Load management system for distributed simulation

Yuan, Zijing

2005

https://hdl.handle.net/10356/2563

https://doi.org/10.32657/10356/2563

NANYANG TECHNOLOGICAL UNIVERSITY

Load Management System for Distributed Simulation

A thesis submitted to the Nanyang Technological University
in Fulfillment of the Requirement for the degree of
Master of Engineering

by

Yuan Zijing

Supervisor: Assoc. Prof. Cai Wentong

School of Computer Engineering

2005

Acknowledgement

This project cannot have reached this stage without the help of my supervisor, ***Dr. Cai Wentong***. I am grateful to his thorough help during my graduate study at NTU. He helps me clarify my design and new concepts in distributed simulations. Without his patience, understanding and encouragement, I cannot stick to and dip into my project, and the project would not be successful.

I would like to thank ***Dr. Low Yoke Hean, Malcolm*** for his patience to listen and discuss with me about the project. His valuable advice inspires me greatly.

I would also like to extend my thanks to staff and technicians in the Parallel and Distributed Computing Center, they kindly assisted me in many aspects, and provided me with all the convenience I need.

Last but not least, I would like to thank all my friends Ms. Xue Ning, Mr. Wang Lizhe, Mr. Zeng Yi and Ms. Wang Lihua, who helped me with the difficulties I met.

.

Abstract

Network of workstations are gaining popularity against parallel and supercomputers for running large-scale simulations. The Grid enables resource sharing of computing resources at geographically distributed locations and provides the simulations with immense computing power. At the meantime, the High Level Architecture (HLA) paradigm provides a software platform and interoperability interface for simulation components to utilize distributed resources. Running HLA-based simulation over the Grid environments would shorten the simulation time, improve the hardware availability and be beneficial for the simulation community.

Despite the convenience and benefits the Grid can provide, issues that hinder the adoption of Grid as the simulation hardware platform still remain. The HLA lacks the capability of resource management for running simulation in the aspect of resource discovery, federate deployment, dynamic load-balancing and fault-tolerance. Hence, this project is motivated to develop a programming framework to facilitate the HLA simulation development, with the consideration for possible integration with the Grid environment.

Both dynamic load-balancing and fault-tolerance require frequent checkpointing of the simulation and saving the simulation state. Consequently, the amount of information saved will greatly impact the simulation performance. HLA simulation normally has a large amount of information to be extracted and saved, hence requiring more time and computing power for federate migration and fault recovery. It is of great importance to develop a federate model that supports easy checkpointing and has minimal state information for saving and restoration. It is also noticed that substantial effort is required for writing programs that conform to the HLA Runtime Infrastructure (RTI) specification because of its complexity. In this project, a framework with easy checkpointing capability was developed, and a code library was implemented. The code library is meant for automatic code generation from user design with a Graphical User Interface (GUI), which is part of the framework. HLA Data Distribution Management is used to route simulation events (interactions) to achieve efficient use of network bandwidth.

Federate migration protocols have been introduced by various research projects. However, existing protocols achieve migration by using either federation wide synchronization or a third party host, such as FTP servers, to handle the saving and restoring of migration states. Based on the framework, federate migration protocols that bypass the shortcomings identified above were developed and better migration performance is achieved. To eliminate message loss during migration process, a counter mechanism was employed. Study also shows that federate join time contributes significantly to the migration overhead. Therefore, our protocol was further modified to further reduce the migration overhead incurred.

Contents

List of Figures

List of Tables

.

Chapter 1

Introduction

Computer simulation has become an important tool for investigating and evaluating complicated scenarios in the area of industrial production, business, financial service, education, science and military [14]. With the recent expansion and advancement in the computing industry, computer hardware price has been declining. This makes network of workstations a more viable choice for large-scale parallel and distributed simulation (PADS) [14] due to its high performance, high availability, high scalability and low cost.

1.1 Background

1.1.1 Distributed Simulation

Distributed simulation is the simulation of some real or imagined system on distributed computers. Distributed computers are connected by network and may be spatially separated by any distance. Distributed computers are characterized by *heterogeneity* and the network used to interconnect the machines. Distributed systems are often composed of stand-alone computers from different vendors, running different operating systems (OSes). Interoperability and reusability eliminate the need to transform existing simulations to new platforms, and enable participation of modelers with different computing equipment in distributed simulations.

By running simulation on distributed systems, the simulation execution time is reduced, and geographical distribution of the simulation is also made available. Distributed simulation also improves the fault tolerance capability. If one computer goes down, other computer may take up the unfinished job of the failed node and keep the simulation going despite the failure.

Distributed simulation is also characterized by its lack of global clock. Simulation components at individual computers have no knowledge of the correct time of the simulation. Fortunately, simulation frameworks usually provide mechanisms for synchronization and time management.

The High Level Architecture (HLA) was developed by the Defense Modelling and Simulation Office (DMSO) to provide a common architecture that facilitates simulation interoperability and reusability across all classes of simulations in the distributed environment. HLA has been adopted as IEEE standard 1516 in September 2000 [22].

In the HLA, a distributed simulation is called a federation, and each individual simulation component is referred to as a federate. Each federate is a single point of attachment to the Runtime Infrastructure (RTI), the software implementation of the HLA standard. Federates in the same federation communicate with each other by passing *interactions*[1] and update objects. A federate can be a computer simulation, it can also be an instrumented physical device or a passive data viewer. HLA simulation must conform to the federation rules [22], the HLA interface spec-

[1]Interaction is the HLA definition of transient message. In this thesis, we will use interaction, message and event interchangeably.

ification [23] and the Object Model Template [24].

1.1.2 Grid Computing

Grid computing has emerged as an important computing technology focusing on large-scale resource sharing at geographically distributed organizations [13]. The Grid integrates "networking, communication, computation and information to provide a virtual platform for computation and data management in the same way that the Internet integrates resources to form a virtual platform for information" [4].

The Grid infrastructure provides the ability to dynamically link together resources to support the execution of large-scale resource-intensive, and distributed applications. The heart of any Grid is the tool that links together geographically distributed resources and allows a single application to execute using these resources collectively. The resources on the Grid can be of any level of computing power and capability, some of them are high-performance machines or clusters. Such high-performance machines form the Grid 'nodes' which provide major resources for simulation, data mining and other computing-intensive executions.

