Efficient and fault tolerant HLA-based simulation

Li, Zengxiang

2012

Li, Z. X. (2012). Efficient and fault tolerant HLA-based simulation. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/48170>

<https://doi.org/10.32657/10356/48170>

NANYANG TECHNOLOGICAL UNIVERSITY

Efficient and Fault Tolerant HLA-based Simulation

Ph.D Thesis

By

Li Zengxiang

Supervisor: Prof. Cai Wentong

School of Computer Engineering

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

Jan, 2012

Acknowledgments

Firstly, I would like to express my sincere thanks to my supervisor Dr. Cai Wentong and co-supervisor Dr. Stephen John Turner for their continuous, remarkable and patient guidance throughout my five years' study. Their expertise and invaluable advice has left a profound impact on my work, especially the way I think and solve new problems.

I would also like to thank my senior Pan Ke for the precious discussions. I am greatly indebted to Irene Goh, Poliran Kenneth Caballes and Tay Siew Eng, the technicians of Parallel and Distributed Computing Center (PDCC) for their patient help along the way.

Last, but not least, I would like to thank all of my family members: my parents, sisters, brothers, wife and my two-years old daughter for their strong support and all of my friends for their consistent help.

Summary

Distributed simulation subdivides a complex simulation (federation) into a group of simulation components (federates) and executes them in distributed manner. The High Level Architecture (HLA), an IEEE 1516 standard, provides a general framework for developing large-scale distributed simulations. The Runtime Infrastructure (RTI) is a middleware that controls the communication among federates according to the HLA interface specification. The simulation executions may involve a large number of computationally intensive federates and thus are time and resource consuming. What is worse, these federates may be subject to crash-stop and Byzantine failures and the risk of federation failure increases with the federation scale. In this thesis, we propose mechanisms to support efficient and fault tolerant HLA-based simulation by exploiting the advantages of decoupled federate architecture, in which a federate connects to federation through its corresponding Decoupled RTI Component (DRC).

Workload imbalance generally leads to poor distributed simulation performance. To achieve load balancing, we propose to migrate federates from heavily-loaded computing nodes to lightly-loaded ones. Using the decoupled federate architecture, only needs the federate to be migrated to the destination computing node; whereas the DRC can stay at the same place and keep the connection to the federation. One-phase migration protocol is first proposed to illustrate the federate migration process. Then, two-phases and relay-based migration protocols are further developed to reduce migration overhead by overlapping federate migration with continuous federate execution.

Inspired by the concepts of active replication and software diversity, we propose to improve simulation performance by replicating simulations with alternative synchronization approaches. In the decoupled federate architecture, federates can be replicated without increasing federation scale. Barrier and No-Barrier replication structures are first introduced. However, they are not able to support concurrent executions of federate replicas

which employ different synchronization approaches. For this purpose, an Un-Identical replication structure is further proposed. The simulation performance is enhanced by choosing the fastest replica to represent the federate in the federation, taking the advantages of different synchronization approaches.

For fault tolerance issues, the DRC, which is decoupled from the federate, can survive from the federate failures and keep the connection to the federation. A crash-stop fault tolerance mechanism is proposed using checkpoint and message logging techniques. A basic state recovery protocol is first developed to recover the state of the crashed federate by rolling back its execution to the latest checkpoint and repeating execution identically to that before the failure. Then, an optimized protocol is further developed to accelerate the state recovery procedure. At the meantime, the Un-Identical replication structure is also extended to tolerate crash-stop failures occurred in federates, while keeping the advantages of performance enhancement.

Finally, a three-phases (i.e., failure detection, failure location, and failure recovery) Byzantine Fault Tolerance (BFT) mechanism is proposed. By combining the replication, checkpointing and message logging techniques, redundant executions of federate replicas are reduced to a low level. The BFT mechanism is implemented based on either Barrier or No-Barrier federate replication structures. Mechanisms to remove the epidemic effects of Byzantine failures also are proposed.

The mechanisms proposed in this thesis are implemented using *Service Oriented HLA RTI* (SOHR) with a decoupled federate architecture. Synthetic (e.g., pingpong and P-Hold) and realistic (e.g., borderless Fab) simulation models are used to evaluate these mechanisms in terms of performance enhancement and fault tolerance. In addition, their scalability and introduced overhead also are investigated.

Contents

Acknowledgments	i
Summary	ii
List of Figures	vii
List of Tables	ix
Index of Abbreviations	x
Index of Definitions	xiii
1 Introduction	1
1.1 Background	1
1.1.1 Parallel and Distributed Simulation	1
1.1.2 HLA-based Simulation	2
1.1.3 Decoupled Federate Architecture	8
1.1.4 SOHR	11
1.2 Motivations	14
1.3 Objectives And Scopes	16
1.3.1 Efficient HLA-based Simulation	16
1.3.2 Fault Tolerant HLA-based Simulation	18
1.3.3 Implementations and Evaluations	20
1.4 Contributions of the Thesis	21
1.5 Structure of the Thesis	22
2 Related Work	24
2.1 Efficient HLA-based Simulation	24
2.1.1 Migrating Federate Executions	24
2.1.2 Replicating Federate Executions	28

2.1.3	Improving Synchronization Efficiency	32
2.2	Fault Tolerant HLA-based Simulation	36
2.2.1	Fault Tolerance in Distributed Systems	36
2.2.2	Fault Tolerance in HLA Community	41
2.3	Summary	45
3	Federate Migration Based on Decoupled Federate Architecture	46
3.1	Migration Protocols	46
3.1.1	One-phase Migration	49
3.1.2	Two-phases Migration	50
3.1.3	Relay-based Migration	50
3.2	Experiments and Results	54
3.2.1	Experiment Design	54
3.2.2	Experiment Results	55
3.3	Summary	57
4	Federate Replication for Simulation Performance Enhancement	58
4.1	Un-Identical Federate Replication	59
4.1.1	Message Exchange Mechanism	62
4.1.2	TM Mechanism	66
4.1.3	Correctness Proofs	72
4.2	Experiments and Results	78
4.2.1	Configuration of Experiments	78
4.2.2	Performance Enhancement of Un-Identical Replication Structure .	80
4.2.3	Scalability of Un-Identical Replication Structure	82
4.3	Summary	87
5	Checkpoint-based Crash-stop Fault Tolerance Mechanism	88
5.1	Federate Fault Tolerance Mechanism	89
5.1.1	Basic State Recovery Protocol	92
5.1.2	Optimized State Recovery Protocol	94
5.2	Experiments and Results	100

5.2.1	Experiment Design	100
5.2.2	Experiment Results	101
5.3	Summary	104
6	Replication-based Crash-stop Fault Tolerance Mechanism	105
6.1	Extended Federate Replication Structure	106
6.1.1	Crash-stop Fault Tolerance Mechanism	106
6.1.2	Transfer State of CON Replica to OPT Replica	109
6.1.3	Transfer State of OPT Replica to CON Replica	110
6.2	Experiments and Results	111
6.2.1	Experiment Design	111
6.2.2	Experiment Results	112
6.3	Summary	117
7	Replication- and Checkpoint-based Byzantine Fault Tolerance Mechanism	118
7.1	Byzantine Fault Tolerance Mechanism	119
7.1.1	Three-Phases Mechanism	120
7.1.2	Remove Epidemic Effect of Byzantine Failure	124
7.2	Experiments and Results	131
7.2.1	Borderless Fab Simulation Model	131
7.2.2	Experiment Design	134
7.2.3	Experiment Results	135
7.3	Summary	143
8	Conclusions and Future Work	144
8.1	Conclusions	144
8.2	Future Work	148
	Appendix	151
A	Author's Publications	151
	References	154

List of Figures

1.1	An overview of an HLA federation	4
1.2	The interface between federates and the RTI	5
1.3	Federate architecture	8
1.4	SOHR architecture	12
1.5	Objectives of the thesis	20
2.1	No-Barrier federate replication	31
3.1	Federate migration on SOHR	48
3.2	One-phase migration protocol	49
3.3	Two-phases migration protocol	50
3.4	Relay-based migration protocol	51
3.5	DRC design in relay-based migration	52
3.6	Migration overheads of different migration protocols	56
4.1	Overview of Un-Identical replication structure	60
4.2	Message exchange mechanism	63
4.3	An example of message exchange mechanism	65
4.4	GALT calculation (TM Module)	67
4.5	Time advancement of federate replicas (Replication Module)	69
4.6	An example of TM mechanism	71
4.7	Execution times in different scenarios	81
4.8	Scalability of federate replication structure in separated DRC series . .	83
4.9	Scalability of federate replication structure in congested DRC series . .	85
4.10	Δ Time in separated and congested DRC series.	86

5.1	Checkpoint-based federate fault tolerance mechanism	90
5.2	Example of a crashed OPT federate	98
5.3	Execution times in different scenarios	101
5.4	Advantages of the optimized state recovery protocol	102
6.1	Replication-based federate fault tolerance mechanism	107
6.2	Performance of Replicated and Replicated & FT scenarios when $P_{ExternEvent}$ equals to extreme values	115
6.3	Performance of execution scenarios when $P_{ExternEvent}$ is variable	116
7.1	Overview of three-phases BFT mechanism	119
7.2	BFT mechanism using No-Barrier federate replication	124
7.3	Examples of epidemic effect removal mechanisms	128
7.4	External workload introduced on computing nodes	134
7.5	Execution times in BFT scenarios in No-Workload case	137
7.6	Execution times in BFT scenarios in Workload case	137
7.7	Overhead of BFT scenarios in No-Workload case	138
7.8	Overhead of BFT scenarios in Workload case	139
7.9	Scalability of different scenarios in No-Failure case	140
7.10	Scalability of BFT scenarios in the presence of Byzantine failrues	141
7.11	Overhead of BFT scenarios with respect to federation scale	142

List of Tables

3.1	Relay-based migration overhead versus state saving time	56
4.1	Additional variables and flags	62
4.2	Variable $P_{ExternEvent}$	82
5.1	Advantages of optimized protocol	97
5.2	Federate checkpoint	99
5.3	Repeated federate execution	99
6.1	Types of injected failures	112
6.2	Execution time (sec) when $P_{ExternEvent} = 50\%$	113
7.1	Execution time (sec) in No-Failure case	135

Index of Abbreviations

<i>ARML</i>	Active Replication Management Layer	10
<i>BFT</i>	Byzantine Fault Tolerance	19
<i>CCGS</i>	Committed Consistent Global States	42
<i>CI</i>	Conditional Information	34
<i>CMB</i>	Chandy-Misra-Bryant	6
<i>CON</i>	Conservative	6
<i>COPS</i>	Cocktail Party Simulation	42
<i>CrashedFed</i>	Crashed Federate	91
<i>CRC</i>	Central RTI Component	8
<i>DM</i>	Declaration Management	14
<i>DRC</i>	Decoupled RTI Component	9
<i>FDD</i>	Federation Object Model Document Data	48
<i>FEDEP</i>	Federation Development and Execution Process	4
<i>FM</i>	Federation Management	13
<i>FOM</i>	Federation Object Model	4
<i>FQR</i>	Flush Queue Request	7
<i>FT</i>	Fault Tolerance	10
<i>FT – RSS</i>	Fault Tolerant Resource Sharing System	41
<i>GALT</i>	Greatest Available Logical Time	6
<i>GT</i>	Globus Toolkit	12
<i>GVT</i>	Global Virtual Time	29
<i>HLA</i>	High Level Architecture	2
<i>IFSpec</i>	Interface Specification	3
<i>LA</i>	Lookahead	6

<i>LAN</i>	Local Area Network	2
<i>LFS</i>	Local Factory Service	13
<i>LGT</i>	Last Granted Time	6
<i>LIS</i>	Local Instance Service	13
<i>LRC</i>	Local RTI Component	9
<i>LRI</i>	Local Resource Instance	13
<i>LS</i>	Local Service	13
<i>MigFed</i>	Migrating Federate	46
<i>NMR</i>	Next Messages Request	7
<i>NMRA</i>	Next Messages Request Available	7
<i>OGSA</i>	Open Grid Services Architecture	12
<i>OM</i>	Object Management	14
<i>OMT</i>	Object Model Template	3
<i>OOM</i>	Object and Ownership Management	14
<i>OPT</i>	Optimistic	6
<i>RecoveredFed</i>	Recovered Federate	91
<i>ResFed</i>	Restarting Federate	46
<i>REST</i>	Representational State Transfer	14
<i>RO</i>	Receive Order	5
<i>RR</i>	Request Retraction	8
<i>RSS</i>	Resource Sharing System	41
<i>RTI</i>	Run-Time Infrastructure	4
<i>RVT</i>	Receive Virtual Time	6
<i>SOAP</i>	Simple Object Access Protocol	14
<i>SOHR</i>	Service Oriented HLA RTI	9
<i>SOM</i>	Simulation Object Model	4
<i>STM</i>	Smart Time Management	149
<i>SVT</i>	Send Virtual Time	6
<i>TAR</i>	Time Advance Request	7
<i>TARA</i>	Time Advance Request Available	7
<i>TM</i>	Time Management	5

<i>TMInfo</i>	TM Information	67
<i>TS</i>	Time Stamp	6
<i>TSO</i>	Time Stamp Order	5
<i>UI</i>	Unconditional Information	34
<i>WAN</i>	Wide Area Network	2
<i>WSDL</i>	Web Services Description Language	14
<i>WSPRC</i>	Web Services Provider RTI Component	9
<i>WSRF</i>	Web Services Resource Framework	12

Index of Key Definitions

<i>Application Specific State</i>	92
<i>Barrier Federate Replication Structure</i>	44
<i>Checkpoint Time</i>	91
<i>Committed Equivalent</i>	60
<i>Committed Events</i>	60
<i>Concurrent Failures</i>	39
<i>Consistent Checkpoint</i>	120
<i>External Event</i>	6
<i>Failure Time</i>	91
<i>Internal Event</i>	6
<i>Logical Time</i>	5
<i>No-Barrier Federate Replication Structure</i>	30
<i>Piecewise Deterministic</i>	51
<i>Piecewise Equivalent</i>	30
<i>Receive Virtual Time</i>	6
<i>Send Virtual Time</i>	6
<i>Simulation Engine State</i>	92
<i>Simulation Time</i>	5
<i>Simultaneous Failures</i>	39
<i>Software Diversity</i>	17
<i>Stable Checkpoints</i>	123
<i>StableLGT</i>	123
<i>Un-Identical Federate Replication Structure</i>	59

Chapter 1

Introduction

1.1 Background

1.1.1 Parallel and Distributed Simulation

Simulation is a process of designing a model which represents a physical or abstract system, and conducting experiments with the model for understanding the key characteristics or behaviors of the selected real-life or hypothetical scenarios. Computer simulation refers to the technique which designs and executes the model in a computation environment. It is a low-cost and safe alternative to solve complex problems. Nowadays, computer simulation has been widely used in various areas such as military, manufacture, business, science and education, to name only a few.

Parallel and distributed simulation facilitates the construction of a simulation with a group of logical processes. It allows the execution of complex simulation which cannot be executed on a single computing node, and reflects the structure of the modeled physical system which usually consists of several parallel components. In a parallel simulation, physical processes in the real system are modeled by the logical processes that are executed on a parallel computing infrastructure, such as a shared memory multiprocessors platform or a cluster with a high-speed internal connection. In a distributed simulation, various components of the simulation are executed on a computation environment with

heterogenous and geographically distributed computing resources connected either by *Local Area Network* (LAN) or by *Wide Area Network* (WAN) (e.g., the Grid [41, 87] or the Cloud [6, 96]). While the main concern of parallel simulation is execution speedup, distributed simulation encourages the collaboration among a great number of participants from different organizations and may involve huge computing resources and data sources from different corners of the earth.

1.1.2 HLA-based Simulation

The *High Level Architecture* (HLA) IEEE 1516-2000 [56] is a general purpose software architecture for creating a distributed simulation out of multiple simulation components. It provides a general framework within which simulation developers can describe their simulation models and collaborate with each other seamlessly.

1.1.2.1 HLA and RTI

Many complex simulations can be viewed as compositions of multiple individual simulation components which were developed previously for other purposes. It is attractive to build a new simulation by combining a group of the existing simulation components. However, the simulation models developed in ad-hoc manner may need to be modified extensively before integrating into a new simulation. The HLA is proposed to support the reusability and interoperability of the simulation components. Reusability means simulation components can be reused in different simulation applications and interoperability means that the reusable components can collaborate with each other without re-coding. In HLA terminology, the simulation components are denoted as federates, and the HLA-based simulation which consists of a number of federates is called a federation.

The HLA standard consists of four major components:

- (i) The HLA Framework and Rules [56] define the responsibilities of federate and federation.

- (ii) The HLA *Interface Specification* (IFSpec) [57] identifies how federates can interact with the federation and ultimately with one another. The IFSpec includes six HLA management service groups:

Federation Management controls federation-wide activities, such as creation and destruction of a federation execution, and federation-wide state saving and restoration. Federates are allowed to join in and resign from the federation execution dynamically.

Object Management manages the lifecycle of object instances, including registration and deletion of object instances. It is also responsible for exchanging messages, e.g., updating/reflecting attribute values or sending/receiving interactions, among federates.

Declaration Management controls the message exchanges among federates with a class-based publication and subscription mechanism. Federates are allowed to publish/subscribe or unpublish/unsubscribe object class attributes or interaction classes during the federation execution.

Ownership Management supports cooperative modeling by allowing attribute ownership of object instances to be transferred among federates in a federation.

Time Management coordinates time advancement of each federate, for the purpose of preserving causal dependency and event ordering.

Data Distribution Management provides a flexible and extensive mechanism for further limiting publication and subscription interests, for the purpose of reducing communication traffic over the network.

- (iii) *Object Model Template* (OMT) [58] prescribes the format and syntax for HLA object model and fosters interoperability and reusability of simulations and their

components. Each federate should have a *Simulation Object Model* (SOM), which presents objects and interactions that can be exposed or consumed by the federate. The whole federation should have a common *Federation Object Model* (FOM) to introduce all shared information among federates.

- (iv) *Federation Development and Execution Process* (FEDEP) [59] defines a standardized and recommended process for developing and executing interoperable HLA-based federations.

Figure 1.1: An overview of an HLA federation

An overview of a federation is shown in Figure 1.1. Besides computer simulations, federates in the federation can be human operators/players, embedded hardware and passive viewers for data collation, analysis, and display. *Run-Time Infrastructure* (RTI) is a communication middleware that provides the implementation of the HLA IFSpec. Federates in the federation communicate with each other through the underlying RTI.

The interface between federates and the RTI usually is implemented via ambassadors as shown in Figure 1.2. In addition to the internal functionality of the simulation component, the federate code invokes RTI services by calling the corresponding functions of the *RTI Ambassador*. It also contains the actual implementations of the callback functions of the *Federate Ambassador*. When the federate joins a federation, it provides the RTI

Figure 1.2: The interface between federates and the RTI

with a reference of the Federate Ambassador so that the RTI can deliver callbacks to it via invoking the corresponding functions of the Federate Ambassador. For instance, a federate sends messages through invoking the corresponding RTI services, e.g., *Update Attribute Values* and *Send Interaction*; the RTI delivers messages to federates in the form of callbacks, e.g., *Reflect Attribute Values* and *Receive Interaction*.

1.1.2.2 Time Management in HLA

A group of *Time Management* (TM) services are defined in the HLA to control the time advancement of each joined federate along the federation time axis [45, 46, 57]. Federates can be divided into four categories based on their *time-regulating* and *time-constrained* status. Among the federates, *Receive Order* (RO) messages and *Time Stamp Order* (TSO) messages are exchanged. RO messages can be delivered to federates in an arbitrary order; whereas the TSO messages can only be delivered to federates under the control of TM mechanism. Only time-regulating federates can send TSO messages and time-constrained federates receive TSO messages. For clarity of illustration, the federates discussed in this thesis are both time regulating and constrained, among which only TSO messages are exchanged.

Two values, i.e., *logical time* and *simulation time*, are used to describe federate execution progress. The logical time of a federate is updated only if its time advancement

request has been granted by the RTI. Hence, the logical time is also referred to as the *Last Granted Time* (LGT). The simulation time of a federate equals to the *Time Stamp* (TS) of the event which is being processed currently. Every message exchanged between federates is attached with both *Send Virtual Time* (SVT) and *Receive Virtual Time* (RVT). The SVT equals to the simulation time of the sending federate when the message is sent, while the RVT indicates when the message must be processed by the receiving federate. On receiving the message, the federate generates an external event with $TS = RVT$ of the message. Correspondingly, the events scheduled by the federate itself are referred to as internal events. According to the meaning of *lookahead* (LA)¹ [43], we can easily get the following:

$$RVT \geq SVT + LA \quad (1.1)$$

In HLA/RTI, federates need to guarantee that they do not send any message with $RVT < LGT + LA$ at any point in the future. To ensure that federates never receive any message with $RVT < LGT$, the RTI prevents a federate from advancing its logical time beyond its *Greatest Available Logical Time* (GALT), which expresses the lower bound of RVT of any future message destined for the federate.

Both conservative (e.g., Chandy-Misra-Bryant (CMB) protocol [15, 20]) and optimistic (e.g., Time Warp protocol [60]) synchronization approaches are supported in the HLA standard and the federates using conservative or optimistic time management mechanism are called conservative (CON) or optimistic (OPT) federates accordingly. Conservative approaches preserve the causality constraint by ensuring that each federate processes events in strictly non-decreasing TS order. A CON federate can only execute those events with $TS \leq$ logical time (i.e., LGT). Hence, the simulation time is smaller than or equal to the logical time. Both of them are monotonic increasing. In contrast,

¹Lookahead refers to the ability to predict the simulated future. If a federate at simulation time T can predict all events it will schedule have $TS \geq T + L$, the federate is said to have lookahead L .

optimistic approaches allow free processing of events and thus potentially have better exploitation of parallelism among federates. However, an OPT federate needs to rollback its execution when a causality error happens (e.g., receiving a straggler message whose RVT is smaller than the simulation time of the federate). In other words, the simulation time of an OPT federate can be greater than its logical time and may decrease in an execution rollback. To support the execution rollbacks, state saving and rollback mechanisms are required in the OPT federate. Since the federate never receives any message with RVT smaller than its logical time, the logical time is the lower bound of future execution rollbacks, and thus can be used as the upper bound for the fossil collection performed on the saved state.

The messages destined for a federate are buffered in the RTI. They are delivered to the federate according to its time advancement and employed synchronization approach. The *Next Messages Request* (NMR) service with parameter t can be used by an event driven CON federate, where t is typically the TS of the earliest event in the eventlist. The time advancement request is granted by a *Time Advance Grant* (TAG) callback only if $\min(t, \text{minimum RVT of the messages buffered in the RTI}) < \text{GALT of the federate}$. The federate is granted to time t and no message is delivered, if $t < \text{minimum RVT of the buffered messages}$. Otherwise, the federate is granted to this minimum RVT, and the buffered messages with RVT equals to the minimum RVT are delivered to the federate. Similar to NMR, the *Next Messages Request Available* (NMRA) can be used by an event driven CON federate with zero LA. The *Time Advance Request* (TAR) and *Time Advance Request Available* (TARA) are used by a time stepped CON federate.

The *Flush Queue Request* (FQR) service with parameter t is used by an OPT federate. It forces the RTI to deliver all the messages buffered in the RTI to the OPT federate. In this case, time advancement request can always be granted immediately, without waiting for other federates to advance. The granted time is the smallest value of t ,

GALT of the federate and the minimum RVT of the messages buffered in the RTI. Upon an execution rollback, an OPT federate may invoke the *Retract* service to unsend the incorrect messages which were sent previously. SVT (or RVT) of the retractions are equal to SVT (or RVT) of their corresponding incorrect messages. If the messages have already been delivered to the receiving federates, the RTI informs those federates to remove the effect of the incorrect messages through *Request Retraction* (RR) callbacks. This may cause secondary execution rollbacks in the receiving federates.

1.1.3 Decoupled Federate Architecture

Figure 1.3: Federate architecture

Figure 1.3 shows traditional and decoupled federate architectures. In these architectures, the roles of the *Central RTI Component* (CRC) and federates are the same. The CRC manages the entire federation. It keeps the global RTI state (e.g, federation name and federate handles) and handles the centralized part of the computation of the HLA management services (e.g., time management and data distribution management services). Each federate performs the internal functionality of the corresponding simulation component by keeping its local state and carrying out its local computation.

An RTI component, which includes RTI ambassador, usually is provided by the RTI developers to connect a joined federate to the federation. It keeps part of the RTI state, which concerns with the federate. For instance, federate's logical time and its publishing/subscribing interaction classes. It also deals with the distributed part of the computation of the HLA management services, and takes care of the federate's need to exchange messages with other federates and to request time advancement along federation's time axis. In summary, the federate communicates with the outside world through the RTI component. It works as a communication agent of the federate in the federation execution.

A *traditional federate architecture* is shown in Figure 1.3.a. The RTI component mentioned above is referred to as *Local RTI Component* (LRC), as it is tightly coupled with the federate. The federate and its LRC are executed on the same computing node or more precisely in the same process. The LRC is a library provided by RTI developers. The federate is compiled and linked with its LRC to form an execution process. Each federate has one and only one tightly coupled LRC. They communicate with each other through function calls. Traditional federate architectures are implemented in most RTI implementations, such as RTI NG [33], FDK [49] and pRTI [92].

A *decoupled federate architecture* is shown in Figure 1.3.b. The RTI component mentioned above is referred to as *Decoupled RTI Component* (DRC), as it is decoupled from the federate. The federate and its DRC are executed in different processes, or even on different computing nodes. They communicate with each other through an external communication channel, such as Inter-Process Communication, Shared memory, Socket, or Web/Grid Services invocations. In some more recent RTI implementations, decoupled federate architectures are implemented. Möller [79] has proposed a *Web Services Provider RTI Component* (WSPRC) to provide RTI services to the corresponding federate using Web Services API. Pan et al. [87] have proposed a *Service Oriented HLA RTI* (SOHR)

framework, in which the DRCs of federates are implemented by Grid services providing the RTI services using Grid Services API.

A decoupled federate architecture can also be implemented by breaking the tight coupling between the federate and its LRC in a traditional federate architecture. Chen et al. [22] have developed a customized RTI++ library, a middleware between federate and its LRC, to detect RTI failures and restored RTI state without interrupting the executions of federates. An *Active Replication Management Layer* (ARML), a message proxy layer between federate and its LRC, was developed by Quaglia [95] to support federate replication for performance enhancement purpose. The *Fault Tolerance* (FT) manager developed by Berchtold and Hezel [11] connects the replicas of the same federate to the same LRC. It is also responsible for detecting the failures of federate replicas and starting new replicas if necessary. A Federate-Proxy-RTI based framework has been proposed by Xie et al. [114]. A RTI service invoked by the federate is translated into a Grid services invocation of the proxy. To finish the RTI service, the proxy invokes the same RTI service through the LRC to the RTI.

Compared with a traditional federate architecture, a decoupled federate architecture may introduce more overhead (e.g., message packing and unpacking overhead, as well as message delivery overhead on the external communication channel). However, the overhead can be well controlled if some optimized protocols are used in particular implementations [24].

Federate and its corresponding RTI component play different roles in the federation execution. By breaking the tight coupling between them, the decoupled federate architecture has the following advantages:

- (i) The federate and its DRC are executed in different processes.
- It is easy to distinguish the federate state from the RTI state kept in the DRC,

as they belong to independent memory spaces; and

- It is possible to isolate the failures occurred in the federate and those occurred in its DRC, and handle them separately using different mechanisms.

(ii) The execution of the federate and its DRC can be managed separately.

- It is able to migrate a federate and connect the migrating and restarting federates to the federation through the same DRC;
- It is able to replicate federate executions and connect federate replicas to the federation through a single DRC; and
- It is able to terminate and restart a federate in the runtime, since the DRC keeps the connection to the federation for the federate.

(iii) A middleware can be inserted between the federate and its DRC.

- The middleware can intercept the RTI services and callbacks of the federate and controls their deliveries if necessary;
- The middleware can detect the failure of the federate and its DRC by monitoring their hearbeats and outputs; and
- The middleware can provide a stable storage to save checkpoint and message loggings of the federate and its DRC.

1.1.4 SOHR

Traditionally, federates interoperate with each other through a vendor-specific RTI software, such as the DMSO RTI NG [33]. Resource arrangement and security measures must be carried out before the actual execution of a distributed simulation application.

RTI Management Services implement the centralized part of the computation of the corresponding six service groups defined by the HLA interface specification [57].

Federate communicates with the federation and ultimately with other federates through the *Local Service* (LS). The design of LS follows the WS-Resource factory/instance design pattern [103]. The *Local Factory Service* (LFS) creates a *Local Resource Instance* (LRI) when a federate joins a federation and destroys the LRI when the federate resigns from the federation. The LRI is used to keep the RTI state concerned with the federate. The *Local Instance Service* (LIS) provides RTI services to the federate by accessing the LRI state and communicating with the HLA RTI Management Services and LISes of other federates. In summary, the federate only deals with the local computation; whereas the LIS/LRI (composition of LIS and the corresponding LRI) works as the communication agent of the federate in the federation. In terms of decoupled federate architecture, LIS/LRI plays the role of DRC. The federate and its LIS/LRI (i.e., DRC) communicate with each other through Grid services invocations. As shown in Figure 1.4, federates which use Grid services RTI interface can invoke the RTI services provided by the LIS/LRI directly. To support a legacy federate using a traditional RTI interface (e.g., C++ and Java API), a light weight library called Local RTI Component (LRC) is linked to the federate. This LRC is different from the LRC in the traditional federate architecture. It is just simply responsible for translating the RTI services initiated by the federate into Grid services invocations of the LIS/LRI; obtaining callbacks from the LIS/LRI by invoking the *getcallbacks()* service; and delivering the callbacks to the federate.

