similar results in this situation, and the decoupled federates incur only slightly more latency than the normal ones.

As shown in Figure 5.11, when a bulky payload as large as 10,000 bytes is applied, the decoupled federates incur about 5 milliseconds extra latency to the normal ones. However the extra latency remains nearly constant with the number of federate pairs. The latencies for both types of federates increase more rapidly than the small payload cases. This is due to the extra overhead incurred by Inter-Process Communication, which becomes obvious with bulky data transmission between the physical federate and virtual federate. When the payload size and the number of participating federates are not too large, the decoupled federate has a similar performance to the normal federate in terms of latency.

Figure 5.9: Latency benchmark on decoupled federate vs. normal federate with payload 100 Bytes

Figure 5.10: Latency benchmark on decoupled federate vs. normal federate with payload 1000 Bytes

Figure 5.11: Latency benchmark on decoupled federate vs. normal federate with payload 10,000 Bytes

5.5.3. Time Advancement Benchmark Results

Figure 5.12: Time advancement benchmark on decoupled federate vs. normal federate

Another series of experiments is carried out to compare the decoupled federates and normal federates in terms of the speed of time synchronization. Figure 5.12 presents the

experimental results reported as the TAG rate. In the time advancement benchmark, the TAG rate decreases with the number of federates for both decoupled and normal federates. The rate decreases less rapidly when the number of federate pairs is greater than 4 (8 federates in total). The TAG rate is about 300 times per second for one pair of federates down to about 40 times per second for 7 pairs of federates. The results indicate that the performance of the decoupled and normal federates is very similar in terms of time advancement.

5.6. Summary

In this chapter, we have investigated the issues related to a decoupled federate architecture in large scale HLA-based distributed simulations. A federate is separated into a virtual federate process and a physical federate process, where the former executes the simulation model and the latter provides RTI services at the backend. A standard RTI interface is presented to support user transparency, while the original RTI component is substituted with a customized library. The architecture enables a relatively generic method of replicating the simulation model and also facilitates state saving and replication at the RTI level. The proposed approach guarantees the correctness of executing RTI services calls and reflecting RTI callbacks to the simulation model.

Benchmark experiments have been performed to investigate the overhead incurred by a decoupled federate architecture. The experimental results are compared for a decoupled federate and normal federate in terms of latency and time advancement performance. The results indicate that the decoupled architecture incurs only a slight extra latency in the case of a bulky payload and has a very close performance of time advancement compared with a normal federate. The results also indicate that an IPC mechanism, such as the *Message Queue*, can provide efficient communication that bridges the virtual and physical federates and is appropriate in designing a framework for federate cloning.

Chapter 6: Managing Scenarios

A federate spawns multiple clones to explore different new scenarios at decision points, thus the overall distributed simulation comprises multiple concurrent scenarios. Data Distribution Management (DDM) provides a method to partition concurrent scenarios in an HLA-based distributed simulation. This chapter details our design of two DDM-based solutions and analyzes their advantages and drawbacks.

6.1. Problem Statement

As multiple dynamic scenarios operate in a common overall distributed simulation, a mechanism is necessary to identify the scenarios and partition them, so as to manage them in a precise and efficient manner. Moreover, to minimize the overhead incurred by creating more and more scenarios, we need to reduce bandwidth requirements by only transmitting interactions or attribute updates where necessary. As a shared clone may operate in different scenarios dynamically as described in chapter 3, there needs to be a dynamic event “routing” and “filtering” mechanism to support cloning. To provide reusability of existing simulations, it is also necessary to hide the complexity of the above operations. In an HLA-based simulation, this means we need to maintain the standard HLA interface to the user’s programs while providing extended functionalities.

This chapter extends the idea of using the Data Distribution Management (DDM) mechanism to partition concurrent scenarios in a distributed simulation. In the benchmark experiments on DDM and Non-DDM cloning solutions in chapter 4, we compared the performance of these two approaches in terms of execution time using both Time Stamp Order and Receive Order cases. The performance of the DDM enabled approach significantly exceeds the Non-DDM one. This superiority is obvious in a large scale distributed simulation.

Two alternative solutions are introduced in this chapter for managing scenarios and identifying each clone and scenario. A **recursive region division solution** and a **point region solution** are discussed and compared. The former solution initializes each original federate with a region occupying the full dimension of the routing space. During the cloning procedure, the new clones inherit split sub-regions from their parent. The latter solution specifies a point region for each original federate and a new clone is given an additional point region on birth. Dynamic region combining is used for shared clones.

Our potential solutions for managing scenarios aim to provide the standard interface of Object Management (OM) or DDM services while employing additional DDM methods in the underlying middleware. Thus, it is possible to hide the DDM solution implementation behind the normal OM or DDM services interface in a transparent way. The solutions to be discussed focus on addressing the manipulation of regions and the mapping between each region and the scenarios.

Figure 6.1: Formation of a scenario tree

As discussed in chapter 3, the active cloning of federates incurs the spawning of new scenarios in the simulation. The concurrent scenarios can be represented as a tree developing along the simulation time axis. Figure 6.1, which is a simplified version of

Figure 3.4, illustrates how a scenario tree is formed. The issues to be considered in comparing the two alternative solutions include coding scenarios, region specification for each scenario and implementing the middleware approach. To illustrate the details of each solution, the example in Figure 6.2 is also used in the following sections in studying the coding scheme. Seven scenarios (labeled “a” to “g”) are present in the overall simulation following two active clonings at times T_1 and T_2 respectively.

Figure 6.2: An example of a scenario tree

In order to minimize the computation involved in DDM, we use only one single routing space having a single dimension for cloning. To ease discussion, we assume the federates being studied do not use DDM services in their models. However, federates that already use DDM services can also easily apply the solutions without changing the federate code. This can be achieved by associating another “cloning” dimension in the middleware to the existing DDM-enabled interactions & attribute updates when necessary.

The underlying middleware subscribes and publishes with the same region for a given clone. Each scenario is associated with an exclusive region, and there is no overlap with any other scenario’s region at any point during execution. Clones within the same scenario utilize a common scenario-specific region extent¹ unless they are shared clones. A shared clone has a merged region that exactly covers the scenario-specific extents of all

¹ In this thesis, extent is used to mean the interval [Lower_Bound, Upper_Bound] that defines the region.

the scenarios in which it operates. DDM also aids interactive control of the scenarios at runtime; external commands can be easily routed to the clones within a given scenario by associating the proper region to them. The approach will lessen the work of recognition and processing at the clone side.

6.2. Recursive Region Division Solution

The basic idea of the recursive region division solution is to divide the full dimension (i.e. $[\text{MIN_EXTENT}, \text{MAX_EXTENT}]$) from top to bottom. A federation is initialized with all federates having an associated region with extent $[\text{MIN_EXTENT}, \text{MAX_EXTENT}]$, thereby the initial scenario has a full dimension region. Once new scenarios are created, each scenario inherits a sub-region from its “parent” scenario under a region division algorithm. Thus the active cloning federate should divide its original region for the child clones, while its partner may still keep the region unchanged. In the example shown in Figure 6.1, immediately after time $T1$, *Fed B[0]*’s original region is split into two parts, *Fed B[0]* and *Fed B[1]* modify their region with the first and the second part respectively. The new regions indicate that *Fed B[0]* and *Fed B[1]* belong to scenario $S[0]$ and $S[1]$ respectively, however *Fed C[0]* will remain associated with the region of both scenarios $S[0]$ and $S[1]$. Thus the events from both *Fed B[0]* and *Fed B[1]* will be automatically routed to *Fed C[0]* without any modification to its region. This solution requires minimal work for shared clones. Region division keeps taking place along the development of the scenario tree.

The scenario tree can be reconstructed as a binary tree to ease coding. When n new scenarios are developed from one scenario branch, the following rule will be applied:

Given that $2^{k-1} < n \leq 2^k$, n is rewritten as $n = 2^{k-1} + m$, $m \leq 2^{k-1}$. Then the branch extends k levels downwards, the leftmost m nodes at level $k-1$ will always be expanded one further level.

Figure 6.3: Development of binary scenario tree

Following this rule, the scenario tree in Figure 6.2 is converted to a binary tree. Suppose that five new scenarios are created from the predecessor scenario node at time T_2 , the development of this branch is as indicated in Figure 6.3. The leaf nodes represent the newborn scenarios including the original one.

Figure 6.4: Binary scenario tree and scenario region code

As shown in Figure 6.4(A), we code a node's left branch as "0" and the right branch as "1" accordingly. Based on the binary tree, we link the branch codes together along the path from the root to the given leaf node, thus the exclusive code of each scenario is obtained. For example, scenario e is identified as "0011". The position of the scenario can be easily traced according to its identity.

A shared clone needs to distinguish events from different clones that are spawned by the same federate. We may also need to control the cloning procedure or update the system

state of one particular clone. Therefore it is necessary to identify each clone accurately. A clone always joins one or multiple scenarios; naturally the clone identity needs to cover this information. A direct scheme is to combine scenario codes and the federate name of the original ancestor to this clone, with format “<Scenario ID>&&<Federate Name>”. For example, a clone joins scenario e and its ancestor federate is named as “fab1”, then this clone may be coded as “0011&&fab1”. In the case of shared clones, a delimiter symbol is placed between scenarios; if necessary a wildcard is also introduced to reduce the identity length. The format is “<Scenario ID 1>:<Scenario ID 2>: ... :<Scenario ID N>&&<Federate Name>”, in which ‘:’ is a delimiter and ‘*’ is a wildcard. For example, if another clone of ancestor “fab2” is shared amongst scenarios a , b and g , this clone is coded as “0000*:1&&fab2”.

Figure 6.4(B) gives the region-coding scheme. The relationship between the position of a scenario node and its specific region is illustrated explicitly. The full dimension is segmented more and more densely with the increase in level depth.

At the k th level (mapping the nodes in the binary tree with depth equal to k), the full dimension is evenly partitioned into $n = 2^k$ segments. Each segment is given a binary code according to its index with the code length equal to the level depth. The segment code is length sensitive, e.g. the code “011” at level 3 differs from the code “0011” at level 4. For the purpose of illustration, assume that MIN_EXTENT = 0x00000000 and MAX_EXTENT = 0xFFFFFFFF, the following simple formula gives the calculation of the extent of the i th segment at level k :

$$\text{Lower Bound} = i \times 2^{32-k},$$

$$\text{Upper Bound} = (i + 1) \times 2^{32-k} - 1 \quad (6.1)$$

The code of each scenario coincides with the segment extent perfectly. The extent of the scenario-specific region can be directly obtained from the scenario code. Firstly we assign

the length of the scenario code to k , secondly we calculate $i = \text{atol}(\text{scenario code})$, then the extent of the region is available immediately from equation 6.1. For example, the code of scenario e is “0011”, then we have $k = \text{strlen}(\text{“0011”}) = 4$ and $i = \text{atol}(\text{“0011”}) = 3$:

$$\text{Lower Bound} = 3 \times 2^{32-4},$$

$$\text{Upper Bound} = (3 + 1) \times 2^{32-4} - 1.$$

The region extent of scenario e is [0x30000000, 0x3FFFFFFF]. The one-to-one map between a scenario code and the region extent is constant for any simulation. This feature implies that we do not have to record the map between a scenario ID and its specific region extent. Using the recursive region division solution avoids the need for a searching procedure.

The full dimension [MIN_EXTENT, MAX_EXTENT] contains 2^{32} unique extents at most. Thereby 2^{32} concurrent scenarios are allowed when making full use of the dimension. Such a large number is able to meet any practical requirement for classifying scenarios. However the binary division algorithm may incur problems in some extreme situations. Let us look at a special example, in which on active cloning only the leftmost branch in the binary scenario tree splits into multiple new scenarios. Thus the region extent of the leftmost scenario will shrink rapidly in an exponential way, much faster than other scenarios. It can be seen that region allocation is densely concentrated at the left end of the dimension whereas only a few scenarios occupy the remainder of dimension. As this continues, when the depth of the binary tree reaches 32, one scenario’s region extent becomes a point and child scenarios are not able to inherit extents any more. This potential limitation exists even if this is unlikely to occur.

Once the extent is exhausted in the single clone dimension, one possible solution is to redistribute the extents of the full dimension. Undoubtedly the redistribution incurs extra complexity and it damages the natural harmony of the one-to-one map between scenario

identity and its region extent. Alternatively we can specify a multi-dimensional routing space for cloning beforehand. A clone region is created with these dimensions, and the recursive division algorithm is initially applied to the first dimension. When the extent is exhausted in this dimension, the subsequent dimensions are still available for extent allocation using the same recursive division algorithm. This approach is able to keep the coding scheme intact. As long as the number of clone dimensions is set large enough, it should meet any request in practice.

6.3. Point Region Solution

An alternative solution, namely the point region solution, is proposed to distribute the extents of the dimension as evenly as possible in a bottom-up manner. A point region has an extent defined as $[Lower_Bound, Lower_Bound+1)$. As the name implies, the point region is a single element used in associating a region with a scenario. The initial federates are assigned a start point region. Instead of inheriting any region from their predecessor, the new scenarios get different point regions from the dimension. A shared clone has to combine its original region with the new ones associated with additional scenarios.

Thus, scenarios can be coded based on the scenario tree. Figure 6.5 illustrates a coded scenario tree based on the example given in Figure 6.2. In this solution, on an active cloning, no matter how many new scenarios are created from an existing scenario, the scenario tree extends only one level down with all the new scenarios (and the existing one). From left to right, sibling branches of any scenario are labeled with “0” to “ n ”. For the scenarios that remain unsplit, in the scenario tree, the corresponding nodes still extend one level down with one single child to keep consistency in representing current scenarios and the cloning history along the simulation time axis. By simply linking codes starting from the root, the identity of a scenario is obtained. To ease representing and resolving

the scenario identity, we can record the label of each branch in hex format, where each byte holds a sibling's code, thus scenario e 's identity is written as "0004". The clone identity follows the format defined in the previous recursive division solution. Given the same examples as in section 6.2, we can code the descendant clone of original federate "fab1" that joins scenario e as "0004&&fab1". As for another clone with ancestor "fab2", which is shared by scenarios a , b and g , we may code it as "0000:0001:0200&&fab2". This shared clone has a region that is the union of region extents associated with scenarios a , b and g .