Currently available toolkits for Grid computing include Globus [12], Legion [16], MultiCluster [45] and Sun Grid Engine [41]. Among these toolkits, Globus is the most popular one for its availability and completeness. The Globus project aims to develop fundamental technologies needed to build computational grids. Globus Toolkit provides convenient services to users to fully utilize the Grid resources. Grid Security Infrastructure (GSI) provides services for authentication, communication protection and authorization. The Grid Resource Allocation and Management (GRAM) service provides resource management services and allows programs to be restarted on remote resources, despite local heterogeneity. The Grid Information Service (GIS) provides access to static and dynamic information regarding heterogeneous resources. Monitoring and Discovery Service (MDS) simplifies Grid information services by providing a single standard interface and schema for the many information services that are used within a virtual organization.

1.2 Motivation

Although the Globus provides the mechanisms for forming large-scale resource sharing and HLA provides the convenience for simulation interoperability, neither the Grid nor the HLA/RTI provides the functionality of *resource management* for distributed simulations. Resource management for running distributed simulation over the Grid environment includes resource discovery at runtime and load management.

Resource discovery collects the available computing resource for the HLA-based simulation deployment. Executing simulation in distributed environment may shorten simulation execution. However, the imbalance of load level at the participating computing nodes may degrade the performance gain resulted from the adoption of distributed environment. Load balancing technique, that migrates some of the simulation components from heavily-loaded computing nodes to less-loaded ones, could overcome the problem and lead to higher utilization rate of the hardware resources.

Migrating a process (or federate in our scenario) involves checkpointing the migrating process and restoring the checkpointed process state to the restarting process at the new host. But, HLA federates generally contain tremendous application specific information and are too difficult to checkpoint. Thus, it is necessary to develop a generic framework that makes check-pointing and migration easier and modular.

Despite the effort to checkpointing federate execution state, migration protocol also plays an important role. Poorly designed protocols may incur large overhead and render the benefit of load balancing less effective. Hence, efficient migration protocol is essential in federate migration.

It is also noticed that developing HLA-based simulation requires tremendous effort to comply with the HLA/RTI specifications. Hence, it is also important to implement an automatic code generation tool so that the modeler could concentrate on the simulation, rather than programming-level implementation.

1.3 Objectives

This project aims to provide a load management system for distributed simulation. The load management system is comprised of two major components, namely a framework that facilitates modelers with easy HLA-based simulation development as well as provides support for easy checkpointing and state restoration, and a migration support system with efficient migration protocol implementation. The objectives can be further classified as follows:

- Design a Graphical User Interface for modelers to specify simulation at a higher level. The modeler only needs to provide the logical processes (LPs), the message types between LPs and how messages of each type are handled.

- Design an extensible and customizable generic framework that takes modeler input from the GUI and produces the final executable simulation code for deployment. The framework would encapsulate the HLA/RTI activities and has the flexibility for modelers to extend for their customized needs.

- The framework would support migration by providing the facilities that allow easy state saving and restoration at migration time.

- Design a federate migration protocol that improves the migration performance and reduces overhead.

1.4 Organization of Thesis

This thesis is organized as follows: Chapter 2 briefly reviews some related works published in the area of federate modeling, code generation, load balancing and federate migration. Chapter 3 describes the overall framework architecture and its operating mechanisms. In Chapter 4, load management system is introduced, with the focus on the federate migration protocols based on the framework proposed. Chapter 5 describes the experiment setup and presents results of various migration protocols. Chapter 6 concludes this thesis and proposes potential future work.

Chapter 2

Related Works

Although the HLA provides the paradigm for simulation interoperability and reusability, there are still important issues that impose barrier for researchers adopting the standard to develop and deploy their simulations in HLA/RTI. The two most important reasons are the complexity to write program conforming to the HLA/RTI requirements and the lack of resource management in the runtime. This chapter reviews some of the related works in these field with the focus on code generation. As our main objective is to achieve load balancing, federate migration techniques are also surveyed.

2.1 HLA Tools

One of the HLA's aims is to foster simulation reuse. Hence, many researchers focus on reuse of existing simulation components or development of reusable simulation components. Some also attempted to integrate simulation development with object-oriented technology. Dobbs [11] noticed that although the HLA Object Model Template (OMT) defines documenting format for HLA-related information about object classes, attributes and interactions are derived based on Object-

Oriented Analysis and Design (OOAD) methodologies. No genuine object-oriented paradigm is implemented. An OOAD representation of the federation using Unified Modeling Language (UML) was proposed and Rational Rose is used to develop and maintain both the federation object model (FOM) and federation design.

Various groups have also explored federate reuse in the HLA environment. In [3], the authors argue that to be successful, the HLA needs to create a set of conditions under which optimal reuse is a natural and assured outcome. Reuse in this case means utilizing the same federate in different federations. Since different federations have different Federation Object Models (FOMs), federates need to be built to support different FOMs. To facilitate federation formation, each federate is required to develop a Simulation Object Model (SOM). The SOM provides means for federation developers to determine the suitability of simulation systems to assume specific roles within a federation. Once the federation has been formed, the FOM is developed. An FOM is an identification of the essential classes of objects, object attributes, and object interactions that are supported by an HLA federation. In short, simulation modelers develop SOMs, SOMs support the development of FOMs, and FOMs support the creation and execution of federations. Five case studies were carefully described in [3] to further illustrate how SOMs can be reused and FOMs are formulated from the given SOMs and simulation requirements.

A Component-Based Development (CBD) [17, 42] methodology using fixed code-base was discussed in [32]. Parr et al. noticed that traditionally, HLA development has been seen as an FOM-centric activity, where all inter-federate interactions (object and interaction classes) are described and then manually coded in the federate using a publish and subscribe pattern. This development approach tends to lead to FOM lock-in, poor federate reuse and an unwieldy code-base. They further argued that significant improvements in the reuse and portability of federates and federations can be achieved by applying a CBD methodology to the HLA environment. These improvements are realized through the use of abstraction to insulate from RTI API code, the improvement of translation and transformation services, and the improvement of component aggregation in both RTI and non-RTI based component integration infrastructures. These improvements lead to better simulation granularity and fidelity, and improved simulation performance by enabling non-RTI integration between aggregated federates using the same unchanged code-base. A similar approach was also introduced in SIMULTAAN [6].