The LIS/LRI is designed in modular manner. It contains a set of modules corresponding to HLA management service groups. For instance, the *Federation Management (FM) module* is responsible for joining/resigning the federate in/from the federation; the *Time Management (TM) module* is responsible for performing federation wide time synchro-

nization; the *Object Management (OM) module*² is responsible for exchanging messages in the federation; and the *Declaration Management (DM) module* is responsible for publishing/subscribing object attributes and interaction classes. Besides these modules, the *Callback Module* is also developed to buffer callbacks destined for the federate. When the LIS/LRI receives a message from LIS/LRIs of other federates, a callback will be generated and saved in the corresponding message queue depending on the message type.

The performance of SOHR in both LAN and WAN environments is measured in [87]. The large latency in SOHR is mainly caused by the overheads of Grid service invocations. Fortunately, persistent TCP/IP connection is supported in GT4.1.1. Hence, multiple service invocations between the same client and server can share a single TCP/IP connection, so the overhead of the connection setup and tearing down is amortized among multiple service invocations. However the Grid service invocations, as discussed in [89], are still too heavy due to the use of old standards, i.e., *Simple Object Access Protocol* (SOAP) [110] and *Web Services Description Language* (WSDL) [111]. We could expect much better performance if SOHR were implemented based on Representational State Transfer (REST) [5, 89], which was proposed as the simpler alternative to SOAP- and WSDL-based Web/Grid services.

1.2 Motivations

Large scale simulations are usually developed to study complex systems, e.g., supply chains, computer networks, and ecological systems. Their executions may involve a large number of computationally intensive federates, and thus, are time and resource consuming. Moreover, a number of simulation executions are required to obtain trusted simulation results. Thus, accelerating the executions is important to make simulation

²The object and ownership management modules are combined as OOM module in [87]

economical and attractive to both industry and research communities. Speedup of simulation executions is especially important for symbiotic simulations [8, 42], in which real time responses are required.

The HLA standard IEEE 1516-2000 [56] provides a method to create a federation execution by connecting a group of federates. However, it is not concerned with the performance of the federation execution. Therefore, performance enhancement mechanisms are required to accelerate HLA-based simulation executions.

Federates in HLA-based simulation are subject to failures due to various reasons, because they may be developed by different participants and executed on heterogeneous and geographically distributed platforms. Crash-stop failures might happen because of the failure of power supply and the collapses of federate process, operating system, and hardware. Byzantine failures might happen because of the bugs in the complex hardware and software. In addition, some participants might play as traitors or adversaries and plot malicious attacks. A crash-stop failure of a single federate may interrupt the entire federation execution. A Byzantine failure may propagate in the federation execution and finally result in wrong simulation results. Consequently, the risk of federation failure increases with the federation scale.

The HLA standard IEEE 1516-2000 [56] does not provide a formal fault tolerance model. Hence, the failed federation execution may need to be restarted from the beginning. As a result, the simulation execution time increases significantly and a large amount of resources are wasted in the repeated federation execution. For some symbiotic simulations, re-execution due to a failure is unacceptable, as the outputs of the re-executed simulation are probably out of date and invalid. In addition, fault tolerance is important especially for simulation applications in safety critical domains, e.g., military mission rehearsal and airport traffic control. As the next version of the HLA standard, IEEE 1516-2010 [55] which is commonly referred to as the HLA Evolved [81]

provides interfaces to implement fault tolerance mechanisms. However, the actual mechanisms, e.g., data recovery, are absent. Fault tolerance mechanisms are indispensable to guarantee successful and correct federation execution in the presence of federate failures.

1.3 Objectives And Scopes

Motivated by the problems identified in the last section, the objective of this thesis is to develop mechanisms to support efficient and fault tolerant HLA-based simulation by exploiting the advantages of the decoupled federate architecture. The efficiency issue will be addressed by federate migration and federate replication, and both crash-stop failure and Byzantine failure will be investigated.

1.3.1 Efficient HLA-based Simulation

1.3.1.1 Federate Migration

In HLA-based simulation, the slowest federate may hold back the execution of all other federates in the federation. Therefore, the workload imbalance among federates' underlying execution platforms generally leads to poor execution performance. It is beneficial to migrate federates from heavily-loaded computing nodes to lightly-loaded ones. Federate migration can also optimize the communication traffic and reduce the transmission latency. Commonly, it is beneficial to migrate logically interrelated federates, which communicate with each other frequently, to geographically close computing nodes with high speed network connections. Federates also need to be migrated if their required execution resources are not available on their underlying computing nodes. In addition, federate fault tolerance can also be supported by migrating a federate from a faulty computing node to a non-faulty one. Currently, the HLA framework does not provide any formal

support for federate migration.

1.3.1.2 Federate Replication

Nowadays, the computation resources can be obtained more and more easily because of the rapid development in computer industry and availability of new technologies, especially Grid [41] and Cloud [6] computing technologies. The active replication technique inspires a direction to enhance simulation performance [3, 95], though more resources are consumed due to the redundant computation of multiple replicas of the same federate. The replicas of the same federate are likely to exhibit different performances if (i) they are executed on heterogenous or geographically distributed platforms; and/or (ii) they are developed in a software diversity manner [34, 75]. Software diversity, also known as N-Version Programming [74], is a method used to improve software performance and reliability by developing multiple functionally equivalent programs independently following the same initial specifications.

To ensure that events in the federation are executed in TS order, federates in the federation must be synchronized in their time advancements. The synchronization overhead is substantial, especially for large scale distributed simulations. As observed in [39, 84], conservative and optimistic synchronization approaches are likely to perform differently for the same federation execution.

Inspired by the concept of active replication and the diversity of synchronization approaches, we propose to enhance simulation performance by concurrently running multiple replicas with different synchronization approaches for the same federate. These replicas are likely to exhibit different runtime performances. Therefore, the simulation performance can be enhanced by always choosing the fastest replica to represent the behavior of the federate in the federation.

To support the concurrent execution of federate replicas, there are some challenges

that need to be tackled. Simply replicating federates in a federation execution will increase the federation scale, and thus increase the overhead of federation wide time synchronization as well as the risk of federation execution failure. What is more, federate replicas with different synchronization approaches employed may behave differently, e.g., invoke different RTI services for time advancement requests and delivery different messages following different sequences. Hence, the proposed replication structure should not increase federation scale and should also handle the different behaviors of federate replicas and coordinate their time advancement.

1.3.2 Fault Tolerant HLA-based Simulation

1.3.2.1 Crash-stop Failures

It is challenging to ensure the continuity and correctness of federation execution in the presence of crash-stop failures. When a federate is crashed, it is disconnected from the federation. Other federates in the federation cannot exchange messages with the crashed federate. Their time advancement requests are also held back. Simply resigning the crashed federate from the federation is not enough, because wrong simulation results may be produced without the participation of the crashed federate in the remaining federation execution.

To ensure correct federate execution, a new federate should be established to replace the crashed federate in the federation execution. It must join the federation and build up connections to other federates. Before participating in the federation execution, the federate must restore its execution state which was lost during the failure. What is more, some messages exchanged among federates may also be lost due to the failure. Hence, a mechanism is needed to ensure the correctness of message delivery in the presence of federate failures. It is even more challenging but attractive to tolerate federate failures

transparently and efficiently. That is, recovery of a crashed federate should not affect the execution of other federates in the federation and the overhead of failure recovery should be minimized.

Initially, the lost federate state and messages are recovered relying on the checkpoint and message loggings which were taken by the federate before the failure. With federate replication, the federation execution will not be interrupted if there is at least one replica of each federate that can survive the failures. However, it might suffer from performance degradation and might fail to tolerate future failures due to the decreased number of working replicas. Therefore, a new replica should be established to substitute the crashed one. To avoid repeated computation from the beginning, the new replica is initialized using the state of one of the working replicas. Since different synchronization approaches are used, the replicas of the same federate may have different execution states. One of the research issues is how to initialize the state of the new replica using the state of a working replica which may employ a different synchronization approach.

1.3.2.2 Byzantine Failures

When a federate encounters a Byzantine failure, it might modify its local state arbitrarily and invoke arbitrary RTI services. As a result, wrong simulation results will probably be produced. The existing *Byzantine Fault Tolerance* (BFT) mechanisms [77] can be adopted to tolerate Byzantine failures of federates. Federates are replicated in multiple locations. Byzantine failures can be detected and located by comparing the outputs of the replicas of the same federate. However, to reduce the high cost of the existing BFT mechanisms, redundant executions of federate replicas should be reduced. Moreover, Byzantine failures propagate quickly due to the frequent communication among federates in the federation. An epidemic effect might happen if the BFT mechanism fails to isolate the Byzantine failures before their propagation. Such an epidemic effect, if it happens,

must be removed to guarantee the correctness of federation execution.

1.3.3 Implementations and Evaluations

Figure 1.5: Objectives of the thesis

To achieve the objectives of this thesis, additional mechanisms need to be provided in HLA-based simulation. In the decoupled federate architecture, a federate and its DRC are executed in different processes. Hence, it is possible to deploy a middleware between them, as shown in Figure 1.5, to intercept and log the messages (i.e., RTI service calls and callbacks) and checkpoints of the federate. Based on the checkpoint and message logs, performance enhancement and fault tolerance mechanisms are provided by implementing the corresponding functional modules in the middleware.

Since the DRC is decoupled from the federate, it can survive from the federate failures and keep the connection to the federation. In this thesis, we assume that the DRC is robust, i.e., it does not suffer from crash-stop or Byzantine failure. The federate migration and replication modules deployed for the performance enhancement purpose can be reused for the fault tolerance purpose. For example, fault tolerance can be supported by

migrating a federate from a faulty computing node to a non-faulty one.

The mechanisms proposed in this thesis can be integrated with the SOHR [86, 87] by either (i) implementing the corresponding functional modules as individual processes between the federate and DRC; or (ii) embedding the corresponding functional modules in the DRC. The former does not require any modification of the DRC. However, the latter is chosen in this thesis for efficiency consideration. As mentioned above, the DRC in SOHR is designed in a modular manner. The additional modules can cooperate with the existing modules in the DRC seamlessly. They are able to share some functions and data structures without introducing extra communication overhead.

To evaluate the proposed approaches/mechanisms in this thesis, synthetic (e.g., ping-pong and P-Hold) and realistic (e.g., borderless Fab) simulation models are developed. The experiments using these simulation models are carried out to demonstrate that our approaches/mechanisms can enhance simulation performance and ensure correct federation execution in the presence of crash-stop and Byzantine failures. At the meantime, the overhead and the scalability of the mechanisms are also investigated.

1.4 Contributions of the Thesis

The contributions of the thesis include:

- (i) Efficient federate migration protocols are proposed based on the decoupled federate architecture;
- (ii) An Un-Identical replication structure is developed based on the decoupled federate architecture to support concurrent executions of conservative and optimistic replicas of the same federate;

- (iii) State recovery protocols are proposed to tolerate crash-stop failures of federates in both non-replication and replication structures; and
- (iv) A three-phases mechanism is developed to tolerate Byzantine failures of federates in both Barrier and No-Barrier federate replications.

1.5 Structure of the Thesis

The thesis is organized into eight chapters.

Chapter 2 introduces the related work. For the performance enhancement issue, three research directions, i.e., balancing workload, improving synchronization efficiency, and replicating federate executions, are introduced. For the fault tolerance issue, failure categories and fault tolerance mechanisms in distributed systems are discussed first. Then, the research effort on fault tolerance issue in the HLA community is introduced. The existing Barrier and No-Barrier federate replication structures using the concept of the decoupled federate architecture are also described in detail.

Chapter 3 proposes one-phase, two-phases and relay-based federate migration protocols. The experiments that compare the performance of these protocols are also included.

Chapter 4 proposes an Un-Identical replication structure to support concurrent executions of CON and OPT replicas of the same federate. Correctness proof is provided. Experimental results are reported to demonstrate the benefit of the Un-Identical replication in terms of performance enhancement and to study its scalability.

Chapter 5 proposes a crash-stop fault tolerant mechanism using the checkpoint and message logging techniques. A basic and an optimized state recovery protocols are proposed to recover the state of the crashed federate. Experiments are carried out to verify the correct failure recovery and to demonstrate the advantages of the optimized protocol.

Chapter 6 extends the Un-Identical replication for crash-stop fault tolerance purposes.

Experimental results are reported to demonstrate that the extended replication structure can provide fault tolerance while maintaining performance advantages.

Chapter 7 first proposes a three-phases BFT mechanism using either Barrier or No-Barrier federate replication structure. Then, epidemic effect removal mechanisms are proposed in case of Byzantine failure propagation in the federation execution. Finally, experiments are carried out to verify the correctness of the BFT mechanism; demonstrate the performance advantages of the BFT mechanism using No-Barrier replication; and investigate the scalability of the BFT mechanism.

Finally, Chapter 8 concludes the thesis and discusses the future work.

Chapter 2

Related Work

In this chapter, we introduce the related work on the issues of performance enhancement and fault tolerance for HLA-based simulation.

2.1 Efficient HLA-based Simulation

Execution performance of HLA-based simulation depends on various factors [10, 50]. The research work on enhancing execution performance of HLA-based simulation can be classified into three directions: migrating federate executions, replicating federate executions, and improving synchronization efficiency.

2.1.1 Migrating Federate Executions

Workload imbalance may lead to poor simulation performance, because the slowest federate may hold back the execution of all other federates in the federation execution. Some frameworks [4, 13, 16, 72, 105] were proposed to balance federates among available computing resources. To achieve workload balance, federates need to be migrated from heavily-loaded computing nodes to lightly-loaded ones. In addition, federates may also

need to be migrated to reduce the communication load and latency [32].

Federate migration means transferring a federate from one computing node to another during execution. It can be realized at different levels. At the operating system level, a federate can be migrated transparently by moving the federate process to the destination computing node. However it introduces considerable overhead, especially when the source and destination computing nodes are heterogeneous. In order to achieve heterogeneity, process state needs to be saved in a machine-independent representation [78]. At the application level, a federate can be restarted on a different computing node much more easily. The amount of saved state can be much less than that at the operating system level. However, the programmer or compiler has the responsibility to generate state saving and restoring code. This thesis focuses on application level federate migration, assuming that state saving and restoring functions are provided by migration-enabled federates. A federate at the source computing node saves and transfers its state, and then it restores its state and resumes execution at the destination computing node.

However, federate migration is much more difficult than normal application migration. In a traditional federate architecture, the federate and its LRC are tightly coupled to form an execution process. Thus, they need to be migrated together to the destination computing node. The RTI state kept in LRC (e.g., information of declaration, ownership, time and data distribution management services) needs to be saved and restored during the migration. Moreover, other federates in the federation should be informed about the migration. There are mainly two types of migration approaches proposed in the literature, i.e., Federation-wide Save and Restore and “*Freeze free*” migration approaches. For simplicity, this thesis uses the abbreviation “**MigFed**” to denote the migrating federate at the source computing node, and “**ResFed**” to denote the restarting federate at the destination computing node.

2.1.1.1 Federation-wide Save and Restore

Zajac et al. [118] have proposed a straight-forward approach to migrate federates by utilizing the HLA Federation Management API [57] to save and restore the RTI internal state, without considering the RTI implementation detail. *RequestFederationSave()* is called to freeze the federation execution. Then, federate state is encapsulated and sent to the destination computing node. Subsequently, **MigFed** resigns from the federation, and **ResFed** joins the federation. Finally, **ResFed** restores its state and invokes *RequestFederationRestore()* to restore the RTI internal state and resume the federation execution. This approach does not require the **ResFed** to rebuild the RTI state explicitly. Message loss and duplication problems, as described in [17], are also avoided. However, the migration overhead is significant as federation execution is frozen during the migration process. Operations in the above migration process are time-consuming in some RTI implementations such as the DMSO RTI NG [33]. For instance, to join or resign from a federation execution, a federate needs to establish or destroy TCP/IP connections to all other federates in the federation. Moreover, federation wide synchronization is required and the complete RTI internal state needs to be saved and restored during the migration procedure. As mentioned in [118], the migration overhead increases linearly with respect to the federation scale.

2.1.1.2 “Freeze free” Migration

Yuan et al. [117] and Tan et al. [106] have proposed “Freeze free” federate migration approaches. In their approaches, a federate is migrated without affecting other federates in the federation. Instead of saving and restoring the complete RTI internal state, their approaches only handle the RTI state concerned with the migrating federate. **MigFed** and **ResFed** modify the RTI state via HLA API invocations. For instance, **MigFed** performs

unpublishing and unsubscription requests and resigns from the federation; whereas **ResFed** joins the federation and performs publishing and subscription requests. As mentioned above, joining and resigning from a federation execution are time-consuming operations. Similarly, performing (un)publishing and (un)subscription are also time-consuming, as all concerned federates need to be informed. Approaches have been developed to reduce the overhead of these time-consuming operations. For instance, **MigFed** is not suspended immediately after being informed about migration. It keeps running while **ResFed** is joining the federation and performing publishing and subscription requests. Thus, these operations performed by **ResFed** are overlapped by continuous execution of **MigFed**. After these operations are completed, **ResFed** requests the federate state from **MigFed**. **MigFed** unpublishes and unsubscribes before resigning from the federation. Finally, **ResFed** restores the received state and resumes execution.

“Freeze free” approaches can improve migration performance. But some limitations are applied on migration-enabled federates. They need to be constructed based on a federate template [106] or an execution model [117]. Federates are not allowed to publish and subscribe after entering the simulation main loop. Both approaches in [106, 117] only consider the declaration management and object management concerning interactions. The situation becomes more complex if object-classes and other management services are considered. Tan et al. [107] have developed an approach to migrate federates with ownership management. Before entering the simulation main loop, **ResFed** has to obtain all ownerships of **MigFed**. Ownership transfers cannot be overlapped by **MigFed** execution, because only one federate can have the ownership of an individual attribute of each object instance at any given time. Hence, it is inefficient if many ownership transfers are required during the migration.

The other problems of “Freeze free” migration are message duplication and loss. To solve these problems, a counter-based solution is proposed in [17]. **MigFed** will continue

execution even after the federate state is saved and sent. It still receives messages and transfers them to **ResFed** if there is a potential risk of message loss. However, the solution relies on a complex protocol and requires the maintenance of a large number of counters.

In summary, federate migration is a complex process, since the federate and its LRC must be migrated as a whole unit due to the tight-coupling between them in traditional federate architectures. Federation-wide Save and Restore migration avoids handling RTI state explicitly. However federation wide synchronization introduces significant overhead during the migration. The “Freeze free” migration can reduce overhead, but it requires the migrating and restarting federates to modify RTI state via some HLA API invocations explicitly. It also requires a complex approach to solve the problems of message duplication and loss. Differently, the migration protocols proposed in this thesis exploit the advantages of the decoupled federate architecture. According to our knowledge, there is not any prior research in this direction.

2.1.2 Replicating Federate Executions

In the context of simulation, a simulation model may need to be executed multiple times in order to generate statistically trustable outputs. To evaluate different what-if scenarios, execution of a simulation model may also need to be replicated, each with a different set of input parameters. As an alternative to the execution of a linear sequence of the replications, CR-PADS proposed by Bononi et al. [12] enhances the execution performance and resource utilization via interleaving the computation and communication phases of the replications. Ewald et al. [37] have also proposed an efficient and adaptive approach for parallel simulation replication that combines portfolio theory with reinforcement learning. A simulation model may be also executed with different scenarios due to different decision policies (i.e., different choices at decision points). Distributed

simulation cloning technology [23, 54] enables dynamic creation of simulation replication for alternative scenarios of a distributed simulation within the same simulation execution session. Particularly, the incremental cloning [23] can improve the performance by avoiding some repeated computation amongst independent scenarios.

Differently, federate replication discussed in this thesis refers to the development of multiple replicas of the same federate with a single scenario (i.e., the same input parameters and decision policy). Since replicas of the same federate are executed concurrently, the federate replication discussed here is active replication (to be defined in Subsection 2.2.1.1). Active replication of federates discussed in this subsection is used for the purpose of improving performance of federation execution. However, it can also be used for fault tolerance purposes (to be discussed in Subsection 2.2.2.2).

Agrawal and Agre [3] have introduced the concept of replicated objects in Time Warp-based, distributed discrete event simulation systems. Frequently used objects are replicated in multiple sites to reduce the communication traffic and delays. The event messages are classified into two categories: read messages (state observing events) and write messages (state modifying events). Read messages are only delivered to the local or closest replica of the object denoted as the primary replica; whereas write messages are delivered to all replicas of the object to maintain the consistency of all replicas. Replicas of the same object are allowed to receive and process different events (either correct or incorrect events) in different orders. The state saving and rollback mechanisms of Time Warp ensure that the effect of incorrect events are removed and the same correct events are processed in the replicas in TS order eventually. The simulation performance can be enhanced significantly if the number of read messages is much greater than that of write messages. However, the replicated objects are treated as individual objects in the simulation execution. As a result, the simulation scale increases, and thus the overhead of *Global Virtual Time* (GVT) calculations increases accordingly. This problem may also

happen in replicating federates under the traditional federate architecture. A federate and its LRC must be replicated as an unit due to the tight-coupling between them, and each replica is treated as an individual federate in the federation.

Another federate replication structure has been proposed by Quaglia [93, 94, 95], using the concept of the decoupled federate architecture implicitly. The tight-coupling between federate and its LRC in the traditional federate architecture is removed by inserting an *Active Replication Management Layer* (ARML) between them. Multiple replicas are developed for the same federate and connected through ARML to one RTI component (i.e., the equivalence of the DRC in the decoupled federate architecture). Since only one DRC is involved in the simulation management for the replicas of one federate, the federation scale does not increase because of the federate replication. The ARML is a kind of message proxy layer between federate replicas and the DRC. For an RTI service invoked by a federate, the ARML only sends the DRC the first instance of the RTI service invoked by the replicas of the federate; for a callback destined to a federate, the ARML distributes it to all replicas of the federate in the correct order and buffer the callback if necessary. The performance of a federation is improved, because the simulation time of a federate is always advanced by the fastest running replica.

In this thesis, we shall classify the federate replication structure proposed in [93, 94, 95] as *No-Barrier federate replication structure*. In such a structure, replicas of a federate connects to the same DRC through a replication module (e.g., ARML) and they are required to be *Piecewise Equivalent*. That is, replicas with the same initial state and the same incoming callbacks must process the same events, update to the same federate state, and invoke the same RTI services. They are allowed to have different runtime performances. The federation execution performance is improved by forwarding the first instance of the same RTI services invoked by the replicas to the DRC. The other instances are discarded to avoid redundancy. For each replica, a service counter is maintained in

the replication module to record the number of RTI services invoked by the replica. When an RTI service is invoked by the replica, the corresponding service counter is increased by one and attached to the RTI service call as its sequence number. The fastest replica, i.e., the replica whose corresponding service counter has greatest value, is chosen to represent the federate in the federation. The federate's logical time is equal to the logical time of the fastest replica. The RTI service invoked by the fastest replica is processed by the DRC. Its return value is buffered in the replication module with the sequence number attached. The RTI service invoked by other replicas is returned immediately using the buffered return value with the same sequence number. When a callback is delivered to the fastest replica, it is also buffered in the replication module with the sequence number attached. The buffered callback is delivered to other replicas only if its sequence number is equal to the value of the corresponding service counters. Hence, other replicas are consistent as the fastest one and will invoke the same RTI services following the same sequence.

Figure 2.1: No-Barrier federate replication

As shown in Figure 2.1, Replica2 executes faster than Replica1. Service_k and Ser-

vice_ $k+1$, where k and $k+1$ are the sequence numbers of the RTI services, are processed by the DRC when they are invoked by **Replica2**. When **Service_ k** is invoked by **Replica1**, the DRC returns the buffered **Return_ k** immediately. **CallbackA** and **CallbackB** are buffered with sequence number k and $k+1$ respectively when they are delivered to **Replica2**. After **Replica1** invokes **Service_ k** , **CallbackA** is eligible to be delivered.

In this thesis, we propose to enhance simulation performance by replicating federate using either a conservative or an optimistic synchronization approach. We show that CON and OPT replicas of the same federate cannot satisfy the Piecewise Equivalent requirement. Hence, a No-Barrier federate replication structure cannot be used for our purpose. Instead, an Un-Identical federate replication structure is proposed.

2.1.3 Improving Synchronization Efficiency

2.1.3.1 Conservative and Optimistic Synchronization

The other well studied direction to improve the performance of parallel and distributed simulation is to reduce the synchronization overhead. As mentioned in Subsection 1.1.2.2, two kinds of synchronization approaches are commonly used: conservative and optimistic approaches. The efficiency of a conservative approach is critically reliant on LA extraction, which is simulation model dependent. An optimistic approach may outperform a conservative approach if causality errors seldom occur. Many research works have been done on evaluating and predicting the performances of conservative and optimistic approaches [39, 84]. It is generally agreed that a conservative approach cannot outperform an optimistic approach in every situation, and vice versa. Different simulation applications need to employ different synchronization approaches to achieve higher efficiency. Even for the same simulation application, different inputs and parameters may require different synchronization approaches in order to achieve better performance [112]. It may

also happen that some federates in the federation are executed efficiently using conservative approach, while others are executed efficiently using optimistic approach. Moreover, the suitability of a synchronization approach for the same simulation application or federate may also change dynamically.

To take advantage of different synchronization approaches, a number of hybrid protocols have been proposed since the beginning of 1990s. They are classified into two groups as follows:

- *Find the compromise between conservative and optimistic approaches:* conservative and optimistic approaches go towards two extremes in a spectrum of possibilities with respect to the treatment of external events: conservative approaches impose strict constraint in time advancement and event processing; whereas optimistic approaches allow free processing of events with the price of error recovery. To reach the compromise, one can either introduce optimism in a conservative approach (e.g., the optimistic-conservative synchronization [115]), or introduce conservatism in an optimistic approach (e.g., Moving Time Window protocol [102] and Adaptive Time Warp protocol [9]). A probabilistic distributed simulation protocol [38] was also proposed as a performance compromise between conservative and optimistic approaches based on information extracted from the simulation model.
- *Mix and switch between different synchronization approaches:* the Local Time Warp method [98] enables mixing of synchronization approaches hierarchically. At the lower level, an optimistic approach is used to exploit parallelism, while a conservative approach is adopted to control cascaded rollbacks at the higher level. The parallel simulation protocol proposed in [61] and the μ sik framework [90] not only allow mixing of different synchronization approaches, but also support switching different approaches dynamically during the simulation.

Unfortunately, it is difficult to find the compromise between conservative and optimistic approaches for real simulation applications [38]. Switching between different synchronization approaches dynamically is an attractive method. However, it is still challenging to determinate when the switching should take place. Moreover, special code should be added to do the switching and the switching overhead is considerable. For example, to switch from optimistic to conservative synchronization, the federate should rollback its execution to the logical time, discarding the optimistic execution between the logical time and the simulation time.

For federated simulation in general, only one synchronization approach is employed at any one time by each federate during the federation execution. Different from the above existing approaches, we propose to support alternative synchronization approaches simultaneously for the same federate. Hence, the performance of the federate can be governed by the best approach among the employed approaches during the federation execution.

2.1.3.2 GALT Calculation Algorithm

As mentioned in Subsection 1.1.2.2, a bound, known as the GALT, is placed on the logical time advancement of both CON and OPT federates. Therefore, the synchronization overhead is directly related to the performance of the GALT calculation algorithm. The GALT value can be calculated in either an asynchronous or a synchronous manner. In general, asynchronous GALT calculation algorithms utilize Unconditional Information (UI) [46] while synchronous GALT algorithms utilize Conditional Information (CI) [46]. UI can be guaranteed to be true based on local information. For instance, a federate can unconditionally guarantee logical time plus LA is a lower bound on the RVTs of its future sending messages. CI can only be guaranteed to be true if the condition is true. For instance, TS of the next event plus LA is the lower bound of RVTs of future sending

messages of the federate, if it will never receive any future message with RVT smaller than TS of the next event. For the implementation, UI and CI can be exchanged among federates in the federation in either a “pushing” [88] or a “pulling” [49] style.

The well-known null message algorithm [15, 20] is a typical asynchronous algorithm. The null message including UI reports may asynchronously trigger the GALT calculation in the receiving federates. With a small lookahead, the performance of asynchronous algorithms degrades significantly due to the “time creep” problem [21] where federates only advance in LA increments. In synchronous algorithms [7, 46, 70], all federates in the federation are required to send CI reports to calculate a new value of GALT. Hence, the time advancements of federates may be blocked by a federate which sends CI reports less frequently. In addition, synchronous GALT algorithms need to deal with transient messages in a proper way to ensure that they are considered in the GALT calculation. Solutions to the transient messages have been proposed using message counters [46], message acknowledgements [99], and a barrier mechanism [83].

Pan et al. [88] have implemented both asynchronous and synchronous GALT algorithms on SOHR. To address the drawbacks of the synchronous and asynchronous GALT algorithms, a hybrid GALT algorithm was also proposed in [88] based on both UI and CI. The GALT calculation algorithm is out of scope of this thesis. For simplification, the performance enhancement and fault tolerance mechanisms proposed in this thesis are implemented on SOHR using the synchronous GALT algorithm described in [88]. However, these mechanisms can be extended with minor modification to other GALT calculation algorithms.

2.2 Fault Tolerant HLA-based Simulation

2.2.1 Fault Tolerance in Distributed Systems

Fault tolerance is a well studied research issue in the domain of distributed computation. Cristian [29] has introduced some rules in designing and understanding fault-tolerant distributed systems, including understanding failure semantics, masking failure and balancing design cost. Failures in distributed systems can be classified into two major categories: crash-stop and Byzantine failures. Upon a crash-stop failure, the component in the distributed system stops producing outputs. However, as long as it produces outputs, the outputs must be correct. Upon a Byzantine failure, the component in the distributed system can still respond. However, it might produce arbitrary outputs. Three mechanisms are required in handling various types of failures in distributed systems.

- Failure detection and location: A failure must be detected within reasonable time. Moreover, the source of the failure should be identified and the type of the failure should be recognized too.
- Load redistribution: The failed component needs to be redistributed on a non-faulty computing node. The load balance issue should be taken into account while redistributing the failed component. Load redistribution algorithms are out of scope of this thesis. They will be discussed briefly in our future work in Chapter 9.
- State restoration: In the case of a failure, some parts of data and messages may be lost. Mechanisms have to be provided to allow the recovery of the failed component, while keeping the consistency of the system.