Figure 6.5: Coding the scenario tree

To optimize the usage of communication channels provided by DDM in RTI-NG, the single "cloning" dimension is divided into multiple segments evenly according to the *NumPartitionsPerDimension* (*NPPD*) in the "RTI.rid" file [RTING02]. Each segment will contain an identical number of point regions. To ease discussion, here we still assume that $\text{MIN_EXTENT} = 0x00000000$ and $\text{MAX_EXTENT} = 0xFFFFFFFF$. The following matrix describes the distribution of all available point regions with the number of rows equal to *NPPD* and the number of columns equal to $(\text{MAX_EXTENT} - \text{MIN_EXTENT} + 1) / \text{NPPD}$. It is obvious that the points in one row belong to same segment in the dimension.

$$Rgn = \begin{bmatrix} R_{0,0} & R_{0,1} & \dots & R_{0,(2^{32}/NPPD-1)} \\ R_{1,0} & \dots & & \vdots \\ \vdots & & & \vdots \\ R_{(NPPD-1),0} & \dots & & R_{(NPPD-1),(2^{32}/NPPD-1)} \end{bmatrix},$$

$$\text{where } R_{i,j} = (2^{32} / NPPD) \times i + j \quad (6.2)$$

In total, there can be up to 2^{32} available point regions in which the j th point region in the i th row is written as $[R_{i,j}, R_{i,j}+1)$. The initial scenario starts with a region $[R_{0,0}, R_{0,0}+1)$. A new scenario will always be assigned the first unused point region in the next row. Once the regions in same column are fully used, the allocation will start from the beginning of the next column. Figure 6.6 illustrates this region specifying flow as indicated by the arrowed line. The scenario ID maps to its specified region in a one-to-one way. The scenario tree records the scenario ID and the region in the node for that scenario, and can be used for resolving the region of one particular scenario from its identity by searching the tree. The scenario ID is resolved to locate the scenario node in the tree, this search is only performed along a given path instead of making a full search of the scenario tree.

Figure 6.6: Point region allocation flow

The point region solution has an added advantage in that it maximizes the use of available communication channels. For each routing space, RTI-NG will create $NPPD^{(Number\ of\ Dimension)}$ reliable and best-effort channels. The channels are mapped to the

dimension in a grid-like fashion. As we only define one dimension, RTI provides exactly *NPPD* pairs of channels for the “cloning” space in our solution. The point regions in the same row (see the matrix in equation 6.2) occupy a common data channel. The point region ensures that the interactions or updates with which it is associated use a unique channel when possible and avoids the overhead of sending data through multiple channels unnecessarily. However this solution does not support a direct conversion between the scenario ID and region, the scenario manager module needs to record this map relationship. Furthermore shared clones need to modify their regions on the latest active cloning. This incurs extra effort compared with the recursive region division solution.

6.4. Summary

Table 6.1: Comparison between the recursive region division and the point region solutions

Features	Recursive region division	Point region
Characteristic region	An extent	A point
Region Manipulation for cloning	Top-down division, a clone inherits region from its parent	Bottom-up distribution, each clone's region is independent
Number of scenarios that can be handled	In the worst case, 33 times the number of “cloning” dimensions	2^{32}
Support of clone sharing	Clone sharing is facilitated automatically	Extra region manipulation is required for shared clones
Map scenario ID to region	Natural one-to-one map	Specific map is needed
Use of communication channels	Not optimized	Optimized

To address the complexity of the overall cloning-enabled distributed simulation due to increasing scenario spawning, we investigate an efficient and precise scheme to identify and represent scenarios. This chapter describes and compares two alternative scenario management algorithms, namely the recursive region division solution and the point region solution. The two solutions are designed to code scenarios as well as to manipulate region extents in the “cloning” dimension. The former solution initializes each original federate with a region occupying the full dimension of the routing space. During the cloning procedure, the new clones inherit split sub-regions from their parent. The latter

solution specifies a point region for each original federate and a new clone is given an additional point region on birth. Shared clones may have changing region combinations when cloning occurs in the scenarios in which they operate.

Table 6.1 gives a concise comparison between the recursive region division solution and the point region solution. The recursive region division solution has some advantages in that: (1) it avoids the extra operation of manipulating regions for shared clones which is required by the point region solution and (2) it provides a natural one-to-one mapping between scenario identity and region extent. The coding mechanism harmonizes with the region specification perfectly. However, the recursive region division solution has a limitation in dealing with some extreme situations. The point region solution has advantages over the former one in that: (1) it can meet the region allocation requirements even in extreme situations and (2) it optimizes the use of data channels. The latter solution is therefore chosen and developed in designing our cloning technology due to these superior features.

Chapter 7: Algorithms for Distributed Simulation Cloning

The cloning management mechanism is designed to ensure correct federate replication, and its tasks include creating clones, manipulating states and coordinating the federation. This chapter first introduces the overall cloning infrastructure, and details the algorithms for distributed simulation cloning, including active cloning and passive cloning. Subsequently, this chapter discusses incremental cloning and the algorithms for managing shared clones.

7.1. Overview of Simulation Cloning Infrastructure

An RTI++ infrastructure enabling simulation cloning is built as the middleware¹ between the simulation model and the real RTI. The infrastructure contains several major modules which perform the necessary functionalities related to simulation cloning while presenting a standard RTI interface to the simulation model (as shown in Figure 7.1).

Prior to using simulation cloning, the user can specify the conditions and/or rules according to which the cloning should be triggered and the different candidate actions to be taken.

The Control Module monitors the states in which the user is interested and evaluates the conditions for cloning the federate at a decision point. The Cloning Manager module creates new clones for the request issued by the Control Module, and it initiates the creation and update of the scenarios. The Scenario Manager module creates and stores the scenario tree, from which the identity and corresponding DDM region of each scenario can be fetched. The Scenario Manager keeps the history of the overall cloning procedure.

¹ The physical federate can also be regarded as a component of the whole middleware between the simulation model and the real RTI. In this discussion, we are only concerned with the RTI++ library that directly interacts with the simulation model in the virtual federate.

The Region Manager module creates DDM regions and manages the regions. The Region Manager services the Scenario Manager by answering enquiries and providing the clone specific region. The Region Manager also deals with any request for modifying a region. The Scenario Manager presents the region to the middleware for partitioning event transmission. The RTI++ Object Management services invoked by a federate are executed via the corresponding DDM methods by associating the region obtained from the Scenario Manager. Eventually it is the physical federate who calls the real RTI services and conveys the callbacks to the Callback Processor in the RTI++ middleware (see chapter 5).

Figure 7.1: RTI++ and internal modules

In the context of the middleware approach, the Cloning Manager provides services directly to the Control Module that handles the decision point and cloning trigger. It also interacts with the Scenario Manager (see chapter 6) to update/fetch scenario information, such as region and scenario ID etc. Once the condition for cloning is met, the cloning trigger invokes the Cloning Manager and the latter performs cloning of the federate

accordingly. External stable storage is used to store RTI States of the federate. Figure 7.2 illustrates the internal components inside the Cloning Manager, which is designed to:

- Log and update the system state at the RTI level to stable storage
- Replicate the virtual federate process at runtime according to the parameters given by the cloning trigger
- Coordinate with other federates for the cloning operation
- Recover system states to initialize the physical federates of all clones
- Link the cloned virtual federates to the corresponding physical federates and resume simulation execution

Figure 7.2: Cloning Manager module

The Cloning Executor works as the key component of the Cloning Manager module. It answers the cloning request issued by the Control Module. Upon cloning it interacts with the Scenario Manager to update and retrieve scenario and characteristic region information. The Cloning Executor makes replicas of the simulation model. In addition, it

initiates the same number of new physical federate instances. The saved RTI States are loaded from the external stable storage to initialize the physical federates. Once the state of each clone is initialized to the snapshot of the parent federate at the decision point, the Cloning Executor resumes the whole simulation after applying the action defined for the decision point to each clone. The Cloning Executor also handles the in-transit events on cloning using the RTI federation save services, which forces all in-transit events to be delivered to all receivers.

The RTI States Manipulator saves RTI states and replicates them when needed. The RTI states include federation information, publication/subscription information, object registration information, time granted, etc. A new clone needs to register object instances that represent the same entities known to the simulation model in the parent federate. However, the RTI assigns different handles (Kuhl 1999) to the “cloned” objects, which are unknown to the simulation model. The RTI States Manipulator deals with this problem using an entity mapping approach. The approach maintains the relationships amongst original entities and “cloned” entities at the RTI level. All incoming events will be mapped (processed) by the Callback Processor (see chapter 5) prior to conveying them to the simulation model.

During cloning of a federate, the Federation Coordinator should synchronize other federates within the whole simulation execution including the sibling clones. After cloning is initiated, execution of the simulation model of a federate (clone) will be paused, while the middleware keeps operating in coordination with the cloning procedure. The coordinator is designed to ensure that prior to continuing the simulation execution, every clone has finished initialization and all federates have updated their states for the current cloning operation. Only when these conditions are met, can the simulation be reactivated and resumed.

7.2. Active Simulation Cloning

Figure 7.3: Active simulation cloning

When a federate reaches a decision point, the Control Module evaluates the states in which the user is interested. Once the cloning conditions are met, the Cloning Manager

will make clones immediately to explore alternative executions as predefined in the cloning triggers. Then an active simulation cloning occurs. The partner federates may perform passive cloning or operate as shared clones.

From the perspective of the federate (parent federate) making clones, the process of an active simulation cloning can be described as follows (see Figure 7.3). In this process, the RTI services used by the middleware are directly executed by the physical federate, remaining transparent to the simulation model.

- **Initiating cloning.** The Cloning Executor calls *cloningManager::createClone* (*number of clones*) to initiate the active cloning. The federate being replicated enters cloning mode, in which control is with the RTI++ middleware.
- **Synchronizing federation.** The Federation Coordinator notifies and synchronizes the remaining federates by requesting a federation save. The federation save label is coded to contain the following information: (1) whether the current simulation cloning is an active one or passive one, (2) the number of new clones to be created, (3) the handle of the federate making clones, and (4) the scenario within which cloning occurs (cloning scenario). The remaining federates retrieve this label via their callbacks. When they have identified this synchronization is for the purpose of cloning, their Callback Processors extract the coded information for reference. When the whole federation has completed synchronization, all federates enter cloning mode.
- **Handling in-transit events.** The current design utilizes the federation save services to ensure all in-transit events reach their destinations prior to cloning. The Callback Processor will present those events to the virtual federate when cloning is accomplished. Those events with timestamp greater than the current federate time have to be passed to the simulation model with the advancing of simulation time.

- Updating scenario tree.** The Scenario Manager of every federate updates its scenario tree (see chapter 6) based on the cloning scenario ID and clone number. The scenario updating algorithm ensures all federates maintain identical scenario trees. Those federates remaining intact check whether they operate in the existing scenario as a partner federate or not. A partner federate will work as a shared clone in the current and new scenarios; the region manager needs to merge the new scenarios' point region extents into its existing region and notify the RTI about this region update. The updated region can be written as $region(Scen[1]) \cup \dots \cup region(Scen[n])$, in which $region(Scen[i])$ represents the characteristic point region associated with the i th scenario in which the shared clone will execute. Non-partner federates directly block until the end of the current cloning.
- Making clones.** The Cloning Executor makes the specified number of replicas of the simulation model and initiates an individual physical federate for each replica. Within each replica, the Cloning Executor causes the virtual federate to link to the corresponding physical federate, which joins the existing federation. Thereafter a clone of the parent federate comes into being, which leads to the construction of a new scenario.
- Replicating system states.** A clone's physical federate needs to be initialized with the stored system states from the parent federate. The system states to be replicated include: (1) object classes/interactions that have been published/subscribed, (2) registered object instances and (3) federate time. The state replication of the new clones will be detailed later. The parent federate and partner federates block for the new clones to complete state replication. Each clone reports to the parent federate when fully initialized with its inherited system states. All clones' identities will be collected and forwarded to the partner federates by the parent federate.

- **Leaving cloning mode.** After all clones announce that they are ready, the Federation Coordinator of the parent federate initiates another federation synchronization to declare the termination of the cloning process. The synchronization label codes the relationship between each clone's federate handle and the scenario in which it will operate. The partner federates' Callback Processors can decode the mapping relationships and use them to distinguish the source of incoming events in the future. As soon as synchronization is achieved, the parent federate, its clones and the remaining federates obtain control from the middleware.

After leaving cloning mode, the paused simulation execution resumes with the new scenarios starting operation. The new clones and the parent federate each execute in a single scenario while the partner federates of the parent initially become shared clones that execute in the original scenario and the new ones. All federates work in normal mode until the next occurrence of simulation cloning.

Figure 7.4 depicts the state replication procedure of a new clone. This procedure involves the parent federate, the new clones and the partner federates, but does not affect the remaining federates. Prior to creating clones, the parent federate sends a "system" interaction (namely "*cloningNotification*", defined for cloning) to the partner federates. This interaction encapsulates the published object/interaction classes and the list of registered objects of the parent federate. The receiver's Callback Processor can simply take the intersection of this remote object list and the original object instances discovered previously, and then it can predict what image object instances (see section 3.3.1) it should discover from each new clone. Hence, information contained in this interaction provides criteria for the partner federate to detect the state replication status of each clone. Furthermore the receiver's Callback Processor will refer to this remote publication and registration information in processing events from the parent federate's sibling clones.

A new clone's state replication can be broken down into three phases. When a clone is created, first it enables time constrained and/or time regulating depending on the parent federate's settings. The federate time and lookahead recorded on cloning should be specified accordingly.

Figure 7.4: Replicating states for new clones

Figure 7.5: Coordinating federates in state replication

Secondly, the clone’s Cloning Manager associates a scenario-specific point region to the clone, which is obtained from the scenario tree according to the scenario in which the

clone will execute. The States Manipulator instructs the physical federate to subscribe/publish the same object/interaction classes as the parent federate has subscribed/published, but with the newly retrieved region. The partner federates (shared clones) merge the designated point regions of the new scenarios into their current region.