Although these works can relieve portion of the development effort from the modelers, the modelers still need to carefully fulfill the required cooperation between each approach's additional specifications and FOM extensions for both the publishing and subscribing federates. This requires the modelers to have not only the knowledge of the system they are simulating, but also the necessary HLA/RTI requirements. Hence, it would be beneficial to the modelers if there is a system which would allow them to focus on the simulation rather than implementation. To accomplish this objective, an automatic simulation code generation tool could be of great help to the modelers. The code generator should allow the modeler to specify the essential information of the simulation, and performs other works at the background. The modeler needs not have the knowledge on how the final executable code is generated.

2.2 Code Generation

The need for automated tools to assist in various phases of the Federation Development and Execution Process (FEDEP) is widely recognized and has been addressed in a number of papers [7, 21, 35, 37]. Extensibility, openness and interoperability are considered crucial to the widespread use of these tools in the federation development and execution process.

Automatic code generation approach was investigated in the Laguna [20] project to migrate legacy flight simulation code into HLA-compliant code using Runtime Communication Infrastructure (RCI), which is a middleware layer code generator that abstracts simulation applications from the underlying interoperability standards (such as HLA or DIS). In the case of HLA, the RCI takes HLA object models as input, and from these the RCI code generator produces necessary RTI communication and administrative work based on the HLA object model.

Having 'ready made' federates to model a portion of the federation is useful for federate development and compliance testing as well as federation testing and integration [15]. These 'ready made' federates are 'RTI ready' without additional modification. To satisfy this purpose, Graham et al [15] proposed an approach for automatic software generation with a bridging federate, or FedProxy. FedProxy allows a user to generate proxy federates from an HLA object model specification in Java programming language. FedProxy provides a complete integrated Java development environment that lets the user quickly customize and adapt the generated federate code to his specific requirements. FedProxy also implements a simulation execution environment that provides all of the simulation services necessary to execute FedProxy as a fully compliant HLA federate.

FedProxy takes a straightforward approach to treat an HLA object model specification as a software object model specification. It simply generates a class for each class and interaction specified in the HLA object model along with any necessary supporting data types. The FedProxy tool allows the user to specify a subset of the classes and interactions she/he is interested in publishing and subscribing so that only the code necessary to support the user's publication and subscription interests is generated. In short, the user is allowed to select from FOM only those classes and interactions that define the SOM for his target federate and then the FedProxy generates the code that implements the federate corresponding to the derived SOM.

Another code generation tool using component-based development model was also presented in [33], which is similar to the CBD approach discussed in the previous section except that automatic code generation is also supported.

2.3 Resource Management for HLA-based Simulations

When the simulation code is ready to execute in the distributed environment, the modeler must make the decision on how to map the simulation components to the available resources. In the scenario of HLA-based simulation, the resources are the computers used to run the simulation, and the simulation components are the federates. Resource management in such circumstances includes discovering the available computers and mapping the simulation federates to these computers.

General resource management implementations, such as Condor [27] and Sun Grid Engine [41], achieves process migration at the kernel level. However, these implementations do not consider the specific requirements of distributed simulations, for example, HLA services, such as object management and time management, that need to be handled during the migration.

Various resource management approaches for distributed simulation have been investigated in the research community. In [28], the resource sharing decision is made by the end user of the computer. If the user has work for the computer, she/he may opt not to participate in the resource sharing system. Otherwise, the user indicates the willingness to share the resource to a *manager*. The manager will 'tell' a *communication federate* that new resource is available. Based on the outcome of the load balancing algorithm, one or more federates are selected by the manager and migrated to the appropriate destination host, and the simulation execution proceeds after migration is successful. When the user decides that the resource should be no longer available to the simulation, the manager performs the load balancing algorithm again to move the running federate(s) to other nodes. Resource availability information is managed by the manager and relayed to the communication federate. All communication between the communication federate and the simulation federates is done using the RTI *interactions*, and the communication federate can be viewed as part of the HLA federation.

An alternative approach was proposed in [8], where a federate is encapsulated in a *job* object. Each job, implemented with multi-threading architecture to increase concurrency, consists of two interfaces: one to the RTI and the other to the Load Management System (LMS). The LMS incorporates two subsystems: *job management* subsystem and *resource management* subsystem. The job management subsystem monitors the execution of federates and performs load balancing activities if necessary. The resource management subsystem, with the help of services provided by the Globus Toolkit, performs resource discovery and monitoring in the computing Grid. The major modules of Globus that are used in LMS include the Grid Security Infrastructure (GSI), the Grid Resource Allocation Manager (GRAM), and Grid Information Service (GIS).

While [8, 28] focus on developing resource management systems for HLA-based simulations, Zając et al. [48] attempted to solve the resource management problem from a Grid perspective. HLA management service, migration support service, broker support service and broker service are implemented as Grid services for the purpose of resource management and federate migration. The migration support service starts HLA-application monitoring and triggers the broker support service to benchmarking and analyzing the site's performance. The broker service will make migration decision based on the analyzed result provided by the broker support service.

2.4 Federate Migration

Load management and load balancing have attracted many researchers in the community [5, 43, 49]. Load management frequently involves dynamic load balancing during runtime, and load balancing is achieved by migrating selected 'victim' process to appropriate destination computer.

Process migration is defined as "the act of transferring a process between two machines" [29] during its execution time. Normally, the information migrated will include data used by the process, a stack, register contents, and the state specific to the underlying operating system, such as parameters related to process, memory, and file management. If the process is multi-threaded, the content of each thread's stack and register must be extracted and restored correctly. Generally, process migration consists of extracting the state information of the process on the source node, creating a new instance of the process at the destination node, transferring the state information there, and updating the connections with other processes on communicating nodes (Figure 2.1).

Many implementations exist for process migration such as Condor [27], Mach [1], Sprite [31], and LSF [50]. These implementations generally migrate whole process at the kernel level or the user application space level in the Unix environment.

It is critical to notice that migration cannot occur at arbitrary point of program

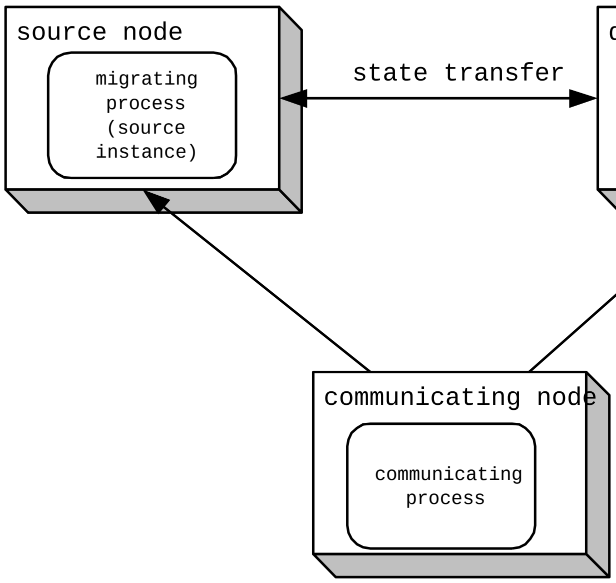


Figure 2.1: **High Level View of Process Migration**

execution. In [9, 40], a pre-compiler is used to examine user program, perform migration point analysis, and insert migration macros with data transfer algorithms. Migration point is carefully selected and annotated so that the migration cost is minimized.