2.2.1.1 Crash-stop Failures

A classical crash-stop failure detection method is based on heartbeat delivery among components in distributed systems. For example, Stelling et al. [104] have proposed a heartbeat-based fault detection service for wide area distributed computations. This method can be implemented in either a centralized or distributed manner. In the centralized manner, a failure detection manager monitors the heartbeats from a group of components. In the distributed manner, a virtual token ring is constructed among a number of components, and heartbeats are delivered along the ring. If a heartbeat from a component is not received within a specified time, the component is said to be failed. Detecting failure using heartbeat-based method is unreliable, since it can erroneously indicate that a component has failed as the heartbeat does not arrive on time. This error can be corrected when the heartbeat from the component is received at a later time.

Two techniques commonly used for the state recovery of crashed components in distributed systems are: checkpointing [36] and replication [53]. Stable checkpoint refers to the state of a component saved on stable storage to survive the failure. Stable storage can be a stable disk on the computing node or the volatile memory on another computing node. Local checkpoints are taken by components independently as the snapshots of their local states. Unfortunately, the domino effect, i.e., the components may roll back to their initial states in worst case, may occur while attempting to recover the state of a distributed system. A global checkpoint consists of a set of coordinated local checkpoints, such that they form a consistent system state [19]. It can be used to recover the distributed system directly, avoiding the domino effect. However, components may lose their autonomy and a higher communication overhead is needed to take the checkpoints in coordination with other components in the distributed system.

An alternative solution to overcome the domino effect is to take both the local check-

point and message logging for the components [62]. This is because the component state can be recreated by restoring the process from the local checkpoint and repeating its execution according to the sequence of logged messages. Pessimistic and optimistic message logging protocols are proposed in [62]. Pessimistic message logging protocols log message synchronously. That is, the component is blocked from handling the received messages until the messages have been logged. In contrast, optimistic message logging protocols log the message asynchronously. They can significantly reduce the overhead caused by message logging. However, if some messages have not been logged when a component fails, a more expensive recovery procedure must be used to restore the system. Sender-based message logging [63] introduces low overhead, as the messages are saved asynchronously in local memory on the sender side. However, a process which receives messages from multiple senders must send back the receiving sequence number of the messages to their corresponding senders for the sender-based message logging. Otherwise, the recovered process cannot receive the logged messages in the same sequence as the crashed process before the failure.

Replication requires maintaining multiple on-line copies of the same component on separate computing nodes. It can be distinguished between active and passive replication. In the active replication, the computation is performed at each replica independently and concurrently. In the case that one replica crashes, the remaining replicas of that component can continue the computation without interruption. In the passive replication, also called the primary-backup replication, computation is only performed on the primary replica, and then its state is transferred to the backup replicas. In the case that the primary replica crashes, the computation will be interrupted and one of the passive, backup replicas will be chosen as the new primary replica.

2.2.1.2 Byzantine Fault Tolerance

In a distributed system, a component encountering a Byzantine failure (i.e., faulty component) might process request incorrectly, thus corrupting its local state and/or producing incorrect or inconsistent outputs. As a result, the distributed system might behave in an unpredictable way. A *Byzantine Fault Tolerance* (BFT) mechanism guarantees the distributed system works in the correct manner, performing the specified functionality in the presence of Byzantine failures. It should detect the Byzantine failures, locate the faulty components, and finally recover the distributed system from the Byzantine failures. Simply excluding the faulty components from the execution is not enough. The BFT mechanism should recover the faulty components with correct states, as well as remove the impact of the arbitrary outputs generated by the faulty components from the distributed system.

Since the Byzantine generals' problem [66] was proposed at the beginning of 1980s, a lot of research has been done using replication and agreement protocols to deal with Byzantine failures. Some state-of-the-art *Byzantine Fault Tolerance* (BFT) mechanisms are proposed in [18, 28, 65] which describe replication protocols that correctly survive Byzantine failures in an asynchronous network. The failures happen on the replicas of the same process are referred to as the *simultaneous failures*, if they occur before any one of them is detected. We assume that the maximum number of simultaneous failures is f . The failures happen on the replicas of different processes are referred to as *concurrent failures*, if they occur before all other failures are recovered. For clarity of illustration, we assume that concurrent failures never happen.

Simply monitoring the heartbeats of distributed components is not enough for Byzantine failure detection. Instead, an agreement protocol, e.g., majority voting policy, should be run on their outputs. To make the agreement, at least outputs from $2f + 1$ replicas

should be collected. In an asynchronous system, e.g., internet, it is impossible to wait for more than $n - f$ replicas (n is the number of executed replicas) to send the outputs, because the last f replicas might never reply [14]. Therefore, $n - f$ must be greater than or equal to $2f + 1$, that is, $n \geq 3f + 1$. Therefore, BFT mechanisms proposed in [18, 28, 65] require at least $3f + 1$ replicas for each process. They are able to detect and locate Byzantine failures after receiving outputs of $2f + 1$ replicas. However, they are overly expensive in terms of communication costs and redundant executions of replicas.

To reduce redundant executions of replicas, a replication- and checkpoint-based approach is proposed in [1]. Each computation task is carried out by $f + 1$ replicas (**Group1**) of the process. In the case that **Group1** replicas have reported the same checkpoint, the reported checkpoint becomes a consistent checkpoint and no Byzantine failure is detected. Otherwise, another f replicas (**Group2**) are started from the latest consistent checkpoint. By comparing the checkpoints reported from **Group1** and **Group2** replicas, the faulty replicas can be identified and finally excluded from the execution. In the case that some replicas (at most f replicas) in **Group1** and **Group2** never report their checkpoints, they should be substituted by new replicas.

Similar to the approach in [1], our BFT mechanism starts **Group2** replicas of the federate, when it is necessary, instead of executing them during the entire execution. However, it is assumed in [1] that the computation task can be completed by each process without exchanging messages with other processes. Hence, the Byzantine failure never affects executions of other processes in the distributed system. Unfortunately, this assumption is no longer true for HLA-based simulation where federates communicate with each other frequently through the RTI. As a result, a Byzantine failure might propagate quickly in the federation execution if the corresponding faulty federate is not isolated in time. How to deal with this epidemic effect will be investigated in our study.

2.2.2 Fault Tolerance in HLA Community

The research work on fault tolerance has been fallen behind in the HLA community [64]. The HLA standard IEEE 1516-2000 [56] itself has no formal fault tolerance model. The updated HLA standard IEEE 1516-2010 [55], commonly known as HLA Evolved, covers more extensively the issue of fault tolerance. A common semantic for failure is defined and mechanisms for fault-detection are provided in [81, 82]. Two types of failures are studied: (i) Federate lost which means that a federate has been unexpectedly lost from a federation; and (ii) Connection lost which means that a federation has been unexpectedly lost from a federate. The HLA Evolved only focuses on crash-stop failures rather than Byzantine failures. Pitch pRTI [92] is the pioneer of adopting HLA Evolved. It prevents a crashed federate from blocking the federation execution by resigning it from the federation. However, the mechanisms for data recovery of the crashed federate, which are indispensable to ensure the correctness of simulation execution, are not provided.

The requirements for fault tolerant distributed simulation and possible approaches to meet these requirements were discussed in [71]. The *Resource Sharing System* (RSS) [72] and *Fault Tolerant Resource Sharing System* (FT-RSS) [73] frameworks have been proposed to balance federates among available computing resources and integrate fault tolerance mechanisms into distributed simulations respectively. As mentioned in [71], the fault tolerance of HLA-based simulation needs to be supported from two directions: implementing a dependable RTI and providing fault tolerance support for federates. The unpredictable failures of the RTI are handled in [22, 81] by dynamically substituting the crashed RTI (including RTI components of all federates in the federation) with newly created or standby one. In this thesis, we focus on federate failures. The existing federate fault tolerance mechanisms can be generally categorized into two groups: checkpoint-based and replication-based.

2.2.2.1 Checkpoint-based Fault Tolerance Mechanisms

Some checkpoint-based mechanisms have been proposed to tolerate crash-stop failures of federates. Upon a failure, a global checkpoint of an HLA-based simulation can be used to recover the execution directly. Grošelj [52] has developed an algorithm to record consistent global checkpoints of a distributed simulation model named *Cocktail Party Simulation* (COPS). Most of the synchronization overhead in building the global checkpoints can be eliminated by executing the simulation in phases, which are defined naturally in COPS. Cucuzzo et al. [30] has proposed a lightweight heuristic-based mechanism to construct a global checkpoint named *Committed Consistent Global States* (CCGS), using the snapshots of application specific state taken for rollback purpose in OPT federates. A global checkpoint can also be taken by invoking *RequestFederationSave* RTI service [57]. However, the overhead is significant, as the federation execution is blocked for the state saving of both federates and the RTI.

Different from global consistent checkpoints, local checkpoints are taken by federates independently without the coordination with other federates in the federation. Fault tolerance mechanisms relying on the local checkpoints of federates were proposed in [2, 31, 35]. These mechanisms are based on the traditional federate architecture. They must require the crashed federate to resign from the federation and the recovered federate to join the federation. The RTI state concerned with the federate is lost in the failure, as it is kept in the tightly coupled LRC. For the correct federation execution, the lost RTI state (e.g., publishing/subscribing interaction classes) must be rebuilt for the recovered federate. To ensure the correctness of message delivery in the presence of a federate failure, sender-based message logging should be taken by other federates in the federation. In order to recover the failure of the federate, the other federates are required to resend logged messages. Moreover, the recovered federate should also synchronize its

time advancement with other federates to guarantee the correctness of the federation execution.

Similarly, the federate fault tolerance mechanisms proposed in this thesis rely on a federate's local checkpoints. However, our mechanisms are based on the decoupled federate architecture. The DRC keeps the connection to the federation and maintains the RTI state concerned with the federate. The federate failure can be recovered without interrupting the execution of other federates in the federation. Furthermore, the fault tolerance mechanisms proposed in [2, 31, 35] are workable on OPT federates only, whereas our mechanisms can be applied on both CON and OPT federates.

2.2.2.2 Replication-based Fault Tolerance Mechanisms

Berchtold and Hezel [11] have proposed an active replication approach named *R-Fed* for fault tolerance purpose. The concept of the decoupled federate architecture was used implicitly. The tight-coupling between federate and its LRC in the traditional federate architecture is removed by inserting fault tolerance components between them. Multiple replicas are developed for each federate. The fault tolerance components, which lie on a single computing node, connect these replicas to the same RTI component (i.e., the DRC in the decoupled federate architecture). Similar to the No-Barrier federate replication structure discussed in Subsection 2.1.2, the replicas of the same federate are also required to be *Piecewise Equivalent*. Moreover, replicas of the same federate are executed synchronously. They have the same logical time (which is also referred to as the federate's logical time), and the same callbacks are delivered to them at the same time. As a result, the replicas will invoke the same RTI services. Each RTI service invoked by a replica is blocked by the barrier until it has been invoked by all other replicas of the same federate. Only one instance of the RTI service invocations is forwarded to the DRC. In this way, the execution is held back by the slowest replica for each RTI service

invocation. Consequently, the simulation performance might degrade significantly. This replication structure is referred to as *Barrier federate replication structure* in this thesis.

The fault tolerance components in R-Fed are able to tolerate both crash-stop (at least $f + 1$ replicas) and Byzantine failures (at least $3f + 1$ replicas). The failures can be detected and located by monitoring the outputs of replicas. The failed replicas can be recovered using the state of a correct replica. Byzantine failures never propagate in the federation execution, as only the same RTI services invoked by all replicas are processed by the DRC. However, as we will discuss in Chapter 7, comparing the RTI services invoked by replicas of the same federate, as reported in [11], is not enough for Byzantine failure detection of a federate.

Based on the study of the above mentioned work, it is evident that combining efficiency with fault tolerance is an attractive but difficult task. Checkpoint based fault tolerance approaches require memory redundancy and repeated execution. The federation execution is disrupted to recover the failed federate. Fault tolerance using active federate replication approaches can overcome this shortcoming. However, computation redundancy is required and overhead is introduced to maintain replicas of the same federate.

In this thesis, the Un-Identical federate replication proposed for performance enhancement purpose is extended to tolerate crash-stop federate failures. Efficient Byzantine Fault Tolerant (BFT) mechanism is also proposed based on Barrier and No-Barrier federate replication structures by exploiting checkpoint and message logging techniques. Compared with the Barrier federate replication structure, the No-Barrier one has advantages in terms of performance enhancement. However, it may suffer from the epidemic effect of a Byzantine failure. To remove the epidemic effect from the federation execution, additional overhead will be introduced.

2.3 Summary

This chapter introduces the related work on performance enhancement and fault tolerance issues for HLA-based simulation. For simulation performance enhancement, research works have been carried out mainly along the following three directions, i.e., balancing workload, improving synchronization efficiency, and replicating federate executions. Accordingly, a number of federate migration, hybrid synchronization, and federate replication approaches have been proposed. For fault tolerance issue, we first introduced the problems of crash-stop and Byzantine failures in the distributed system, as well as the potential solutions. After that, we presented the efforts in HLA community to address the failures occurred in simulation executions. Checkpoint-based and replication-based mechanisms are discussed. In addition, Barrier and No-Barrier federate replication structures based on the decoupled federate architecture are also introduced in this chapter.

Discussion in this chapter helps to identify the gap between the existing work and our objectives and provides the basis for the work that will be discussed in the following chapters:

- Discussion in Subsection 2.1.1 leads to the work to be described in Chapter 3;
- Discussion in Subsection 2.1.2 and Subsection 2.1.3 lead to the work to be described in Chapter 4;
- Discussion in Subsection 2.2.2.1 leads to the work to be described in Chapter 5; and
- Discussion in Subsection 2.2.2.2 leads to the work to be described in Chapter 6 and 7.

Chapter 3

Federate Migration Based on Decoupled Federate Architecture

To achieve workload balance and thus improve simulation performance, one of the objectives of this thesis is to propose efficient federate migration protocols by exploiting the advantages of decoupled federate architecture. This chapter describes one-phase, two-phases, and relay-based federate migration protocols and compares their migration overheads.

3.1 Migration Protocols

In the traditional federate architecture, a federate is tightly coupled with its LRC. Both of them need to be migrated to the destination computing node. Besides the federate state, the RTI state kept in the LRC also needs to be saved and restored during the migration. Although migrating federate (**MigFed**) and restarting federate (**ResFed**) execute the same simulation model, they are regarded as different federates. **MigFed** should resign from the federation; whereas, **ResFed** should join in the federation. Moreover, other federates in

the federation should be informed about the migration. Otherwise, they cannot redirect communications to the **ResFed** after the migration. As mentioned in Subsection 2.1.1, the federate migration protocols [106, 117, 118] based on the traditional federate architecture are usually complex and inefficient.

In the decoupled federate architecture, the federate can be migrated to the destination computing node while connecting to the same DRC. It is the DRC instead of the federate itself that connects to the federations. Since the DRC keeps the connection to the federation for the federate, the federate can be migrated without informing other federates in the federation. To the other federates in the federation, **MigFed** and **ResFed** are exactly the same. There is no need for **MigFed** to resign from the federation and for **ResFed** to join the federation. The DRC keeps the RTI state of the corresponding federate. For instance, the DRC maintains federate’s publishing and subscription information. Thus, there is no need for **MigFed** to unpublish and unsubscribe, and **ResFed** to publish and subscribe object and interaction classes. Similarly, **ResFed** does not have to re-enable time management, re-build data distribution management regions, and obtain ownerships from **MigFed**. Since the DRC buffers all RTI service calls and callbacks of the federate, the problems of message duplication and loss, as described in [17], are also avoided.

The migration protocols proposed in this chapter were implemented on SOHR. As shown in Figure 3.1, a migration module is embedded in the DRC (i.e., LIS/LRI). Following components were also provided in the execution environment to support federate migration. A federate server (a Grid service) is deployed at each computing node. It registers with the RTI Index service on its creation, reports the workload of the computing node periodically, and manages all federates executed on the same computing node. To ensure efficiency, low-level communication (e.g., Java Sockets) is utilized for the communication between the federate server and federates on the same computing node. A federate can report its status (e.g., execution stage) to the federate server. It

Figure 3.1: Federate migration on SOHR

can also understand and execute commands from the federate server, such as start and terminate execution. To support data transfer, a Grid FTP server is deployed at each computing node. A workload balance module in the RTI Index service is used to determine which federate needs to be migrated, when and where it should be migrated. The load-balancing algorithm is outside the scope of this thesis. Instead, we focus on efficient federate migration protocols.

The federate server deployed on each computing node plays an important role during federate migration. It informs the federate to migrate after receiving the migration decision from the RTI Index service and informs the federate server at the destination computing node to restart the federate. It also suspends **MigFed** to save its state and finally terminates **MigFed**. To migrate a federate, some files need to be transferred to the destination. These files can be divided into two types: static files and dynamic files. Static files are read-only, such as federate code files and the *Federation Object Model Document Data* (FDD) file [56]; while dynamic files can be modified by the federate during its execution, such as database files and the federate execution state file. Files are transferred to the destination computing node by the federate server via Grid FTP.

Figure 3.2: One-phase migration protocol

In the following, a one-phase migration protocol is first proposed to illustrate the federate migration procedure. To achieve higher efficiency, two-phases and relay-based migration protocols are further developed. As mentioned in Subsection 2.1.1, we focus on application level federate migration and assume that state saving and restoring functions are provided by migration-enabled federates.

3.1.1 One-phase Migration

The detail of the one-phase migration protocol is shown in Figure 3.2. **Computing Node1** is the source computing node; while **Computing Node2** is the destination computing node. In this protocol, the federate server on **Computing Node1** is informed about the migration of a federate. **MigFed** is suspended immediately after receiving the “migrate” command from the federate server. Its state is saved into a state file via a federate-saving function provided by the federate. **MigFed** provides the files that need to be transferred, which are then uploaded by the federate server to the Grid FTP service deployed on **Computing Node2**. After the federate files are sent, the federate server terminates **MigFed** and informs the federate server on **Computing Node2** to restore **ResFed** via a federate-restoring function provided by the federate. Finally, **ResFed** resumes execution on **Computing Node2**.

Figure 3.3: Two-phases migration protocol

3.1.2 Two-phases Migration

In the two-phases migration shown in Figure 3.3, static and dynamic files are sent separately. MigFed on Computing Node1 provides the static files to the federate sever after being informed to migrate. It keeps running, while the federate server uploads the static files to the Grid FTP server on Computing Node2. Thus, the transfer of static files is overlapped with the continuous execution of MigFed. The time taken for transferring static files, which may be considerable for complex federates, is excluded from the migration overhead. After static files are uploaded, the federate server on Computing Node1 informs MigFed to suspend and save execution state into a state file. After that, MigFed provides the federate server with the dynamic files including the state file. These files are sent to Computing Node2 via Grid FTP transfer. Finally, MigFed on Computing Node1 is terminated and ResFed is started on Computing Node2.

3.1.3 Relay-based Migration

The relay-based migration protocol is proposed to further overlap the transfer of dynamic files and the restoration of ResFed with the continuous execution of MigFed. In this way, federate state saving becomes the only overhead of federate migration. In the relay-

Figure 3.4: Relay-based migration protocol

based migration protocol shown in Figure 3.4, **MigFed** will be terminated after **ResFed** has restored its state and caught up with **MigFed** in execution. In the case that **ResFed** cannot catch up, **MigFed** will be terminated after a certain period of time after **ResFed** has been restored.

Similar to the two-phases migration protocol, static and dynamic files are sent separately in this protocol. **MigFed** and **ResFed** are executing concurrently and are connected to the same DRC. It is assumed that the execution of migration-enabled federates are *Piecewise Deterministic* [48]. This means that when a federate receives callbacks, it performs internal computation and invokes some RTI services, and then blocks its execution to receive future callbacks. All these actions are deterministic and determined by the callbacks received and the execution state when the callbacks are received. Hence, **ResFed** restored from the execution state is able to repeat the execution identically as **MigFed**, if the same callbacks are delivered to **ResFed** following the same sequence as **MigFed**. Thus, the same federate-initiated services will be invoked by both **MigFed** and **ResFed**. However, a federate-initiated service should only be handled once.

To solve the above problems, two service counters, a service buffer and a callback

Figure 3.5: DRC design in relay-based migration

buffer, as shown in Figure 3.5, are deployed in the migration module in DRC. The service counters are used to record the number of services invoked by **MigFed** and **ResFed** after DRC is informed about the federate migration. When a service is invoked by **MigFed** or **ResFed**, the corresponding service counter is increased by one. Thus, the service counters keep track of the sequence numbers of the latest received service invocations from **MigFed** and **ResFed** respectively. The service buffer is used to buffer the invoked service represented by the service sequence number and its return value. Initially, **MigFed** is likely to run ahead of **ResFed**, as **ResFed** can only resume execution after the federate state is restored. In this case, only the service invoked by **MigFed** is handled. The service counter of **MigFed** and the return value of the invoked service are saved in the service buffer. When the service is invoked by **ResFed** later, instead of handling it again, the DRC searches the service buffer using the service sequence number and returns the corresponding return value to **ResFed**. Thus, the same federate-initiated service is handled only once. Both **MigFed** and **ResFed** can get the expected return values from the DRC.

The callback buffer in the migration module of the DRC is used to buffer the callbacks fetched between federate-initiated services. For each callback, the sequence number of the last federate-initiated service is attached. Similarly, **MigFed** runs ahead of **ResFed** initially. It fetches callbacks from message queues in the callback module of the DRC. The service counter of **MigFed** and the fetched callbacks are saved in the callback buffer in the migration module of the DRC. When **ResFed** tries to get the callbacks later, the DRC directly returns all the callbacks in the callback buffer whose sequence number is equal to the service counter of **ResFed**. Thus, both **MigFed** and **ResFed** can fetch the same callbacks follow the same sequence from the DRC.

An example of relay-based migration is shown in Figure 3.5.b. The federate gets one callback after it has sent one interaction. The change of the service counters, service buffer and callback buffer in the migration module of the DRC is shown step by step. The changed elements are shaded. On receiving **Interaction1** from **MigFed**, the DRC increases the service counter of **MigFed** by one. As **MigFed**'s service counter is greater than **ResFed**, the request is handled and the interaction is sent to subscribers. The return value is buffered in the service buffer with **MigFed**'s service counter. **Nil** is returned for the service invocation of sending interaction. **MigFed** fetches **CallbackA** from the callback module of the DRC, as it runs ahead of **ResFed**. **CallbackA** is buffered in the migration module of the DRC with the current service counter of **MigFed**. Similarly, **MigFed** sends **Interaction2** and gets **CallbackB**. After state restoration, **ResFed** sends **Interaction1** to the DRC. As service counter of **ResFed** is smaller than that of **MigFed**, **Interaction1** is not sent to subscribers again. The return value (**Nil**) is retrieved from the service buffer and returned to **ResFed** immediately. Subsequently, **ResFed** tries to get callbacks. It will get the callbacks from the callback buffer and only **CallbackA** will be returned since the sequence number of **CallbackB** is greater than the current service counter of **ResFed**.

Generally, a federate is migrated to a lightly-loaded computing node. Hence, **ResFed**

runs faster than **MigFed**. **ResFed** runs faster also because its service invocations of the DRC are returned immediately and the callbacks are fetched from the callback buffer in the migration module of the DRC directly, without waiting for the messages from other federates in the federation. **ResFed** catches up **MigFed** in execution if its service counter is greater than that of **MigFed**. After **ResFed** catches up with **MigFed**, the DRC informs the federate server to terminate **MigFed**. It is also possible that **ResFed** may never catch up with **MigFed**. To avoid this situation, **MigFed** is terminated after a period of time after **ResFed** has been restored. If **MigFed** is terminated because of time out, the return values in the service buffer and callbacks in the callback buffer are returned to **ResFed** initially in response to its service invocations and get callbacks requests. After the service buffer and callback buffer in the migration module of the DRC are exhausted, the DRC will start processing service invocations and get callback requests received from **ResFed**.

In the relay-based migration protocol, storage overhead is introduced in the DRC to buffer the services invoked by the **MigFed** and the callbacks delivered to the **MigFed**. However, the storage will be reclaimed once the services are invoked by the **ResFed** and the callbacks are delivered to the **ResFed**. The storage overhead depends on how fast the **ResFed** can catch up the **MigFed**.

3.2 Experiments and Results

3.2.1 Experiment Design

A ping-pong simulation model described in [87] is used in our experiments to evaluate the performance of the proposed federate migration protocols. It is composed of two federates. **Federate1** sends an interaction to **Federate2** and then waits to receive an interaction from **Federate2** before repeating the same process. **Federate2** sends an interaction

back to **Federate1**, after receiving an interaction from **Federate1**. Spin-loops are added to emulate event processing times before an interaction is sent and after a callback is received. **Federate1** is migrated to another computing node once during the federation execution. In total, 100 ping-pong iterations are simulated. In the experiment, all computing nodes used have the same workload. The migration overhead is measured as the difference between the execution times of the simulation with migration and that without migration.

Experiments are carried out in a cluster, which has an infiniband connection and each of its computing nodes is installed with 2*Dual core Xeon 3.0GHZ CPU, 4G RAM and Redhat Enterprise 4 OS. The RTI Index service and the HLA RTI Management Services in SOHR are deployed on separate computing nodes. Two DRCs (i.e., LIS/LRI) are deployed on different computing nodes, each of which is connected to only one federate. Federates are also executed on different computing nodes. For each execution scenario (i.e., with or without migration), the federation execution is repeated five times. Since the execution time of the repeated executions have small variance, we simply use their average value to calculate the migration overheads reported below.

3.2.2 Experiment Results

The first series of experiments are carried out to compare one-phase and two-phases migration protocols. The size of the dynamic file is fixed at 1 *MBytes*; whereas, the size of the static files is increased from 5 *MBytes* to 25 *MBytes*. As shown in Figure 3.6.a, the migration overhead of the one-phase migration is greater than that of the two-phases migration and increases linearly with respect to the size of the static files. In the two-phases migration, the migration overhead almost stays at the same level. This is because the transfer of static files is overlapped with **MigFed** execution in the two-phases migration.

The second series of experiments are carried out to compare two-phases and relay-based migration protocols. The size of the static file is fixed at 5 *MBytes*; whereas, the size of the dynamic files is increased from 1 *MBytes* to 5 *MBytes*. Figure 3.6.b shows the migration overheads of the two-phases and relay-based migration. Compared with the two-phases migration, the migration overhead of the relay-based migration is much smaller and increases less significantly with respect to the size of the dynamic files. We can also observe from Table 3.1 that the migration overhead of the relay-based migration is close to the state saving time of MigFed. This is because the transfer of dynamic files and federate state restoration are overlapped with MigFed execution in the relay-based migration.

3.6.a: One-phase versus two-phases migration

3.6.b: Two-phases versus relay-based migration

Figure 3.6: Migration overheads of different migration protocols

Table 3.1: Relay-based migration overhead versus state saving time

Size of dynamic files (MBytes)	1	2	3	4	5
Relay-based migration overhead (ms)	191.08	282.71	426.94	580.46	766.05
State saving time of MigFed (ms)	185.61	275.56	414.85	544.56	733.45

3.3 Summary

This chapter has proposed federate migration protocols based on the decoupled federate architecture. Only does the federate need to be migrated. **MigFed** and **ResFed** are connected to the federation through the same DRC. One-phase migration protocol illustrates the federate migration procedure, which is transparent to other federates in the federation. Two-phases and relay-based migration protocols can reduce the migration overhead by overlapping the migration procedure with the continuous execution of **MigFed**. As experimental results shown, the relay-based migration protocol introduces the lowest migration overhead that approximately equals to the state saving time required by **MigFed**.

Chapter 4

Federate Replication for Simulation Performance Enhancement

One of the objectives of this thesis is to improve HLA-based simulation performance by replicating simulations with alternative synchronization approaches. It is inspired by the concepts of active replication and software diversity. For the same federate, multiple replicas are developed in software diversity manner by employing either a conservative or an optimistic synchronization approach. Due to different characteristics of the employed synchronization approaches, replicas of the same federate are likely to exhibit different runtime performances (see discussion in Subsection 2.1.3.1). The simulation performance is therefore enhanced by choosing the fastest replica to represent the federate dynamically and automatically during the execution.

To support concurrent executions of CON and OPT replicas of the same federate, an Un-Identical federate replication structure is proposed in this chapter. It consists of a message exchange mechanism and a TM mechanism. In addition to presenting design and implementation details, a proof of correctness is also provided. Experiments using a synthetic P-hold simulation model are carried out to evaluate the Un-Identical federate

replication structure in terms of performance enhancement and scalability.

4.1 Un-Identical Federate Replication

In traditional federate architecture, the federate and its LRC should be replicated as one unit due to the tight-coupling between them. Each replica of a federate is treated as an individual federate in the federation. As a result, the federation scale increases significantly. Consequently, the overhead of federation wide time synchronization and the risk of federation failures increase. Replicas of federates in the federation communicate with each other through RTI communication channel. Thus, the number of messages exchanged in the RTI communication channel increases sharply. In addition, one replica of each federate should be chosen to take attribute ownerships, because only one federate can have the ownership of an individual attribute of each object instance at any given time.

In decoupled federate architecture, a replication module can be deployed between the federate and its DRC, as shown in Figure 1.5. It is able to connect replicas of the same federate to the same DRC and to support their concurrent executions without increasing federation scale. The replication of a federate is transparent to the federate itself. Its replicas do not know the existence of replication module and other replicas of the same federate. The replication of a federate is also transparent to remote federates. They have no idea about how many replicas are executed for the federate and which replica of the federate they are actually communicating with.

The overview of *Un-Identical federate replication structure* is shown in Figure 4.1. Federate replicas are allowed to employ either a conservative or an optimistic synchronization approach. They are denoted as CON and OPT replicas respectively. A replication module is deployed to support the concurrent executions of any number of replicas

Figure 4.1: Overview of Un-Identical replication structure

of the same federate and connect them to the same DRC.