Thirdly, the **new clone** needs to register image objects to the objects owned by the parent federate. The physical federate invokes the RTI method *RTI::registerObjectClassInstanceWithRegion(objectClass, Name, ...)* (Kuhl 1999) and conveys the returned object handle to the middleware, and the latter maps the original objects to the image ones. The specified “name” of an image object follows the format “<federate handle>&&<original object instance handle>”. The middleware of other **partner federates** which are subscribers will discover these image objects (their regions overlap with the clone’s region) and can map them to the original ones from their names. The mapping relationship (recorded in Mapping Tables) will be used by the Callback Processors to process events from different scenarios. The Callback Processor blocks until all image objects have been discovered, and hides these object discoveries from the simulation model to keep correct HLA semantics. Otherwise, the simulation model will “rediscover” object instances it has already discovered before cloning. On the other hand, the **new clones** may also detect objects registered by the partner federates before cloning (due to the overlapped region). As the cloned simulation model has “already” discovered those objects, these object discoveries should also be hidden to avoid repeated discovery. The **parent federate** waits for new clones to be created and initialized. As soon as each clone completes initialization, it reports to the parent federate with a “*cloningReady*” message containing its federate handle and scenario ID. This design puts constraints (see Figure 7.5) on the parent federate and partner federates before leaving cloning mode. Only when all clones finish replicating states may the whole federation resume normal execution. Thus the state consistency of the overall simulation can be achieved.

Let n denote the number of new scenarios to be explored at a decision point and r the number of RTI object instances to be re-registered. The computational complexity of the cloning process for an individual clone is $O(r)$. As the new sibling clones execute on the same processor as the parent federate, the overall computational complexity of the active cloning algorithm is $O(n \times r)$. This does not include the cost of communication and synchronization with other federates which will depend on the underlying RTI implementation. The overall performance of the cloning mechanism is analysed experimentally in chapter 8.

7.3. Passive Simulation Cloning

The partner federates of the federate making active cloning may remain intact and become shared clones in related scenarios, given that their system states are not affected by the cloning immediately. Incoming events of a shared clone are checked by the Callback Processor. The result of checking an event decides whether the shared clone remains shared or requires passive cloning. The event checking algorithm determines which events and how these events should be conveyed to the simulation model as well as when the passive cloning should be triggered.

As mentioned previously, a shared clone performs passive cloning to handle different events from existing scenarios rather than to explore new scenarios on its own initiative. The passive cloning does not cause creation of new scenarios, and this is different to active cloning. A passive cloning procedure is illustrated in Figure 7.6 from the shared clone's point of view, and can be described as follows:

- **Initiating passive cloning.** The Cloning Executor calls *cloningManager::splitClone* to initiate the passive cloning, and the shared clone enters cloning mode. The in-transit events are handled in the same way as for active cloning.

Figure 7.6: Passive simulation cloning

- **Synchronizing federation.** The Federation Coordinator notifies and synchronizes with the remaining federates by requesting a federation save. These federates retrieve

the federation save tag and identify the current passive cloning from the extracted information. When the whole federation has completed synchronization, all federates enter cloning mode.

- **Creating clones and updating region.** The Cloning Executor creates new clones, and the number of clones is set depending on the number of scenarios in which the parent federate operates beforehand. The original region of the parent federate (denoted as $region(Scen[1]) \cup \dots \cup region(Scen[n])$) should be split into n individual point regions. The Region Manager of the parent federate and each new clone associates an exclusive point region to the clone according to the individual scenario in which it may participate. The parent federate becomes a normal clone and so do its new clones, while the remaining federates continue to operate as normal or shared clones. Their existing scenarios are kept intact without generating new ones.
- **Replicating system states** (see Figure 7.4). The mapping tables of partner federates will be updated by their Callback Processors according to the re-registered image object instances.
- **Leaving cloning and conveying buffered callbacks.** After each clone finishes initialization, another federation synchronization will be initiated to denote the ending of passive cloning. The Callback Processor of the parent federate or each new clone needs to convey the buffered events (received in pending-passive-cloning mode, see section 7.5.2) to the Federate Ambassador. The events are inherited from the parent federate (for new clones). Only those events with timestamp not greater than the current federate time have to be conveyed. Subsequently, each Callback Processor returns control to the simulation model and the latter obtains control again. Thereafter the whole federation resumes normal execution.

Let n denote the number of independent scenarios from which a shared federate (clone) has received different events and r the number of RTI object instances to be re-registered. The computational complexity of the cloning process for an individual clone is $O(r)$. As the new sibling clones execute on the same processor as the parent federate, the overall computational complexity of the passive cloning algorithm is $O(n \times r)$. Again, this does not include the cost of communication and synchronization with other federates.

7.4. Mapping Entities

A federate simulation model produces or consumes information in its object model via RTI services using handles assigned by the RTI. For example, when an object instance is registered, a federation unique handle is returned to identify that object instance. This handle is used to represent an entity known to the model and other federates who have discovered this object instance.

A clone inherits identical states from the original federate, including the RTI entities known to the simulation model. In order to keep the state consistent and federate code transparent, our cloning approach needs to ensure that the clones of a federate use the same reference to the original entities at the RTI level as before cloning. The approach should correctly manage the events associated with these entities within the overall federation, for example a shared clone may receive updates of different object instances even though they refer to the same object in the simulation model (see Figure 7.7).

The consistency can be achieved using a mapping approach in the middleware. The middleware maps the original handles with the image object handles to ensure user transparency and consistency. For one original object instance referred to by the simulation models of all clones, there can be different image object instances accessed by the physical federates. The middleware keeps transparency of object instances in the object model by translating the handle as appropriate. The same principle is applied in

processing other entities at the RTI level. An example of processing object instances is given in Figure 7.7. The physical federate is omitted in the figure to ease discussion.

Figure 7.7: Mapping RTI entities

The overall simulation starts with a federation formed by a pair of federates (*Fed A[0]* and *B[0]*). Assume that *Fed A[0]* publishes object class “machine” and *Fed B[0]* subscribes to it. As shown in Figure 7.7(A), when *Fed A[0]* registers an object instance of

“machine” with object handle 6533 via its RTI++, its simulation model recognizes that a local machine object “6533” is created. *Fed B[0]* detects the object via its RTI++ and its simulation model also knows about the creation of this remote machine.

As illustrated in Figure 7.7(B), *Fed A[1]* is created as a clone of *Fed A[0]* at some point, and *Fed B[0]* is shared by *Fed A[0]* and *A[1]*. *Fed A[1]* re-registers an image object of class “machine” with the original object instance 6533 during state replication and gets a unique handle 8872 from the RTI. However the simulation model of *Fed A[1]* still refers to the original object “6533”, the original machine object. The object 8872 is an image object and is mapped to the original object 6533 by the RTI++. Similarly, *Fed B[0]*’s RTI++ discovers the new object and records its mapping to the original one. The new image object handle will not be passed to *Fed B[0]*’s simulation model, as *Fed B[0]* has already discovered the original object instance.

As simulation execution proceeds, *Fed A[1]* needs to change the state of machine “6533” (Figure 7.7(C)). Its RTI++ converts the update call of “6533” to the image object 8872 and passes it to the RTI services in the underlying layer. *Fed A[0]* invokes an update of image object 6533 directly on its simulation model’s behalf. At the *Fed B[0]*’s side, these updates are automatically translated into the updates of machine “6533” for the simulation model.

7.5. Incremental Distributed Simulation Cloning

When a federate makes clones on its own initiative and creates new scenarios, other federates in the original scenario have to interact with each of these clones properly in the new scenarios. One direct solution is to clone all other federates immediately, thus each independent set of clones forms a new standalone scenario. In this case, a full set of independent federates are exploited to examine each scenario after cloning. However, when performing distributed simulation cloning, it is desirable to replicate only those

federates whose states will alter at a decision point. The remaining federates may keep intact and become shared between the original scenario and the new ones; only when absolutely necessary will those shared federates be cloned. Hence such an incremental simulation cloning mechanism is expected to further share computation amongst scenarios.

7.5.1. Illustrating Incremental Distributed Simulation Cloning

Figure 7.8 illustrates a simple supply chain simulation comprising three federates, namely *simAgent* (SA), *simFactory* (SF) and *simTransportation* (ST). A cloning trigger is predefined for federate *simFactory*, which contains a cloning condition “*OrderSize > MAX?*” and four candidate policies. The simulation models the supply chain operation of one-year duration. The *simFactory* reports the cost incurred in each order and in the whole year at the end of the simulation. Chapter 8 will give a detailed description of this example.

Figure 7.8: A distributed simulation example

Figure 7.9 depicts the simulation execution using incremental simulation cloning to examine three of the candidate policies. Each scenario is marked as *Scen[i]* ($i = 0, 1, 2$), in which *Scen[0]* denotes the initial scenario. The incremental simulation cloning occurs along the time axis as follows:

At time 0, the simulation is initialized with a single scenario *Scen[0]*. When simulation progresses to time T_1 , *SF[0]* performs active cloning due to an order with extra large

volume, which results in the creation of clones $SF[1]$ & $SF[2]$, and new scenarios $Scen[1]$ & $Scen[2]$. The remaining federates do not need to be cloned immediately, and they only need to expand their associated region to enable them to continue interacting with $SF[1]$ & $SF[2]$. Thus $SA[0]$ and $ST[0]$ become shared clones in both scenarios. The event flow from $SF[i]$ ($i = 0,1,2$) to $ST[0]$ is named as $ev_F[i]$ ($i = 0,1,2$). $ST[0]$ keeps intact as long as $ev_F[i]$ ($i = 0,1,2$) remain identical.

Figure 7.9: Executing simulation with incremental cloning

At simulation time T_2 , $ev_F[0]$ deviates from $ev_F[1]$ and $ev_F[2]$, this triggers a passive cloning of $ST[0]$ and results in the creation of clones $ST[1]$ & $ST[2]$. This passive cloning does not trigger any change of existing scenarios. $SA[0]$ persists as a shared clone after that.

Using the incremental cloning algorithm, clones are created as required according to the changing external conditions. For the whole simulation session, we always have:

Total no. of federates $\leq \sum$ No. of federates executing in each scenario. For example, as shown in Figure 7.9, from simulation time T_0 to T_1 there exists only 5 federates simulating 3 scenarios whereas there has to be 9 federates examining the same scenarios in the context of traditional distributed simulations or using the entire cloning approach. Both incremental and entire cloning avoid repeating the computation of the original scenario before cloning. However, the incremental cloning approach also enables the sharing of computation amongst independent co-existing scenarios after cloning.

7.5.2. Managing Shared Clones

Figure 7.10: Internal design of the Callback Processor

A shared clone is capable of operating in multiple scenarios as long as it keeps receiving identical events from all scenarios in which it participates. The shared clone persists in this mode until the condition of triggering passive cloning is met. Thus during this time, the computation of this clone can be shared by different scenarios. The incremental cloning mechanism aims to make full use of the interdependencies amongst related scenarios, which is supported by a sensitive event checking algorithm.

Sensitive events are those received from sibling clones in different scenarios, and sensitive events are comparable if they are updates of the same original object instance or interactions belonging to the same class (see chapter 3 for details). Sensitive events are checked by the Callback Processor which is one part of the RTI++ middleware built upon the decoupled federate architecture (see chapter 5). Figure 7.10 illustrates the primary elements inside the Callback Processor designed for checking events. The Sensitive Event Checker checks events and invokes the Control Module to trigger passive cloning when necessary. Mapping Tables maintain the relationships amongst scenarios and federates and the object instances (original/image) registered by related sibling clones. These tables are established and updated during the state replicating procedure on cloning. The checker can identify the source clone and scenario of each event via the tables, thus it can verify which events are comparable. A queue *Pending_Sensitive_Events_Queue* (*PSEQ*) stores the **target** sensitive events with which other incoming sensitive events must be compared. The queue can be either empty or contain events with the same timestamp at any point in the simulation, this timestamp is referred to as the characteristic timestamp of *PSEQ*. A set of TSO event queues, *TSO_Queue_Scen[i]* abbreviated as *TQS[i]* ($i = 1, 2, \dots, n$), are established to buffer the events from each scenario in the corresponding queue. Events in those queues can be presented to the simulation model as appropriate.

The event checking algorithm determines which events and how these events should be conveyed to the simulation model. The event checking decides whether or not a passive cloning is required and at which point the cloning should be triggered. A shared clone is said in to be in **pending-passive-cloning** mode during the interval from deciding that a passive cloning is required to carrying out the cloning. Event checking is performed in the Callback Processor when control of a federate process is still with the RTI. Thus cloning will only be carried out when the RTI returns control to avoid potential problems incurred

by replicating a federate while the RTI invokes callbacks. Figure 7.11 illustrates the algorithm of checking sensitive events by the Callback Processor, which are as follows:

Figure 7.11: Checking sensitive events

- **Testing sensitive event.** A received update/interaction is tested according to definition 1 and 2 in section 3.3.2. In the case where the event is a sensitive one, the checking continues. The processing of non-sensitive events will be covered later.

- **Identifying event source.** Mapping Tables are referenced to locate the source of this event. Hence the event checker can enqueue this event into the corresponding *TQS* queue.
- **Checking pending sensitive event queue.** If *PSEQ* is empty, the event checker pushes this event into *PSEQ* and sets its characteristic timestamp equal to the event's, after which current processing ends. When *PSEQ* is not empty, the event checker compares its characteristic timestamp with the event's. In the case where they are not equal (event's timestamp > *PSEQ*'s), the shared clone will enter pending-passive-cloning mode, otherwise the processing continues.
- **Locating target comparable event.** The event checker searches *PSEQ* to locate the comparable event to the event being processed. If *PSEQ* does not contain any comparable event, the event will be pushed into *PSEQ* and the processing ends. Otherwise, the event checker checks if the received event and the target event in *PSEQ* are identical. If they are not identical, the shared clone will also enter pending-passive-cloning mode, otherwise the processing continues.
- **Checking the progress status of processing.** The event checker examines whether the shared clone has received identical comparable events from all other scenarios. If so, the event checker presents the target event from *PSEQ* to the Federate Ambassador and deletes the comparable events inside the *TQS* queue set. If not, then the event checker waits for the next event.

The last step ensures the Federate Ambassador receives/reflects only one single event for one full set of identical events obtained from all related scenarios. This design hides the complexity of checking events from multiple scenarios. As a result, shared clones operate in multiple scenarios as if they only interact with one single scenario independently.