Checkpointing[2] often comes with process migration to provide fault-resilience and fault-recovery capability. Techniques used in checkpointing include periodic checkpointing, aperiodic checkpointing, adaptive checkpointing [2, 25, 38] and probabilistic checkpointing [30]. Most checkpointing implementations were built at the system level and are transparent to the user, with few exceptions that allow users to define what data should be checkpointed [36].

Federate migration can be achieved at various levels. Obviously, general purpose process migration schemes could be modified and be used to migrate HLA federates. In this project, however, we focus only on application-level federate migration.

[2]Checkpointing and state saving are both saving essential execution state information. In this thesis, we use the two terms interchangeably.

The easiest approach to migrate a federate is to utilize the HLA standard interfaces: *federationSave* and *federationRestore*. The drawback is apparent: federation wide synchronization is required. One side-effect is that all non-migrating federates are required to participate in the federation save and restore process for every migration request. As seen in Zając et al [48], the migration overhead increases almost proportionally with the number of federates in the simulation. They further argue that the overhead is mainly due to the time taken to join the federation [47]. Another side-effect of this approach is that no HLA-activity is allowed during the whole federation save/restore process.

Other implementations generally adopt the checkpoint-and-restore approach. In both [8, 28], the migrating federate's essential state is checkpointed and uploaded to an FTP server. The restarting federate will reclaim the state information from the FTP server and perform a restoration. These implementations introduce further latency since communicating with the FTP server is more time consuming.

Hence, minimizing the migration latency would be of great interest. As migration overhead stems mainly from synchronization and communication with a third party (such as FTP), mechanisms avoiding these issues would be desirable. Such algorithms exist in other migration studies. An interesting *freeze free* algorithm for general purpose process migration was proposed by Roush [34]. In this algorithm, the source host receives messages before the communication links are transferred. Any message arriving while the communication links are in transit will be held temporarily and will be sent to the destination when the links are ready at the new host. Message receipt is only delayed while the communication links are in transit. This greatly reduces process freeze time since non-migrating processes are not involved in the migration. The *migration mailbox* is another approach [18] where a predefined address, called a migration mailbox, receives messages for the migrating process. After the process is successfully migrated, it will retrieve the messages from the mailbox and inform other processes to send messages directly to it. The shadow object approach used in avatar migration [19] also achieves the same target. In this approach, each server monitors an interest area and neighboring servers' interest areas overlap in a migration region. An avatar is maintained by a server. When an avatar moves into the migration region, a shadow object is created on the neighboring server. Once the avatar is out of the original server's scope, the shadow object is upgraded to an avatar on the neighboring server and the old avatar at the original server is destroyed.

It is critical to ensure state consistency of the migrating federate during the migration process. Of all aspects, message integrity is the most difficult to ascertain since messages are transmitted between federates dynamically. Above-mentioned migration algorithms solve the message integrity problem under the specific circumstances. In Chapter 4, our approach will be discussed based on the migration protocol proposed.

In this project, we focus on development of the framework and migration of HLA-based simulation federate. In HLA-based simulations, each federate's state information includes its local attributes, published objects, updates and publication/subscription information, the received and outgoing interactions, and the internal state of the federate program itself. Thus, in addition that HLA simulations are difficult to checkpoint, migrating a federate may need a large amount of data to be transmitted. Therefore, one of our objectives is to devise a federate model which will simplify federate checkpointing and minimize the data to be transferred during migration. Our framework will be discussed in Chapter 3.

Chapter 3

SimKernel Framework

Although HLA/RTI provides the facility for simulation interoperability and reuse, the complicated implementation requirements of an RTI-compliant program still impose a barrier for people not specialized in HLA-based programming but still like to design and deploy their PADS simulation in HLA [3].

The communication mechanism of HLA is based on the producer-consumer paradigm where federates publish and subscribe certain type of object/events based on their interests. A typical HLA-based federate contains its simulation code, a *FederateAmbassador* that receives information from the RTI and an *RtiAmbassador* that allows the simulation program to issue service requests to the RTI (see Figure 3.1). When a modeler develops a federate complying to the HLA specification, he/she needs to modify the simulation code to include service requests to the RtiAmbassador and to implement all the required callbacks of the FederateAmbassador for federation management, time management, object management, and declaration management. However, this process adds a significant amount of complexity to user applications. For example, the HLA "HelloWorld" example program contains only 15 lines of simulation code, but the entire program sums up to ap-

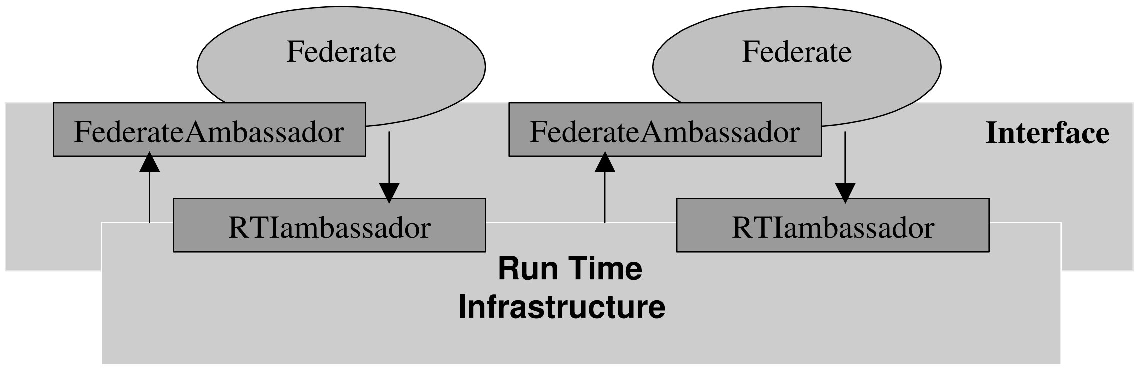


Figure 3.1: **HLA/RTI Simulation Interface**

proximately 2700 lines. So, it takes tremendous effort for the modeler to develop HLA-compliant simulation.