The existing Barrier and No-Barrier replication structures, which were introduced in Chapter 2, are unable to support concurrent executions of CON and OPT replicas. Firstly, CON and OPT replicas are not *Piecewise Equivalent*. They invoke different RTI services to request time advancement. CON replicas only process events conservatively, so that they modify federate state and generate messages correctly. In contrast, OPT replicas are allowed to process events optimistically and undo the event processing when a causality error happens. They may modify federate state incorrectly and cancel the incorrect modification in the future. They may also generate incorrect messages and retract them in the future. Secondly, the callbacks are delivered to CON and OPT replicas in different manners. The callbacks generated from correct messages are delivered to CON replicas in TS order. In contrast, the callbacks generated from correct messages, incorrect messages, and retractions may be delivered to OPT replicas out of TS order.

In the Un-Identical replication structure, CON and OPT replicas are required to be *Committed Equivalent*, which is a weaker restriction compared to *Piecewise Equivalent*. The replicas with the same initial state should be updated to the same federate state and generate the same messages while processing the same *committed events* (i.e., the events with $TS < \text{logical time}$). The replicas of the same federate are consistent, if they

process the same committed events in TS order. To achieve this, the replication module shown in Figure 4.1 should perform the following tasks:

- (i) Provide a *message exchange mechanism* to support exchanging messages for federate replicas. One and only one instance of each correct message generated by replicas should be delivered to all replicas of the receiving federates. The incorrect messages and their retractions should be delivered properly to ensure that the effects of incorrect messages are removed.
- (ii) Provide a *TM mechanism* to perform federation wide time synchronization and to coordinate time advancements of federate replicas. It grants the time advancement requests from replicas and delivers eligible callbacks to replicas according to their logical time and synchronization approaches.

Section 4.1.1 and Section 4.1.2 will respectively describe these mechanisms in details. The simulation performance is improved because the replication module chooses the fastest replica (i.e., the replica with the largest simulation time) to represent the federate in the federation by using its time information for federation wide time synchronization and sending its generated messages to receiving federates.

As mentioned in Subsection 1.3.3, the Un-Identical replication structure is implemented based on the decoupled federate architecture of SOHR. The replication module is embedded in DRC (i.e., LIS/LRI) to achieve higher efficiency. For the implementation simplicity, the current version of the replication module supports only two replicas, i.e., a CON and an OPT replica, for each federate. Some additional variables and flags are defined in Table 4.1. Since CON and OPT replicas have different runtime performance, they may have different requested time (ConReqTime and OptReqTime) and different logical time (ConLGT and OptLGT). LGT, the logical time of the federate, equals the bigger value between ConLGT and OptLGT, as Equality (4.1) shows. A replica is in time

Table 4.1: Additional variables and flags

Name	Type	Description
ConReqTime	Double	Requested time from CON replica
OptReqTime	Double	Requested time from OPT replica
ConLGT	Double	Last Granted Time of CON replica
OptLGT	Double	Last Granted Time of OPT replica
LGT	Double	Last Granted Time of Federate
ConInTAS	Boolean	CON replica is in time advancing or granted state
OptInTAS	Boolean	OPT replica is in time advancing or granted state
InTAS	Boolean	Federate is in time advancing or granted state

advancing state if its corresponding flag, i.e., ConInTAS or OptInTAS, is true. Otherwise, it is in time granted state. As Equality (4.2) shows, a federate is in time advancing state, if time advancement request from either CON or OPT replica is pending.

$$LGT = \text{Max}(\text{ConLGT}, \text{OptLGT}) \quad (4.1)$$

$$\text{InTAS} = \text{ConInTAS} || \text{OptInTAS} \quad (4.2)$$

4.1.1 Message Exchange Mechanism

The message exchange mechanism among replicated federates is shown in Figure 4.2. The solid lines represent the traces of outgoing messages sent from the federate, while the dashed lines show the flow of incoming messages destined to the federate. The message exchange mechanism requires the replication module to cooperate with the existing *Object Management* (OM) module in the original SOHR. The OM module is responsible for exchanging messages and retractions with other federates; whereas, the replication module is responsible for distributing incoming messages to replicas of the federate and filtering redundant outgoing messages generated by the replicas.

Figure 4.2: Message exchange mechanism

Mechanism for handling incoming messages: According to the HLA IFSpec [57], different callbacks are eligible to be delivered to CON and OPT federates. Hence, two individual TSO queues are deployed: one for CON replica (ConTSOQueue) and the other for OPT replica (OptTSOQueue). When a message (either a correct or an incorrect message) is received by the OM module (step R1), it is transferred to the replication module. Then, a callback is generated and buffered in both TSO queues (step R2). The callbacks buffered in TSO queues are sorted by their RVTs. ConTSOHead (OptTSOHead) refers to the minimum RVT of callbacks in the ConTSOQueue (OptTSOQueue respectively). Finally the buffered callbacks are delivered to the corresponding replicas and removed from the TSO queues (step R3). Only the callbacks in ConTSOQueue with $RVTs \leq ConLGT$ are delivered to CON replica; whereas, all callbacks in OptTSOQueue are delivered to OPT replica.

When a retraction is received, it is also transferred to the replication module. In the case that the corresponding incorrect message is kept in the TSO queues, it will be

annihilated with the retraction. Otherwise, a *RR* callback will be generated and finally delivered to the corresponding replica to remove the effect of the incorrect message. As will be proved in Subsection 4.1.3.2, the callbacks generated from incorrect messages and their retractions are annihilated in *ConTSOQueue* instead of being delivered to *CON* replica. In contrast, they might be delivered to *OPT* replica and the effects of the incorrect messages are removed using state saving and rollback mechanisms. In addition, two individual Non-message Callback Queues (not shown in the figure) are deployed: one for *CON* replica to buffer the *TAG* callbacks, and the other for *OPT* replica to buffer the *TAG* and *RR* callbacks.

Mechanism for handling outgoing messages: Outgoing messages are generated by *CON* and *OPT* replicas (step S1). To filter the redundant messages, two message filters *ConMsgFilter* and *OptMsgFilter* are deployed for *CON* replica and *OPT* replica respectively. The messages buffered in message filters are sorted by their RVTs. *ConMsgHead* (or *OptMsgHead*) denotes the minimal RVT of the messages in *ConMsgFilter* (or *OptMsgFilter*). When a message is generated by a replica, the replication module searches the same message (i.e., with the same SVT, RVT, and message content) in the message filter of the other replica (step S2). If the same message is found, the message must have been generated by the other replica. So, the message is discarded and the same message is also removed from the message filter of the other replica. Otherwise (that is, if the same message is not found), the message is forwarded to the OM module and buffered in the message filter of the replica (step 3). In this way, the first instance of each correct message is sent to the receiving federates, while the second one is discarded. The incorrect messages generated by *OPT* replica will be kept in *OptMsgFilter* after they are forwarded to the OM module. They will be removed from *OptMsgFilter* when their corresponding retractions are generated and forwarded to the OM module.

An example of the message exchange mechanism is shown in Figure 4.3. The incom-

Figure 4.3: An example of message exchange mechanism

ing messages, *Msg_k* and *Msg_k+1*, are buffered in ConTSOQueue and OptTSOQueue in the form of callbacks. They are eligible to be delivered to OPT replica. While processing them, OPT replica generates *Msg_l* and *Msg_l+1* respectively. These outgoing messages are sent to **Federate2**, as they are not generated by CON replica previously. In contrast to the OPT replica, *Msg_k+1* is not delivered to CON replica because its RVT is greater than ConLGT. Since replicas are committed equivalent, CON replica generates *Msg_l* while processing *Msg_k*. The message is simply discarded, as the same message is kept in OptMsgFilter. Some time later, **Federate2** sends retraction of *Msg_k+1*, which is identified as an incorrect message. The retraction is annihilated with *Msg_k+1* in ConTSOQueue. However, it is delivered to OPT replica, as *Msg_k+1* has been removed from OptTSOQueue when it was delivered to OPT replica. While processing the retraction, OPT replica retracts *Msg_l+1* and the OM module finally sends the corresponding retraction to **Federate2**.

In addition, messages in the following cases can be simply discarded without searching the same messages in the message filters:

Remark 4.1 (i) *The message generated by a replica is simply discarded if its $RVT < LGT + LA$. (ii) The message generated by OPT replica is discarded if its $RVT < \min(ConReqTime, ConTSOHead) + LA$ and $ConInTAS$ is true.*

In the first case, the message can be discarded because the other replica must have logical time = LGT and have generated all correct messages with RVTs $< LGT + LA$. For this reason, OptMsgFilter and ConMsgFilter can be reclaimed once LGT is updated, by removing those messages with RVTs $< LGT + LA$. In the second case, if the DRC will not receive any message with $RVT < ConTSOHead$, CON replica will be granted to $\min(ConReqTime, ConTSOHead)$, and thus, it will not send any message with $RVT < \min(ConReqTime, ConTSOHead) + LA$. Otherwise, CON replica may only generate messages with $RVTs < \min(ConReqTime, ConTSOHead) + LA$ while handling the new messages received by the DRC. These generated messages are obviously different from those messages generated by OPT replica before the new messages are received by the DRC. So, CON replica will not generate the same message as the message generated by OPT replica in case (ii). In summary, in the case that CON replica has generated the same message previously, the message is a correct message and can be discarded. Otherwise, the message is an incorrect message and can be discarded as well.

4.1.2 TM Mechanism

Pan et al. [88] have proposed a synchronous TM algorithm based on Conditional Information (CI) [46]. However, the algorithm is only designed for the federation where each federate has only a single instance that uses a conservative synchronization protocol. In Un-Identical federate replication structure, the time management requires the replication module to cooperate with the *Time Management* (TM) module in the original SOHR. The TM module calculates the value of GALT using the synchronous algorithm proposed

in [88] (see details in Subsection 2.1.3.2). The replication module handles the time advancement requests from CON and OPT replicas using the latest GALT value. In the case that the GALT value is not great enough, replication module will send TM information (TMInfo) of replicas to the TM module for a new round of GALT calculation. In general, the TMInfo of the faster replica is likely to be used to increase the value of GALT. With the increased GALT value, the time advancement request from the slower replica is likely to be granted immediately.

<pre> bool CalculateGALT(TMInfo) { 1. If(CISeqNum>GALTSeqNum) 2. return false; 3. CI=min(TMInfo+LA, minTSOfTransientMessages); 4. CISeqNum++; 5. send CI_CISeqNum to DRCs of other federates 6. reset minTSOfTransientMessages; 7. if (CI_CISeqNum from a DRC is not ready) 8. return false; 9. GALT = min (all CI_CISeqNum); 10. GALTSeqNum=CISeqNum; 11. return true; } </pre>	<pre> void ReceiveCI(sendingFederate, CI_CISeqNum) { 12. if (CI_CISeqNum from a DRC is not ready) 13. return; 14. GALT = min (all CI_CISeqNum); 15. GALTSeqNum=CISeqNum; 16. ReplicationModule.GALTUpdated(); } </pre>
---	---

Figure 4.4: GALT calculation (TM Module)

As shown in Figure 4.4, a GALT calculation may be carried out by TM module in two cases: (i) receiving TMInfo from the replication module (CalculateGALT() function); and (ii) receiving CI reports from other federates (ReceiveCI() function). CISeqNum (or GALTSeqNum) keeps track of the number of sent CI reports (or calculated GALT values).

On receiving TMInfo, a new CI report may be created. The CI report with CISeqNum attached is denoted as CI_CISeqNum. In synchronous TM algorithm, it is unnecessary to create a new CI report, if the previous CI report has not been used for GALT calculation (i.e., if CISeqNum > GALTSeqNum (Lines 1 and 2 of Figure 4.4)). If a new CI report is created, it is sent to other federates by invoking their ReceiveCI() functions (Lines 3-6). In the case that CI reports with sequence number equal to CISeqNum have been received from all other federates, the value of GALT can be calculated by taking the minimum

value of the sending and receiving `CI_CISeqNum` (Lines 9-10) and the return value of `CalculateGALT()` function is true (Line 11). The calculated GALT value is denoted as `GALT_GALTSeqNum`, where `GALTSeqNum` equals `CISeqNum`. If the GALT value is not updated, the return value of `CalculateGALT()` function will be false (Lines 2 and 8). The transient message problem, which might happen in the synchronous TM mechanism, is solved using Grid service acknowledgements in a similar way to the Global Virtual Time (GVT) algorithm in [99]. The variable `minTSOfTransientMessages`, which keeps track of the minimum RVT of transient messages and retractions sent by the DRC, is considered when creating a CI report (Lines 3 and 6). A new value of GALT may be also calculated when a `CI_CISeqNum` is received from one of other federates (Lines 12-15). In this case, the TM module informs the replication module about the updated GALT by invoking `GALTUpdated()` function (Line 16).

The pseudo code of NMR service (refer to Section 1.1.2.2) is shown in Figure 4.5.a. As mentioned in case(ii) of Remark 4.1, after entering time advancing state, CON replica will not generate the same message as that generated by OPT replica with $RVT < TMInfo$ (i.e., $\min(ConReqTime, ConTSOHead) + LA$). Hence, the messages in `OptMsgFilter` with $RVTs < TMInfo + LA$ are only generated by the OPT replica. They are incorrect messages and can be retracted before the OPT replica invokes the corresponding retract services (Lines 6 and 7 of Figure 4.5.a). The replication module sends `TMInfo` to TM module for GALT calculations until either the GALT value is greater than `TMInfo` or GALT can no longer be updated (Lines 8-12). If the GALT value is greater than `TMInfo`, `ConLGT` and `LGT` are updated; `ConMsgFilter` and `OptMsgFilter` are reclaimed; and the NMR request is granted through a TAG callback (Lines 13-17). Unfortunately, the GALT calculation, as shown in Figure 4.4, might be blocked if CI reports from other federates are not ready. In this case, the GALT cannot be updated to a value greater than `TMInfo` immediately. As a result, the NMR request is simply returned without granting the

```

void NMR(t)
{
1.   if(ConInTAS)
2.       throw InTimeAdvancingStateException;
3.   ConReqTime=t;
4.   ConInTAS=true;
5.   TMInfo = min(ConReqTime, ConTSOHead);
6.   while(OptMsgHead<TMInfo + LA)
7.       Retract Msgs with RVT==OptMsgHead;
8.   do{
9.       if (GALT>TMInfo) {
10.           ConGranted(TMInfo);
11.           break;
12.       }
13.   } while(TMModule.CalculateGALT(TMInfo))
14. }

void ConGranted(t)
{
13.   ConLGT=t;
14.   LGT=max(ConLGT, OptLGT);
15.   reclaimMsgFilters(LGT);
16.   Insert TAG(ConLGT) into
    Non-message Callback Queue of CON;
17.   ConInTAS=false;
18. }

void GALTUpdated()
{
18.   if(ConInTAS) {
19-26 Line 5-12 in NMR(t)
20.   }
21. }

```

```

void FQR(t)
{
1.   if(OptInTAS==true)
2.       throw InTimeAdvancingStateException;
3.   OptReqTime=t;
4.   OptInTAS=true;
5.   if(RR callbacks are buffered) {
6.       OptGranted(OptLGT);
7.       return;
8.   }
9.   TMInfo = min(OptReqTime, OptTSOHead);
10.  do{
11.      if (GALT>=TMInfo) {
12.          OptGranted(TMInfo);
13.          return;
14.      }
15.  } while(TMModule.CalculateGALT(TMInfo))
16.  OptGranted(GALT);
17. }

void OptGranted(t)
{
15.   OptLGT=t;
16.   LGT=max(ConLGT, OptLGT);
17.   reclaimMsgFilter(LGT);
18.   insert TAG(OptLGT) into
    Non-message Callback Queue of OPT;
19.   OptInTAS=false;
20. }

```

4.5.a: Time advancement of CON replica

4.5.b: Time advancement of OPT replica

Figure 4.5: Time advancement of federate replicas (Replication Module)

time advancement request. While CON replica remaining in time advancing state, the ConReqTime is a constant value; whereas the ConTSOHead may change on receiving new messages and retractions. TM module waits for CI reports from the other federates for GALT calculation. Once the GALT value is updated, it informs the replication module by invoking GALTUpdated() function, whose pseudo code is also shown in Figure 4.5.a. When the GALT value is great enough, the pending time advancement request of CON replica can be granted. Based on the above discussion, we can have the following remark:

Remark 4.2 *The following must be true when NMR request is granted:*

$$ConLGT = \min(ConReqTime, ConTSOHead) \quad (4.3)$$

$$ConLGT < GALT \quad (4.4)$$

Figure 4.5.b shows the pseudo code of FQR service (refer to Section 1.1.2.2). In the case that there are some *RR* callbacks buffered in Non-message Callback Queue, the replication module delivers them to OPT replica in response to the FQR service, without advancing its logical time (Lines 5-7 of Figure 4.5.b). The replication module sends TMInfo (i.e., $\min(\text{OptReqTime}, \text{OptTSOHead})$) to TM module for GALT calculations (Lines 8-13). If $GALT \geq \text{TMInfo}$, OPT replica advances its logical time to the minimum of OptReqTime and OptTSOHead (Lines 16-20). Otherwise, instead of waiting for CI reports from other federates for further GALT calculations, the replication module simply grants OPT replica to advance its logical time to the current value of GALT. Thus we have the following remark:

Remark 4.3 *FQR request will be always granted immediately and when it is granted, the following must be true:*

$$OptLGT = \min(\text{OptReqTime}, \text{OptTSOHead}, GALT) \quad (4.5)$$

$$OptLGT \leq GALT \quad (4.6)$$

An example of TM mechanism is shown in Figure 4.6. When CON replica of **Federate1** invokes an NMR request, the replication module requests the TM module to calculate a new value of GALT, as $GALT_k-1 \leq \min(\text{ConReqTime}, \text{ConTSOHead})$. Since **CI_k** has been received from **Federate2**, **GALT_k** can be calculated immediately. As **GALT_k** is still not greater than TMInfo, the replication module requests for another GALT calculation.

Figure 4.6: An example of TM mechanism

However, the TM module cannot calculate GALT value before receiving CI_{k+1} of **Federate2**. Hence, the NMR request is returned and CON replica remains in time advancing state. After that, the $ConTSOHead$ might change on receiving some messages and retractions; $GALT_{k+1}$ is calculated by the TM module on receiving CI_{k+1} from **Federate2**. The new GALT value is passed to the replication module through $GALTUpdated()$. Since $GALT_{k+1}$ is now greater than $TMInfo$, the CON replica is granted to advance its logical time to $\min(ConReqTime, ConTSOHead)$. When OPT replica of **Federate1** invokes a FQR request, the replication module also requests for GALT calculation, as $GALT_{k+1}$ is smaller than $TMInfo$ of the OPT replica. Although the TM module cannot calculate $GALT_{k+2}$ immediately, the replication module can simply grant OPT replica to advance its logical time to $GALT_{k+1}$.

In summary, Un-Identical federate replication structure can enhance simulation performance as the faster replica is chosen to represent the federate. The messages are delivered to receiving federates when they are first generated by the faster replica. The

faster replica is also more likely to produce greater TMInfo for GALT calculation. The slower replica is able to get benefit from the replication structure and catch up with the faster one easier. If CON replica is slower, its time advancement requests might be granted immediately using the GALT value increased by OPT replica. If OPT replica is slower, it follows the execution of CON replica without suffering from any execution rollback. On the other side of the coin, the replication structure may introduce some overheads. For instance, message comparisons are needed before sending or discarding the messages generated by replicas.

4.1.3 Correctness Proofs

We first prove the correctness of the TM mechanism by ensuring that the TM module calculates GALT correctly and the replication module grants correct logical time to federate replicas. Then we prove the correctness of the message exchange mechanism by ensuring that the messages (correct messages, incorrect messages and retractions) are properly sent and received under the control of the TM mechanism.

4.1.3.1 Correctness of TM Mechanism

Lemma 4.1 *GALT value is calculated correctly.*

Proof: In Un-Identical replication structure, the GALT value is calculated as the minimum value of CI reports received from all the federates in the federation with the same sequence number (see Lines 9 and 14 in Figure 4.4). As proved below, the value of each CI report sent by a DRC is the lower bound of RVTs of future outgoing messages and retractions, if the DRC will not receive any new incoming message or retraction with $RVT < CI - LA$. Hence, the calculated GALT value represents the lower bound of RVTs of messages and retractions which might be generated in the entire federation.

Obviously, it is also the lower bound of RVTs of messages or retractions which might be received by the DRC in the future. Consequently, we can prove that GALT value is calculated correctly.

As described in Subsection 4.1.2, a CI report is created in the following two cases (see Figure 4.5): (i) TMInfo of CON replica is sent by replication module via CalculateGALT() request while processing NMR() or GALTUpdated() function; (ii) TMInfo of OPT replica is sent by replication module via CalculateGALT() request while processing FQR() function. For case (i), if the DRC will not receive any new incoming message or retraction with $RVT < CI - LA$, we can prove:

- (a) *CON replica will not generate any message with $RVT < CI$.* When the CI report is created, we can get (see Line 5 of Figure 4.5.a and Line 3 of Figure 4.4)

$$CI \leq \min(ConReqTime, ConTSOHead) + LA \quad (4.7)$$

The future messages generated by CON replica should have $RVTs \geq \min(ConReqTime, ConTSOHead) + LA$, if the DRC will not receive any new message or retraction with $RVT < CI - LA$. Hence, we can derive that CON replica will not generate any message with $RVT < CI$.

- (b) *Messages generated by OPT replica with $RVTs < CI$ are discarded.* If OPT replica is slower than CON replica, it might generate some messages with $RVTs < CI$. According to Inequality (4.7), these messages must have $RVTs < \min(ConReqTime, ConTSOHead) + LA$. If these messages are generated before the NMR request is granted, they are discarded (see case (ii) of Remark 4.1). Otherwise, if NMR is granted, $ConLGT = \min(ConReqTime, ConTSOHead)$ (see Remark 4.2). So, these messages are also discarded as their $RVTs < ConLGT + LA \leq LGT + LA$ (see case (i) of Remark 4.1).

- (c) *The DRC will not send out retractions with RVTs < CI.* As shown in Figure 4.5.a, the messages in OptMsgFilter with RVTs < min(ConReqTime, ConTSOHead) + LA are retracted before the CI report is created. Hence, we can get OptMsgHead $\geq$ min(ConReqTime, ConTSOHead) + LA $\geq$ CI. OptMsgHead $\geq$ CI is true as well because the messages generated by OPT replica with RVTs < CI are discarded (see part (b) above). Since a retraction is sent by the DRC only if there is a corresponding incorrect message kept in OptMsgFilter, any retraction sent by the DRC should have RVT $\geq$ OptMsgHead. Consequently, we can derive that DRC will never retract messages with RVTs < CI.

For case (ii), if the DRC will not receive any new incoming message or retraction with RVT < CI - LA, we can prove:

- (a) *OPT replica will not generate any message and retraction with RVT < CI.* When the CI report is created, we can get (see Line 8 of Figure 4.5.b and Line 3 of Figure 4.4))

$$CI \leq \min(\text{OptReqTime}, \text{OptTSOHead}) + LA \quad (4.8)$$

Since the buffered retractions have been handled (see Lines 5-7 of Figure 4.5.b) and the DRC will not receive any new messages or retractions with RVT < CI - LA, we can get that OPT replica will never receive any message or retraction with RVT < CI - LA. Hence, OPT replica will not send any message or retraction with RVT < CI. In other words, OPT replica has generated all correct messages with RVTs < CI and retracted all incorrect messages with RVTs < CI.

- (b) *Messages generated by CON replica with RVTs < CI are discarded.* These messages are correct messages. As described in part(a), the same messages must have been generated by OPT replica previously. They have been sent to receiving federates

and are kept in OptMsgFilter. Hence, these message generated by CON replica are discarded either because their RVTs $< \text{LGT} + \text{LA}$ or because the same messages are found in OptMsgFilter.

- (c) *DRC will not send out retractions with RVTs $< \text{CI}$.* Again, as described in part(a), all incorrect messages with RVTs $< \text{CI}$ have been retracted by OPT replica.

In summary, the CI report sent by the DRC in either case (i) or case (ii) is correct. It represents the lower bound of RVTs of future messages and retractions sent by the DRC, if the DRC will not receive any new message or retraction with $\text{RVT} < \text{CI} - \text{LA}$. $\square$

Since GALT value is calculated correctly, we can derive the following lemma:

Lemma 4.2 *Time advancement requests from replicas are granted correctly. That is, no message with $\text{RVT} \leq \text{ConLGT}$ will be delivered to CON replica in the future; no message or retraction with $\text{RVT} < \text{OptLGT}$ will be delivered to OPT replica in the future.*

4.1.3.2 Correctness of Message Exchange Mechanism

Lemma 4.3 *For a sending federate, one and only one instance of each correct message is sent by the DRC; each incorrect message and its corresponding retraction are either both sent or both discarded by the DRC.*

Proof: Since replicas of the same federate are committed equivalent, each correct message should be generated by both CON and OPT replicas. In the case that the correct message is generated by CON replica first, we can get its $\text{RVT} \geq \text{ConLGT} + \text{LA}$. Since OPT replica will generate the same message some time later, we can also get its $\text{RVT} \geq \text{OptLGT} + \text{LA}$. Hence, the message will be sent by the DRC to the receiving federates as its $\text{RVT} \geq \text{LGT} + \text{LA}$ and the same message must not be found in the OptMsgFilter. The same message generated by OPT replica some time later will be discarded either

because one of the cases in Remark 4.1 is true or because the same message can be found in the ConMsgFilter.

In the case that the correct message is generated by OPT replica first, we can similarly get that its $RVT \geq LGT + LA$. It is also obvious that the condition $RVT < \min(\text{ConReqTime}, \text{ConTSOHead}) + LA$ is not true (otherwise, CON replica would have generated the same correct message previously). So, the message will be sent out by the DRC to the receiving federates as the conditions for both cases in Remark 4.1 are not true and the same message must not be found in the ConMsgFilter. The same correct message generated by CON replica some time later will be discarded either because the first case of Remark 4.1 is true or because the same message can be found in the OptMsgFilter.

In summary, one and only one instance of each correct message is sent by the DRC, regardless that it is first generated by CON or OPT replica.

If an incorrect message generated by OPT replica is discarded by the DRC, it will not be buffered in the OptMsgFilter. Consequently, the DRC will never send out its corresponding retraction. If an incorrect message generated by OPT replica is sent by the DRC, it will be kept in OptMsgFilter and its corresponding retraction must be sent before LGT (i.e., either ConLGT or OptLGT) becomes greater than its $RVT - LA$. Since OPT replica must have processed all the events with TS smaller than OptLGT correctly, it must have retracted all the incorrect messages with $RVT < \text{OptLGT} + LA$. As shown in Figure 4.5.a (Lines 6-8), before ConLGT increases to $\min(\text{ConReqTime}, \text{ConTSOHead})$, messages kept in OptMsgFilter are identified as incorrect messages and are retracted if their $RVT < \min(\text{ConReqTime}, \text{ConTSOHead}) + LA$. Hence, an incorrect message, if kept in OptMsgFilter, must have been retracted before ConLGT becomes greater than its $RVT - LA$.

In summary, the incorrect message and its corresponding retraction are either both sent or both discarded by the DRC. $\square$

Lemma 4.4 *For a receiving federate, correct messages are delivered to both replicas; incorrect messages and their corresponding retractions are only sent to OPT replica rather than CON replica.*

Proof: The correct messages received by the DRC are buffered in TSO queues of CON and OPT replicas. As a result, CON and OPT replicas will receive the same correct messages. The incorrect messages received by the DRC are also buffered in TSO queues of the CON and OPT replicas. According to Lemma 4.1 and 4.3, we can deduce that the retractions of an incorrect message must have been received by the DRC when GALT becomes greater than its RVT. Since only those messages with $RVT \leq ConLGT$ in ConTSOQueue are eligible to be delivered to the CON replica and $ConLGT < GALT$ (see Remark 4.2), we can deduce that incorrect messages will not be delivered to CON replica. Hence, all incorrect messages are annihilated with their retractions instead of being delivered to CON replica. In contrast, the incorrect messages kept in OptTSOQueue might be delivered to OPT replica. For an incorrect message, the DRC must have received its corresponding retraction before GALT becomes greater than its RVT. Hence, the retraction of an incorrect message must have been already delivered to the OPT replica when its OPTLGT ($OPTLGT \leq GALT$, according to Remark 4.3) is greater than the RVT of the message. Hence, the effects of incorrect messages are removed from OPT replica. $\square$

Based on Lemma 4.1 to 4.4, we can then have the following Theorem:

Theorem 4.1 *With the same set of federates, the federation executed on Un-Identical federate replication structure exhibits the same behavior as that executed on non-replication structure.*

Proof: According to Lemma 4.3 and 4.4, in Un-Identical federate replication structure, correct messages are delivered to CON and OPT replicas of receiving federates without message loss and duplication. Although some of the incorrect messages might be delivered to OPT replicas, they will be retracted. Hence, the committed events processed in CON and OPT replicas of the same federate are the same correct events. Since the replicas are committed equivalent and advance their logical time correctly (see Lemma 4.2), they process committed events in the same manner following the same TS order as the federate executes on the non-replication structure. So, with the same set of federates, the federation on the Un-Identical federate replication structure and that on the non-replication structure exhibit the same execution behavior. $\square$

4.2 Experiments and Results

Experiments are carried out to verify that the Un-Identical replication structure can enhance performance while ensuring the correctness of the simulation executions. Besides that, the issues of scalability are also studied.

4.2.1 Configuration of Experiments

A synthetic P-Hold simulation model [44] is used in our experiments. It is composed of two symmetric federates, which are denoted as **Federate1** and **Federate2**. Initially, each federate has an internal event. When processing an internal event, in addition to generate another internal event, the federate may send an external event to the other federate. The probability of generating an external event is denoted as $P_{ExternEvent}$. In the case that an external event is received, the federate just processes the external event without generating either an internal or external event. Since Grid service invocations usually introduce considerable overheads, most of the simulations executed over the Grid are

coarse-grained. Therefore, spin-loops are added in the P-hold simulation to emulate the event processing time. Similarly, spin-loops are also added to emulate the state saving and restoration overheads in OPT federates.