In the case where a non-sensitive event is received, the *PSEQ*'s characteristic timestamp also needs to be compared when *PSEQ* is not empty. If the event's timestamp is greater than the characteristic timestamp, the shared clone requires passive cloning. This is because the shared clone will no longer receive identical events from any of the related scenarios to the target events in *PSEQ*. In the case that their timestamps are identical or *PSEQ* is empty, the event checker delivers this event to the Federate Ambassador directly.

If *PSEQ* contains target sensitive events, the decision of triggering a passive cloning depends on both the incoming events and the next granted time. Once the Callback Processor gets a granted time greater than (or equal to) *PSEQ*'s timestamp², the shared clone enters pending-passive-cloning mode.

Figure 7.12: Processing events in Pending-Passive-Cloning

When a shared clone is in pending-passive-cloning mode, the Callback Processor buffers the incoming events as illustrated in Figure 7.12. Sensitive events should be

² If the granted time is equal to *PSEQ*'s timestamp, setting pending-passive-cloning or not depends on whether the shared clone requests the last time advance by calling *timeAdvanceRequest* (TAR) / *nextEventRequest* (NER) or *timeAdvanceRequestAvailable* (TARA) / *nextEventRequestAvailable* (NERA).

enqueued to the corresponding *TQS* queue, whereas non-sensitive events should be inserted into all *TQS* queues unselectively. Thus, when new clones are created later on, each clone can straightforwardly inherit the events belonging to the scenario in which it operates. The Callback Processor logs the *timeAdvanceGranted* callback and the granted time. All callbacks are retained and not delivered to the simulation model until the pending cloning has been completed. Such a design aims to keep the semantics of the HLA specification and minimize the complexity of dealing with potential callbacks during pending-passive-cloning.

7.6. Summary

In this chapter, we have introduced the infrastructure for HLA-based distributed simulation cloning. Our design enables distributed simulation cloning using a decoupled federate architecture. The Cloning Manager module is developed to ensure correct cloning of distributed simulations when preset conditions are met at decision points. During cloning, coordination and synchronization are required to maintain the state consistency. We have presented the mechanism and algorithms for managing distributed simulation cloning, including active cloning and passive cloning.

The incremental cloning mechanism initiates cloning only when strictly necessary. Clone sharing amongst multiple scenarios is supported by a sensitive event checking algorithm. The algorithm facilitates accurate sharing of clones and delays the passive cloning as long as possible. An entity mapping approach is designed to identify events, which maps the new HLA entities created for the clones in state replication to the existing ones known to the simulation models. The proposed incremental cloning mechanism supports correct HLA semantics and user transparency.

Chapter 8: Experiments and Results

8.1. An Application Example

Figure 8.1: A simple distributed supply chain simulation

This chapter presents a simulation example (see Figure 8.1) to verify the correctness and investigate the performance of the proposed distributed simulation cloning technology. The example studies a simple supply chain that comprises an agent company, a factory and a transportation company. The agent keeps issuing orders to the factory, and the latter processes these orders and plans production accordingly. The transportation company is responsible for delivering products of the factory and reporting the delivery status. The factory has a limited manufacturing ability, which means it needs to adjust its daily operation policy in the case of fulfilling an extra large order. One of the following candidate policies can be chosen:

- a) Keeping normal operation policy (NORMAL policy), thus the factory may get penalized since the order is not fulfilled in time.
- b) Encouraging overtime work to ensure on time delivery (OVERTIME policy), the factory has to pay for the extra manpower.

- c) Sharing a partial order with other competing factories (SUBCONTRACT policy), this also incurs profit loss and negative impact on future business development.
- d) Expanding manufacturing ability by purchasing assembly lines and recruiting more workers (EXPAND policy), thus extra cost will be incurred.

The analyst focuses on optimizing the extra cost/profit loss of the factory operation in processing large orders. Four different scenarios need to be constructed to examine the candidate policies.

8.2. Configuration of Experiments

The three nodes in the supply chain can be modeled as three federates as shown in Figure 8.1, namely *simAgent*, *simFactory* and *simTransportation*. These federates form a simple distributed supply chain simulation. Two object classes “*Order*”, “*Products*” and one interaction class “*deliveryReport*” are defined in the FOM [IEEE1516] to represent the types of events exchanged amongst the federates. Table 8.1 gives the classes published and/or subscribed by the federates. The *simFactory* reports the extra cost incurred for each order and for the whole year (from simulation time 0 to 361) at the end of the simulation. The experiments use four computers in total (three workstations and one server), which are interlinked via a 100Mbps-based backbone. Each federate (together with its clones if any) occupies one individual computer, with the RTIEXEC and FEDEX processes running on another computer.

The experiment architecture and platform specification are listed in Table 8.2. Using the same codes for the simulation models, the federates are built into three different versions by linking to: (1) the DMSO RTI library directly (TRADITIONAL), (2) an RTI++ middleware library supporting entire cloning (CLONING_ENTIRE) and (3) an RTI++ middleware library supporting incremental cloning (CLONING_INCREMENTAL). We can also configure the federates of the 2nd (or the 3rd) version to execute each of the

policies without using cloning (CLONING_DISABLED) to examine the correctness of using the decoupled architecture to build federates. The simulation execution using incremental cloning is discussed in section 7.5.1 and illustrated in Figure 7.9.

Table 8.1: Declaration information of the federates

Federate	Object Classes and Attributes		Interaction Classes and Parameters
	Order	Products	deliveryReport
	<i>Index, Size</i>	<i>Amount, Index, Date</i>	<i>Index, Status</i>
simAgent	Publish	NIL	NIL
simFactory	Subscribe	Publish	Subscribe
simTransportation	NIL	Subscribe	Publish

Table 8.2: Configuration of experiment test bed

Specification	Computers				
	Workstation 1	Workstation 2	Server 1	Workstation 3	Workstation 4~13
Operating System	Sun Solaris OS 5.8	Sun Solaris OS 5.8	Sun Solaris OS 5.8	Win2000 Professional	Sun Solaris OS 5.9
CPU	Sparcv9 CPU, at 900 MHz	Sparcv9 CPU, at 900 MHz	Sparcv9 CPU * 6, at 248 MHz	Intel 1700 MHz Pentium IV	Sparc II CPU, at 400 MHz
RAM	1024M	1024M	2048M	256M	512M
Compiler	GCC 2.95.3	GCC 2.95.3	GCC 2.95.3	MS VC++ 6.0	GCC 2.95.3
Underlying RTI	DMSO NG 1.3 V6	DMSO NG 1.3 V6	DMSO NG 1.3 V6	DMSO NG 1.3 V6	DMSO NG 1.3 V6
Processes running on	simAgent	simTransportation	simFactory	RTIEXEC & FEDEX	SimAgent or simTransportation

8.3. Correctness of Distributed Simulation Cloning

The ten sets of experiments in total are indicated in Table 8.3. To verify the correctness of the cloning mechanism, we specify federate *simAgent* to generate the same set of orders in different runs. We first execute the TRADITIONAL federates, in which the collected results are used as a reference in subsequent experiments. Secondly, we repeat these experiments using CLONING_DISABLED federates. The last two experiments adopt CLONING_ENTIRE and CLONING_INCREMENTAL federates to explore multiple policies in each of the sessions. The outputs obtained using TRADITIONAL federates are summarized as follows:

- *simAgent* issues 240 orders, in which the first extra large order and the last order carries timestamp 83 and 362.5 respectively (note the last order is not received as it is after the simulation end time).
- *simFactory* receives 239 orders and makes products according to the policy adopted. Different costs are reported in experiments T_Norm , T_Over , T_Sub and T_Exp respectively.
- *simTransportation* receives all product updates issued earlier than the end time and sends *deliveryReport* interactions with respect to these updates.

Table 8.3: Experiments for verifying the correctness of cloning technology

Type of Federates	Candidate Policies			
	NORMAL	OVERTIME	SUBCONTR ACT	EXPAND
TRADITIONAL	T_Norm	T_Over	T_Sub	T_Exp
CLONING_ DISABLED	D_Norm	D_Over	D_Sub	D_Exp
CLONING_ ENTIRE	$Ec_Multiple$			
CLONING_ INCREMENTAL	$lc_Multiple$			

In order to evaluate the correctness of the cloning mechanisms, we examine the orders issued and received, products produced and delivered as well as the costs. We have the following observations from the experiments using federates linked with the other libraries:

- Outputs in the experiments with CLONING_DISABLED federates (D_Norm , D_Over , D_Sub and D_Exp) match exactly those using TRADITIONAL federates.
- In experiments $Ec_Multiple$, *simFactory* makes one, two or three new clones at simulation time 83 to execute two, three or four policies concurrently. Immediately after this, passive cloning is forced in both *simAgent* and *simTransport*. For example, in the experiment to examine 3 policies, after cloning there are 9 federates in total to

explore 3 scenarios concurrently. Each clone of *simFactory* reports exactly the same results to those in T_{Norm} , T_{Over} , T_{Sub} and T_{Exp} respectively.

- In experiments *Ic_Multiple*, *simFactory* makes clones in the same way as in the *Ec_Multiple* experiments. The difference is that federate *simAgent* keeps intact all the time and *simTransportation* remains shared until simulation time 101 when it performs passive cloning, required by the event checking algorithm (see section 7.5). In the experiment to examine 3 policies, there are 5 federates from time 83 to 101 while there are 7 federates after time 101 to explore 3 scenarios (see Figure 7.9). Each clone of *simFactory* reports exactly the same results to those in T_{Norm} , T_{Over} and T_{Sub} respectively.

Outputs of the above experiments indicate that the technology provides a correct cloning mechanism for HLA-based distributed simulations. This also proves that the decoupled federate architecture ensures complete fidelity in the way it bridges the simulation model and the Local RTI Component. In other words the simulation cloning technology does not introduce any variation to the simulation results.

8.4. Efficiency of Distributed Simulation Cloning

To investigate the performance of the cloning technology, we carry out a series of experiments to collect the overall execution time using federates linked to different libraries to execute all policies (Table 8.4). For traditional federates, we execute the policies one by one, and for each policy we carry out a number of runs with the extra large order generated randomly at any time (ranging from the start time to the end time) in each run. The execution times of all runs are averaged, and the result is referred to as the average time of executing one single scenario. As for experiments with cloning-enabled federates, we let federate *simAgent* generate different orders. We select three runs

in which the extra large order occurs at time 80, 203 and 320, thus federate *simFactory* may trigger active cloning at different stages in each run. These points represent cloning at the start, middle and end stages respectively. Furthermore we also specify *simFactory* to make a different number of clones to examine alternative policies in different experiments (2, 3 or 4 policies respectively).

Table 8.4: Experiments for studying the efficiency of cloning technology

Type of Federates	Experiments								
Cloning Stage	Start			Middle			End		
Number of Policies	2	3	4	2	3	4	2	3	4
CLONING_ENTIRE	<i>Ec_s_2</i>	<i>Ec_s_3</i>	<i>Ec_s_4</i>	<i>Ec_m_2</i>	<i>Ec_m_3</i>	<i>Ec_m_4</i>	<i>Ec_e_2</i>	<i>Ec_e_3</i>	<i>Ec_e_4</i>
CLONING_INCREMENTAL	<i>lc_s_2</i>	<i>lc_s_3</i>	<i>lc_s_4</i>	<i>lc_m_2</i>	<i>lc_m_3</i>	<i>lc_m_4</i>	<i>lc_e_2</i>	<i>lc_e_3</i>	<i>lc_e_4</i>

The average CPU utilization of a single traditional federate (in workstation 1 or 2) is reported as above 80%. In the case of enabling simulation cloning, the average CPU utilization of each clone is reported as ~44%, ~30% or ~21% respectively when there are two, three or four clones running on a single workstation. Physical federates have an average CPU utilization as low as ~1%. Experimental results are recorded in seconds in Figure 8.2. The average time of executing one single scenario per run is 561 seconds using traditional federates. The percentage of saved execution time using cloning technology is shown in Figure 8.3. This uses the execution times reported by traditional federates running scenarios sequentially as a reference.

Figure 8.2 shows that the cloning enabled federates can significantly reduce execution time compared with traditional federates. The experimental results indicate that the more computation there is in common amongst different scenarios the more execution time can be reduced using simulation cloning. It also shows that the larger the number of scenarios to be examined using cloning the more execution time can be reduced. Results in Figure 8.3 indicate that the incremental cloning approach has an obvious advantage over the

entire cloning approach in terms of execution efficiency under all given configurations. This is because the incremental cloning approach can further save computation by supporting federate sharing amongst scenarios.

Figure 8.2: Execution time for examining multiple scenarios

Figure 8.3: Percentage of saved execution time using entire and incremental cloning

8.5. Scalability of Distributed Simulation Cloning

Another series of experiments are performed to test the scalability of the cloning technology. This consists of six sets of experiments in total, with the *initial* number of federates varying from 3 to 13. Each set of experiments uses a federation which always contains one *simFactory* and one or multiple instances of *simAgent* and *simTransportation*. In the n th set of experiments, the federation initially has $2n+1$

federates as shown in Figure 8.4. Similar to the previous experiments, the execution time for examining the first three policies (see section 8.1) is measured and compared using different types of federates (TRADITIONAL, CLONING_ENTIRE and CLONING_INCREMENTAL). The experiment architecture and platform specification for the scalability study are listed in Table 8.2. Federates *simAgent.1~n* are configured to issue extra large orders at the middle stage of the simulation. The first extra large order received by the *simFactory* acts as a trigger of an active cloning, but after cloning, each particular scenario applies to all subsequent orders, including extra large ones.

Figure 8.4: Initial federation for the n th set of experiments for the scalability test

The average CPU utilization of each federate is the same as in previous experiments. The experimental results are recorded in Figure 8.5. The sum of the execution times using traditional federates increases smoothly when the federation contains an increasing number of federates (starting from 1683 seconds for 3 federates to 2169 seconds for 13 federates), and the same trend can be observed when using cloning enabled federates. Figure 8.5 indicates that cloning technology can reduce the execution time significantly. Figure 8.6 shows that using incremental cloning the percentage of saved execution time is about 47~49%. The entire cloning approach gains about 30~39% time saving. The experimental results show that the proposed cloning technologies scale well with increasing distributed simulation size, and the incremental cloning approach consistently performs better than the entire cloning approach.