Despite the complexity involved in the development phase, the HLA specification does not provide consideration for efficient migration support. HLA does specify API for federation save and restore for migration and fault tolerance purpose. However, during the whole federation save or restore operation, all federates in the simulation are forced to participate, and no other HLA-related activity is allowed. This limitation effectively halts all simulation federates even if the federate is not involved in the migration operation. Obviously, this approach is ineffective.

In this project, we aim to develop a framework that separates the simulation design from detailed implementation. The framework not only provides the modelers with an easy-to-use interface, but also supports migration by providing easy state saving and restoration APIs. In this chapter, we introduce the *Sim*ulation *Kernel* framework, or *SimKernel* in short. We demonstrate how the SimKernel framework [46] can facilitate efficient federate migration in the following chapters.

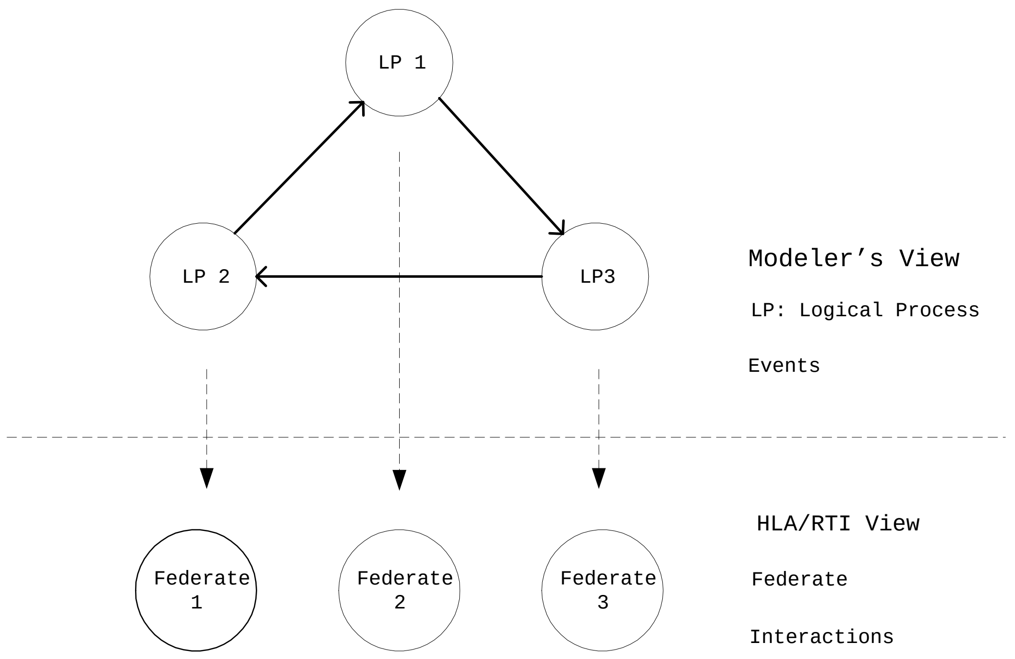


Figure 3.2: **Logical Process to RTI Federate Mapping**

3.1 Introduction

A user-friendly modeling interface is developed to facilitate the adoption of HLA as simulation platform. The interface hides the RTI implementation from the PADS modelers. It is based on a commonly used model for PADS: physical processes are modeled as logical processes (LPs); and interaction between physical processes are modeled using message-passing between LPs. From the modeler's view, she/he only sees a set of LPs with appropriate communication links between them. However, at the implementation level, these LPs will be mapped to a set of federates, and all communications between the LPs will be implemented using RTI interactions.

When a modeler wants to design a simulation running on the RTI environment, she/he can simply draw the LPs and specify the events between them with the help of a GUI. The modeler can then specify the details on how each event is handled at each receiving LP. Note that in this case the modeler does not need to know anything about the HLA/RTI.

Once the configuration is specified, it will be transformed into an intermediate specification file. Based on the configuration file, the target federate code is automatically generated by extending a code library. This process greatly reduces the modeler's work in developing and deploying HLA simulations. Figure 3.2 shows a mapping of PADS simulation configuration from a modeler's view to the HLA/RTI view.

This chapter explains the SimKernel framework in detail. Section 3.2 presents a brief overview of the framework and Section 3.3 describes how the SimKernel functions. Time Management and Data Distribution Management issues are illustrated in Section 3.4 and 3.5 respectively. The framework code library is presented in Section 3.6 and the Code Generator that generates final executable based on the modeler's input is stated in Section 3.7. Finally, Section 3.8 summarizes the characteristics of the SimKernel framework.

3.2 Framework Overview

Figure 3.3 shows the overview of the framework architecture. From a modeler's point of view, a GUI is available for her/him to specify the configuration of a parallel simulation. The modeler can specify the LPs in the simulation and the events that are sent or received by the LPs. The modeler also needs to specify event handling behaviors for each type of events transmitted between two LPs. Additional attributes for LPs can also be defined. The modeler may perform other initialization activities in the LPs. The modeler's design will be transformed to a specification file (i.e., the *LPConf.txt* file), and event handling and initialization

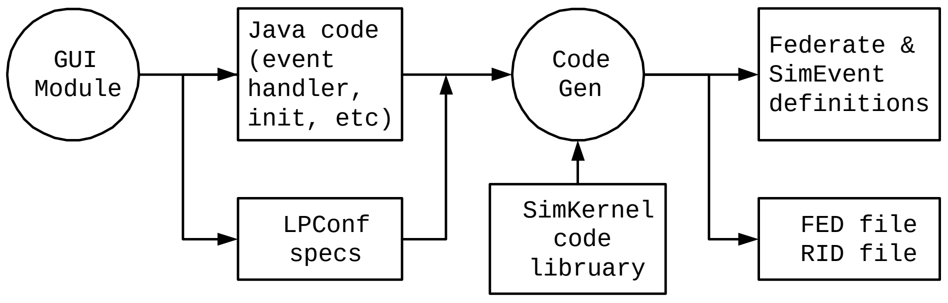


Figure 3.3: **SimKernel Framework Overview**

codes are specified in Java programming language. These are all that are required from the modeler, and the remainder of the work will be done by the framework. Subsequently, a code generator will generate the RTI Initialization Data (RID) file, the Federation Execution Data (FED) file and the executable federate code based on the modeler specifications by extending the SimKernel code library. The SimKernel code library components, developed in Java, are described in Section 3.6.