With an increasing $P_{ExternEvent}$, events are exchanged more frequently between federates, and thus, the probability of execution rollbacks in OPT federates increases. In addition to $P_{ExternEvent}$, other parameters of the P-hold simulation, e.g., simulation time of generated internal and external events, LAs of federates, event processing time and federate state saving and restoration time in OPT federates, also affect the performance of conservative and optimistic synchronization approaches. The default setting of the parameters is presented as follows.

It is meaningless to run fine-grained simulations over Grid environment, as the communication overhead is too large. Hence, a coarse-grained P-hold benchmark is used in our experiments. The parameter settings for event processing time and state saving time are close to the real world coarse-grained simulation models described in [27]. The spin-loop for processing an internal event is set to 300 milliseconds, while the spin-loop for processing an external event is set to 100 milliseconds. The spin-loops for state saving or restoration in OPT federates are set to 50 milliseconds¹. The simulation time of the generated internal event equals the TS of the processed internal event plus a random number between 10 and 110 following uniform distribution. The external event is sent using an interaction with RVT equal to TS of the processed internal event plus LA. The LAs of both federates are set to 1. The termination time for both federates are set to 12,000 simulation units.

To meet committed equivalent condition, replicas of the same federate should use pseudo-random number generators with the same seed. The simulation starts with an

¹For simplification, we do not consider the secondary performance effects related to the change of locality proper of contexts where state saving relies on real memory copy operations.

initialization procedure which includes create/join the federation, enable time regulating/constrained, and publish/subscribe the interaction. After that an execution barrier is used to ensure all federate replicas enter the simulation procedure at the same time. At the end of the simulation, each federate outputs its state, e.g., the number of processed events and the TS of processed events. In the following experiments we are only concerned with the execution time of simulation procedure.

Three execution scenarios are investigated in our experiments. In the Conservative and Optimistic scenarios, federates are not replicated. They are executed on the original SOHR without replication structure. In the Conservative scenario, federates employ conservative synchronization; and in the Optimistic scenario, federates employ optimistic synchronization. In the Replicated scenario, each federate has two replicas: CON and OPT replicas. They are executed concurrently on SOHR with the Un-Identical federate replication structure.

The experiments are carried out in the same cluster as that described in Subsection 3.2.1, with SOHR deployed in the same manner. Depending on the execution scenarios, the DRC (i.e., LIS/LRI) deployed on separate computing nodes may be connected with a CON replica, an OPT replica, or both CON and OPT replicas of a federate. Each replica executes on a separate computing node. For each execution scenario, the federation execution is repeated five times. Since the execution time of the repeated executions have small variance, only the average execution time is reported in the experiment results.

4.2.2 Performance Enhancement of Un-Identical Replication Structure

Two series of experiments are carried out to illustrate the advantages of the Un-Identical replication structure in terms of performance enhancement. In the first series of ex-

4.7.a: Constant $P_{ExtrenEvent}$ 4.7.b: Variable $P_{ExtrenEvent}$

Figure 4.7: Execution times in different scenarios

periments, the $P_{ExtrenEvent}$ value is a constant value during the simulation execution. As shown in Figure 4.7.a, the Optimistic scenario outperforms the Conservative scenario when $P_{ExtrenEvent} < 50\%$. The reason is that the federates seldom send external events to each other and only a small number of rollbacks occur in both federates. So, the optimistic approach can explore parallelism more effectively than conservative one. With an increasing $P_{ExtrenEvent}$, the performance advantage of the Optimistic scenario decreases. When $P_{ExtrenEvent} = 50\%$, Conservative and Optimistic scenarios have comparable performance. When $P_{ExtrenEvent} > 50\%$, Optimistic scenario performs worse than the Conservative scenario due to the frequent execution rollbacks. The most important observation is that the Replicated scenario can outperform both the Conservative and Optimistic scenarios. For extreme cases, the Replicated scenario performs similarly to the Optimistic scenario when $P_{ExtrenEvent}=0\%$, and performs similarly to the Conservative scenario when $P_{ExtrenEvent}=100\%$.

As shown in the first series of experiments, Conservative scenario and Optimistic scenario can outperform each other depending on the value of $P_{ExtrenEvent}$. If the value of $P_{ExtrenEvent}$ varies in the same execution, the Optimistic scenario can outperform the

Table 4.2: Variable $P_{ExternEvent}$

Cases	Change patterns of $P_{ExternEvent}$	Description
1	0% , 25% , 50% , 75% , 100%	Increasing $P_{ExternEvent}$
2	100% , 75% , 50% , 25% , 0%	Decreasing $P_{ExternEvent}$
3	0% , 100% , 0% , 100% , 0%	Alternative $P_{ExternEvent}$
4	0% , 50% , 100% , 50% , 0%	Highest $P_{ExternEvent}$ in the middle
5	100% , 50% , 0% , 50% , 100%	Lowest $P_{ExternEvent}$ in the middle

Conservative scenario during one execution phase, while the Conservative scenario can outperform the Optimistic scenario during another execution phase.

In the second series of experiments, we measure the execution times of the federation with variable $P_{ExternEvent}$, which is changed every 2,400 simulation units (i.e., one fifth of the termination time of the P-hold simulation). Five cases of variable $P_{ExternEvent}$ are investigated, as shown in Table 4.2. The execution times of different execution scenarios are shown in Figure 4.7.b. Compared to the better result between the Conservative and Optimistic scenarios, the simulation performance is enhanced from 10% to 16% in the Replicated scenario.

4.2.3 Scalability of Un-Identical Replication Structure

Two series of experiments are carried out to study the scalability of the Un-Identical replication structure with the federation scale increased from two to eight. Constant $P_{ExternEvent}$ (equal to 50%) is used in the simulation executions. The external events are sent to all other federates in the federation. In the first series of experiments (i.e., separated DRC series), DRCs of federates are executed on separate computing nodes. In the second series of experiments (i.e., congested DRC series), DRCs of all federates in the federation are executed on a single computing node. Figures 4.8 and 4.9 present the performance results in these two series of experiments respectively. Besides simula-

tion execution time, the numbers of execution rollbacks and GALT calculations are also counted.

4.8.a: CPU utilization

4.8.b: Federation execution time

4.8.c: Execution rollbacks (Federate1)

4.8.d: GALT calculations

 Figure 4.8: Scalability of federate replication structure in **separated** DRC series

The messages and CI reports are delivered through Grid service invocations, which have considerable overhead. According to our experience, a Grid services invocation inside our cluster takes around eight milliseconds. The major overhead of the Grid service invocation in our case is the overheads caused by protocol stacks, since the communication cost is low due to the fast intra-cluster network connections. With the increasing federation scale, the numbers of messages exchanged among federates, the number of GALT calculations, and the number of CI reports for each GALT calculation increase accordingly. The larger the federation scale, the more resources will be required by each

DRC. In the Replicated scenario, a DRC is connected with CON and OPT replicas. Hence, it requires even more resources than that in the Conservative and Optimistic scenarios. According to our observation, the workload of the computing node with a single DRC deployed in the separated DRC series is acceptable. Its CPU utilization, as shown in Figure 4.8.a, is less than 30% in the Replicated scenario when the federation scale is equal to eight. In contrast, the workload of the computing node with multiple DRCs deployed in the congested DRC series increases significantly with increasing federation scale, as shown in Figure 4.9.a.

The execution times of different scenarios in separated DRC series are shown in Figure 4.8.b. As we can see, the Replicated scenario outperforms other scenarios by 11% to 17%. The execution times in the Conservative and Optimistic scenarios increase linearly with respect to the federation scale, and the same trend can be observed for the Replicated scenario. Figure 4.8.c reports the number of execution rollbacks in **Federate1**. As we can see, OPT replica in the Replicated scenario encounters fewer rollbacks compared with the OPT federate in the Optimistic scenario. Figure 4.8.d reports the numbers of GALT calculations conducted in different scenarios. As we can see, the Replicated scenario requires fewer GALT calculations compared with the Conservative scenario.

The execution times of different scenarios in the congested DRC series are shown in Figure 4.9.b. The execution time in the Conservative scenario increases faster than the Optimistic scenarios when the federation scale increases. As shown in Figure 4.9.d, the numbers of GALT calculations in the Conservative scenario are much larger than those in the Optimistic scenario. As a result, the workload of the computing node where the DRCs are deployed increases more significantly in the Conservative scenario than that in the Optimistic scenario, as shown in Figure 4.9.a. The DRCs on the overloaded computing node will spend more time to process each Grid service invocation. Hence, the Conservative scenario performs worse than the Optimistic scenario in large-scale

4.9.a: CPU utilization

4.9.b: Federation execution time

4.9.c: Execution rollbacks (Federate1)

4.9.d: GALT calculations

 Figure 4.9: Scalability of federate replication structure in **congested** DRC series

federation executions.

In congested DRC series, the Replicated scenario can outperform the Conservative and Optimistic scenarios when the federation scale is smaller than seven. But it performs worse than Optimistic scenario when the federation scale is equal to seven or eight. Compared to the Optimistic scenario, each federate executed in the Replicated scenario has one more CON replica. As a result, greater number of GALT calculations are required (as shown in Figure 4.9.d) which consume more resources in DRCs (as shown in Figure 4.9.a).

We have further investigated the execution of **Federate1** with federation scale equal to eight in the Replicated scenario. $ConTime_i$ (or $OptTime_i$) records the wall clock time of the computing node when CON replica (or OPT replica) processes the i^{th} committed

Figure 4.10: $\Delta Time$ in separated and congested DRC series.

event. In the same cluster, the computing nodes have synchronized time clocks. $\Delta Time_i$ is defined as $ConTime_i - OptTime_i$. In the case that $\Delta Time_i > 0$, the i^{th} committed event is processed earlier in OPT replica. Otherwise, it is processed earlier in CON replica. Figure 4.10 shows $\Delta Time_i$ of 200 successive committed events. In separated DRC series of experiments, CON and OPT replicas outperform each other frequently during the execution. In contrast in congested DRC series of experiments, OPT replica outperforms the CON replica during almost the entire execution.

As illustrated in Figure 4.10, CON replica falls behind OPT replica during almost all execution phases in large-scale federation executions in congested DRC series of experiments. Consequently, OPT replica is difficult to get performance benefits from CON replica. This also explains why OPT replica in the Replicated scenario generates more execution rollbacks in congested DRC series (Figure 4.9.c), compared with that in separated DRC series (Figure 4.8.c) when federation scale becomes large.

The congested DRC series of experiments show that Replicated scenario might perform worse than the scenarios without federate replication in some situations. It is not suggested to use the Un-Identical replication structure, if DRCs are deployed on overloaded computing nodes and if one replica can hardly catch up with the other one in the

entire execution.

4.3 Summary

This chapter proposes an Un-Identical replication structure to support concurrent executions of CON and OPT replicas of the same federate. It is composed of the message exchange mechanism and the TM mechanism. The former is in charge of message exchange amongst federates and federate replicas; and the latter performs federation wide time synchronization and coordinates the time advancement of federate replicas. The simulation performance is improved by choosing the fastest replica to represent the federate in the federation dynamically and automatically during the execution, taking the advantages of different synchronization approaches. In addition to presenting implementation details, a proof of correctness is also provided. As experimental results show, the Un-Identical replication structure enhances performance and also has good scalability. As observed, all execution scenarios produce exactly the same simulation results in every single experiment reported in this chapter. This double-confirms the correctness of our Un-Identical federate replication structure.

Chapter 5

Checkpoint-based Crash-stop Fault Tolerance Mechanism

To support fault tolerant HLA-based simulation, one of the objectives of this thesis is to tolerate the crash-stop failures which may occur on either CON or OPT federates. By exploiting the decoupled federate architecture, federate failures can be masked from the federation and recovery can take place without interrupting the executions of other federates. A basic state recovery protocol is proposed first to recover the state of the failed federate relying on the checkpoint and message logs taken before the failure. Then, an optimized protocol is developed further to accelerate the state recovery procedure. To demonstrate the overhead of proposed fault tolerance mechanism and illustrate the advantages of the optimized state recovery protocol over the basic one, experiments are carried out using a synthetic P-hold simulation model.

5.1 Federate Fault Tolerance Mechanism

In the traditional federate architecture, we cannot separate the failures of a federate from those of its LRC, and vice versa. When a federate or its LRC crashes, the federate is lost from the federation. Other federates in the federation cannot exchange messages with the federate any more. The federate must re-join the federation and build up connections to other federates through a new LRC. To ensure the consistency of the federation execution, both the federate state and LRC state lost during the failure should be restored. What is more, some messages exchanged among federates may also be lost due to the failure [2, 31, 35]. Hence, a mechanism is needed to ensure the correctness of message delivery in the presence of federate failures.

Using the decoupled federate architecture, the failures of a federate and its DRC can be isolated and handled separately. In this thesis, we assume that the DRC is robust. It can survive from the failures which occur on the federate and its underlying execution platform. To tolerate crash-stop failures of the federate, a Fault Tolerance (FT) module is developed and embedded in the DRC (refer to Figure 1.5). As described below, the federate failures can be detected by the FT module within a reasonable time. The FT module also plays an important role in the failure recovery. It provides a reliable storage to save checkpoints of the federate. It can also log all the incoming messages (callbacks) and outgoing messages (RTI service calls) of the federate. Relying on the checkpoint and message logs, the message and execution state lost during the failure can be recovered. The DRC keeps the connection between the federate and the federation. It also maintains the RTI state concerned with the federate. Hence, the federate can be reconnected to the federation through the same DRC.

Some components described in Subsection 3.1, i.e., the workload balance module, federate server, and Grid FTP server, are provided in the simulation execution environment

to support federate migration. These components also are needed in our federate fault tolerance mechanism.

Since failure detection is not our major research focus, we simply apply a heartbeat-based failure detection method [104] in our fault tolerance mechanism. The federate server sends heartbeats of federates located in the computing node to their DRC periodically. The FT module in the DRC monitors the heartbeats. If a heartbeat of a federate is not received within a specified time period, the federate is determined as crashed. The heartbeat-based failure detection is unreliable, because a federate can be erroneously determined as crashed if its heartbeat is simply delayed. In this case, the federate will be still treated as crashed and will be terminated by the federate server. Our experiments are conducted in a cluster with stable network connections among computing nodes. A heartbeat is sent by a federate every three seconds. A federate is determined as crash, if the DRC does not receive a new heartbeat of the federate in five seconds.

Figure 5.1: Checkpoint-based federate fault tolerance mechanism

An overview of the proposed checkpoint-based federate fault tolerance mechanism is shown in Figure 5.1. At the beginning of a federation execution, a copy of static files for each federate, such as federate code files and the FDD (Federation Object Model Document Data) file [56], is sent to its DRC and kept in the FT module. During its execution, the federate is requested to take its checkpoint periodically and the checkpoint is reported to its DRC and buffered in the FT module. Hence, the static files and the checkpoint of the federate are available in the DRC when the federate crashes. When a failure is detected, the workload balance module determines where to start a new federate (**RecoveredFed**) to substitute the crashed one (**CrashedFed**) in the remaining federation execution. The **CrashedFed** and **RecoveredFed** are connected to the federation through the same DRC. They are the same federate in the federation. In this way, the DRC with the FT module can mask the federate failure from the federation. Furthermore, the failure detection and recovery procedures are transparent to other federates in the federation.

The static files and the checkpoint of **CrashedFed** kept in the FT module of the DRC are uploaded to the GridFTP server deployed in the computing node where **RecoveredFed** is to be started. **RecoveredFed** has the same implementation as **CrashedFed**, as they have the same static files. However, the state of **RecoveredFed**, restored from the checkpoint, will generally be outdated. We can easily get Inequality (5.1), in which *checkpoint time* refers to the logical time of the federate when the checkpoint was taken; *failure time* refers to the logical time of the federate when the failure occurred.

$$Checkpoint\ time \leq Failure\ time \tag{5.1}$$

Before substituting **CrashedFed** in the federation, **RecoveredFed** restored from the checkpoint must repeat some federate execution which was conducted by **CrashedFed** before the failure. In the following subsections, a basic state recovery protocol is first proposed, in which **RecoveredFed** repeats the execution identically using the message log taken in

the FT module embedded in the DRC. Then, an optimized protocol is further developed to accelerate the state recovery procedure.

5.1.1 Basic State Recovery Protocol

In HLA-based simulations, federate state is composed of application specific state and simulation engine state. Application specific state includes data structures implemented by the simulation model. For the simulation engine state, different data structures are used in CON and OPT federates. In a conservative federate, an eventlist is used to keep events that will occur in the future. In an optimistic federate, an input queue is used to keep the processed and unprocessed events. When an event is rolled back, a mechanism is required to undo modifications to application specific state and unsend messages scheduled by the event. To support the recovery of application specific state, a snapshot of the state associated with an event just before it is processed is saved in a state queue. Each snapshot is associated with a TS which is the current simulation time when the snapshot is taken. As shown in Figure 5.2 (a), before processing event **E21**, the simulation time of the federate is 12. The snapshot of application specific state **S12** is saved in the state queue and associated with **E21**. To support unsending messages, an output queue is used to store corresponding anti-messages (i.e., retractions) for the messages generated by each processed event.

In the basic protocol, the checkpoint is taken by saving the entire federate state, including application specific state, and the entire simulation engine state. **RecoveredFed** restored from the checkpoint obtains the same federate state as that of **CrashedFed** when it took the checkpoint. After that, the FT module delivers the same callbacks to **RecoveredFed** following the same sequence as those delivered to **CrashedFed**. Similar to the relay-based migration protocol proposed in Chapter 3, the execution of the feder-

ate is assumed to be Piecewise Deterministic [48]. Hence, the execution conducted by **CrashedFed** from taking the checkpoint to encountering the failure is repeated by **RecoveredFed** identically. The logical time of **RecoveredFed** is increased from the checkpoint time to the failure time accordingly. During the repeated execution, **RecoveredFed** will invoke the same RTI services following the same sequence as those invoked by **CrashedFed**. As described below, the DRC will not handle these duplicate RTI service invocations.

The basic state recovery protocol performs in the same manner regardless of the synchronization approach employed by the federate. The data structure of FT module is similar to that of the Migration module shown in Figure 3.5 in Chapter 3. Two service counters are used to record the number of services invoked by **CrashedFed** and **RecoveredFed** respectively. The service buffer is used to buffer the service calls invoked by **CrashedFed**, represented by the service sequence number and its return value. The callback buffer is used to buffer the callbacks delivered to **CrashedFed**, with the service counter of **CrashedFed** attached. When a checkpoint is reported by the federate, the checkpoint maintained in the DRC is updated. During the meantime, the service counters of **CrashedFed** and **RecoveredFed** are set to 0, and the service buffer and callback buffer are cleared.

During the repeated execution, the FT module delivers callbacks in the callback buffer to **RecoveredFed** if their sequence numbers are equal to the service counter of **RecoveredFed**. In this way, **RecoveredFed** recovered from the latest checkpoint will identically repeat the execution conducted by **CrashedFed** before the failure. In the case that the service counter of **RecoveredFed** is smaller than or equal to that of **CrashedFed**, the RTI service invoked by **RecoveredFed** must have been invoked by **CrashedFed**. Instead of handling the RTI service call again, the corresponding return value kept in the service buffer is returned directly. Otherwise, **RecoveredFed** has finished the repeated execution and is ready to participate in the federation execution.

5.1.2 Optimized State Recovery Protocol

The optimized protocol is developed to accelerate the state recovery procedure by using the information kept in the DRC (e.g., failure time of **CrashedFed** and the callbacks delivered to **CrashedFed** before the failure). The optimized protocol described below focuses on RTI service and callbacks concerned with federate time advancement and message exchange¹. Depending on the synchronization approach employed by the federate, these RTI service and callbacks are handled in different manners, and thus, the optimized protocol performs differently.

For a CON federate, the checkpoint is the same as that taken in the basic protocol. It includes a snapshot of the current application specific state and a copy of the eventlist in the simulation engine. However, the repeated execution is conducted in the **RecoveredFed** in a manner different from that in the basic state recovery protocol. **RecoveredFed** restored from the checkpoint can be granted to failure time directly, without repeating a number of time advancement requests and the corresponding TAG callbacks as in the basic state recovery protocol. The incoming messages, in the form of callbacks, delivered to **CrashedFed** after the checkpoint are buffered in the FT module in the DRC. The buffer callbacks whose RVT is greater than checkpoint time and is smaller than or equal to failure time are bundled together and delivered to **RecoveredFed** at one time. **RecoveredFed** restored from the checkpoint is able to conduct the repeated execution by processing events between checkpoint time and failure time in TS order. During the repeated execution, the outgoing messages sent by **RecoveredFed** are filtered out by the RTI interface instead of being delivered to the DRC. This is because the same messages must have been already sent by **CrashedFed** before the failure. After processing all events with $TS \leq \text{failure time}$, **RecoveredFed** is ready to substitute **CrashedFed** in the federation

¹Other RTI services and callbacks are handled in a similar manner as the basic state recovery protocol

execution.

For an OPT federate, the checkpoint is different from that taken in the basic protocol. As mentioned above, the latter is a copy of the entire federate state, including the current application specific state and the simulation engine state. As a part of the simulation engine state, the state queue with multiple snapshots of application specific state is saved in the checkpoint. Consequently, the size of the checkpoint can be huge and considerable overhead may be introduced while the checkpoint is taken and reported to the DRC. The snapshots of application specific state kept in the checkpoint are unsafe if their $TS \geq$ checkpoint time. They will be discarded if the federate receives an event with smaller TS . In the optimized protocol, the checkpoint includes one snapshot of application specific state only, i.e., the latest snapshot kept in the state queue with $TS <$ checkpoint time. Hence, this snapshot of application specific state in the checkpoint is safe and the size of the checkpoint is also greatly reduced. Besides this snapshot, the events in the input queue are also saved in the checkpoint except for the following cases: (i) the events with $TS <$ checkpoint time; (ii) the events that are received from other federates with $TS \geq$ checkpoint time; and (iii) the events that are generated by the federate while processing events with $TS \geq$ checkpoint time. The events in case (i) are correct events processed by **CrashedFed** before taking the checkpoint. The events in cases (ii) and (iii) are received or generated by the federate optimistically. They are not included in the checkpoint because **RecoveredFed** will repeat the execution from the checkpoint time as explained below. Hence, only the events generated by the federate while processing events with $TS <$ checkpoint time are saved in the checkpoint.

Similar to the recovery of a CON federate, **RecoveredFed** of the OPT federate restored from the checkpoint can be granted to failure time directly. **RecoveredFed** restored from the checkpoint starts its execution from the checkpoint time, i.e., the simulation time of **RecoveredFed** is equal to checkpoint time. That is, all events with $TS \geq$ checkpoint time

will be processed by **RecoveredFed** after the state restoration. This is different from the basic state recovery protocol, where **RecoveredFed** starts its execution from the simulation time of **CrashedFed** when the checkpoint was reported².

The incoming messages, in the form of callbacks, with $TS \geq$ checkpoint time are saved in the callback buffer in the FT module when they are delivered to **CrashedFed**. A callback is removed from the callback buffer if its corresponding retraction is also delivered to **CrashedFed**. Upon a failure, **RecoveredFed** is established and recovered from the checkpoint. After that, the FT module sorts the remaining callbacks in the callback buffer in TS order and delivers them to **RecoveredFed** at one time. In this way, incorrect messages with TS in the interval [checkpoint time, failure time) and their retractions are not delivered to **RecoveredFed**; correct messages with TS in the same interval are delivered to **RecoveredFed** in TS order. Consequently, **RecoveredFed** is able to process events between checkpoint time and failure time without any rollbacks and increase its logical time to failure time directly.

The outgoing messages generated by **RecoveredFed** while processing events with $TS <$ failure time are filtered out by the RTI interface, because **CrashedFed** must have processed these events and sent the same messages correctly before the failure. In addition, some optimistic messages might be generated by **CrashedFed** while processing events with $TS \geq$ failure time. These messages are buffered in FT module of the DRC. A mechanism similar to lazy cancelation [46, 47, 113] is applied to handle these optimistic messages. Messages generated by **RecoveredFed** are compared with the optimistic messages generated by **CrashedFed**. If duplicates are found, they are simply discarded. Otherwise they are sent by the DRC. The optimistic messages generated by **CrashedFed** are retracted, if it can be ensured that **RecoveredFed** will never generate the same message in the future

²Generally, the simulation time is greater than the checkpoint time. However, **RecoveredFed** which starts its execution from the simulation time may rollback its execution to the checkpoint time in the future in worst case.

Table 5.1: Advantages of optimized protocol

Advantages of optimized protocol	CON Federate	OPT Federate
Advance to failure time directly	✓	✓
Receive buffered messages at one time	✓	✓
Avoid sending messages during repeated execution	✓	✓
Reduce overhead of taking checkpoint	N/A	✓
Process events with $TS \geq$ checkpoint time and $TS <$ failure time without any rollbacks	N/A	✓

(i.e., if RVT of the optimistic message $<$ logical time + LA of RecoveredFed).

The advantages of the optimized protocol over the basic one are shown in Table 5.1, some of which are only applicable to OPT federates.

An example is given to illustrate the failure recovery of an OPT federate using the basic and optimized state recovery protocols. In Figure 5.2, we have traced the execution of a federate from taking the checkpoint to encountering the failure. For clarity of illustration, fossil collection is not shown in the figure. Events, states, messages, and retractions shown in the figure are time stamped (e.g., E12, S9, M19, and R21). It is assumed that all events saved in the input queue are external events. As shown in Figure 5.2(a), the checkpoint is taken by the federate after processing event E35. The checkpoint time, i.e., the logical time of the federate when the checkpoint is taken, is equal to 12. When the straggler message with $TS = 18$ is received, the application specific state is rolled back to S12; M42 which was sent previously is retracted; and finally E18 is inserted in the input queue (see Figure 5.2(b)). After that, the federate processes events E18, E21, and E35. M44 is sent by the federate while processing event E35 (see Figure 5.2(c)). The federate experiences from another rollback, when the retraction R21 is received. It rolls back its application specific state to S18; retracts M44; and removes E21 from the input queue (see Figure 5.2(d)). After processing events E35 and E41,

Figure 5.2: Example of a crashed OPT federate

the federate crashes ((see Figure 5.2(e))). The failure time, i.e., the logical time of the federate when the failure happens, is equal to 36. To compare the basic and optimized protocols, Table 5.2 and Table 5.3 are provided, which describe the federate's checkpoint and repeated execution respectively.

In the basic protocol, the checkpoint includes the entire federate state. As shown in Table 5.2, a copy of the current application specific state and a copy of the simulation engine state are saved in the checkpoint. The size of checkpoint can be huge due to the multiple snapshots of application specific state (i.e., S9, S12, S21, and S35). After recovering from the checkpoint, **RecoveredFed** gets the same callbacks and invokes the same RTI services following the same sequence as those of **CrashedFed**. It experiences

Table 5.2: Federate checkpoint

Protocols	Data structure		Content
Basic	Application specific state		S35
	Simulation engine state	Input Queue	E12, E21, E35, E41
		State Queue	S9, S12, S21
		Output Queue	M19, M42
Optimized	Application specific state		Not Required
	Simulation engine state	Input Queue	Empty
		State Queue	S9
		Output Queue	Not Required

Table 5.3: Repeated federate execution

Protocols	Rollbacks	Events	Messages	Retractions	FQR
Basic	M18, R21	E18, E21, E35, E35, E41	M44, M46, M57	R42, R44	Multiple
Optimized	None	E12, E18, E35, E41	M57	None	One

from rollbacks, when the straggler messages M18 and the retraction R21 are delivered. Before advancing to failure time, **RecoveredFed** invokes a number of FQR requests and gets corresponding TAG callbacks. During the repeated execution, **RecoveredFed** sends messages (M44, M46, and M57) and retractions of messages (R42 and R44) to DRC. As explained above, these messages are simply discarded by the DRC since they have been previously sent by **CrashedFed**. After processing E41, **RecoveredFed** can substitute **CrashedFed** in the federation execution.

In the optimized protocol, the size of the checkpoint is reduced sharply. Only a safe snapshot of application specific state (i.e., S9) is saved in the checkpoint. The current application specific state and the output queue are not saved in the checkpoint. The events in the input queue are not saved in the checkpoint, because they are external events with $TS \geq$ checkpoint time. Before the failure, messages received with $TS \geq$ checkpoint

time are kept in the callback buffer in DRC. They are M12, M18, M35, and M41. M21 is not in the callback buffer because of the retraction R21. After `RecoveredFed` restores from the checkpoint, callbacks (M12, M18, M35, and M41) are delivered to `RecoveredFed` in TS order. In this way, `RecoveredFed` will not experience from the rollbacks during its repeated execution from checkpoint time to failure time and it can be granted to advance its logical time to 36 (i.e., the failure time) directly. It needs not send messages to the DRC while processing events E12 and E35, since the corresponding messages (i.e., M19 and M46 which are correct messages) have been already sent to other federates by the DRC. `RecoveredFed` sends a message M57 to DRC on processing E41. However, as explained above, the message will be discarded.

5.2 Experiments and Results

5.2.1 Experiment Design

The federate fault tolerance mechanism proposed in this chapter is implemented based on the decoupled federate architecture of SOHR [86, 87]. In order to evaluate the proposed mechanism, experiments are carried out using a P-hold simulation model described in Subsection 4.2.1. The simulation executions are conducted on the cluster described in Subsection 3.2.1, with SOHR deployed in the same manner. In addition, we assume that the total size of static files of each federate is *5MBytes*, and the size of its application specific state is *1MByte*. The former affects the overhead of transferring static file at the beginning of the simulation execution; whereas, the latter affects the overhead of taking and reporting checkpoint during the simulation execution.

Three execution scenarios are investigated. In the Normal scenario, the federation executes on SOHR without a fault tolerance mechanism. In the Basic and Optimized

scenarios, our fault tolerance mechanism is implemented using the basic and optimized state recovery protocols respectively. Two types of federation executions (denoted as CON and OPT federation) are conducted, in which federates employ conservative and optimistic synchronization respectively. For each execution scenario, the federation execution is repeated five times. Since the execution times of the repeated executions have small variance, only the average execution time is reported in experiment results.