Figure 8.5: Execution time for examining three policies with increasing number of federates

Figure 8.6: Percentage of saved execution time with increasing number of federates

8.6. Optimizing the Cloning Procedure

The simulation cloning procedure is defined as the interval from the point where the parent federate starts cloning by calling the *createClone* method (or *splitClone* in the case of passive cloning) to the point where all new clones complete initialization, the remaining federates are ready for resuming normal execution and control is returned to the simulation models (see Figure 7.3 and Figure 7.6). The cloning algorithm requires the

following RTI related operations to be performed in sequence during the cloning procedure (as described in section 7.2):

- Save the federation
- For each new clone, join the existing federation
- Synchronize the federation

Such a simulation cloning procedure takes about 20 seconds in the previous experiments. This overhead is mainly due to above operations which incur costly federation-wide coordination, especially the *joinFederationExecution* call. In DMSO RTI-NG, this operation requires opening TCP sockets to all other federates in the federation, which is expensive [Rycerz05]. Although the time taken by these operations will increase with the number of federates, the exact relationship will depend on the internal design of these RTI services, which is transparent to the user.

It is therefore worthwhile to optimize the cloning procedure. A possible solution is to avoid the *joinFederationExecution* call during the cloning procedure itself. We attempt to solve this problem using a physical federate (**phyfed**) pool approach as shown in Figure 8.7. This approach creates phyfed instances concurrently with the normal simulation execution, which can potentially serve the new clones in the future.

Before cloning occurs, several existing phyfed instances are created, which join the federation and form a phyfed instance pool. In the context of the pool approach, a phyfed instance may operate in two modes: (1) **idle mode**, calling *tick* regularly to maintain connection session with the RTI while checking for invocation from a virtual federate, and (2) **working mode**, servicing a virtual federate as normal. An idle phyfed instance is neither time regulating nor constrained, and has minimum interaction with other federates.

Figure 8.7: Physical federate pool approach

When cloning occurs, a phyfed instance can be fetched from the pool by the virtual federate to provide the required RTI services immediately. Thus, this approach avoids new clones taking part in the *joinFederationExecution* prior to state replication. However, maintaining spare phyfed instances will incur extra overhead in that (1) each instance consumes system resources and (2) the RTI has to monitor the session with more federates.

Figure 8.8: Experiments for studying physical federate pool approach

In order to measure the performance of this approach, a series of experiments are carried out. The configuration and structure of the experiments are similar to the supply chain example and are given in Figure 8.8 (see Table 8.2 for the platform specification). The simulation contains three federates (*Fed[1]~[3]*), which are built into two different versions by linking to (1) the previous RTI++ middleware supporting incremental cloning and (2) an RTI++ middleware supporting the phyfed pool approach. *Fed[2]* is configured

to perform active cloning (spawning 1~3 clones in different runs) when it is granted federate time 100, and this leads to passive cloning of *Fed[3]* afterwards. The average CPU utilization of each federate is the same as in previous experiments.

In these experiments, we compare the initialization time of federates, the time for the cloning procedure and the overall execution time using the two different versions of middleware. The initialization time denotes the interval from when a federate is started to the point immediately after it completes the following operations: join the federation, enable time regulating/constrained, publish/subscribe object/interaction classes and register object instances. The phyfed pool is created at the time when the original federate invokes *createFederationExecution*, and a different number of idle phyfed instances are created according to the number of potential policies. Table 8.5 lists the results of the experiments.

Table 8.5: Comparison between normal and pool approach

Number of Clones	Idle Phyfed Instances	Initialization Time (seconds)		Cloning Time (seconds)		Execution Time (seconds)	
		Normal	Pool Approach	Normal	Pool Approach	Normal	Pool Approach
1	1	19~27	21~27	~20	~3	~93	~83
2	2			~23	~3	~117	~107
3	3			~23	3~4	~140	~131

The initialization time in the pool approach is close to the normal one, which indicates the overhead of phyfed instances joining the federation is hidden by the other initialization operations. The time used for cloning is dramatically reduced from ~20 seconds to ~3 seconds and most of the time is spent in the federation save operation. Using the pool approach can reduce the overall execution time compared with the normal cloning approach. However, the improvement in cloning time is reduced by the overhead for maintaining idle phyfed instances.

To achieve the best performance using this approach, phyfed instances should be added to the pool close to the point of cloning. In this way, the reduction in cloning time can best outweigh the redundancy of idle phyfeds.

Another concern is to optimize the time consumed in the federation save operation which is only used to ensure in-transit messages will arrive at destinations prior to cloning. An alternative candidate approach for dealing with in-transit messages is introduced in chapter 9.

8.7. Summary

In this chapter, an example of distributed supply chain simulation is used for examining the correctness, performance and scalability of the distributed simulation cloning technology. Experiments are carried out to compare two cloning mechanisms, namely entire cloning and incremental cloning, in terms of execution time using normal federates as a reference.

The simulation outputs demonstrate that the technology provides a correct cloning mechanism for existing distributed simulations. The performance results show that the technology can reduce the time of executing multiple scenarios significantly. The results also show that the proposed cloning technology has a promising scalability. Furthermore, incremental cloning can provide further savings by sharing the computation of alternative scenarios compared with an approach using entire cloning.

Chapter 9: Exploiting the Decoupled Federate Architecture

In chapter 5, we have discussed the decoupled federate architecture, which separates a federate's simulation model from the Local RTI Component (LRC). The architecture enables efficient simulation cloning while facilitating reusability of legacy federate code and user transparency. The decoupled federate architecture can also be used in supporting fault tolerance in HLA-based distributed simulations. Section 9.1 describes a fault-tolerant model, which adopts a “dynamic LRC substitution” approach to deal with failure. Section 9.2 gives a scheme for recovering in-transit events in the case of failure. As future work, section 9.3 discusses the feasibility of using the decoupled federate architecture to achieve a Web/Grid enabled architecture and to provide load balancing.

9.1. Fault Tolerance

9.1.1. Background and Motivation

Simulation federates running at different locations are liable to failure, and the failure of one federate (or a part of a federation) can lead to either incorrect simulation results or even the crash of the overall distributed simulation. With more federates participating in the federation, the risk of such failure increases steadily. Restarting the whole simulation session might be costly. Thus, we need to investigate the fault tolerance issue and whether the decoupled federate architecture can be used for this purpose.

People have developed many technologies for facilitating fault tolerance in distributed applications. Cristian pointed out some principles about fault-tolerance in distributed system architectures [Cristian91], and these are: understanding failure semantics, masking failure and balancing design cost.

The checkpoint and message-logging approach is an alternative to the replication

methods described in chapter 2. As proposed in [Johnson89], a process records each message received in a message log while the state of each process is occasionally saved as a checkpoint. Message logging can be pessimistic assuming the occurrence of a failure at any time or optimistic assuming that a message will always be properly delivered before process failure. A failed process can be restored using some previous checkpoint of the process and the log of messages. This approach assumes the process execution to be deterministic between received input messages.

The current IEEE 1516 HLA specification does not support a fault tolerance mechanism for HLA-compliant simulations. Although fault tolerance support has been proposed in the latest “HLA Evolved” specification and some design patterns for fault-tolerant federations are suggested in [Möller05], the proposal is still under discussion and no standard implementation is available.

The RTI supports federation save and restore services, which can be adopted in a checkpoint or simulation migration approach. However, using such approach means a federate has to be specially modeled. For example, given that a checkpoint method is adopted, a federate should have the additional functionalities to save its simulation model’s states and recover from the last checkpoint. It is desirable to allow users to model their federates freely while enabling robustness. To optimize execution, it is preferable that the fault tolerance approach can handle the failure of the RTI immediately without repeating or disrupting the execution of the simulation model.

In component-based simulations, one or more components can be substituted during simulation with a functionally equivalent set of components. Rao et al proposed Dynamic Component Substitutions to realize fault-tolerance in web-based simulation [Rao00]. Rao’s idea of component substitution can be used in designing a fault-tolerant mechanism without concern over manipulating states for simulation models.

A normal federate usually exists as a single process at runtime, and the simulation

model shares the same memory space with the LRC (see Figure 5.1). In the case where the RTI crashes or faces congestion, the failure of any LRC prevents the simulation execution from proceeding correctly even though the simulation model contains no error at all. Besides the challenge to develop a generic state saving and recovery mechanism, the fault-tolerant model requires an approach to “isolate” the error of the LRC from the simulation model. We can see that the simulation model and the LRC have very different characteristics. Therefore, it suggests that the distinction made between these two modules when designing the mechanisms for cloning a federate can also be exploited to deal with failure.

9.1.2. Fault-tolerant Model

Many factors can contribute to the failure of a federate, for example, network congestion, the platform crash of the RTIEXEC process [DMSO02] etc. In a large-scale distributed simulation, the crash of one federate can lead to failure of the overall simulation. The more federates that participate in the same federation, the higher the probability that simulation failure occurs. Although users may model the federates properly, an RTI failure may still cause simulation collapse. It is also difficult to handle such failures at runtime because most RTI implementations cannot be accessed or modified by the developers. Simply restarting a new federate to substitute the crashed one is not applicable since the consistency of the overall simulation state is lost. Considering the complexity and distribution of the individual simulation models and the number of federates in a large scale distributed simulation, it is costly to restart the overall distributed simulation.

The federation save and restore services provided by the RTI could be used to recover a crashed federation. Using these services, the state of each federate is occasionally saved at a checkpoint. In the case of failure, a new federation can be created to “restore” the

federation with the saved states. This approach requires the simulation model to have the functionality to manipulate the states at the model level, and it repeats the computation from one of the checkpoints onwards. The overhead for executing federation save and recovery can also be significant [Zajac03, Rycerz05].

We propose using the decoupled federate architecture to design a fault-tolerant model, which aims to minimize the cost of providing robustness. The model is designed to tackle unpredictable failure of RTI services regardless of the cause. No implementation can ensure that any program is immune from all faults, and the focus of this study is to develop a robust infrastructure for facilitating distributed simulations rather than to free developers from validating their simulation models. Therefore, the fault-tolerant model does not consider federate crashes due to the incorrect implementation of its simulation model or address deadlock in federation synchronization. It assumes also that the messages sent and received in the network are not corrupted.

As shown in Figure 9.1, the model contains a Management Module and a Failure Detector in the middleware. The Management Module comprises an RTI States Manipulator (see sections 5.4 and 7.1) and a Buffer Manager. The RTI States Manipulator saves RTI states immediately before passing an RTI call to the physical federate to execute it¹. The Buffer Manager saves attribute updates and interactions in a similar manner for dealing with in-transit events (see section 9.2). The Failure Detector monitors the status of the LRC or even the RTIEXEC if necessary. An RTI failure in the current design can be: (1) the time-out of an RTI invocation, (2) a critical RTI exception², (3) any other unknown error from the RTI, or (4) the crash of the physical federate or RTIEXEC. The first three cases can be detected passively via the physical federate while the fourth

¹ For example, when the virtual federate invokes *publishObjectClass*, the RTI States Manipulator intercepts this call and saves the information, after which it will call the physical federate via ExtComm.

² e.g. *RTI::RTIInternalError* or any other exception specified as critical by the user.

requires the failure detector actively checking the status of the physical federate or RTIEXEC. Subsequent to confirming an occurrence of an RTI failure, the Management Module will take control and start a failure recovery procedure. Meanwhile, the execution of simulation models will be paused.

The Management Module within the virtual federate will cut off the connection from its physical federate and terminate it. Subsequently the Management Module will initiate a new physical federate instance. Other virtual federates can also perform the same action and form a new workable federation together. After that, the new physical federate recovers the RTI states from the stable storage. The same model of state saving and replication used in cloning federates also applies in fault tolerance (see Figure 5.6). Finally the virtual federate continues execution with the new physical federate. Therefore, physical federates work as plug-and-play components, and they can be replaced at runtime. The fault-tolerant model functions as a firewall to prevent failure of local or remote LRCs from stopping the execution of the simulation model.

Figure 9.1: Fault-tolerant model upon dynamic LRC substitution

The fault-tolerant model provides user transparency, which frees the user from including extra “fault-tolerant codes” in modeling federates, or intervening into the running simulation to deal with RTI failure. The model exploits a “dynamic LRC substitution” method, which allows a simulation to be resumed from exactly the point

where the failure occurs. This means the simulation model's execution is immune from the RTI failure. The model provides robustness to legacy federates and avoids extra effort for facilitating rollback into the simulation models.

9.2. Dealing with In-transit Events

The RTI treats updates of attributes and some other events as transient ones. From the RTI's point of view, a new physical federate (required by failure recovery or simulation cloning) operates as a late joining federate. Thus, the federate recovering from failure (or new clones) may miss some events generated previously, which should be delivered to them. Figure 9.2 illustrates the problem that new LRCs face in dealing with events generated previously. Although the example is discussed based on the timestamp ordered (TSO) events, it is similar for receive order (RO) events.

Figure 9.2: Problem in dealing with in-transit events

As illustrated in Figure 9.2, at simulation time T , Federate $X[0]$ sends a TSO event EvX with timestamp $T + \Delta t_1$ ($\Delta t_1 > Lookahead$) to subscribers (e.g. Federate $Y[0]$). In the case that Federate $Y[0]$ fails at time $T + \Delta t_0$ (Figure 9.2(A)), a new Federate $Y[0]$ resumes with a new physical federate. But the recovered federate will never receive the event EvX as it has already

been lost due to the failure.

It is similar in simulation cloning (Figure 9.2(B)). At simulation time $T + \Delta t_0$, both federates are cloned, and new clones (*Fed X[1]* and *Y[1]*) are created. With the advance of simulation time, the subscribers in the original scenario will receive this TSO event whereas those in the new scenario may not receive this event. This event loss can lead to incorrect simulation results in the new scenario. The RTI federation save forces all in-transit events to be delivered to the receivers (even though they are not necessarily presented to the simulation models, see chapter 7). Thus a simple solution is to issue a federation save on cloning, and this ensures that the federate to be cloned can access all “future” events sent to it via its TSO queue. However, the federation save is an expensive operation (see section 8.6). Moreover, this solution does not apply to failure recovery, as the time the failure occurs cannot be predicted.