In HLA, federates communicate with each other by sending and receiving interactions. Each event type defined by the modeler will correspond to an HLA interaction class. In the implementation level, we define the *SimEvent* class. Each event defined between two LPs will correspond to a SimEvent subclass. Hence, each event subclass can be uniquely defined by the event type, the source and destination LPs. Since events of the same type could be processed in totally different ways at different receiving LPs, we define an abstract method *consume()* in the SimEvent class. Each event subclass defines the handling behavior in the *consume()* method. Thus, with the help of Java's polymorphism, events can be processed in a uniform way. SimEvent defines the *name*, *data* (event parameters encoded by the modeler), *timestamp*, and *destination* of the event. Interfaces for parameter encoding and decoding are also defined in the SimEvent class. Additional attributes can be defined in SimEvent subclass, but the modeler must manage the attributes properly.

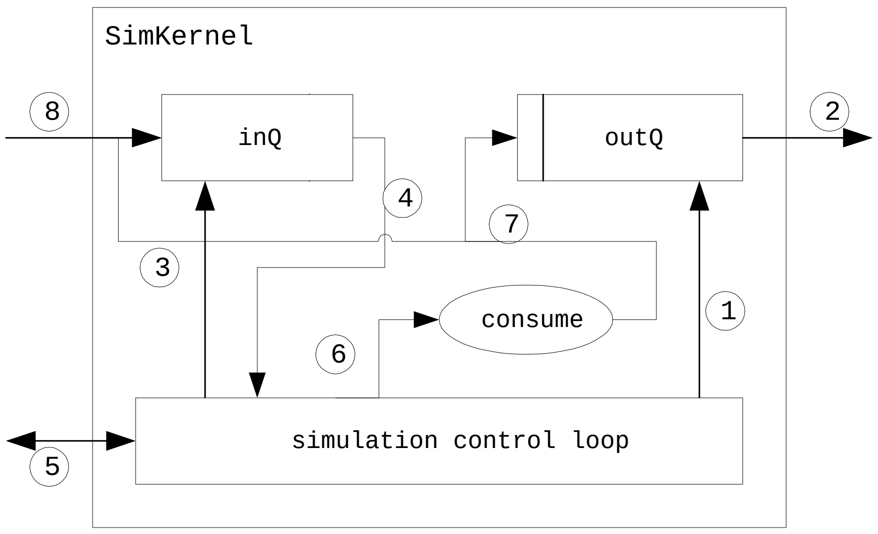


Figure 3.4: **Control Flow in SimKernel**

The SimKernel maintains two queues, namely *inQ* and *outQ*. The *inQ* will maintain the incoming interactions in SimEvent format, and the *outQ* contains interactions to be sent to the RTI. The federate ambassador will dynamically translate incoming interactions to SimEvent objects using Java dynamic class loading technique [26], and put them into the *inQ*. When a SimEvent object is consumed, interactions may be generated by the SimKernel and they will be put into the *outQ*. RTI ambassador will later remove the interaction from the *outQ* and send it to the appropriate destination.

3.3 SimKernel Operation

After the SimKernel is created and joins the federation execution, it will enter a main simulation loop. The main loop iteratively checks the internal queues, processes events and performs RTI related tasks until a termination condition is met (see Figure 3.4).

The SimKernel first checks the *outQ* (operation 1). If there are some pending interactions in the *outQ*, SimKernel will send them out using the HLA Data Distribution Management (DDM) service (operation 2). The federate ambassador that receives an interaction will convert it into a SimEvent object and push it to *inQ* (operation 8).

Next, the *inQ* is checked (operation 3). If there are some events, the topmost event's timestamp (referred to as eventTime in algorithm 1) is compared with the current simulation time (operation 4). If the timestamp of the event is greater than the current simulation time, the SimKernel will request time advancement from the RTI (operation 5). Once time advancement is granted, the events with timestamp less than or equal to the granted time will be processed with the SimEvent's *consume* method (operation 6). Processing an event may generate new events. Events generated for other LPs are converted into RTI interactions and inserted into the *outQ*. Events generated for the same LP will be inserted into the *inQ* (operation 7).

This process is repeated until a termination criterion is met. The modeler can end the simulation by setting the *fedStatus* variable to "`terminating`" directly.

The queue structure used in the implementation also simplifies the checkpointing process. Since the main simulation loop will iteratively process events in the *inQ*, checkpointing is made easy by simply saving the local attributes, the object/interaction publication/subscription information, the *inQ* and *outQ* at the beginning of each iteration. When a migration request is issued, only the objects in the *inQ*, the local attributes and the interaction publication/subscription information need to be transferred.

3.4 SimKernel Time Management

The SimKernel employs the HLA event-based time management scheme. That is, the SimKernel's simulation time is calculated based on the event received from the federation execution. Algorithm 1 presents the time advancement mechanism.

Algorithm 1 Time Advancement Algorithm

```
while fedStatus != terminating do
  sendEventToRTI();  // send the interactions in outQ to RTI

  if inQ.size() == 0 then
    nextEventRequestAvailable(INFINITY);
    while timeAdvanceGrant == false do
      rtiAmb.tick();
    end while
    // CurrentTime set by RTI callback
    if inQ.size() == 0 then
      continue;
    end if
  end if

  eventTime = inQ.getTopTime();
  if eventTime > CurrentTime then
    nextEventRequestAvailable(eventTime);
    while timeAdvanceGrant == false do
      rtiAmb.tick();
    end while
    // CurrentTime set by RTI callback
  end if

  repeat
    inQ.getTopEvent().consume();
    if inQ.size() == 0 then
      break;
    else
      eventTime = inQ.getTopTime();
    end if
  until CurrentTime < eventTime
end while
```

The control flow is described in the previous section. The interactions in the *outQ* are sent out using the *sendEventToRTI()* method. Next, SimKernel will make request to the RTI to advance its simulation time. If the *inQ* has no event, the SimKernel will attempt to advance the current simulation time to INFINITY. The *nextEventRequestAvailable()* method is invoked here to handle potential zero lookahead problem. The RTI will grant appropriate logical time to the federate. When time advancement is granted, the *inQ* is checked again. If there is no new event waiting for processing, this process is repeated. Otherwise, SimKernel will attempt to process the first event in the *inQ*.