5.2.2 Experiment Results

5.3.a: CON federation

5.3.b: OPT federation

Figure 5.3: Execution times in different scenarios

The first series of experiments are carried out to illustrate the benefit of the proposed fault tolerance mechanism. In this series of experiments, the $P_{ExternEvent}$ is set to 50%. A federate takes its checkpoint when its logical time is equal to $1000+2000k$ ($k=1,2\dots5$). Failures are triggered on **Federate1** when its logical time is equal to 2000, 6000, and 10000. They are denoted as F@S, F@M, and F@E respectively. The execution times under different execution scenarios are reported in Figure 5.3.a and Figure 5.3.b. In No-Failure case, the Normal scenario outperforms other scenarios slightly. This shows only marginal overhead is introduced in the Basic and Optimized scenarios for taking

checkpoints.

In the Normal scenario, the triggered failures interrupt the federation execution; and thus, the federation execution has to restart from the beginning. It is assumed that the failures are not repeated, i.e., the same failures are not triggered again in the restarted federation execution. In our experiments, the sum of the elapsed times of both the failed and restarted federation executions is used in the comparison. Consequently, the execution time of the Normal scenario increases significantly in the presence of federate failures. The later the failure happens, the more time is wasted on the failed federation execution. For instance, the execution times are much longer in the presence of F@E failures than those in the presence of F@S failures.

In the Basic and Optimized scenarios, the federation execution can complete successfully in the presence of the triggered failures. The simulation results are the same as those in No-Failure case. This verifies that the proposed mechanism can recover from federate failures correctly. Comparing with the No-Failure case, the execution times of the Basic and Optimized scenarios in the presence of failures increase slightly due to the overhead of failure detection and recovery.

5.4.a: Reduction of federation execution time

5.4.b: Reduction of execution rollbacks

Figure 5.4: Advantages of the optimized state recovery protocol

The second series of experiments are carried out to illustrate the advantages of the optimized state recovery protocol over the basic one in terms of performance enhancement. In theory, the optimized protocol can enhance performance more significantly, if the repeated execution is longer. In the experiments, a failure is triggered on **Federate1**, when its logical time is between 5000 and 7000. Consequently, the length of the repeated execution increases from 0 to 2000. The value of $P_{ExternEvent}$ is set to 50% or 100%, denoted as $p = 50\%$ or $p = 100\%$. The reduction of execution time, i.e., federation execution time of the Basic scenario minus that of the Optimized scenario, is plotted in Figure 5.4.a. The reduction of execution time is minimal for CON federation executions. As shown in Table 5.1, the Optimized scenario can outperform the Basic scenario for CON federation executions, because the RTI service calls and callbacks communicated between **RecoveredFed** and the DRC are combined or even avoided. However, the communication overhead between the federate and the DRC is very small compared to the event processing time of federates in our experiments. Hence, the Optimized protocol can only enhance performance of CON federation executions slightly. In contrast, the reduction of execution time is considerable for OPT federation executions. It increases linearly with respect to the length of the repeated execution.

The reduction of rollbacks, i.e., the number of execution rollbacks in **RecoveredFed** in the Basic scenario minus that in the Optimized scenario, is shown in Figure 5.4.b. As we can see, **RecoveredFed** in the Optimized scenario can avoid much more rollbacks when $p = 100\%$ compared to the case when $p = 50\%$. This explains why the reduction of execution time of the Optimized scenario is more significant when $p = 100\%$ compared to the case when $p = 50\%$, as shown in Figure 5.4.a.

5.3 Summary

In this chapter, a checkpoint-based mechanism is proposed to tolerate crash-stop failures of federates. Using the decoupled federate architecture, the crash-stop failures occurred on federates can be isolated from the DRC and recovered without interrupting the execution of other federates in the federation. Relying on the checkpointing and message logging techniques, the basic and optimized protocols are developed to recover the state of a crashed federate. In the basic protocol, the federate execution from taking the checkpoint to encountering the failure is repeated identically. In the optimized protocol, the state recovery procedure is accelerated by using the information kept in the DRC. Experimental results show that the proposed mechanism can tolerate federate failures at the price of acceptable overhead (e.g., time needed for checkpointing and message logging). The results also indicate that the optimized protocol can outperform the basic one, especially for the case where the crashed federate employs an optimistic synchronization approach.

Chapter 6

Replication-based Crash-stop Fault Tolerance Mechanism

One of the objectives of this thesis is to tolerate crash-stop failures of federates by replicating their executions in the federation. To enhance simulation performance, we have proposed an Un-Identical federate replication structure in Chapter 4 to support the concurrent executions of CON and OPT replicas of the same federate. In this Chapter, we will extend the Un-Identical replication structure so that it supports fault tolerance while keeping the performance advantages.

Using the decoupled federate architecture, the DRC, which is assumed to be robust, is able to survive from the failures which occur in the federate and its underlying execution platform. In the case that some replicas of the federate are crashed, the DRC is still able to keep the connection to the federation and choose other replicas to represent the federate. Hence, the federation execution will not be interrupted, as long as there is one replica of each federate that can work properly. From this point of view, the Un-Identical federate replication structure already supports fault tolerance. However, the performance of the federation execution may degrade if the crashed replica is the

fastest one, since Un-Identical replication structure improves performance by choosing the fastest replica to represent the federate. Furthermore, the federation execution may be blocked if all replicas of a single federate crash. For these reasons, the Un-Identical replication structure needs to be extended.

In the extended replication structure, a new replica (i.e., the recovered replica) is established and connected to the same DRC to substitute the crashed replica. To avoid repeated computation from the beginning, the recovered replica is initialized using the state of one of the working replicas. Since replicas are allowed to employ different synchronization approaches, one of the research issues investigated in this chapter is how the state of the recovered replica can be initialized using the state of a working replica which may employ a different synchronization approach. Experiments using P-hold simulation model are carried out to illustrate the benefits of extending the Un-Identical federate replication structure.

6.1 Extended Federate Replication Structure

6.1.1 Crash-stop Fault Tolerance Mechanism

In the extended Un-Identical federate replication structure, a replication-based crash-stop fault tolerance mechanism is implemented as shown in Figure 6.1. Some components described in Subsection 3.1 to support federate migration, i.e., the workload balance module, federate server, and Grid FTP server are also needed in our federate fault tolerance mechanism. Similar to Chapter 5, a Fault Tolerance (FT) module embedded in the DRC is responsible for detecting the failures of federate replicas within reasonable time and recovering the crashed replica on a workable computing node.

A heartbeat-based failure detection method described in Subsection 5.1 is reused

Figure 6.1: Replication-based federate fault tolerance mechanism

to detect the crash-stop failures of federate replicas. When a failure is detected, the recovered replica is started on a workable computing node to substitute the crashed replica. It is assumed that the static files of each replica, such as federate code files and the FDD (Federation Object Model Document Data) file [56], are kept in the FT module embedded in its DRC at the beginning of the federation execution. Hence, static files are available in the presence of federate failures. They are uploaded to the GridFTP server deployed on the computing node where the recovered replica is to be started. Instead of executing the recovered replica from the beginning, we initialize it using the state of the working replica (i.e., the other replica of the same federate which survives the failure). The working replica is informed to save its state and transfer its dynamic files, such as database files and the federate execution state file. Finally, the recovered replica initializes its state, connects to the same DRC and starts its execution. The working replica executes as normal except for suspending for state saving. The processes of detecting failure, sending static and dynamic files, and restoring the recovered replica state are overlapped with the execution of the working replica.

Time advancement of the federate will not be held back in the Un-Identical replica-

tion structure, because the working replica will continue its execution and represent the federate in the federation. To ensure the correctness of federation execution, the LGT of each federate, i.e., $\max(\text{CONLGT}, \text{OPTLGT})$, must keep monotonically increasing despite the presence of failures. If the crashed replica has a smaller logical time than the working replica, the recovered replica can be initialized using the current state of the working replica. However, in the case that the logical time of the working replica is smaller than that of the crashed replica, the LGT would decrease if the recovered replica were initialized using the current state of the working replica. To keep LGT monotonically increasing in this case, the FT module delays informing the working replica to save its state until its logical time is great enough. Therefore, we can guarantee the following inequalities in both cases:

$$\text{LGT of the federate before failure} \leq \text{LGT of the federate after failure} \quad (6.1)$$

$$\text{Logical time of the crashed replica} \leq \text{Logical time of the recovered replica} \quad (6.2)$$

In the Un-Identical replication structure, each federate has multiple replicas employing different synchronization approaches. When a replica is crashed, a new replica (recovered replica) should be created using the same synchronization approach as the crashed one. In the case that, there is a working replica, employing the same synchronization approach as the crashed one, the state of the recovered replica can be created straightforwardly using the state recovery protocols presented in Chapter 5. Otherwise, the state of the recovered replica should be created based on a working replica employing a different synchronization approach. In Sections 6.1.2 and 6.1.3, we illustrate how to create the state of a recovered OPT replica based on the state of a working CON replica and vice versa.

As explained in the last chapter, in HLA-based simulation, federate state is composed

of application specific state and simulation engine state. The application specific state includes data structures implemented by the simulation model. It is assumed that the saving and restoration functions of the application specific state are provided by the federate developer. For the simulation engine state, different data structures are used in CON and OPT federates. In CON federate, an eventlist is used to keep events that will occur in the future. In OPT, an input queue is used to keep the processed and unprocessed events; a state queue is used to save snapshots of application specific state; an output queue is used to log corresponding anti-messages (i.e., retractions) for the messages scheduled by each processed event.

6.1.2 Transfer State of CON Replica to OPT Replica

The application specific state of the working CON replica is transferred to the recovered OPT replica by invoking the corresponding saving and restoration functions. The initialized logical time of the recovered OPT replica, denoted as `RecoveredOPTLGT`, is set to `CONLGT`. The restored application specific state is safe, and thus its snapshot can be kept in the state queue as the lower bound of future rollbacks. The events in the eventlist of the working CON replica are copied to the input queue of the recovered OPT replica. Thus, the recovered OPT replica has the same internal events as the working CON replica. To ensure correct message delivery in the federation, the content saved in the data structures of message exchange mechanism in the Un-Identical federate replication structure (see Figure 4.2 in Chapter 4) needs be modified accordingly. The callbacks in `OptTSOQueue` are replaced by those in `ConTSOQueue`, and thus, the recovered OPT replica has the same external events as the working CON replica. In addition to initialize `OptTSOQueue`, messages in `OptMsgFilter` also need to be cleared. They can be divided into two groups: (i) messages generated by the crashed OPT replica while processing

events with $TS \leq \text{RecoveredOPTLGT}$; and (ii) messages generated by the crashed OPT replica while processing events with $TS > \text{RecoveredOPTLGT}$. The first group of messages are incorrect messages, as the working CON replica whose logical time is equal to RecoveredOPTLGT did not sent these messages. To ensure the correctness of the federation execution, these incorrect messages must be retracted. The second group of messages also need to be retracted to remove the effect of events with $TS > \text{RecoveredOPTLGT}$ which were processed by the crashed OPT replica. After the failure recovery, all events with $TS > \text{RecoveredOPTLGT}$ will be processed by both the working CON replica and the the recovered OPT replica.

6.1.3 Transfer State of OPT Replica to CON Replica

The current application specific state of the working OPT replica cannot be transferred directly to the recovered CON replica, since the working OPT replica may experience from rollbacks in the future. The recovered CON replica is initialized using the latest snapshot in the state queue with $TS < \text{OPTLGT}$. Its logical time (denoted as RecoveredCONLGT) is set equal to the TS of the snapshot. For the simulation engine state, it is unnecessary to transfer the state queue and the output queue of the working OPT replica to the recovered CON replica, as the latter never encounters execution rollbacks. The events in the input queue are copied to the eventlist of the recovered CON replica except for the following cases: (i) the events with $TS \leq \text{RecoveredCONLGT}$; (ii) the events that are generated by processing events with $TS > \text{RecoveredCONLGT}$ (i.e., optimistic internal events); (iii) the events that are received from other federates with $TS > \text{RecoveredCONLGT}$ (i.e., optimistic external events). The events in case (i) are correct events that have been processed by the working OPT replica before transferring its state to the recovered CON replica. The events with $TS \geq \text{RecoveredCONLGT}$ are not yet processed by the

recovered CON replica. So, the events in case (ii) should not be included in the eventlist. Since the logical time of the recovered CON replica is set to `RecoveredCONLGT`, the events in case (iii) are not allowed to be delivered to the recovered CON replica.

Besides federate state restoration, the message exchange with other federates through the DRC must also be taken into account. According to Inequality (6.2), we can get $\text{CrashedCONLGT (logical time of the crashed CON replica)} \leq \text{RecoveredCONLGT}$. In other words, the logical time of CON replica is increased from `CrashedCONLGT` to `RecoveredCONLGT` directly after restoration. Accordingly, the content saved in the data structures of message exchange mechanism in the Un-Identical federate replication structure needs be modified. The callbacks with $\text{TS} \leq \text{RecoveredCONLGT}$ must be removed from `ConTSOQueue`. The messages kept in `ConMsgFilter` were generated by the crashed CON replica while processing events with $\text{TS} \leq \text{CrashedCONLGT}$. All these messages must have been removed from `ConMsgFilter`, as the same messages must have been sent by OPT replica with logical time greater than `CrashedCONLGT`.

6.2 Experiments and Results

6.2.1 Experiment Design

In order to verify that the extended Un-Identical federate replication structure can enhance performance in the presence of failures, experiments are carried out using a synthetic P-Hold simulation model described in Subsection 4.2.1. The simulation executions are conducted on the cluster described in Subsection 3.2.1, with SOHR deployed in the same manner. Each federate replica executes on an individual computing node. In addition, we assume that the total size of static files is 5 *MBytes*, and the total size of dynamic files is 1 *MBytes*, which affect the file transfer overhead respectively.

Table 6.1: Types of injected failures

Failure Type	Crashed Replica(s)	Failure(s)	Description
No-Failure	None	0	No replica crashes
CON@S	CON	1	CON crashes at 2000
CON@M	CON	1	CON crashes at 6000
CON@E	CON	1	CON crashes at 10000
OPT@S	OPT	1	OPT crashes at 2000
OPT@M	OPT	1	OPT crashes at 6000
OPT@E	OPT	1	OPT crashes at 10000
CON@S&OPT@E	CON&OPT	2	CON crashes at 2000; OPT crashes at 10000
OPT@S&CON@E	CON&OPT	2	OPT crashes at 2000; CON crashes at 10000

Four execution scenarios are investigated. In the Conservative (or Optimistic) scenario, each federate has only one CON replica (or OPT replica) and executes on SOHR without replication structure. In the Replicated scenario, each federate has a CON and an OPT replica, which execute concurrently on the Un-Identical replication structure. In the Replicated scenario, failed replica will not be recreated. Each federate can tolerate the crash of one replica. However, the federate would have to be restarted when the other replica fails in the future. In Replicated&FT scenario, replicas execute on the extended Un-Identical federate replication structure with fault tolerance mechanisms as described in the last section of this chapter. For each execution scenario, the federation execution is repeated five times. Since the execution times of the repeated executions have small variance, only the average execution time is reported in experiment results.

6.2.2 Experiment Results

In the first series of experiments, the $P_{ExternEvent}$ is a constant value during the federation execution. Several types of failures as illustrated in Table 6.1 are triggered on the replicas of **Federate1** in the P-hold simulation. A failure is tolerated, if the federation execution is not interrupted by the failure. Otherwise, the failure is not tolerated in the execution scenario. Consequently, the entire federation execution has to restart from the beginning.

Table 6.2: Execution time (sec) when $P_{ExternEvent} = 50\%$

Failure Type	Failure Group	Federation Execution Scenarios			
		Conservative	Optimistic	Replicated	Replicated&FT
No-Failure	(i)	163.106	164.756	144.810	145.204
CON@S	(ii)	190.453	N/A	160.508	147.598
CON@M	(ii)	243.463	N/A	155.992	147.846
CON@E	(ii)	296.700	N/A	149.049	147.045
OPT@S	(ii)	N/A	191.869	150.143	147.470
OPT@M	(ii)	N/A	245.702	148.378	147.920
OPT@E	(ii)	N/A	303.600	147.357	147.529
CON@S&OPT@E	(iii)	N/A	N/A	277.625	148.459
OPT@S&CON@E	(iii)	N/A	N/A	269.510	149.288

It is assumed that the failures are not repeatable, i.e., the same failures are not triggered again in the restarted federation execution. In our experiments, the sum of the elapsed times of both the failed and restarted federation executions is used in the comparison.

Table 6.2 reports the federation execution times of different execution scenarios in the presence of different types of failures, when $P_{ExternEvent} = 50\%$. According to the failure types, the results can be categorized into three groups:

- (i) **No replica crashes:** The Replicated scenario can obtain better performance than those without using the replication structure (similar result also is illustrated in Subsection 4.2.2). The Replicated&FT scenario executes in the same manner as the Replicated scenario except for the heartbeat reports for failure detection. As shown by the result, the overhead is marginal.
- (ii) **Either CON or OPT replica crashes:** This group of failures cannot be tolerated in the execution scenarios without federate replication, i.e., the Conservative and Optimistic scenarios. The execution time increases significantly as the re-execution of the entire federation is needed. In the case that the failure is triggered near the end of federation execution, more time is consumed on the failed federation execution. Taking Conservative scenario as an example, the execution time is

much longer in the presence of CON@E failure than that in the presence of CON@S failure. In the Replicated scenario, these types of failures are tolerated and better performance is achieved. However, the performance might degrade compared to the No-Failure case due to the loss of a replica in the execution after the failure. The earlier the failure happens, the more significant the performance degrades. For instance, the Replicated scenario performs worse in the presence of CON@S failure compared to the situation with CON@E failure. In the Replicated&FT scenario, the crashed replica is substituted by the recovered replica, so that similar performance is achieved as that in No-Failure case.

- (iii) **Both CON and OPT replicas crash:** In this group of failures, both replicas of the federate crash at different times. The re-execution of the federation is needed in the Replicated scenario, as no working replica of the federate remains. As a result, the execution time is increased significantly. In the Replicated&FT scenario, two replicas are always available during the entire execution except for the failure recovery period. Both failures are tolerated and hence, similar performance is achieved as that in the No-Failure case.

To further compare the Replicated and Replicated&FT scenarios, Figure 6.2 reports their execution time in the presence of failures when $P_{ExternEvent}$ represents extreme values. For the cases when both replicas of **Federate1** crash, the Replicated&FT scenario outperforms Replicated scenario significantly. When $P_{ExternEvent} = 100\%$, conservative synchronization is much more efficient than optimistic synchronization. The performance of the Replicated scenario in the presence of OPT replica failures is similar to that in No-Failure case. However, the performance of Replicated scenario degrades by 30.9%, 20.1% and 5.8%, in the presence of CON@S, CON@M and CON@E failures respectively. In contrast, the Replicated&FT scenario obtains similar performance no matter whether

 6.2.a: $P_{ExtEvent} = 100\%$

 6.2.b: $P_{ExtEvent} = 0\%$

 Figure 6.2: Performance of Replicated and Replicated &FT scenarios when $P_{ExtEvent}$ equals to extreme values

or not there is a failure. When $P_{ExtEvent} = 0\%$, optimistic synchronization is much more efficient than conservative synchronization. The execution time of the Replicated scenario does not increase due to the loss of CON replica. We can also observe that the execution time of the Replicated scenario also does not increase considerably due to the loss of OPT replica. The reason is that, the OPT replica of **Federate2** can execute quickly without time constraints and execution rollbacks. This advances GALT, and thus the time advancement requests from CON replica of **Federate1** can always be granted immediately. Consequently, Replicated and Replicated&FT scenarios achieve comparable performance.

In the second series of experiments, we measure the performance of the federation executions with variable $P_{ExtEvent}$, which is changed every 2,400 simulation units. Five cases are investigated, which are shown in Table 4.2 in Chapter 4. In this series of experiments, failures are triggered on replicas of both federates in the federation execution. CON1&CON2@M (or OPT1&OPT2@M) means CON replicas (or OPT replicas) of both **Federate1** and **Federate2** in the P-hold simulation crash concurrently at the middle of the federation execution. Triggering of failures is not applicable if the corresponding replica

6.3.a: CON1&CON2@M

6.3.b: OPT1&OPT2@M

Figure 6.3: Performance of execution scenarios when $P_{ExternEvent}$ is variable

does not exist in the execution scenario. For instance, CON1&CON2@M is not applicable in the Optimistic scenario. Again, failures cannot be tolerated in the Conservative and Optimistic scenarios. A restarted federation execution is needed to conduct the simulation successfully. In contrast, these concurrent failures can be tolerated in both the Replicated and Replicated&FT scenarios. As shown in Figure 6.3, they outperform the Conservative and Optimistic scenarios significantly. In the Replicated scenario, only one type of replica (either CON or OPT replica) remains in the execution for each federate after the concurrent failures; whereas in the Replicated&FT scenario, a crashed replica is always restored. Hence, both CON and OPT replicas execute for each federate during the entire federation execution (except for the recovery period). Since the Un-Identical federate replication structure is able to choose the faster replica to represent the behavior of the federate during the entire federation execution, the Replicated&FT scenario is expected to outperform the Replicated scenario. As shown in Figure 6.3, for the cases tried, the Replicated&FT scenario obtains around 2% to 15% performance enhancement compared to the Replicated scenario.

6.3 Summary

The Un-Identical federate replication structure can tolerate crash-stop federate failures by simply removing the crashed replicas. Unfortunately, this reduces the number of active replicas of the federate. As a result, the replication structure might suffer from execution performance degradation and might fail to tolerate future failures. In this chapter, the Un-Identical federate replication structure is extended for the purpose of enhancing performance as well as providing fault tolerance for federation executions. When a federate replica crashes, a new replica is established to substitute the crashed one during the remaining execution. To avoid repeated computation from the beginning, the new replica is initialized using the execution state of a working replica. Since replicas of the same federate might employ different synchronization approaches, mechanisms are proposed to initialize the state of the new replica using that of the working replica which employs a different synchronization approach. As experiments show, compared with the original one, the extend replication structure can tolerate more types of federate failures and achieve better performance in the presence of federate failures.

Chapter 7

Replication- and Checkpoint-based Byzantine Fault Tolerance Mechanism

One of the objectives of this thesis is to tolerate Byzantine failures of federates by exploiting the advantages of the decoupled federate architecture. In this chapter, a three-phases (i.e., failure detection, failure location, and failure recovery) Byzantine Fault Tolerance (BFT) mechanism is proposed using the existing Barrier and No-Barrier federate replication structures (introduced in Chapter 2). Epidemic effect removal mechanisms are also proposed in case of Byzantine failure propagation in the federation execution. Experiments using a Borderless Fab simulation model are carried out to study the efficiency and scalability of the proposed BFT mechanism.

Figure 7.1: Overview of three-phases BFT mechanism

7.1 Byzantine Fault Tolerance Mechanism

In the decoupled federate architecture, the failures of a federate and its DRC can be isolated. In this chapter, we focus on Byzantine failures of the federate while assuming that its DRC is robust. We also assume that the messages in communication channels are never lost, duplicated or corrupted. The proposed BFT mechanism relies on federate replication. It is provided in HLA-based simulation by implementing the corresponding functional modules, i.e., BFT module and replication module. As shown in Figure 7.1, for efficiency consideration, the BFT and replication modules are embedded in the DRC with other modules (e.g., FM, TM, and OM modules)(see Subsection 1.3.3). The replication module is responsible for supporting the concurrent executions of federate replicas; whereas, the BFT module is responsible for detecting, locating, and recovering from Byzantine failures. The BFT module is implemented based on either Barrier or No-Barrier replication structure.

The replicas of the same federate executed on Barrier and No-Barrier replication structures are required to be Piecewise Equivalent. Thus, they should employ the same

synchronization approach. If a replica encounters Byzantine failure, it is denoted as *faulty replica*. Otherwise, it is denoted as *non-faulty replica*. The executions of non-faulty replicas are normal. In contrast, the executions of faulty replicas are arbitrary. They might modify their local state arbitrarily and invoke arbitrary RTI services. As mentioned in Subsection 2.2.1.2, we assume that the number of simultaneous failures is not greater than f and that there are no concurrent failures.

7.1.1 Three-Phases Mechanism

The RTI service calls, e.g., sending interactions, can affect the execution of other federates in the federation. A faulty federate may invoke arbitrary RTI service calls and thus leads to a wrong federation execution. The Byzantine failure can be detected by replicating the federate in multiple places and comparing their RTI service calls. However, a faulty replica might modify its local state arbitrarily while invoking the same RTI services as the non-faulty ones. In this case, a wrong simulation result will be produced due to the arbitrary federate state. Therefore, both RTI service calls invoked by replicas of the same federate and their local states should be compared for Byzantine failure detection.

To detect arbitrary local state modification, replicas are required to take checkpoints of their states, including application specific state and simulation engine state (described in Subsection 5.1.1); and to report the checkpoints to the BFT module for comparison. They should report checkpoints after invoking every RTI service, but this would be too expensive. Instead, replicas report checkpoints after invoking every K RTI services at the price of greater failure detection latency. Each checkpoint is associated with a logical time, denoted as *checkpoint time*, equal to the replica's logical time when the checkpoint is taken. In the case that all replicas of the same federate have reported the same checkpoint, the checkpoint becomes a *consistent checkpoint*. Besides detecting Byzantine failures, the

checkpoints are also used to start new replicas and rollback replicas' execution state in our BFT mechanism.

As shown in Figure 7.1, the BFT module is composed of a group of functional components. The BFT module keeps a service counter for each replica. On receiving an RTI service, the service counter is increased by one. The service counter is attached to the checkpoint and RTI service as its sequence number. The checkpoints with sequence number attached are buffered in the checkpoint log. The RTI service calls invoked by the replicas and the callbacks delivered to the replicas are buffered in the message log¹. Besides the return value and sequence number, the RTI service call itself should also be included in the message log for Byzantine failure detection. The compare unit is responsible for comparing the checkpoints and RTI service calls of the replicas. Using the comparison results, the failure detector detects the presence of Byzantine failures; whereas, the failure locator identifies the faulty replicas using a majority voting protocol.

Some components described in Subsection 3.1 (i.e., the workload balance module, federate server, and Grid FTP server) to support federate migration and execution also are required to manage the execution of federate replicas in our BFT mechanism. To start a new replica, the BFT module obtains an available computing node from the workload balance module; sends a checkpoint to the computing node via the Grid FTP server; and finally starts the replica through the federate server. The BFT module can also terminate faulty replicas through the federate server. In the decoupled federate architecture, the DRC keeps the connection to the federation and masks the federate replication from the federation. Hence, any number of replicas can be started and terminated dynamically during the federation execution.

The BFT mechanism consists of three-phases: failure detection, failure location, and failure recovery. For the same federate, only $f+1$ replicas (**Group1**) are executed con-

¹The message log can be reused by the No-Barrier replication module.

currently. In the case that **Group1** replicas invoke the same RTI service and report the same checkpoint with the same sequence number, all replicas are non-faulty replicas. Otherwise, one or more Byzantine failures are detected. To identify the faulty replicas, another f replicas (**Group2**) are started from the latest consistent checkpoint of **Group1** replicas. By replaying the message log kept in the BFT module, **Group2** replicas can identically repeat the executions conducted by **Group1** replicas before the failure using the basic state recovery protocol proposed in Chapter 5². After the repeated execution, **Group2** replicas catch up with **Group1** replicas in their executions. By comparing the RTI services and checkpoints sent by **Group1** and **Group2** replicas, faulty replicas can be identified. More replicas are needed, if some replicas fail to respond or **Group2** replicas encounter Byzantine failures during the repeated execution.

In the BFT mechanism using Barrier replication, the replicas of the same federate are blocked, if some of them encounter Byzantine failures (i.e., send arbitrary RTI service calls or report arbitrary checkpoints). So, arbitrary RTI service calls are never processed by the DRC, and thus, epidemic effects of the Byzantine failures never happen. To recover federation execution from the Byzantine failures, the faulty replicas are disconnected from the DRC and their arbitrary RTI service calls or checkpoints are discarded. The executions of replicas before their consistent checkpoint are normal and cannot be affected by Byzantine failures. Hence, the message log and checkpoints taken before the consistent checkpoint in the BFT module can be removed.

In the BFT mechanism using No-Barrier replication, Byzantine failures are detected if a replica invokes a RTI service call or reports a checkpoint different from that invoked or reported by other replicas previously with the same sequence number. In the case that the fastest replica is a non-faulty replica, arbitrary RTI service calls will not be processed by

²The optimized state recovery protocol proposed in Chapter 5 cannot be used, because **Group2** replicas recovered from the checkpoint do not repeat the execution of **Group1** replicas identically.

the DRC, and thus, no epidemic effect happens. The Byzantine failures can be recovered in a similar way as the BFT mechanism using Barrier replication. In the case that the fastest replica is a faulty replica, an epidemic effect happens if its arbitrary RTI service calls are processed by the DRC. To overcome the Byzantine failures, the faulty replicas, their arbitrary checkpoints, and the epidemic effect of the Byzantine failure should be removed from the federation execution.

Due to the epidemic effect, although a federate may have a consistent checkpoint, its execution might have been already affected by the Byzantine failures that occurred in other federates in the federation. Therefore, the checkpoint log in the BFT module has to maintain multiple consistent checkpoints of a federate. The minimal checkpoint time of the latest consistent checkpoints of all federates in the federation is denoted as *stableLGT*. Accordingly, the consistent checkpoints kept in the BFT module with *logicaltime* $\leq$ *stableLGT* are referred to as *stable checkpoints*. Since *stableLGT* defines the lower bound of the future Byzantine failures, the message log and checkpoint log taken before the latest stable checkpoint can be removed.