From the above discussion, there is a need for an alternative solution to ensure in-transit events are delivered to the receivers when the simulation resumes from failure or during simulation cloning. A general solution to this problem is to resend “image” events with an identical content/timestamp to the corresponding in-transit events generated previously. We propose to build a Buffer Manager (see Figure 9.1) at the sender side to record each outgoing TSO event and index the event according to the sequence in which it is created. The buffer can be flushed to stable storage periodically. This approach has similarity to the commonly-used message logging approach in the sense of recording events, but our approach does not require rollback of the model’s execution [Tanenbaum02]. The approach needs to make a tradeoff between redundancy in message passing and complexity of the control mechanism under the condition that the new physical federate must not miss any event that ought to be received. A general principle in designing the resending approach is to ensure that all federates operate in the same way as traditional federates not using fault tolerance or cloning.

To minimize extra networking overhead, the proposed approach requires the sender only to resend those events that (1) have been subscribed and (2) have not been received or buffered in the subscriber's TSO queue. The middleware can be designed to help the subscribers notify the particular sender(s) about the reception status of the events originating from the sender. According to the feedback, the sender can selectively generate the image events.

9.2.1. Scheme for Recovering In-transit Events

The proposed approach is as follows, including preparation before a crash/cloning and the action on failure recovery/cloning:

- **Collecting Subscription/Publication/Registration Data.** Each federate builds a Federate Subscription/Publication/Registration (FSPR) Table, which records the classes subscribed/published and the objects registered by other federates. Each federate broadcasts its subscription/publication information and enables other federates to update corresponding entries in their own **FSPR** Tables. When an object instance is registered (with federate ID encoded using middleware), each subscriber updates the table according to the object class and federate to which it belongs. Thus, when attribute updates of an object instance (events) are received, the receiver can trace the source of this event.
- **Buffering Events.** Each sender records its local updates in time stamp order according to their associated object classes. Thus each sender records what events it has generated. Referring to the **FSPR** Table, each sender also knows which federates should receive these events.
- **Regenerating Events.** (1) On **recovering from failure**, the physical federate being recovered requests the senders to deliver those events with timestamp greater than its

current granted time³. Thus the recovered federate can receive those events and pass them to the simulation model with the advance of time. (2) On **cloning**, the physical federate being cloned flushes the events from its TSO queue (see section 3.1) and gives this information to the senders. Thus each sender can identify the in-transit events that it has generated itself. When cloning occurs and new scenarios are created, if necessary the senders (or clones of a sender) can regenerate these events in the new scenarios.

The same approach is applicable for processing interactions. For processing RO events, the Buffer Manager module indexes the outgoing RO events according to the sequence in which they are created, thus the index can be used to order the RO events. Consequently we can also apply this approach in processing RO events.

9.2.2. Fossil Collection

Using the scheme described in the previous section, sent events are buffered at the senders' side against any potential unpredictable failure. As the simulation execution proceeds, the buffered data will accumulate indefinitely; at some stage this will become a bottleneck as system resources are wasted in maintaining a huge amount of redundant data. The same problem exists in using this scheme to deal with cloning. Therefore, it is necessary to perform fossil collection on the logged events.

Since our fault-tolerant model is designed to handle failure at any point, the fossil collection should ensure that events that any subscriber might miss in case of failure are always available. On the other hand, in order to optimize performance, we should dispose of those events that have been received by all subscribers as soon as possible.

³ It is possible that the timestamps of some events to be resent are less than the sender's granted time plus lookahead. In this case, the RTI++ middleware can be designed to encode the content and timestamps of these events in a special RO message, which can then be decoded in the form of TSO events at the receiver's end.

Our fossil collection approach only concerns the logged TSO events. The approach assumes that simulation federates use the conservative time management scheme and are both time regulating and time constrained [Kuhl99][RTING02]. The fossil collection approach is designed to:

- Minimize extra distributed computing.
- Simplify the decision about which events can be disposed.

The RTI ensures that a federate receives all events with timestamp less than its granted time. Therefore, senders do not need to keep events with timestamp less than the granted time of a receiver. Based on this fundamental assumption, the main task of fossil collection is to determine which logged TSO events are safe for a sender to dispose according to its current granted time.

Figure 9.3: Example for calculating timestamp of events to be disposed

A time constrained federate has an associated Lower Bound Time Stamp (LBTS), which is the timestamp of the earliest possible TSO event that may be generated by any other regulating federate [Kuhl99][RTING02]. A constrained federate cannot advance beyond its LBTS. In the scenario illustrated in Figure 9.3, we denote the lookahead of the i th federate ($Fed[i]$) as La_i , its current granted time as T_i and the time of the next request this federate may make to the RTI to advance time as T_i' . We define a “*timestep*” by which $Fed[i]$ advances its time in each loop as $\delta_i = T_i' - T_i$.

Considering the simplest scenario consisting of only two federates, from $Fed[1]$'s perspective, failure may occur in $Fed[2]$ either (1) after $Fed[2]$ has been granted time T_2 but before $Fed[2]$ makes another request to advance time, or (2) after $Fed[2]$ has made a request to advance time to $T_2 + \delta_2$ but before the request is granted.

$Fed[1]$'s LBTS is $T_2 + La_2$ in the first case (hence $T_1 \leq T_2 + La_2$), and its LBTS is $T_2 + La_2 + \delta_2$ in the second case (hence $T_1 \leq T_2 + La_2 + \delta_2$). It is safe for $Fed[1]$ to dispose of buffered events earlier than $Fed[2]$'s current granted time T_2 , which means any event with timestamp less than $T_1 - (La_2 + \delta_2)$ can be removed immediately.

Generalizing to n federates, suppose that $Fed[k](k \neq 1)$ has the smallest federate time T_k of the other federates, so that it is safe for $Fed[1]$ to dispose of events with time earlier than T_k . It is obvious that:

$$La_k + \delta_k \leq \max\{ (La_i + \delta_i) | i \neq 1 \}.$$

Thus, in the worst case, it is safe for $Fed[1]$ to dispose of all logged TSO events with timestamp less than $T_1 - \max\{ (La_i + \delta_i) | i \neq 1 \}$, and we define this value as $Fed[1]$'s **safe lower bound**. The fossil collection algorithm can determine this safe lower bound easily given that the lookahead and timestep of other federates are available. According to the principles set at the beginning of this section, the fossil collection algorithm operates in the following steps:

- **Disseminating lookahead.** Each federate broadcasts its lookahead through the middleware when it enables time regulating.
- **Notifying timestep.** Each federate calculates the maximum of its first few timesteps and broadcasts it through middleware.
- **Disposing events.** When a sender ($Fed[j]$) is granted a new federate time, it calculates

its safe lower bound from its local information and disposes of TSO events accordingly.

Furthermore, for any federate, its “timestep” may change from time to time. To minimize global propagations, a simulation time window can be defined with an upper and lower bound specified. The window of a federate (say $Fed[i]$) is an interval around $T_i + \delta_i$, i.e. $[(T_i + \delta_i) - \xi_1, (T_i + \delta_i) + \xi_2]$, for some $\xi_1 < \delta_i$ and $\xi_1, \xi_2 > 0$.

When $Fed[i]$ requests to advance its time to T_i' , there are four cases:

- (1) If $T_i' \leq (T_i + \delta_i) - \xi_1$, set new $\delta_i = (\delta_i - \xi_1)$;
- (2) If $(T_i + \delta_i) - \xi_1 < T_i' \leq (T_i + \delta_i)$, keep δ_i unchanged;
- (3) If $(T_i + \delta_i) < T_i' \leq (T_i + \delta_i) + \xi_2$, set new $\delta_i = (\delta_i + \xi_2)$;
- (4) If $T_i' > (T_i + \delta_i) + \xi_2$, set new $\delta_i = (T_i' - T_i)$.

When δ_i decreases, it is safe for the other federates to calculate safe lower bounds using a larger δ value for $Fed[i]$. In case (1), we still need to send other federates the new δ_i , as it is out of the window. For cases (3) and (4), we need to ensure other federates have received the new δ_i before $Fed[i]$ requests the RTI to advance time. This can be achieved easily using a middleware approach. After $Fed[i]$ is granted a new time, the time window will be moved forward to adapt to the change. It might be necessary to specify large time windows to those federates with large change in their δ ; nevertheless this is not often the case. A similar approach applies in the case that the lookahead of a federate is modified.

To further reduce the redundancy of logged TSO events, the approach can compute the safe value for each object class (interaction class) individually. Only those subscribers to the selected object/interaction class will be involved.

The proposed scheme for dealing with in-transit events can also be applied in simulation cloning. In this case, a sender only needs to consider those federates (clones) belonging to the same scenario(s) when calculating the safe lower bound.

9.3. Other Uses of the Decoupled Federate Architecture

The decoupled federate architecture can provide other advantages to distributed simulation technology. Executing HLA-based simulations normally requires users to allocate computing resources to federates and the RTIEXEC (DMSO RTI) statically, which lacks flexibility. The interoperation amongst federates over the Internet is also restrained by security requirements (e.g. firewalls) or heterogeneity of the computing resources in different organizations.

Web services provide interoperability between software applications distributed over the Internet. As an extension of Web services, the Grid has been designed to manage sharable and often heterogeneous computing resources distributed in different organizations.

One of the advantages of distributing simulations over a network is the capability of exploiting sharable computing power. In the case an individual federate experiences high congestion at the host where it executes, load balancing [Zhou88] can improve the overall simulation performance. This can be achieved by migrating load of the congested federate to other lightly loaded hosts [Yuan04].

9.3.1. Web or Grid Enabled Architecture

The Grid infrastructure provides dependable, consistent, pervasive, inexpensive and secure access to high-end computational capabilities [Foster01]. The Open Grid Services Architecture (OGSA) extends the Web services to include additional functionalities. With these features, Web and Grid services offer a promising approach to enhance the

flexibility and interoperability for large scale HLA-based simulations.

The HLA does not define security mechanisms for using RTI services, while the Grid provides a built-in security architecture to support authentication and secure communication in accessing Grid services [Globus04].

The HLA supports interoperability among HLA-compliant federates. However, this feature is often hampered in the context of large scale distributed simulations over the Internet. This is due to various security requirements on the networking resources belonging to different organizations. There is a need to combine the HLA and Web or Grid technologies in order to achieve the advantages of both. Directly utilizing Web or Grid services will incur extra effort from the developers in modeling simulations. An alternative approach is to use a **Web or Grid Enabled Architecture**, which manages the RTI services at the backend while presenting dynamic access of the RTI services to the simulation model or the end users via Web or Grid services. The architecture can be used to relieve the developers from the complication of coding Web or Grid services in existing simulation models.

Web and Grid technologies can help large scale distributed applications in terms of resource coordination and connectivity. Using a decoupled federate architecture can ease the combination of Web or Grid technologies with the HLA.

Figure 9.4 gives an abstract model that supports a Web or Grid Enabled Architecture. The model treats physical federates and the RTIEXEC process as components managed by the Web or Grid services. Therefore, each physical federate (or RTIEXEC) is encapsulated as a Web or Grid service and becomes a Web or Grid-enabled component accessible via web-based communications (such as SOAP, BEEP). A Web or Grid-enabled RTI library is embedded in the middleware to bridge the model and the physical federate.

The model has significant advantages compared with traditional simulation using the HLA RTI in the sense that (1) it can cross firewalls, (2) it enhances the interoperability and connectivity among simulation federates and (3) it eases the interoperation between HLA federates and non-HLA simulations. The communication between the virtual federate and the physical federate is via Web services which can pass through most firewalls. Using Web services allows interoperation of federates physically located at various organizations, while connecting them over a normal RTI may encounter network barriers. The RTI component becomes a Web service in the context of the architecture. Non-HLA simulations can also be wrapped as Web services. Therefore, HLA simulations can easily interoperate with non-HLA ones through Web services.

Figure 9.4: Model supporting Web or Grid Enabled Architecture

Using Grid services has the added benefits over using pure Web services as follows:

- Grid services provide dynamic creation and lifecycle management. The RTI and federate instances are dynamic and transient service instances under the management of the Grid service infrastructure.
- The model also eases the discovery of the Grid-enabled RTI and federate services

using the Grid Index Service [Globus04].

- The Grid provides security in accessing services using its built-in mechanisms such as authentication, secured communication etc.
- With the notifications feature of the Grid, the simulation model does not have to block for the completion of RTI calls and may operate asynchronously from the physical federate.

With the nature of the Grid, this model has the potential in providing fault tolerance and load balancing. The model supports the reusability of legacy federate code and it also allows federates developed upon the Web or Grid Enabled Architecture to interact with other traditional federates.

9.3.2. Load Balancing

A large scale distributed simulation running over the Internet is liable to encounter congestion in some computing hosts or over part of the network. The congestion will cause performance degradation of some federate(s), thus the execution efficiency of the whole federation may be lowered. Load balancing technologies are often used in a distributed simulation (system) to smooth out periods of high congestion, therefore the performance of the system can be improved.

Load management of federates at the simulation model level has been addressed by some researchers, such as the approach proposed in [Cai01]. However, there exists significant overhead and implementation complexity for dealing with RTI related computing during the migration of a federate at runtime [Zajac03]. For example, the federate being migrated needs to resign from the federation and the new federate needs to join the federation. The test bed described in section 8.2 shows that it can take about 20 seconds for a federate to complete the RTI call *joinFederationExecution* [RTING02]. The migration design also needs to ensure correct event delivery and state consistency at the

RTI level.

This section presents a generic model to provide load balancing in the HLA. Our approach treats the simulation model and the physical federate as separate tasks to be managed, and attempts to distribute the model's load to lightly congested nodes when necessary. Appropriate existing algorithms for load management may be adopted. The approach focuses on avoiding the complication of dealing with RTI related operations while migrating running federates.

Figure 9.5: Load balancing model for simulation model

An abstract model for load balancing is illustrated in Figure 9.5, in which the Resource Observer monitors the availability of resources. In the case of host congestion, the model may alleviate the lack of computing resources by migrating the virtual federate to another idle host. The migration process is managed based on the load balancing algorithms. The migrated virtual federate (a replica of the old one) on the new host links to the original physical federate, and both processes form a federate executive to continue the simulation session.

The load balancing model separates the simulation model's migration from the physical federate. The model has advantages in that (1) it does not require the physical federate to perform extra RTI operations to support the model migration, (2) it avoids dealing with in-transit events during migration, (3) it saves the overhead for joining the federation and (4) the physical federate keeps intact while servicing the new virtual federate as before.