Before the event is processed, SimKernel will negotiate with the RTI to advance its simulation time to the event's time. This is done by issuing a *nextEventRequestAvailable()* with the eventTime. Once time advancement is granted, all events in the *inQ* with timestamp less than or equal to the current simulation time are processed. When a new event is generated during the event processing phase, it will be inserted into either the *outQ* or the *inQ*. Events in the *outQ* will be sent to other federates in the federation at the next round of processing.

3.5 DDM Event Routing

In the SimKernel framework, each LP specified by the modeler is given a unique literal name. When an LP sends an event to its destinations, the modeler is allowed to address the destination using the literal name. However, the HLA manages each federate's event interest based on a producer-consumer model. That is, a publishing federate produces interaction instances according to its published interaction classes, and the subscribing federate consumes the instances of the subscribed interaction classes. The RTI tracks all federates' publication and subscription information and sends control signals to the consumer, notifying what is produced based on its subscription interest. Hence an algorithm that transforms the literal name to HLA-compliant implementation is needed, and it should be completely transparent to the modeler.

Suppose the modeler wants the LPs in Figure 3.5 to send and receive the same type of events. The modeler certainly does not want LP3 to receive events sent by LP2 which are in fact destined for LP1. Therefore, there must be some mechanisms to prevent this scenario from happening.

One possible approach is to rename these events to different interaction classes. For such approach, four federates publish and subscribe five distinct types of interactions, although these interactions carry the same type of information. Each federate will subscribe and publish based on its interest. Federates only receive interactions sent from interested publishing federates.

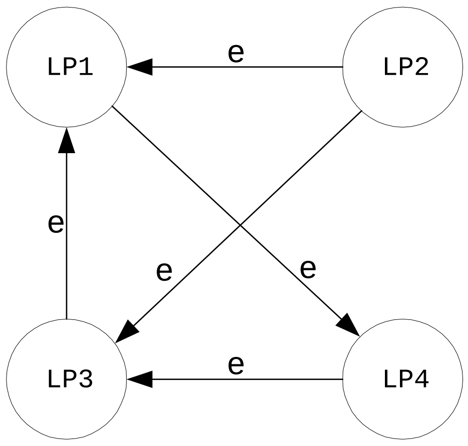


Figure 3.5: **Same Event, Different Destination**

Event renaming is a simple approach to design and implement. However, this approach is not scalable. The difficulty of renaming increases with the complexity of the simulation application. For example, if event e from LP2 to LP1 and LP3 are renamed to e21 and e23 respectively, when LP2 wants to send some information through event e to both LP1 and LP3, it must send each event separately. However, these two messages carry the same information except the destination is different. With DDM, this can be done in one transmission by including both LP1 and LP3's subscription region when the message is transmitted. Event renaming is also inflexible. For example, assume that LP2 will send e to LP1 only on certain condition, and broadcast e to both LP1 and LP3 otherwise. With event renaming, it is not straightforward to implement such scenarios.

3.5.1 DDM Service

Another approach is to employ the HLA Data Distribution Management (DDM) service. HLA specifies the DDM service to promote effective network utilization and reduce the network traffic. DDM provides a flexible and extensive mechanism for isolating publication and subscription interests – effectively extending the sophistication of the RTI's producer/consumer mechanism. The RTI effectively serves as an intelligent switch – matching up data distribution, based on declared interests without knowing details about the data format or content being transported.

DDM further limits the number of messages transmitted in the federation using *Region*[3] with more stringent filtering at both sender and receiver side. This improves the performance and scalability. With DDM, each federate declares a *region* in which it is interested to receive interactions. When another federate sends an interaction with a region overlapping the region a federate subscribes to, the federate will receive such interaction. Otherwise, the interaction will not be received by the federate. A comprehensive discussion on how DDM is used can be found in [10].

[3]A Region is a subset of the default routing space in HLA.

3.5.2 Region Determination

Suppose a collection of LPs send and receive an event class e as shown in Figure 3.5, and the modeler wishes e to be delivered to different destination LPs based on certain criteria. To fulfill this requirement, a generic region determination algorithm is needed.

To allow the modeler to send event to specific destination(s), our framework automatically generates a unique literal name for each LP when it is created. The modeler can change the name, but its uniqueness must be preserved. When an LP needs to send an event to a specific LP or a list of LPs, the modeler only needs to supply a literal string containing all the names of the destination LPs. From the HLA perspective, we use a default region specification with only one dimension. Each federate will subscribe with a unique single point region. When an LP sends an event to a destination, the destination's subscription region will be added to its sending region. Figure 3.6 explains how a region is determined for the example 3.5. Big circle is used to indicate the single point extent for each LP.

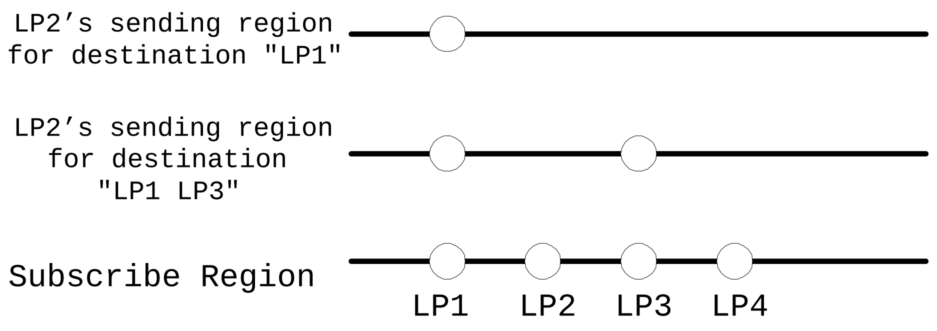

LP2's sending region for destination "LP1"
LP2's sending region for destination "LP1 LP3"
Subscribe Region
LP1
LP2
LP3
LP4

Figure 3.6: **Region Determination for Example 3.5**

Multiple extent region is used when an interaction is sent to multiple destinations. The region will include all the subscription regions of the destination LPs.

For the example in Figure 3.5, when LP2 wishes to send event e to LP1 and LP3, the destination will be specified as "LP1 LP3", and transformed to a region with two extents including the subscription points of both LP1 and LP3. If the event is only sent to LP1, the destination will be specified as "LP1" and transformed to a single extent region including the subscription point of LP1.