Figure 7.2 shows an example of the BFT mechanism using No-Barrier replication, in which f is equal to 1. In the figure, the replication module is combined with other modules (e.g., TM and OM modules) in the DRC for clarity of presentation. **Replica2**, as the faster replica, encounters a Byzantine failure and invokes an arbitrary RTI service **Service'_k**. The Byzantine failure is detected when the normal service **Service_k** is invoked by **Replica1**. To identify the faulty replica, **Replica3** is started from the latest consistent checkpoint of **Replica1** and **Replica2**. After conducting the repeated execution, **Replica3** catches up in execution and invokes **Service_k**. Using a majority voting protocol, **Replica2** is identified as the faulty replica. An epidemic effect of Byzantine failure happens, because the arbitrary service **Service'_k** has been processed by the DRC. After removing the epidemic effect, as explained below, the effect of **Service'_k** is canceled. Then, an instance of **Service_k**

Figure 7.2: BFT mechanism using No-Barrier federate replication

invoked by Replica1 and Replica3 is processed by the DRC.

7.1.2 Remove Epidemic Effect of Byzantine Failure

7.1.2.1 Epidemic Effect Removal Mechanism

When arbitrary time advancement requests invoked by a faulty replica of a federate is handled by the DRC, wrong time information might be provided for federation-wide time synchronization. Consequently, federates in the federation might advance their logical time wrongly. As a result, CON federates might process events out of order; and OPT federates might carry out wrong fossil collection. In addition, the arbitrary messages generated by the faulty replica might be delivered to the receiving federates. On processing the event generated from the arbitrary messages, the replicas of the receiving federates might modify their execution states wrongly and generate other arbitrary messages. That is, a secondary Byzantine failure propagation happens. The execution of

the federate with the faulty replica and that of the federates affected by the Byzantine failure are wrong. In order to remove the epidemic effect, the wrong federate executions should be rolled back. The actions taken to remove such kind of epidemic effect are described below. A general epidemic effect removal mechanism is proposed. It can be applied to purely conservative or purely optimistic federation executions, as well as mixed federation executions, in which federates are allowed to employ either conservative or optimistic synchronization.

Firstly, determine the execution points to which federates should roll back. The message and checkpoint logs for each federate represent the execution procedure of the federate. Besides the sequence number, the federate's logical time is also attached to each entry buffered in the message and checkpoint logs. For the federate with faulty replicas, the *failure point* refers to the logical time when the first arbitrary RTI service or checkpoint is sent. Hence, *failureLGT* (or *failureSeq*) is equal to the attached logical time (or sequence number) of the corresponding RTI service call or checkpoint.

To distinguish the different LA values of federates, we use LA_f to denote the LA of the federate with faulty replica and use LA_o to denote the LA of other federates in the federation. For any other federate in the federation if its logical time $\geq \text{failureLGT} + LA_f$, it is referred to as federate in epidemic effect; otherwise, it is referred to as federate not in epidemic effect. Since, a Byzantine failure never affects the execution of other federates before $\text{failureLGT} + LA_f$, for each federate in epidemic effect, an *epidemic point* is defined as the latest RTI service or checkpoint buffered for the federate in the message or checkpoint log with attached logical time $< \text{failureLGT} + LA_f$. Accordingly, the *epidemicLGT* (or *epidemicSeq*) of the federate is equal to the attached logical time (or sequence number) of the RTI service or checkpoint at the epidemic point. We can easily get Inequality (7.1).

$$epidemicLGT < failureLGT + LA_f \quad (7.1)$$

To remove the epidemic effect, federates with faulty replicas and federates in epidemic effect should roll their executions back to the failure point and epidemic point respectively.

Secondly, restore logical time and execution state of federates to the failure point and epidemic point respectively. For the federate with faulty replicas, we just simply reduce the logical time to failureLGT. Only non-faulty replicas with execution state before the failure point remain in the federation execution. For each federate in epidemic effect, its replicas with logical time greater than epidemicLGT should reduce their logical time to epidemicLGT. They should restore their states using the latest checkpoint with logical time $\leq$ epidemicLGT. By replaying the message log kept in the BFT module, they are able to repeat executions from the checkpoint time to epidemicLGT identically using the basic state recovery protocol proposed in Chapter 5. In this way, the replicas restore their states to the epidemic point.

Thirdly, cancel the behaviors of federates after the failure point and epidemic point respectively. The time advancement of federates are canceled by restoring their logical time to failureLGT and epidemicLGT respectively. Federates exchange messages through their DRCs. When an outgoing message is generated by the fastest replica of a federate, it is sent by the DRC of the federate to the DRCs of the receiving federates. When an incoming message destined for a federate is received by the DRC of the federate, its corresponding callback is generated and buffered in the DRC. The callback is removed from the DRC, when it is delivered to the fastest replica of the federate. To remove the epidemic effect, we should cancel the behaviors of sending and receiving messages during epidemic effect by (i) retracting messages buffered in the message log with sequence number greater than failureSeq or epidemicSeq, for federate with faulty replica or fed-

erates in epidemic effect respectively, by sending the corresponding retraction requests; and (ii) moving callbacks buffered in the message log with sequence number greater than failureSeq or epidemicSeq to the DRC. In this way, the retractions of those sent messages will be annihilated with their corresponding callbacks in the DRCs of their receiving federates. In addition, the invoked RTI services and the reported checkpoints should be respectively removed from the message log and checkpoint log in the BFT module, if their sequence numbers are greater than failureSeq or epidemicSeq .

For OPT federates, an alternative mechanism is proposed to remove the epidemic effect by reusing the state saving and rollback mechanisms provided by optimistic synchronization. The federate with faulty replicas retracts messages which were sent after failure point. On receiving a retraction of an arbitrary message from the federate with faulty replica, the federates in epidemic effect restore their application specific state using the snapshot which was taken before processing the arbitrary message; and retract the messages which were sent after processing the arbitrary message. The epidemic effect of the Byzantine failure is removed from the federation execution, if the retractions of all arbitrary messages are handled. After taking the epidemic effect into account, the upper bound for fossil collection in OPT replicas should be the stableLGT , i.e., the lower bound of Byzantine failures, rather than their logical time.

Two examples of removing the epidemic effect are illustrated in Figure 7.3. The faster replica of **Federate1** is the faulty replica; whereas, a replica of **Federate2** is affected by the Byzantine failure. LA_f and LA_o denote the LA of **Federate1** and **Federate2** respectively. The faulty replica sends an arbitrary message Msg'_k . On receiving the arbitrary message, the replica in epidemic effect sends another message Msg'_l , which is finally delivered to the faulty replica. To remove the epidemic effect, we should determine the failure point and epidemic point, and restore the logical time of **Federate1** and **Federate2** to failureLGT and epidemicLGT respectively. Since Msg'_k is the arbitrary message that

7.3.a: CON federate replica

7.3.b: OPT federate replica

Figure 7.3: Examples of epidemic effect removal mechanisms

causes epidemic effect, we have:

$$RVT \text{ of } \text{Msg}'_k \geq failureLGT + LA_f \quad (7.2)$$

Since Msg'_l was generated after processing Msg'_k , we have:

$$RVT \text{ of } \text{Msg}'_{-1} \geq RVT \text{ of } \text{Msg}'_{-k} + \text{LA}_0 \quad (7.3)$$

From Inequalities (7.1) and (7.2), we can get:

$$RVT \text{ of } \mathbf{Msg}'_k > epidemicLGT \quad (7.4)$$

By combining Inequalities (7.2) and (7.3), we can get:

$$RVT \text{ of } \mathbf{Msg}'_l > failureLGT \quad (7.5)$$

By combining Inequalities (7.3) and (7.4), we can get:

$$RVT \text{ of } \mathbf{Msg}'_l > epidemicLGT + LA_o \quad (7.6)$$

In the case that the replica of **Federate2** is a CON replica, as shown in Figure 7.3.a, it restores execution state to the checkpoint with logical time $\leq epidemicLGT$. Its corresponding service counter is also restored using the sequence number of the checkpoint. After that, the replica conducts the repeated execution until it invokes the RTI service with sequence number equal to $epidemicSeq$. The BFT module of **Federate1** retracts $\mathbf{Msg}'_k$ and moves $\mathbf{Msg}'_l$ callback to the DRC of **Federate1**; the BFT module of **Federate2** retracts $\mathbf{Msg}'_l$ and moves $\mathbf{Msg}'_k$ callback to the DRC of **Federate2**. According to Inequalities (7.2) and (7.6), the retractions of $\mathbf{Msg}'_k$ and $\mathbf{Msg}'_l$ can be sent by **Federate1** and **Federate2** respectively to the DRCs of the receiving federates. According to Inequalities (7.4) and (7.5), $\mathbf{Msg}'_k$ and $\mathbf{Msg}'_l$ callbacks should be in the DRC and not be delivered to the CON replicas after their state restoration. After the retractions and corresponding callbacks are annihilated, the epidemic effect of the Byzantine failure is removed from the federation execution.

In the case that the replica of **Federate2** is an OPT replica, as shown in Figure 7.3.b, the BFT module of **Federate1** retracts $\mathbf{Msg}'_k$ and moves $\mathbf{Msg}'_l$ callback to the DRC

of **Federate1**. On receiving the RR of **Msg'_k**, the replica of **Federate2** rolls back its application specific state and retracts **Msg'_l**. The retraction is finally annihilated with **Msg'_l** callback in the DRC of **Federate1**. After that, the epidemic effect of the Byzantine failure is removed from the federation execution.

7.1.2.2 Exceptions Occurred in the Epidemic Effect

Using the No-Barrier replication, arbitrary RTI services generated by the faulty replicas or the federate replicas affected by the Byzantine failure might be processed by the DRC. Consequently, the federation execution might encounter some exceptions (as defined in HLA standard [56]). For instance, a federate might request to advance its logical time to a value which has already been passed, i.e., the requested time $<$ logical time. It might also send or receive messages with $RVT < \text{logical time} + LA$.

In No-Barrier replication, only the RTI services invoked by the fastest replica are processed. When an exception is caught, the fastest replica is blocked; whereas, the other replicas keep running. In the case that the fastest replica is a faulty replica, it will be removed after the failure is recovered. Thus, the exception can be avoided accordingly. In the case that the fastest replica of the federate is affected by the Byzantine failures occurred in other federates, the other replicas of the federate will invoke the same arbitrary RTI services. In this situation, all replicas will be blocked until the epidemic effect of the Byzantine failures has been removed. Since the exception is a part of the epidemic effect, removing the epidemic effect will also avoid the exception.

7.2 Experiments and Results

7.2.1 Borderless Fab Simulation Model

A Borderless Fab simulation model based on information collected from a real-world fab is used in our experiment to evaluate the proposed BFT mechanism using either Barrier or No-Barrier federate replication structures. The concept of Borderless Fab was introduced in [67], as a means of sharing capacity across wafer fabrication plants within close proximity. It is particularly important in the semiconductor industry as equipment are typically expensive. Wafer lots are transferred from one fab to another if (i) the required equipment is unavailable in the fab; or (ii) the required equipment has insufficient capacity in the fab, while appropriate equipment are available in another fab. After the process on certain equipment is completed, the transferred lots can either continue in the destination fab or be transferred back to the original one, depending on which option is more efficient. Hence, the Borderless Fab concept, if successfully implemented, can minimize the cycle time of products, maximize the equipment utilization rate, and improve efficiency in handling the unplanned situations (e.g., a sudden surge in demand of certain types of products and an unscheduled breakdown of bottleneck equipment).

Due to the confidentiality of the model, we are not able to disclose the details of equipment layout and features as well as lots processing flows and steps. But the important thing to see here is that there are 80 equipment sets modeled in each wafer fab, including furnaces, steppers, implanters, wet benches and metrology tools. In addition, each equipment set is composed of a number of equipment instances. Different equipment have different processing times, which follow exponential distribution with range from 10 to 500 minutes and average value of 60 minutes. The lots are transformed from raw material to finished products through a sequence of processing steps. Different processing steps are required for the lots of different product types. The number of required process-

ing steps ranges from 200 to 300 steps. At each step, the equipment for the processing is claimed. If the claiming is unsuccessful, the lot enters the queue of the equipment and waits for the next equipment availability. Otherwise, the lot is processed by the equipment. The equipment is marked as occupied and the lot is delayed for a pre-defined amount of time which simulates the lot processing time on the equipment. After the processing is completed, the equipment is marked as available and the lot is sent to the subsequent processing step.

Each wafer fab produces ten types of products. For nine types of products, wafer lots are only processed by the equipment within a wafer fab. For the remaining type of products, wafer lots can be transferred to any other fabs with the required equipment. Each fab has approximately 8100 wafer lots released in a year. On average, there is one lot released every 65 minutes. A careful analysis to the simulation has helped us to identify the bottleneck equipment of wafer fabs. Since wafer fabs produce different types of products, they have different bottleneck equipment. To avoid the bottleneck equipment, some lots are transferred among wafer fabs.

In the HLA-based simulation, each wafer fab is simulated by an individual federate. The lots transferred among wafer fabs are simulated as interactions exchanged among federates. The LA of federates are determined by the transferring time, which has been capped at 60 minutes due to the close proximity of wafer fabs. The simulation duration is 60 days, including the first 30 days as the warmup period. The simulation results include the cycle times of products, the number of different types of products, the number of transferred wafer lots, the utilization levels of equipment, and so on.

Our simulation model is event intensive, i.e, around 73 events per simulation minute. However, only a small part of events potentially send interactions to other federates. The simulation runs extremely slowly, if CON federates issue a time advancement request before processing every event with $TS > \text{logical time}$. To solve this problem, an optimized

conservative synchronization approach proposed in [68, 91] is adopted. The requested time is set to the TS of the earliest event that will potentially send out interactions. In this way, the number of time advancement requests can be reduced significantly. For instance in our experiments, one NMR request is invoked in averaged 30 simulation minutes.

For OPT federates, it is inefficient to save application specific state before processing every event, since the state saving time is huge compared to the event processing time. To solve the problem, the infrequent state saving approach described in [25, 40, 46, 69] is applied. OPT federates only save their application specific state after processing 1000 events. When a causality error happens, the federate must roll back to a previous saved state and reprocess coast-forward events in order to recreate the desired execution state. OPT federates are also allowed to send time advancement requests infrequently. In our experiment, an OPT federate will process all the events with $TS \leq \text{last requested time} + 50$ before sending the next FQR to RTI. To avoid over-optimistic event processing, Moving Time Window protocol [102] is adopted. OPT federates are prevented from processing events with $TS \geq \text{logical time} + 500$. The upper bound of fossil collection is logical time (or stableLGT) for BFT mechanism using Barrier (or No-Barrier) replication. Since infrequent state saving mechanism is adopted, the upper bound now refers to the latest event with state saved and $TS < \text{logical time (or stableLGT)}$. Since our simulation model is event intensive, the input queue, state queue, and output queue would increase significantly if stableLGT were used as the upper bound of fossil collection. Consequently, considerable overhead would be introduced while maintaining these data structure, e.g., searching/inserting events or federate states. To reduce the overhead, we use logical time, rather than stableLGT, as the fossil collection upper bound. The information between stableLGT and logical time is saved in an auxiliary data structure which is used for Byzantine failure handling only.

7.2.2 Experiment Design

The proposed BFT mechanism using either Barrier or No-Barrier Replication is implemented on SOHR [86, 87]. Five execution scenarios are investigated. In the Normal scenario, federates execute on the original SOHR. In Barrier and No-Barrier scenarios, two replicas of each federate execute on SOHR with Barrier and No-Barrier replication module respectively. In BFT(Barrier) and BFT(No-Barrier) scenarios, the replicated federates execute on SOHR with the BFT module using Barrier and No-Barrier replication respectively. For each federate, there are two replicas in **Group1** and one replica in **Group2**, to tolerate the Byzantine failure occurred on one of the **Group1** replicas. Both purely conservative (CON) and purely optimistic (OPT) federation executions are conducted. The general and alternative epidemic effect removal mechanisms are applied for CON and OPT federation executions respectively. Experiments are carried out in the following two cases: (i) No-Workload case, i.e., federates execute on computing nodes without any external workload; and (ii) Workload case, i.e, federates execute on computing nodes with external workload. In the latter case, the computing nodes are divided into two sets. As shown in Figure 7.4, external workload is introduced in alternate execution periods. In our experiments, replicas of each federate are executed on different sets of computing nodes. Hence, No-Barrier replication is expected to have performance advantages over the Barrier one.

Figure 7.4: External workload introduced on computing nodes

Table 7.1: Execution time (sec) in No-Failure case

	No-Workload		Workload	
	CON	OPT	CON	OPT
Normal	47.730	51.165	52.320	65.760
			59.494	72.690
Barrier	51.061	59.345	74.712	84.725
No-Barrier	43.759	50.964	44.488	68.412
BFT(Barrier)	54.213	64.129	83.278	91.797
BFT(No-Barrier)	45.669	53.114	46.009	70.237

Federation executions are conducted in the same cluster with SOHR deployed in the same manner as described in Subection 3.2.1. For each execution scenario, the federation execution is repeated five times. Since the execution time of the repeated executions have small variance, only the average execution time is reported in experiment results.

7.2.3 Experiment Results

7.2.3.1 BFT Mechanism Efficiency

To study the efficiency of our BFT mechanism, two wafer fabs are simulated by **Federate1** and **Federate2** respectively in the federation. Firstly, we investigate the case where no failure is injected in the federation execution, i.e., the No-Failure case. The CON and OPT federation execution times in the above mentioned execution scenarios are reported in Table 7.1. With external workload introduced, the federation execution times in the Normal scenario increase accordingly. Two values in the table illustrate the cases where two federates are executed on the same or different set(s) of computing nodes respectively. In No-Workload case, No-Barrier scenario outperforms Normal scenario slightly; whereas, Barrier scenario performs a little worse than Normal scenario. In Workload case, for Barrier scenario, the execution is blocked by the slower replicas which

execute on the computing nodes with the external workload. So its performance is significantly worse than the Normal scenario. For the CON federation in the No-Barrier scenario, the slower replica can easily catch up the faster one, as its time advancement requests are granted immediately. Since the faster replica is chosen to represent the federate, similar performance can be obtained regardless whether external workload is introduced or not. For the OPT federation in the No-Barrier scenario, the slower replica encounters the same execution rollbacks as the faster one; and thus, it can hardly get benefit from the replication structure. As shown in Table 7.1, the federation execution time is between the two values in the Normal scenario.

In the BFT scenarios (i.e., BFT(Barrier) and BFT(No-Barrier)), federate replicas are required to take and report checkpoints after invoking every 100 RTI services; the BFT module is required to compare the RTI services and checkpoints for Byzantine failure detection. The failure detection overhead is equal to the execution time of the BFT(Barrier) or BFT(No-Barrier) scenario subtracted by that of the Barrier or No-Barrier scenario in the No-Failure case respectively. Compared with CON federates, OPT federates invoke fewer RTI services, and thus, report fewer checkpoints. But, their checkpoints usually have greater sizes (as discussed in Subsection 5.1.2). As shown in Table 7.1, similar failure detection overhead is introduced in CON and OPT federation executions.

Secondly, a Byzantine failure is injected in one replica of **Federate1**. In different federation executions, the failure is injected at different execution points between the 10th and 11th checkpoint, and thus, the length of the repeated execution for **Group2** replica of **Federate1** is between 0 to 100 RTI service calls. To further study the performance of the proposed epidemic effect removal mechanisms, the BFT(No-Barrier) scenario is divided into two sub-scenarios: the BFT(No-Barrier)F and BFT(No-Barrier)S, representing the cases where the faulty replica is the faster and the slower one respectively.

7.5.a: CON Federation

7.5.b: OPT Federation

Figure 7.5: Execution times in BFT scenarios in No-Workload case

7.6.a: CON Federation

7.6.b: OPT Federation

Figure 7.6: Execution times in BFT scenarios in Workload case

Figures 7.5 and 7.6 show federation execution times in the BFT scenarios on computing nodes in No-Workload and Workload cases respectively. Compared with the BFT(Barrier) scenario, the BFT(No-Barrier)F and BFT(No-Barrier)S scenarios perform better for both CON and OPT federation execution. The performance advantage is much more significant in Workload case. The BFT(No-Barrier)S scenario outperforms the BFT(No-Barrier)F scenario slightly. It is also observed that the execution time increases linearly with respect to the length of the repeated execution in the **Group2** replica of **Federate1**. It is worthwhile to point out that similar performance can be obtained for

the CON federation execution in the BFT(No-Barrier)F and BFT(No-Barrier)S scenarios regardless of the external workload.

The total overhead of our BFT mechanism can be calculated as the difference between the execution time of the BFT(Barrier) or BFT(No-Barrier) scenario in the presence of the Byzantine failure and the execution time of the Barrier or No-Barrier scenario in the No-Failure case respectively. Besides the above mentioned failure detection overhead, failure location and recovery overheads are also included in the total overhead. The failure location overhead includes transferring federate files, restarting federates, and repeating federate execution from last consistent checkpoint to the failure point. It refers to the time between detecting and locating the Byzantine failure, which can be measured by the BFT module directly. The failure recovery overhead can be calculated by subtracting failure detection and location overheads from the total overhead of our BFT mechanism. Theoretically, failure recovery overhead is close to zero in the BFT(Barrier) and BFT(No-Barrier)S scenarios, as the failure is recovered by simply excluding the faulty replica from federation execution. In the BFT(No-Barrier)F scenario, the failure recovery overhead mainly includes the overhead of our epidemic effect removal mechanisms.

7.7.a: CON Federation

7.7.b: OPT Federation

Figure 7.7: Overhead of BFT scenarios in No-Workload case

Figures 7.7 and 7.8 show fault tolerance overheads in BFT scenarios. In these Figures,

7.8.a: CON Federation

7.8.b: OPT Federation

Figure 7.8: Overhead of BFT scenarios in Workload case

B, NF, and NS stand for the BFT(Barrier), BFT(No-Barrier)F, and BFT(No-Barrier)S scenarios respectively. The failure detection overhead does not change with respect to the length of repeated execution. The failure detection overhead in BFT(Barrier) scenario is greater than that in BFT(No-Barrier) scenarios. The reason is that the execution of replicas in the BFT(Barrier) scenario are blocked during the comparison of RTI services and checkpoints; whereas, the execution of the faster replicas in the BFT(No-Barrier) scenario overlaps with the comparison of RTI services and checkpoints. The failure location overhead increases linearly with respect to the length of the repeated execution. Compared with CON federation execution, the failure location overhead of OPT federation execution is greater. As mentioned above, OPT federates invoke RTI services less frequently. With the same repeated execution length³, OPT federates are likely to process more events. What is worse, they might also encounter execution rollbacks during the repeated execution. Consequently, OPT federates spend more time on failure location. Failure recovery overheads are very low in the BFT(Barrier) and BFT(No-Barrier)S scenarios. However they are much greater in the BFT(No-Barrier)F scenario because of the epidemic effect. We can also observe that the failure detection, location and recovery

³Recall that repeated execution length is measured by the number of RTI service calls

overheads in Workload case are greater than those in No-Workload case.

It is worthwhile to note that the BFT scenarios, with or without failure, produce the same simulation results as the Normal scenario in the No-Failure case. This indicates the correctness of our BFT mechanism.

7.2.3.2 BFT Mechanism Scalability

7.9.a: CON Federation

7.9.b: OPT Federation

Figure 7.9: Scalability of different scenarios in No-Failure case

In order to study the scalability of our BFT mechanism, we increase federation scale from two to eight. Only No-Workload case is investigated in this series of experiments.

The transferred wafer lot is sent to one of the other wafer fabs. Firstly, we investigate the No-Failure case. As shown in Figure 7.9, the execution times of both CON and OPT federation executions in the Normal scenario increase linearly with respect to the federation scale. The same increasing trend can be also observed in other execution scenarios. The No-Barrier scenario outperforms the Normal scenario slightly; whereas, the Barrier scenario performs a little worse than the Normal scenario. Due to the failure detection overhead, federation execution times in the BFT(Barrier) and BFT(No-Barrier) scenarios are greater than those in the Barrier and No-Barrier scenarios respectively. The results in Figure 7.9 show that the failure detection overhead does not increase with the scale of federation.

Figure 7.10: Scalability of BFT scenarios in the presence of Byzantine failrues

Secondly, a failure is injected in a replica of **Federate1** when the federate invokes the 40^{th} RTI service call after the 10^{th} checkpoint. Execution times of the BFT scenarios, as shown in Figure 7.10, increase linearly with the increasing federation scale. We can also observe that the BFT(No-Barrier) scenarios outperform the the BFT(Barrier) scenario; and that the BFT(No-Barrier)S scenario outperforms the BFT(No-Barrier)F scenario slightly.

As shown in Figure 7.11, the total Byzantine fault tolerance overhead does not increase

7.11.a: CON Federation

7.11.b: OPT Federation

Figure 7.11: Overhead of BFT scenarios with respect to federation scale

significantly with the increasing federation scale, which indicates the good scalability of our BFT mechanisms. Similar to two federates case, the failure detection overhead in the BFT(Barrier) scenario is greater than that in the BFT(No-Barrier) scenarios; the failure location overhead in OPT federation execution is greater than that in CON federation execution; and the failure recovery overhead of BFT(No-Barrier)F scenario is significantly greater than other scenarios. We can further observe that the failure recovery overhead in CON federation execution in the BFT(No-Barrier)F scenario obviously increases with the increasing federation scale. This is because the number of federates in epidemic effect increases with the increasing federation scale. Federates in epidemic effect are required to rollback their execution to the latest checkpoint and conduct the repeated execution from the checkpoint to the epidemic point. Due to the federation-wide time synchronization, the federation execution is hold back by the federate in epidemic effect that spends the longest time on the execution rollback. This also explains why the difference between the execution times of the BFT(No-Barrier)S and BFT(No-Barrier)F becomes larger when the federation size increases (see Figure 7.10.a). In contrast, alternative epidemic effect removal mechanism is applied for OPT federation execution. Federates in epidemic ef-

fect rollback their executions when it is necessary, using the state saving and rollback mechanisms provided by optimistic synchronization. Hence, unlike the CON federation execution, there is not an obvious trend that the failure recovery overhead in OPT federation execution in the BFT(No-Barrier)F scenario increases with the increasing federation scale.

7.3 Summary

In this chapter, a three-phases BFT mechanism is proposed for HLA-based simulation. Using the decoupled federate architecture, DRC can survive from the Byzantine failures of federates. It can detect the Byzantine failures by replicating federates and comparing the RTI services and checkpoints of the federate replicas. Only $f+1$ replicas of the same federate are executed to detect the Byzantine failures, where f is the maximum number of simultaneous failures. For failure location and recovery, additional f replicas are started using the checkpoint and message logging. The BFT mechanism is implemented using either Barrier or No-Barrier federate replication structures. To deal with Byzantine failure propagation in the federation execution, epidemic effect removal mechanisms are proposed for CON and OPT federates by rolling back their executions, i.e., reducing logical time, restoring execution state, and retracting arbitrary messages.

Experiments based on a Borderless Fab simulation model are carried out. They have verified that the BFT mechanism ensures correct federation executions in the presence of Byzantine failures. The experiment results show that the BFT mechanism using No-Barrier replication can significantly outperform that using Barrier replication when the replicas of the same federate have different runtime performance. The experimental results have also indicated that our BFT mechanism has good scalability, with acceptable overhead in failure detection, location, and recovery.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Distributed simulation as is has been developed encourages the collaboration among a great number of participants by linking geographically dispersed simulation components. The *High Level Architecture* (HLA) standard IEEE 1516-2000 [56] defines a framework for the reusability and interoperability of simulation components (Federates) in distributed simulations (Federations). *Run-Time Infrastructure* (RTI) is a communication middleware that provides the implementation of the HLA interface specification. Federates in a federation communicate with each other through the underlying RTI.

HLA-based simulation may involve a large number of computationally intensive federates, and thus, is time and resource consuming. Moreover, fast or real time responses are required in many simulation applications (e.g., on-line decision support system). Federates which are developed by different parties and executed on heterogeneous and geographically distributed platforms are subject to crash-stop and/or Byzantine failures. Furthermore, the risk of federation failure increases with increasing federation scale. Therefore, this thesis aims at enhancing simulation execution performance and

simultaneously tolerating failures during simulation execution.

For each federate, an RTI component is provided by the RTI as the communication agent of the federate in the federation. In the traditional federate architecture, the RTI component is a part of the federate and is referred to as *Local RTI Component* (LRC). In the decoupled federate architecture, the RTI component works as a separate process and is referred to as *Decoupled RTI Component* (DRC). By breaking the tight-coupling between the federate and its RTI component, the decoupled federate architecture has some advantages over the traditional federate architecture. By exploiting these advantages, performance enhancement and fault tolerance mechanisms are proposed to achieve the objectives of this thesis. They are implemented on *Service Oriented HLA RTI* (SOHR) [86, 87], which supports the decoupled federate architecture by providing federates with Grid services RTI interface.

Workload imbalance generally leads to poor distributed simulation performance. To achieve load balancing, we propose to migrate federates from heavily-loaded computing nodes to lightly-loaded ones. Using the decoupled federate architecture, only the federate needs to be migrated. **MigFed** and **ResFed** are connected to the federation through the same DRC. This makes the federate migration transparent to the other federates in the federation. In addition to the basic (i.e., one-phase) migration protocol, two-phases and relay-based migration protocols are also proposed in Chapter 3 to reduce the migration overhead by overlapping the migration procedure with the continuous execution of **MigFed**. As experimental results show, the relay-based migration protocol introduces the lowest migration overhead which equals the state saving time in **MigFed**.

In the decoupled federate architecture, federates can be replicated transparently without increasing the federation scale, as the replicas of the same federate are connected to the federation through the same DRC. Barrier and No-Barrier replication structures have been proposed in the literature based on the decoupled federate architecture. The feder-

ation execution with the Barrier federate replication is held back by the slowest replica of each federate. In contrast, the federation execution with the No-Barrier federate replication is pushed forward by the fastest replica of each federate.

Inspired by the concepts of software diversity and active replication, we propose to enhance simulation performance by replicating the federate with replicas that employ either a conservative (CON replica) or an optimistic (OPT replica) synchronization protocol and choosing the fastest replica to represent the federate in the federation. Unfortunately, the existing Barrier and No-Barrier replication structures, which require replicas of the same federate to be Piecewise Equivalent, cannot support the concurrent executions of CON and OPT replicas. For this purpose, an Un-Identical replication structure is proposed in Chapter 4. It is composed of message exchange mechanism and TM mechanism. The former is in charge of message exchange amongst federates and message delivery for the replicas of the same federate; and the latter performs federation wide time synchronization and coordinates the time advancements of CON and OPT replicas. Besides presenting the design and implementation details, correctness proof is also provided. Experiments using P-hold simulation model have demonstrated that the Un-Identical replication structure is able to enhance simulation performance while producing correct simulation results. Experimental results have also indicated that the Un-Identical replication structure exhibits good scalability.