9.4. Summary

This chapter presents an approach for facilitating fault tolerance in distributed simulations using the decoupled federate architecture. The approach can isolate the faults of part of a federation and provide fault recovery without rollback of the simulation execution. In particular, the fault-tolerant model can be used to provide runtime robustness for the Single-Federation solution to distributed simulation cloning (see chapter 4). This chapter also introduces a method for dealing with in-transit events in the case of both fault recovery and cloning. Other possible uses of the decoupled federate architecture, namely a Web or Grid enabled architecture and load balancing, are also discussed. HLA-based simulations may benefit from the flexible resource management and enhanced interoperability provided by the nature of Web or Grid technologies. A federate with a heavy load may be migrated in a networked environment, and our approach frees the developers from handling RTI states or in-transit events when designing their federate migration method as well as minimizing the overhead.

Chapter 10: Conclusions and Future Work

10.1. Achievements

In this thesis, we have investigated important research issues in HLA-based distributed simulation cloning and have presented alternative management mechanisms. Simulation cloning offers the flexibility for a distributed simulation to examine multiple decision policies concurrently at a decision point. Thus, the execution time can be reduced and the analyst may quickly obtain multiple sets of results that represent the impacts of alternative decisions. The technology provides simulation users with a decision support platform much more powerful and flexible than the traditional “linear” approach. All the objectives set at the beginning of the thesis have been fulfilled. The major achievements are summarized as follows:

- Establishment of the foundation theory of distributed simulation cloning
- Investigation and design of efficient and reliable solutions to support distributed simulation cloning
- Development of a generic approach for federate cloning using our decoupled federate architecture
- Development of mechanisms for management of concurrent scenarios in cloning-enabled simulations
- Design and development of an RTI++ middleware to provide reusability of existing simulation models and user transparency
- Investigation and development of alternative distributed simulation cloning mechanisms, including entire and incremental cloning

This thesis establishes the foundation theory of distributed simulation cloning, defining the terms and identifying the issues involved. A federate can perform cloning actively or passively according to its own or its partner's requirements. When cloning of a federate is required, an entire cloning mechanism replicates the whole scenario whereas an incremental mechanism only clones those federates whose states will alter at the decision point. Execution of identical federates (clones) may be shared by multiple scenarios. Thus, there exists dynamic relationships amongst scenarios and federates.

We consider Multiple-Federation (MF) and Single-Federation (SF) solutions to distributed simulation cloning. This thesis compares and analyzes their advantages as well as drawbacks. The Multiple-Federation solution has the advantage of robustness and provides simple synchronization of federates in each individual scenario. Using the Single-Federation solution, management of cloning and computation sharing amongst scenarios is much easier to achieve.

These candidate solutions have been analyzed and compared in terms of efficiency, complexity in management and reliability. In the case where there is a high data exchange, the MF solution can perform much better than the simple SF solution as the concurrent scenarios are not efficiently partitioned when the latter solution is applied. The two solutions exhibit close performance by applying Data Distribution Management services to the Single-Federation solution to give the DSF solution. Furthermore, the two solutions have no significant difference in terms of federate synchronization. As DSF is also superior in convenience for design and scenario management, it has been adopted for developing the infrastructure of distributed simulation cloning.

In order to tackle the problems of replicating a federate instance and providing fault tolerance to HLA-based distributed simulations, a decoupled federate architecture has been designed. This approach decouples the simulation model from the Local RTI Component, with these two basic modules forming a single federate executive. A virtual

federate process executes the simulation model while a physical federate process provides RTI services to the corresponding model at the backend. The decoupled approach successfully supports user transparency by using the standard HLA specification to provide an interface to the simulation model. The approach maintains the semantics of executing RTI service calls and conveying RTI callbacks to the simulation model. The decoupled architecture facilitates state saving and replication at the RTI level. It has been used to develop a mechanism for replicating the simulation federate, which is reliable, correct and relatively model-independent.

Decoupled federates have been compared with normal federates in terms of latency and time advancement performance. The decoupled architecture incurs a negligible latency overhead for small or medium payload and only a slight extra latency in the case of a bulky payload. Furthermore, the decoupled federate architecture has a promising performance of time advancement to normal federates. This makes the architecture an ideal approach to support cloning.

Concurrent scenarios coexist in a single federation session as a result of simulation cloning, each representing a particular execution path. Interactions need to be confined within each scenario to guarantee the correctness of simulation results. On the other hand, a mechanism to share computation is required by the shared clones that cross multiple scenarios. DDM services have been used at the backend to route events and partition scenarios. In this thesis, we have studied issues involved in managing concurrent scenarios and suggested candidate approaches. To obtain the best DDM performance, the internal mechanism of DDM services implemented in DMSO RTI-NG has been analyzed.

With the spawning of scenarios, the number of clones in a cloning-enabled simulation session keeps increasing. We have developed the idea of a scenario tree to identify and denote scenarios correctly and efficiently. In this thesis, we discuss two possible solutions we have designed to code scenarios and to manipulate characteristic region extents

associated with a scenario, namely a recursive region division solution and a point region solution. The former solution has advantages in minimizing the complication for sharing federates (clones) and mapping the scenario identity to the region extent. However, the point region solution has been adopted in our cloning technology since it can meet the region allocation requirements in any situation while optimizing the use of networking channels for DDM services. The scenario management mechanism ensures both proper scenario partitioning and accurate computation sharing across scenarios, which are potentially contradictory requirements. The mechanism also accurately represents and manipulates the dynamically complex relationship amongst increasing number of scenarios.

A middleware approach is used to hide the complexity incurred by simulation cloning. A cloning-enabled infrastructure, namely RTI++, contains the modules for controlling, and managing cloning, managing scenarios as well as handling cloning regions and the underlying DDM services. The RTI++ presents a standard RTI interface to the simulation model. The infrastructure supports the correct HLA semantics and user transparency, which maximizes the reusability of legacy simulation applications and also minimizes the users' effort to utilize cloning technology in developing their models.

One important goal of cloning technology is to optimize execution by avoiding repeated computation amongst independent scenarios. Alternative cloning mechanisms have been developed: an entire cloning mechanism and an incremental cloning mechanism. We have investigated and designed algorithms for managing active and passive cloning, which ensure correct replication of federates when required. These algorithms maintain the state consistency of the simulation session by introducing proper coordination and synchronization on cloning. State saving and replication are designed using the decoupled federate architecture.

Although the incremental mechanism requires a more complex control algorithm, it makes the most use of computation sharing. In this thesis, we have studied the incremental cloning mechanism and established a theory of incremental distributed simulation cloning. The incremental mechanism initiates cloning of a federate only when strictly necessary, while the entire cloning mechanism makes replicas of every federate in a scenario. A sensitive event checking algorithm facilitates clone sharing across scenarios, and avoids/delays the passive cloning as long as possible while guaranteeing the correctness of computation sharing. The checking algorithm employs an entity mapping approach to identify events, which maps the new HLA entities created for the clones in state replication to the existing ones known to the simulation models and vice versa. The incremental cloning mechanism not only avoids repeated computation prior to cloning, but also further saves identical federate execution amongst scenarios without prior knowledge of the behavior of the federate.

A series of experiments has been performed to investigate the correctness and performance of the two alternative mechanisms using an example of a distributed supply chain simulation. The experimental results are compared for traditional, cloning-disabled and cloning-enabled federates in terms of the consistency of outputs and computing efficiency. The experimental results show that the technology provides a correct cloning mechanism for HLA-based distributed simulations. The experimental results also show that the cloning technology can reduce the time of executing multiple scenarios of distributed simulations. The scalability of the cloning mechanisms has also been examined, and experimental results show that the proposed cloning technology has a promising scalability for both mechanisms.

However, there is still not a general solution to the problem of a possible combinatorial explosion of scenarios in distributed simulation cloning in some extreme situations. Although it is unlikely to occur, simulation analysts should be aware of the problem and

avoid defining too many cloning triggers or specifying a large number of new scenarios on active cloning.

The decoupled federate architecture can also be exploited in providing fault tolerance and load balancing as well introducing Web and Grid technologies to HLA-based simulations. In this thesis, we have discussed and proposed candidate solutions to address these issues. An alternative scheme for dealing with in-transit messages is also described. The HLA-based distributed simulations may benefit from the decoupled federate architecture in the following ways:

- Fault tolerance. We can easily isolate the faults of part of a federation and provide fault recovery without rolling back the simulation execution.
- Web or Grid Enabled Architecture. The decoupled architecture can be exploited to combine the advantage of Web/Grid services and the HLA. HLA-based simulations may profit from the flexible resource management and enhanced interoperability provided by a Web/Grid enabled architecture.
- Supporting load balancing. Using the decoupled architecture can free developers from handling RTI states and in-transit events, and it avoids the overhead incurred in saving/restoring RTI states when adopting a federate migration approach to balance the federates' load.

10.2. Future Work

For our future work, we need to explore further the design and implementation of the methods to support other uses of the decoupled federate architecture, that is fault tolerance, a Web or Grid enabled architecture and load balancing. Another interesting issue is to design an optimizer module for cloning-enabled simulations, which can explore different execution paths using simulation cloning and optimize the execution

autonomously to satisfy preset objective functions. Thus, the analysts' effort can be alleviated, and the effectiveness of simulation execution will be enhanced.

The current implementation does not straightforwardly support the cloning of federates built using Commercial Off-The-Shelf (COTS) simulation packages. However, those COTS packages that already provide the functionalities of saving the simulation model's states can be easily incorporated into the decoupled federate architecture to achieve simulation cloning. In the future, when interoperability amongst different COTS packages is standardized [Wang04], simulation cloning can be a desirable scheme to optimize execution. We need to investigate the issues involved in applying our distributed cloning technology in building cloning-enabled models using COTS packages. The technology will have a much wider applicability especially for existing simulations modeled using COTS packages.

10.3. Summary

In this project, we have pioneered the investigation of distributed simulation cloning and have developed the theory and concepts. This project has successfully created cloning mechanisms for large scale HLA-based distributed simulations. A RTI++ middleware has been designed to facilitate correct replication of federates at runtime. The entire and incremental cloning mechanisms have been developed, and the algorithms for cloning have been implemented. The distributed simulation cloning technology considerably improves the execution efficiency for executing HLA-based distributed simulations.

References:

- [AbouRizk00] S. AbouRizk and Y. Mohamed, "Symphony-An Integrated Environment for Construction Simulation", *Proceedings of the 2000 Winter Simulation Conference*, pp 1907-1974, Orlando, Florida, USA, December 2000.
- [Addie01] R. G. Addie, "Quantum Simulation: Rare Event Simulation by Means of Cloning and Thinning", *European Transactions on Telecommunications*, 13:4, pp 387-397, July/August, 2002.
- [Agarwal05] A. Agarwal and M. Hybinette, "Merging Parallel Simulation Programs", *Proceedings of the 19th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pp 227-233, Monterey, California, USA, June 2005
- [Bassiouni97] M. Bassiouni et al., "Performance and Reliability Analysis of Relevance Filtering for Scalable Distributed Interactive Simulation", *ACM Transactions on Modeling and Computer Simulation*, 7:3, pp 293-331, July 1997.
- [Berchtold01] C. Berchtold and M. Hezel. "An Architecture for Fault-tolerant HLA-based Simulation", *Proceedings of the 15th International European Simulation Multi-Conference (ESM) 2001*, pp 616-620, Prague, Czech Republic, June 2001.
- [Birman96] K. P. Birman, "Building Secure and Reliable Network Applications", *Prentice Hall and Manning Publishing Company*, 1997.
- [Cai01] W. Cai, S. J. Turner and B. P. Gan, "Hierarchical Federations: An Architecture for Information Hiding", *15th Workshop on Parallel and Distributed Simulation (PADS '01)*, pp 67-74, Lake Arrowhead, California, USA, May 2001.
- [Cai02] W. Cai, S. J. Turner and H. Zhao "A Load Management System for Running HLA-based Distributed Simulations over the Grid". *Proceedings of the 6th IEEE*

International Workshop on Distributed Simulation and Real-Time Applications, pp 7-14, Fort Worth, Texas, USA, October 2002.

[Ciciani90] B. Ciciani, D. M. Dias and P. S. Yu, "Analysis of Replication in Distributed Database Systems", *IEEE Transaction on Knowledge and Data Engineering*, 2:2, pp 247-261, June 1990.

[Cooper93] K. D. Cooper, M. W. Hall and K. Kennedy, "A methodology for procedure cloning", *Computer Languages*, 19:2, pp 105-118, April 1993.

[Cristian91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of the ACM*, 34:2, pp 57-78, February 1991.

[Cunning99] S. J. Cunning, S. Schulz and J. W. Rozenblit, "An Embedded System's Design Verification Using Object-Oriented Simulation Techniques", *Simulation*, 72:4, pp 238-249, April 1999.

[Dahmann98] J. S. Dahmann, F. Kuhl and R. Weatherly, "Standards for Simulation: As Simple As Possible But Not Simpler, The High Level Architecture for Simulation", *Simulation*, 71:6, pp 378-387, December 1998.

[Dahmann99] J. S. Dahmann, "The High Level Architecture and Beyond: Technology Challenges", *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pp 64-70, Atlanta, Georgia, USA, May 1999.

[Danami98] Om. P. Danami and V. K. Garg, "Fault-Tolerant Distributed Simulation", *Proceedings of the 12th workshop of Parallel and Distributed Simulation*, pp 38-45, Banff, Alberta, Canada, May 1998.

[Das94] S. Das, R. Fujimoto, K. Panesar, D. Allison and M. Hybinette, "GTW: a time warp system for shared memory multiprocessors", *Proceedings of the 1994 Winter*

Ph.D. Thesis

References

Simulation Conference, pp 1332-1339, Lake Buena Vista, Florida, USA, December 1994.

[DIS94] DIS Steering Committee. *The DIS Vision, A Map to the Future of Distributed Simulation*, Technical Report IST-SP-94-01, Institute for Simulation and Training, Orlando, Florida, USA, 1994.

[DMSO95] Defense Modeling and Simulation Office, DoD Modeling and Simulation Master Plan, U.S. Government Printing Office, Washington DC, 1995.

[DMSO03] Defense Modeling and Simulation Office, Software Distribution Center, <https://sdc.dmsomil>, 2003.

[EA04] Electronic Arts Inc., Westwood Studio, <http://www.westwood.com/>, 2004

[Elnozahy93] E. N. Elnozahy, "Fault Tolerance in Distributed systems Using Rollback-Recovery and Process Replication", *Ph.D. Thesis*, Rice University, Texas, USA, 1993.