Since the DRC is decoupled from the federate, it can survive from the federate failures and keep the connection to the federation. A crash-stop fault tolerance mechanism using checkpoint and message logging techniques is proposed in Chapter 5. The crashed federate is recovered without interrupting the executions of other federates in the federation. A basic state recovery protocol is first developed to recover the state of the crashed federate by rolling back its execution to the latest checkpoint and repeating execution identically to that before the failure. Then, an optimized protocol is further developed

to accelerate the state recovery procedure. Experiments using P-hold simulation model show that the proposed mechanism can tolerate federate failures at the price of acceptable overhead. In addition, the optimized protocol can outperform the basic one, especially for OPT federates.

In the Un-Identical federate replication structure, the federation execution will not be interrupted if there is at least one replica of each federate that can survive the failures. However, the performance of the federation execution may degrade due to the loss of the failed replica(s). Therefore, the Un-Identical federate replication structure is extended in Chapter 6, so that it keeps the performance advantages in the presence of failures. A new replica is established to substitute the crashed one. Mechanisms are proposed to initialize the new replica using the execution state of a working replica which employs a different synchronization approach. Experiments using P-hold simulation model are carried out to illustrate the benefit of extending Un-Identical replication structure in terms of fault tolerance and performance enhancement.

A three-phases BFT mechanism using either Barrier or No-Barrier federate replication structure is proposed in Chapter 7. Since the DRC keeps the connection to the federation and masks the federate replication from the federation, any number of replicas can be started and terminated during the runtime. To detect Byzantine failure, $f + 1$ replicas are required, where f is the maximum number of simultaneous failures. After a failure is detected, additional f replicas are started using the checkpoint and message logs for failure location and recovery. The Byzantine failure is recovered by excluding the faulty replica from the federation execution and removing the epidemic effect of the Byzantine failure from the federation execution. Experiments using a Borderless Fab simulation model verify that the BFT mechanism ensures correct federation executions in the presence of Byzantine failures. The experimental results show that the BFT mechanism using No-Barrier replication can outperform that using Barrier replication significantly when

the replicas of the same federate exhibit different runtime performances. The experimental results also indicate that our BFT mechanism has good scalability with acceptable overhead introduced in failure detection, location, and recovery.

In summary, the objectives of this thesis as defined in Subsection 1.3 are accomplished.

8.2 Future Work

The BFT mechanism proposed in Chapter 7 is only implemented using either Barrier or No-Barrier federate replication structure. In the future, the BFT mechanism will be implemented on the Un-Identical federate replication structure for the purpose of enhancing simulation performance and supporting fault tolerance simultaneously.

As we know, CON and OPT replicas of the same federate might have different execution states and invoke different RTI services following different sequences. Hence, new methods should be proposed to detect and locate Byzantine failures, instead of simply comparing their execution states and the invoked RTI service calls. As mentioned in Chapter 4, CON and OPT replicas, which are committed equivalent, must process the same committed events in the same manner in TS order. Mechanisms are proposed in Chapter 6 to transfer execution state from CON to OPT replica, and vice versa. Inspired by these mechanisms, CON and OPT replicas can report their abstract checkpoints, including the latest safe snapshot of application specific state and correct events. In this way, Non-faulty replicas should report the same abstract checkpoints with the same checkpoint time. For message delivery, CON and OPT replicas are allowed to send different messages in different sequences. However, non-faulty replicas should send the same committed messages (i.e., the messages with $SVT \leq \text{logical time of the sender}$). So, we can detect and locate the Byzantine failures by comparing the abstract checkpoint and committed messages of CON and OPT replicas. New replicas, if necessary, can be

started using the consistent abstract checkpoints and message log. Similar to No-Barrier replication structure, the fastest replica is chosen to represent the federate in the Un-Identical replication structure. Hence, a Byzantine failure might also propagate in the federation execution before it is detected. So, the epidemic effect removal mechanisms proposed in Chapter 7 should be also adapted to the Un-Identical replication structure.

Federate migration and fault tolerance mechanisms proposed in this thesis require cooperation of the federate developers. They should provide federate state saving and restoring functions as described in the corresponding mechanisms. In the future, the mechanisms proposed by Santoro and Quaglia [100, 101] can be adopted for the purpose of providing a high level of transparency and efficiency for federate state saving and restoring.

Both conservative and optimistic synchronization approaches are considered in this thesis. However, the level of data structure and RTI services support for TM synchronization, especially for optimistic synchronization, is too low to be easily used by simulation application developers. To relieve the OPT federates from execution state saving and rollback, a rollback controller is proposed in [112] using a middleware approach; a rollback manager is created in [109] using computational reflection techniques; and a transparent state manager is integrated within the RTI in [100] using transparent and machine-independent checkpointing and recovery software architecture. Chiola and Ferscha [26] have demonstrated how to implement multiple different synchronization approaches simultaneously relying on a single program skeleton. A *Smart Time Management* (STM) which also is described in [116], hides the synchronization issues from the federate and provide the federate with a unanimous time management interface. By applying aforementioned approaches, mechanisms can be developed to support TM synchronization on behalf of CON and OPT federates and to provide more friendly interface to federate developers.

Running a large-scale distributed simulation may need a large amount of geographically distributed resources. However, the availability and performance of the resource may change during simulation execution. Therefore, coordinating and managing the resources is a critical issue for efficient and fault tolerant distributed simulation. As described in this thesis, a workload balance module is deployed in RTI Index Service for the purpose of resource management. It can be built on existing resource management systems, e.g., Grid resource sharing system provided by GT [51] and Cloud load balancers provided by Rackspace [97]. Consequently, algorithms can be developed to map federates to the available resources dynamically and to make the federate migration decisions, taking workload balance, communication efficiency, and fault tolerance issues into account. In addition, different load balancing algorithms [76, 108] can be proposed for the distributed simulation, depending on its employed synchronization approach (i.e., either a conservative or an optimistic synchronization approach).

In summary, there are many future works remaining in order to complete a user-friendly facility for efficient and fault tolerant HLA-based simulation on unpredictable distributed execution environment. Furthermore, our works should also be migrated from the HLA standard IEEE 1516-2000 [56] to the updated HLA standard IEEE 1516-2010 [55], exploiting the latter's new features [82], e.g., Web Services (WSDL) support/API, smart update rate reduction [80], and fault tolerance support services.

Appendix A

Author's Publications

Journal Paper:

- (1) **Z. Li**, S. J. Turner, W. Cai and K. Pan, “A Federate Replication Structure for Simulation Performance Enhancement via Employing Different Synchronization Approaches”. Submitted to *ACM Transactions on Modeling and Computer Simulation*
- (2) **Z. Li**, S. J. Turner, W. Cai and K. Pan, “Byzantine Fault Tolerance for HLA-based Simulation”. Submitted to *ACM Transactions on Modeling and Computer Simulation*
- (3) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “A Hybrid HLA Time Management Algorithm Based on Both Conditional and Unconditional Information”. *Simulation*, Vol. 85, No. 9, Pages 559-573, 2009.
- (4) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “A Dynamic Sort-Based DDM Matching Algorithm for HLA Applications”. *ACM Transactions on Modeling and Computer Simulation*, Vol. 21, No. 3, 2011.

Book Chapter:

- (1) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “Design and Performance Evaluation of a Service-Oriented HLA RTI on the Grid”. *Grid Computing: Infrastructure, Service, and Applications, Chapter 18*, Eds. L. Wang, W. Jie and J. Chen, ISBN: 978-1420067668, CRC Press, April 2009.

Conference Paper:

- (1) **Z. Li**, W. Cai, S. J. Turner and K. Pan, “Federate Migration in a Service Oriented HLA RTI”. In *Proceeding of 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'07)*, Pages 113-121, 2007.
- (2) **Z. Li**, W. Cai, S. J. Turner and K. Pan, “Improving Performance by Replicating Simulations with Alternative Synchronization Approaches”. In *Proceedings of Winter Simulation Conference (WSC'08)*, Pages 1112-1120, 2008.
- (3) **Z. Li**, W. Cai, S. J. Turner, and K. Pan, “A Replication Structure for Efficient and Fault-Tolerant Parallel and Distributed Simulations”. In *Proceeding of the 43rd Annual Simulation Symposium*, Pages 151:1-151:10, 2010.
- (4) **Z. Li**, W. Cai, S. J. Turner, and K. Pan, “Federate Fault Tolerance in HLA-based Simulation”. In *Proceeding of the 24th ACM/IEEE/SCS International Workshop on Principles of Advanced and Distributed Simulation (PADS'10)* 2010.
- (5) **Z. Li**, W. Cai, S. J. Turner and K. Pan, “A Three-phases Byzantine Fault Tolerance Mechanism for HLA-based Simulation”. In *Proceeding of the 14th ACM/IEEE International Symposium on Distributed Simulation and Real-Time Applications, (DS-RT'10)*, Pages 149-158, 2010 (Best Paper Candidate)

- (6) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “An Efficient Sort-Based DDM Matching Algorithm for HLA Applications with a Large Spatial Environment”. In *Proceedings of the 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS'07)*, Pages 70-82, 2007.
- (7) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “Service Oriented HLA RTI on the Grid”. In *Proceedings of the 5th International Conference on Web Services (ICWS'07)*, Pages 984-992, 2007.
- (8) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “A Hybrid HLA Time Management Algorithm Based on Both Conditional and Unconditional Information”. In *Proceedings of the 22th Workshop on Parallel and Distributed Simulation (PADS'08)*, Pages 203-211, 2008. (Best Paper Candidate)
- (9) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “implementation of Data Distribution Management Services in a Service Oriented HLA RTI”. In *Proceedings of Winter Simulation Conference (WSC'09)*, Pages 203-215, 2009
- (10) K. Pan, S. J. Turner, W. Cai and **Z. Li**, “Multi-User Gaming on the Grid using a Service Oriented HLA RTI”. In *Proceedings of the 13th ACM/IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'09)* Pages 48-56, 2009. (Best Paper Award)

References

- [1] A. Agbaria and R. Friedman. A replication- and checkpoint-based approach for anomaly-based intrusion detection and recovery. In *Procs of the 2nd International Workshop on Security in Distributed Computing Systems*, pages 137–143, 2005.
- [2] D. Agrawal and J. R. Agre. Recovering from multiple process failures in the Time Warp mechanism. *IEEE Trans. Comput.*, 41(12):1504–1514, 1992.
- [3] D. Agrawal and J. R. Agre. Replicated objects in Time Warp simulations. In *Procs of the 24th conference on Winter simulation*, pages 657–664, 1992.
- [4] E. E. Ajaltouni, A. Boukerche, and M. Zhang. An efficient dynamic load balancing scheme for distributed simulations on a Grid infrastructure. In *Procs of the 12th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '08)*, pages 61–68, 2008.
- [5] K. Al-Zoubi and G. Wainer. Using REST web-services architecture for distributed simulation. In *Procs of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 114–121, 2009.
- [6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of Cloud computing. *Commun. ACM*, 53:50–58, April 2010.
- [7] R. Ayani. A parallel simulation scheme based on the distance between objects. In *Procs of the SCS Multiconference on Distributed Simulation*, pages 113–118, 1989.
- [8] H. Aydt, S. J. Turner, W. Cai, and M. Y. H. Low. Symbiotic simulation systems: An extended definition motivated by symbiosis in biology. In *Procs of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS '08)*, pages 109–116, 2008.
- [9] D. Ball and S. Hoyt. The adaptive Time-Warp concurrency control algorithm. In *Procs of the SCS Multiconference on Distributed Simulation*, pages 174–177, 1990.

- [10] L. G. Ben Watrous and D. Wood. HLA federation performance: What really matters? In *Procs of Fall Simulation Interoperability Workshop*, 2006.
- [11] C. Berchtold and M. Hezel. An architecture for fault tolerant HLA-Based simulation. In *Procs of the 15th International European Simulation Multi-Conference (ESM'01)*, pages 616–620, 2001.
- [12] L. Bononi, M. Bracuto, G. D'Angelo, and L. Donatiello. Concurrent replication of parallel and distributed simulations. In *Procs of 19th Workshop on Parallel and Distributed Simulation (PADS '05)*, pages 234–243, 2005.
- [13] A. Boukerche and R. E. D. Grande. Dynamic load balancing using Grid services for HLA-Based simulations on large-scale distributed systems. In *Procs of the 13th Symposium on Distributed Simulation and Real Time Applications (DS-RT '09)*, pages 175–183, 2009.
- [14] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32:824–840, October 1985.
- [15] R. E. Bryant. Simulation of packet communication architecture computer systems. Technical report, Massachusetts Institute of Technology. Cambridge, MA, USA, 1977.
- [16] W. Cai, S. J. Turner, and H. Zhao. A load management system for running HLA-based distributed simulations over the Grid. In *Procs of the 6th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '02)*, pages 7–14, 2002.
- [17] W. Cai, Z. Yuan, M. Y.-H. Low, and S. J. Turner. Federate migration in HLA-based simulation. *Future Generation Computer System*, 21(1):87–95, 2005.
- [18] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Procs of the 3rd symposium on Operating systems design and implementation*, pages 173–186, 1999.
- [19] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3:63–75, 1985.
- [20] K. M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1979.
- [21] J.-B. Chaudron, E. Noulard, and P. Siron. Design and modeling techniques for real-time RTI time management. In *Procs of Spring Simulation Interoperability Workshop*, number 11S-SIW-045, 2011.

- [22] D. Chen, S. J. Turner, and W. Cai. A framework for robust HLA-based distributed simulations. In *Procs of 20th Workshop on Principles of Advanced and Distributed Simulation (PADS '06)*, pages 183–192, 2006.
- [23] D. Chen, S. J. Turner, W. Cai, B. P. Gan, and M. Y. H. Low. Algorithms for HLA-based distributed simulation cloning. *ACM Transactions on Modeling and Computer Simulation*, 15(4):316–345, 2005.
- [24] D. Chen, S. J. Turner, W. Cai, and M. Xiong. A decoupled federate architecture for high level architecture-based distributed simulation. *J. Parallel Distrib. Comput.*, 68(11):1487–1503, 2008.
- [25] V. Chernyakhovsky, P. Frey, R. Radhakrishnan, P. A. Wilsey, P. Alexander, and H. W. Carter. A formal framework for specifying and verifying time warp optimizations. In *Procs of 4th Workshop Formal Methods for Parallel Programming: Theory and Applications (FMPPTA '99)*, pages 1228–1242, 1999.
- [26] G. Chiola and A. Ferscha. Performance comparable design of efficient synchronization protocols for distributed simulation. In *Procs of 3rd International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems(MASCOTS'95)*, pages 343–348, 1995.
- [27] J. Cleary, F. Gomes, B. Unger, Z. Xiao, and R. Thudt. Cost of state saving & rollback. In *Procs of the 8th workshop on Parallel and distributed simulation (PADS '94)*, pages 94–101, 1994.
- [28] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Procs of the 7th symposium on Operating systems design and implementation*, pages 177–190, 2006.
- [29] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.
- [30] D. Cucuzzo, S. D'Alessio, F. Quaglia, and P. Romano. A lightweight heuristic-based mechanism for collecting committed consistent global states in optimistic simulation. In *Procs of the 11th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '07)*, pages 227–234, 2007.
- [31] O. P. Damani and V. K. Garg. Fault-tolerant distributed simulation. In *Procs of the 12th workshop on Parallel and distributed simulation (PADS '98)*, pages 38–45, 1998.

- [32] R. De Grande and A. Boukerche. Dynamic partitioning of distributed virtual simulations for reducing communication load. In *Procs of the Workshop on Haptic Audio visual Environments and Games (HAVE '09)*, pages 176–181, 2009.
- [33] Defense Modeling and Simulation Office. *High Level Architecture RTI 1.3NG Programmers Guide Version 5*, 2002.
- [34] W. D. Ehrenberger. Software diversity - some considerations on failure dependency. In *Procs of the World Multiconference on Systemics, Cybernetics and Informatics (ISAS-SCIs '01)*, pages 577–582, 2001.
- [35] M. Eklöf, F. Moradi, and R. Ayani. A framework for fault-tolerance in HLA-based distributed simulations. In *Procs of the 37th conference on Winter simulation (WSC '05)*, pages 1182–1189, 2005.
- [36] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.
- [37] R. Ewald, S. Leye, and A. M. Uhrmacher. An efficient and adaptive mechanism for parallel simulation replication. In *Procs of the 23rd Workshop on Principles of Advanced and Distributed Simulation (PADS '09)*, pages 104–113, 2009.
- [38] A. Ferscha and G. Chiola. Self-adaptive logical processes: the probabilistic distributed simulation protocol. In *Procs of the 27th Annual Simulation Symposium*, pages 78–88, 1994.
- [39] A. Ferscha, J. Johnson, and S. J. Turner. Distributed simulation performance data mining. *Future Generation Computer System*, 18:157–174, 2001.
- [40] J. Fleischmann and P. A. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Procs of the 9th workshop on Parallel and distributed simulation (PADS '95)*, pages 50–58, 1995.
- [41] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid - enabling scalable virtual organizations. *International J. Supercomputer Applications*, 15(3):200–222, 2001.
- [42] R. Fujimoto, D. Lunceford, E. Page, and A. Uhrmacher. Technical report of the dagstuhl-seminar grand challenges for modelling and simulation. Available via <http://www.dagstuhl.de/02351/Report/.2002>, 2002.
- [43] R. M. Fujimoto. Lookahead in parallel discrete event simulation. In *Procs of the International Conference on Parallel Processing*, pages 34–41, 1988.

- [44] R. M. Fujimoto. Performance of Time Warp under synthetic workloads. In *Procs of the SCS Multiconference on Distributed Simulation*, pages 23–28, 1990.
- [45] R. M. Fujimoto. Time management in the high level architecture. *SIMULATION Special Issue on High Level Architecture*, 71(6):388–400, 1998.
- [46] R. M. Fujimoto. *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.
- [47] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. *Distributed Simulation*, pages 61–67, 1988.
- [48] V. K. Garg. *Elements of distributed computing*. John Wiley & Sons, Inc., 2002.
- [49] Georgia Tech. Federated simulations Development Kit (FDK). Available via <http://www.cc.gatech.edu/computing/pads/fdk.html>.
- [50] D. Gianni, G. Iazeolla, and A. D’Ambrogio. A methodology to predict the performance of distributed simulations. In *Procs of the 24rd Workshop on Principles of Advanced and Distributed Simulation (PADS ’10)*, 2010.
- [51] Globus Alliance. Globus Toolkit version 4. Available via <http://www.globus.org/>.
- [52] B. Grošelj. Fault-tolerant distributed simulation. In *Procs of the 23rd conference on Winter simulation (WSC ’91)*, pages 637–641, 1991.
- [53] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Procs of Ada-Europe International Conference on Reliable Software Technologies*, pages 38–57, 1996.
- [54] M. Hybinette and R. M. Fujimoto. Scalability of parallel simulation cloning. In *Procs of the 35th Annual Simulation Symposium*, page 275, 2002.
- [55] IEEE. *1516-2010 IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules*, August 2010.
- [56] IEEE 1516-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Framework and Rules*, September 2000.
- [57] IEEE 1516.1-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - Federate Interface Specification*, September 2000.
- [58] IEEE 1516.2-2000. *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - HLA Object Model Template (OMT) Specification*, September 2000.
- [59] IEEE 1516.3-2003. *IEEE Recommended Practice for High Level Architecture (HLA) Federation Development and Execution Process (FEDEP)*, April 2003.

- [60] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, 1985.
- [61] V. Jha and R. L. Bagrodia. A unified framework for conservative and optimistic distributed simulation. *ACM SIGSIM Simulation Digest*, 24(1):12–19, 1994.
- [62] D. B. Johnson. *Distributed system fault tolerance using message logging and checkpointing*. PhD thesis, Houston, TX, USA, 1990.
- [63] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. In *Digest of Papers: 17th Annual International Symposium on Fault-Tolerant Computing*, pages 14–19, 1987.
- [64] T. Kiesling. Fault-tolerant distributed simulation: A position paper. Available via <http://www.unibw.de/inf4/personen/wm/t.kiesling/misc/ftds-position-paper.pdf>, 2003.
- [65] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):1–39, 2009.
- [66] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [67] P. Lendermann, B. P. Gan, Y. L. Loh, H. K. Tan, S. K. Lieu, L. F. McGinnis, and J. W. Fowler. Analysis of a borderless fab scenario in a distributed simulation testbed. In *Procs of the 36th conference on Winter simulation (WSC '04)*, pages 1896–1901, 2004.
- [68] Y. Liang, S. J. Turner, and B. P. Gan. Predictive-conservative synchronization for commercial simulation package interoperability. In *Procs of the 40th Conference on Winter Simulation (WSC '08)*, pages 1103–1111, 2008.
- [69] Y.-B. Lin, B. R. Preiss, W. M. Loucks, and E. D. Lazowska. Selecting the checkpoint interval in Time Warp simulation. In *Procs of the 7th workshop on Parallel and distributed simulation (PADS '93)*, pages 3–10, 1993.
- [70] B. D. Lubachevsky. Efficient distributed event driven simulations of multiple-loop networks. *SIGMETRICS Perform. Eval. Rev.*, 16:12–24, May 1988.
- [71] J. Lüthi and C. Berchtold. Concepts for dependable distributed discrete event simulation. In *Procs of the 14th European Simulation Multiconference on Simulation and Modelling*, pages 59–66, 2000.
- [72] J. Lüthi and S. Großmann. The resource sharing system: dynamic federate mapping for HLA-based distributed simulation. In *Procs of the 15th workshop on Parallel and distributed simulation (PADS '01)*, pages 91–98, 2001.

- [73] J. Lüthi and S. Großmann. FT-RSS: a flexible framework for fault tolerant HLA federations. In *Procs of International Conference on Computational Science*, pages 865–872, 2004.
- [74] M. R. Lyu. *Software Fault Tolerance*. John Wiley & Sons Ltd, 1995.
- [75] M. R. Lyu, J. hong Chen, and A. Avizienis. Software diversity metrics and measurements. In *Procs of the 16th Annual Int. Computer Software and Applications Conference (COMPSAC '92)*, pages 69–78, 1992.
- [76] S. Meraji, W. Zhang, and C. Tropper. A multi-state Q-learning approach for the dynamic load balancing of time warp. In *Procs of the 24rd Workshop on Principles of Advanced and Distributed Simulation (PADS '10)*, 2010.
- [77] M. Merideth. Tradeoffs in Byzantine-Fault-Tolerant State-Machine-Replication protocol design. Technical Report CMU-ISR-08-110, Institute for Software Research, Carnegie Mellon University, March 2008.
- [78] D. S. Milojicic, F. Douglass, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.
- [79] B. Möller. A first look at the HLA Evolved Web Service API. In *Procs of the European Simulation Interoperability Workshop*, number 06E-SIW-061, 2006.
- [80] B. Möller and M. Karlsson. Developing well-balanced federations using the HLA evolved smart update rate reduction. In *Procs of Spring Simulation Interoperability Workshop*, number 05F-SIW-087, 2005.
- [81] B. Möller, M. Karlsson, and B. Löfstrand. Developing fault tolerant federations using HLA evolved. In *Procs of Spring Simulation Interoperability Workshop*, number 05S-SIW-048, 2005.
- [82] B. Möller, K. L. Morse, M. Lightner, R. Little, and R. Lutz. HLA evolved — a summary of major technical improvements. In *Procs of Spring Simulation Interoperability Workshop*, number 08f-SIW-064, 2008.
- [83] D. M. Nicol. Noncommittal barrier synchronization. *Parallel Comput.*, 21:529–549, April 1995.
- [84] E. Niewiadomska-Szynkiewicz and A. Sikora. Algorithms for distributed simulation - comparative study. In *Procs of Conference on Parallel Computing in Electrical Engineering (PARELEC'02)*, pages 261–266, 2002.

- [85] OASIS. *Web Services Resource Framework (WSRF)*.
- [86] K. Pan, S. J. Turner, W. Cai, and Z. Li. A Service Oriented HLA RTI (SOHR). Available via <http://pdcc.ntu.edu.sg/sohr/>.
- [87] K. Pan, S. J. Turner, W. Cai, and Z. Li. A service oriented HLA RTI on the Grid. In *Procs of Conference on Web Services (ICWS 2007)*, pages 984–992, 2007.
- [88] K. Pan, S. J. Turner, W. Cai, and Z. Li. A hybrid HLA time management algorithm based on both conditional and unconditional information. *Simulation*, 85:559–573, September 2009.
- [89] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful web services vs. big web services: Making the right architectural decision. In *Procs of the 17th International World Wide Web Conference (WWW '08)*, pages 805–814, 2008.
- [90] K. S. Perumalla. μ sik - a micro-kernel for parallel/distributed simulation systems. In *Procs of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS '05)*, pages 59–68, 2005.
- [91] B. Ping Gan, M. Yoke Hean Low, X. Wang, and S. J. Turner. Using manufacturing process flow for time synchronization in HLA-based simulation. In *Procs of the 9th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '05)*, pages 148–160, 2005.
- [92] Pitch. Pitch prti. Available via <http://www.pitch.se>.
- [93] F. Quaglia. Enhancing the performance of HLA-based simulation systems via software diversity and active replication. In *Procs of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, 2006.
- [94] F. Quaglia. A middleware level active replication manager for high performance HLA-based simulations on smp systems. In *Procs of 10th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '06)*, pages 219–226, 2006.
- [95] F. Quaglia. Software diversity-based active replication as an approach for enhancing the performance of advanced simulation systems. *International Journal of Foundations of Computer Science*, 18:495–515, 2007.
- [96] A. J. P. R. M. Fujimoto, A. W. Malik. Parallel and distributed simulation in the Cloud. *SCS Modeling and Simulation Magazine, Society for Modeling and Simulation, Intl.*, 1, July 2010.

- [97] Rackspace. Cloud load balancers. Available via <http://www.rackspace.com/>.
- [98] H. Rajaei, R. Ayani, and L.-E. Thorelli. The local Time Warp approach to parallel simulation. *ACM SIGSIM Simulation Digest*, 23(1):119–126, 1993.
- [99] B. Samadi. *Distributed simulation, algorithms and performance analysis*. PhD thesis, University of California, Los Angeles, 1985.
- [100] A. Santoro and F. Quaglia. A version of MASM portable across different UNIX systems and different hardware architectures. In *Procs of the 9th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '05)*, pages 35–42, 2005.
- [101] A. Santoro and F. Quaglia. Transparent state management for optimistic synchronization in the high level architecture. *Simulation*, 82(1):5–20, 2006.
- [102] L. M. Sokol, D. P. Briscoe, and A. P. Wieland. MTW: A strategy for scheduling discrete simulation events for concurrent execution. In *Procs of the SCS Multiconference on Distributed Simulation*, pages 34–42, 1988.
- [103] B. Sotomayor. The Globus Toolkit 4 programmers tutorial. Available via <http://gdp.globus.org/gt4-tutorial/>.
- [104] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.
- [105] G. Tan and K. C. Lim. Load distribution services in HLA. In *Procs of the 8th Symposium on Distributed Simulation and Real-Time Applications (DS-RT '04)*, pages 133–141, 2004.
- [106] G. S. H. Tan, A. Persson, and R. Ayani. HLA federate migration. In *Procs of Annual Simulation Symposium*, pages 243–250, 2005.
- [107] G. S. H. Tan, A. Persson, and R. Ayani. Migration of HLA federates. In *Procs of Spring Simulation Interoperability Workshop*, number 05S-SIW-058, 2005.
- [108] S. Thulasidasan, S. Kasiviswanathan, S. Eidenbenz, and P. Romero. Explicit spatial scattering for load balancing in conservatively synchronized parallel discrete event simulations. In *Procs of the 24rd Workshop on Principles of Advanced and Distributed Simulation (PADS '10)*, 2010.
- [109] F. Vardânega and C. Maziero. Using computational reflection in optimistic distributed simulations. In *Procs of the 19th Conference of the Chilean Computer Science Society*, pages 1–8, 1999.

- [110] W3C. Simple Object Access Protocol (SOAP). Available via <http://www.w3.org/tr/soap>.
- [111] W3C. Web Services Description Language (WSDL) 1.1 . Available via <http://www.w3.org/TR/wsdl>.
- [112] X. Wang, S. J. Turner, M. Y. H. Low, and B. P. Gan. Optimistic synchronization in HLA-based distributed simulation. *Simulation*, 81:279–291, April 2005.
- [113] D. West. *Lazy rollback and Lazy re-evaluation*. Master’s thesis, University of Calgary, Calgary, Alberta, 1988.
- [114] Y. Xie, Y. M. Teo, W. Cai, and S. J. Turner. Servicing provisioning for HLA-Based distributed simulation on the Grid. In *Procs of 19th IEEE/ACM/SCS Workshop on Parallel and Distributed Simulation (PADS ’05)*, pages 282–291, 2005.
- [115] S. Xu and L. F. McGinnis. Optimistic-conservative synchronization in distributed factory simulation. In *Procs of the 38th conference on Winter simulation (WSC ’06)*, pages 1069–1074, 2006.
- [116] J. yao Huang, M.-C. Tung, K. M. Wang, and M.-C. Lee. Smart time management - the unified time synchronization interface for the distributed simulation. *Computer Standards & Interfaces*, 27(2):149–161, 2005.
- [117] Z. Yuan, W. Cai, Y.-H. Low, and S. J. Turner. Federate migration in HLA-Based simulation. In *Procs of Conference on Computational Science*, pages 856–864, 2004.
- [118] K. Zajac, M. Bubak, M. Malawski, and P. M. A. Sloot. Towards a Grid management system for HLA-Based interactive simulations. In *Procs of 7th Symposium on Distributed Simulation and Real-Time Applications (DS-RT ’03)*, pages 4–11, 2003.