[Engelson00] V. Engelson, "Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing", *Ph.D. Thesis*, Dissertation No. 627, Linköpings Universitet, Linköping, Sweden, 2000.

[FDK04] <http://www.cc.gatech.edu/computing/pads/fdk.html>, 2004

[Foster01] I. Foster, C. Kesselman and S. Tuecke. "The Anatomy of the Grid". *International Journal of High Performance Computing Applications*, 15:3, pp 200-222, Fall 2001.

[Franks97] S. Franks, F. Gomes, B. Unger and J. Cleary, "State Saving for Interactive Optimistic Simulation", *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pp 72-79, Lockenhaus, Austria, June 1997.

[Fujimoto00] R. M. Fujimoto, "Parallel Distributed Simulation Systems", *New York: Wiley*, ISBN 0-471-18383-0, 2000.

- [Fujimoto04] <http://www.cc.gatech.edu/computing/pads/tech-highperf.html>, 2004.
- [Gan00] B. P. Gan, L. Liu, S. Jain, S. J. Turner, W. Cai and W. Hsu, “Distributed Supply Chain Simulation Across Enterprise Boundaries”, *Proceedings of the 2000 Winter Simulation Conference*, pp 1245-1251, Orlando, Florida, USA, December 2000.
- [Gilmer98] J. B. Gilmer, Jr. and F. J. Sullivan, “Alternative Implementations of Multitrajectory Simulation”, *Proceedings of the 1998 Winter Simulation Conference*, pp 865-872, Piscataway, New Jersey, USA, December 1998.
- [Gilmer99] J. B. Gilmer, Jr. and F. J. Sullivan, “Multi-trajectory Simulation Performance for Varying Scenario Size”, *Proceedings of the 1999 Winter Simulation Conference*, pp 1137-1146, Phoenix, Arizona, USA, December 1999.
- [Gilmer00] J. B. Gilmer, Jr. and F. J. Sullivan, “Recursive Simulation to Aid Models of Decision Making”, *Proceedings of the 2000 Winter Simulation Conference*, pp 958-963, Orlando, Florida, USA, December 2000.
- [Glasserman96] P. Glasserman, P. Heidelberger, P. Shahabuddin and T. Zajic, “Splitting for rare event simulation: analysis of simple cases”, *Proceedings of the 1996 Winter Simulation Conference*, pp 302-308, Coronado, California, USA, December 1996.
- [Globus04] Globus. <http://www.globus.org/>, 2004.
- [Görg01] C. Görg, E. Lamers, O. Fuß and P. Heegaardy, “Rare Event Simulation”, *Proceedings of the European COST (256) telecommunication Symposium*, Modeling and simulation environment for satellite and terrestrial communication networks, ISBN 0-7923-7547-5, Kluwer Academic Press, pp 365-396, Jan 2001.
- [Henriksen97] J. O. Henriksen, “An Introduction to SLXTM”, *Proceedings of the 1997 Winter Simulation Conference*, pp 559-566, Atlanta, Georgia, USA, December 1997.

Ph.D. Thesis

References

[Henriksen98] J. O. Henriksen, “Stretching the Boundaries of Simulation Software”, *Proceedings of the 1998 Winter Simulation Conference*, pp 227-234, Washington DC, USA, December 1998.

[Huang01] W. Huang, “General Purpose Cellular Automata Programming”, *Master Thesis*, Iowa State University, Ames, Iowa, USA, 2001.

[Hybinette97] M. Hybinette and R. M. Fujimoto, “Cloning: A Novel Method for Interactive Parallel Simulation”, *Proceedings of the 1997 Winter Simulation Conference*, pp 444-451, Atlanta, Georgia, USA, December 1997.

[Hybinette00] M. Hybinette, “Interactive Parallel Simulation Environment”, *Ph.D Thesis*, Georgia Institute of Technology, Atlanta, Georgia, USA, 2000.

[Hybinette01] M. Hybinette and R. M. Fujimoto, “Cloning parallel simulations”, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 11:1, pp 378-407, January 2001.

[Hybinette02] M. Hybinette and R. M. Fujimoto, “Scalability of Parallel Simulation Cloning”, *Proceedings of the 35th Annual Simulation Symposium*, p.0275, San Diego, California, USA, April 2002.

[Hybinette04] M. Hybinette, “Just-in-time cloning”, *Proceedings of the eighteenth workshop on Parallel and distributed simulation*, pp 45-51, Kufstein, Austria, May 2004.

[Hyett02] M. Hyett and R. Wuerfel, “Implementation of the Data Distribution Management Services in the RTI-NG”, *Proceeding of 2002 Spring Simulation Interoperability Workshop*, paper no. 02S-SIW-044, Orlando, Florida, USA, March 2002.

[IEEE1278] IEEE 1278, IEEE Standard for Distributed Interactive Simulation, 1993.

[IEEE1516] IEEE 1516, IEEE Standard for High Level Architecture, 2001.

Ph.D. Thesis

References

- [IEEE1516.3] IEEE1516.3, IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP), 2003.
- [Jie01] W. Jie, “POEMS: A Parallel Object-oriented Environment for Multi-computer Systems”, *Ph.D. Thesis*, Nanyang Technological University, Singapore, 2001.
- [Johnson89] D. B. Johnson, “Distributed System Fault Tolerance Using Message Logging and Checkpointing”, *Ph.D. Thesis*, Rice University, Texas, USA, 1989.
- [Kuhl99] F. Kuhl, R. Weatherly and J. Dahmann, *Creating Computer Simulation Systems: An Introduction to HLA*, ISBN 1-3-022511-8, Prentice Hall, 1999.
- [Law99] A. M. Law, W. David Kelton, *Simulation Modeling and Analysis*, 3rd Edition, ISBN 0-07-116537-1, McGraw-Hill, 2000.
- [Mchany99] R. McHaney, “Integration of the Genetic Algorithm and Discrete-Event Computer Simulation for Decision Support”, *Simulation*, 72:6, pp 401-411, June 1999.
- [McLean00] C. McLean and F. Riddick, “The IMS Mission Architecture for Distributed Manufacturing Simulation”, *Proceeding of the 2000 Winter Simulation Conference*, 1539-1548, Orlando, Florida, USA, December 2000.
- [Möller05] B. Möller, B. Löfstrand and Mikael Karlsson, “Developing Fault Tolerant Federations Using HLA Evolved”, *Proceeding of 2005 Spring Simulation Interoperability Workshop*, paper no. 05S-SIW-048, San Diego, California, USA, April 2005.
- [Morse01] K. L. Morse and M. D. Petty, “Data Distribution Management Migration from DoD 1.3 to IEEE 1516”, *Proceeding of the Fifth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, pp 58-65, Cincinnati, Ohio, USA, August 2001.

- [Morse03] K. L. Morse, D. Drake and R. P. Z Brunton, “Web Enabling an RTI – an XMSF Profile”, *Proceedings of the 2003 European Simulation Interoperability Workshop*, paper no. E03-SIW-034, Stockholm, Sweden, June 2003.
- [Nicola00] M. Nicola and M. Jarke, “Performance Modeling of Distributed and Replicated Databases”, *IEEE Transaction on Knowledge and Data Engineering*, 12:4, pp 645-672, July/August 2000.
- [Pitch04] Pitch Kunsapsutveckling AB, <http://www.pitch.se/>, 2004
- [Plevyak96] J. Plevyak, “Optimization of Object-Oriented and Concurrent Programs”, *Ph.D. Thesis*, University of Illinois Urbana-Champaign, Illinois, USA, 1996.
- [Pope91] A. Pope and R. Schaffer, “The SIMNET Network and Protocols”, *BBN Report No 7627, BBN Systems and Technologies*, June 1991.
- [Rao00] D. M. Rao and P. A. Wilsey, “Dynamic Component Substitution in Web-based Simulation”, *Proceedings of the 2000 Winter Simulation Conference*, pp 1840-1848, Orlando, Florida, USA, December 2000.
- [RTING02] RTI 1.3-Next Generation Programmer’s Guide Version 5, *DoD, DMSO*, February 2002.
- [Rocchetti01] M. Rocchetti, P. Salomoni and M. E. Bonfigli, “A Design for a Simulation-based Multimedia Learning Environment”, *Simulation*, 76:4, pp 214-221, April 2001.
- [Rycerz05] K. Rycerz, M. Bubak, M. Malawski and P. Slood, “A Framework for HLA-Based Interactive Simulations on the Grid”, *Simulation*, 81:1, pp 67-76, January 2005.
- [Schulze99] T. Schulze, S. Straßburger and U. Klein, “Online-Data Processing in Simulation Models: New Approaches and Possibilities Through HLA”, *Proceedings of the 1999 Winter Simulation Conference*, pp 1602-1609, Washington DC, USA, 1999.

- [Schulze00] T. Schulze, S. Straßburger and U. Klein, “HLA-federate Reproduction Procedures In Public Transportation Federations”, *Proceedings of the 2000 Summer Computer Simulation Conference*, Vancouver, Canada, July 2000.
- [Shehory98] O. Shehory, K. Sycara, P. Chalasani and S. Jha, “Agent cloning: an approach to agent mobility and resource allocation”, *IEEE Communications*, 36:7, pp 58-67, July 1998.
- [Stevens98] W. R. Stevens, *UNIX Network Programming, Networking APIs: Sockets and XTI*, Vol. 1, 2nd Edition, Prentice Hall, 1998.
- [Stevens99] W. R. Stevens, *UNIX Network Programming, Inter-Process Communications*, Vol. 2, 2nd Edition, Prentice Hall, 1999.
- [Straßburger98] S. Straßburger, T. Schulze, U. Klein and J. O. Henriksen, “Internet-based Simulation using Off-the-Shelf Simulation Tools and HLA”, *Proceedings of the 1998 Winter Simulation Conference*, pp 1669-1676, Washington DC, USA, December 1998.
- [Sycara96] K. P. Sycara, A. Pannu, M. Williamson, Dajun Zeng and K. Decker, “Distributed Intelligent Agents”, *IEEE Expert*, 11:6, pp 36-46, December 1996.
- [Sycara98] K. P. Sycara, “Multi-agent Systems”, *AI Magazine: The American Association for Artificial Intelligence*, pp 79-92, June 1998.
- [Tanenbaum02] A. S. Tanenbaum and M. van Steen. “Distributed Systems: Principles and Paradigms”, Prentice Hall, 2002.
- [Turner01] S. J. Turner, W. Cai and B. P. Gan, “Adapting a Supply-chain Simulation for HLA”, *Proceeding of the Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications*, pp 67-74, San Francisco, California, USA, August 2001.

- [Vahid99] F. Vahid, "Procedure cloning: a transformation for improved system-level functional partitioning", *ACM Transactions on Design Automation of Electronic Systems*, 4:1, pp 70-96, January 1999.
- [Wang04] X. Wang, S. J. Turner, W. Y. H. Low and B. P. Gan, "A Generic Architecture for the Integration of COTS Packages with the HLA", *Proceedings of the 2004 UK Operational Research Society Simulation Workshop*, pp 225-233, Birmingham, UK, March 2004.
- [Yu02] H. Yu and A. Vahdat, "Design and evaluation of a conit-based continuous consistency model for replicated services", *ACM Transactions on Computer Systems*, 20:3, pp 239-282, August 2002.
- [Yuan04] Z. Yuan, W. Cai, W. Y. H. Low and S. J. Turner, "Federate Migration in HLA-Based Simulation", *Proceedings of the 2004 International Conference on Computational Science*, pp 855-864, Kraków, Poland, June 2004.
- [Zajac03] K. Zajac, M. Bubak, M. Malawski and P. Sloot, "Towards a Grid Management System for HLA-based Interactive Simulations", *Proceedings of the 7th IEEE International Symposium on Distributed Simulation and Real Time Applications*, pp 4-11, Delft, Netherlands, October 2003.
- [Zhou88] S. Zhou, "A Trace-Driven Simulation Study of Dynamic Load Balancing", *IEEE Transactions on Software Engineering*, 14:9, pp 1327-1341, September 1988.

Appendix: Papers Accepted and Published from this Research

“A Framework for Robust HLA-Based Distributed Simulation”, D. Chen, S. J. Turner, Wentong Cai, *20th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2006)*, May 2006, Accepted for publication on Dec. 2005.

“Algorithms for HLA-based Distributed Simulation Cloning”, D. Chen, S. J. Turner, W. Cai, B. P. Gan and M. Y. H. Low, *ACM Transactions on Modeling and Computer Simulation*, paper no. 2004-24, Accepted for publication on Jul. 2005.

“Incremental HLA-based Distributed Simulation Cloning”, D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low, *Winter Simulation Conference 2004*, pp 386-394, Dec. 2004.

“HLA-Based Distributed Simulation Cloning”, D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low, *8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pp. 244-247, Oct. 2004.

“Management of Simulation Cloning in HLA-based Distributed Simulations”, D. Chen, S. J. Turner, B. P. Gan, W. Cai, M. Y. H. Low and J. Wei, *European Simulation Interoperability Workshop 2004*, 04E-SIW-010, Jun. 2004 (SIWzie award, Outstanding paper).

“A Decoupled Federate Architecture for Distributed Simulation Cloning”, D. Chen, S. J. Turner, B. P. Gan, W. Cai, M. Y. H. Low and J. Wei, *15th European Simulation Symposium*, pp 131-140, Oct. 2003 (Outstanding paper award).

Ph.D. Thesis

References

“Data Distribution Management in Distributed Simulation Cloning”, D. Chen, B. P. Gan, S. J. Turner, W. Cai and J. Wei, *European Simulation Interoperability Workshop 2003*, 03E-SIW-024, Jun. 2003.

“Alternative Solutions for Distributed Simulation Cloning”, D. Chen, S. J. Turner, B. P. Gan, W. Cai, J. Wei and N. Julka, *Simulation: Transactions of the Society for Modeling and Simulation International*, Vol. 79, No. 5-6, pp 299-315, 2003.

“Evaluating Alternative Solutions for Cloning in Distributed Simulation”, D. Chen, B. P. Gan, S. J. Turner, W. Cai, N. Julka and J. Wei, *36th Annual Simulation Symposium*, pp 201-208, Apr. 2003.