All new regions created are maintained by the federate using a lookup table. When a region is to be created for a destination list, the lookup table is first checked to see if an entry already exists. If such entry exists, the region is used directly. Otherwise, a new region is created and stored in the lookup table. This region reuse saves the time for creating a new region each time an interaction is sent.

By giving each federate a unique lateral name, it not only makes the destination list of events more meaningful to the modeler but also makes use of the network more efficiently. It can also further reduce the efforts in federate migration, and this will be discussed in the next chapter.

3.6 SimKernel Code Library

The SimKernel code library is developed in Java language. The modeler has two approaches to utilize the code library. One approach is to use the GUI to complete the simulation design. The GUI will generate the LPConf file according to the modeler's specification. The other approach is manual development. That is, the modeler writes the LPConf specification file and any necessary Java code directly. Then, she/he can manually use the code generator to generate the final executable code. Either way, the final code will be generated by the code generator (see Figure 3.3). Appendix B presents examples on how each approach works. In this section, we will describe the SimKernel code library hierarchy and how the library is used.

The SimKernel library consists of six classes, namely SimKernel, SimEvent, SimEventQueue, Interaction, InteractionQueue and MyFederateAmbassador (see Figure 3.7).

The SimKernel encapsulates most of the RTI-related work, such as federation management, time management, declaration management, federate setup, region creation, and other APIs for use in the modeler's code. It also has an abstract method, *init()*. The final federate code generated by the code generator will extend the SimKernel and implement the abstract *init()* method. In *init()*, the modeler may initialize federate attributes and/or create initial events. As previously mentioned, each SimKernel will have two queue objects: the *inQ* of type SimEventQueue and the *outQ* of type InteractionQueue.

The SimEvent class defines the name, data (event parameters encoded by the modeler), timestamp and destination of the event. The modeler can specify additional attributes when creating a subclass of SimEvent. All parameters of an event will be packed into the data field of SimEvent. However, the modeler must explicitly define how to decode information from SimEvent objects through the *decode()* method. The processing detail must be defined in the *consume()* method. Two APIs are provided for the modeler to send event, one taking a SimEvent object, and the other takes the contents of event. Encoding of individual parameters is done either implicitly by the user in *consume()*, or explicitly in *encode()* method. Note that SimEvent will also have a reference to the SimKernel. The reference is necessary because instances of SimEvent are dynamically constructed, and the event handling is done in the SimEvent's *consume()* method. When the modeler wishes to send new events to the RTI, the event object must have the reference to the SimKernel so that appropriate SimKernel method can be used to deliver the

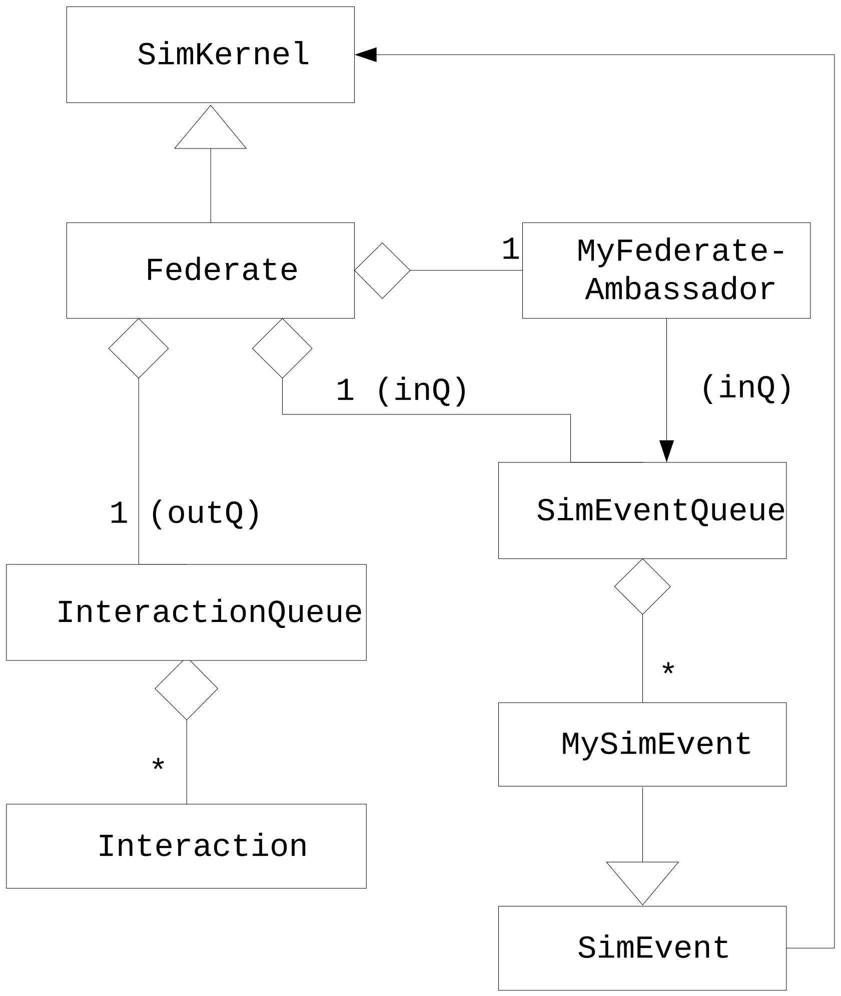


Figure 3.7: **SimKernel Class Hierarchy**

event.

The SimEventQueue class stores event objects converted from RTI interactions in ascending order based on the SimEvent's timestamp. The SimEventQueue allows SimEvent objects to be pushed into and popped from the queue. As both the SimKernel and the FederateAmbassador are accessing the SimEventQueue, synchronized methods are used in the SimEventQueue to ensure mutual exclusion.

The Interaction class wraps an RTI interaction into a Java object. It includes a handle of the interaction, the ParameterHandleValuePair, timestamp, and a tag indicating the source federate and a Region to be sent with. The InteractionQueue class is analogous to the SimEventQueue class in functionality, except that this queue only contains Interaction objects.

The MyFederateAmbassador class handles callbacks from the RTI. It is responsible for coordinating the federate's time advancement. It also uses the received interaction information to dynamically create SimEvent subclass objects and pushes the objects into SimKernel's *inQ*.

3.6.1 Class Definitions

The SimKernel class is the core of the code library. It defines an abstract model of federate. The final user federate will extend the SimKernel class and may define other local attributes. The SimKernel has a vector where the literal names of all federates in the federation is stored, and hashtables are used to store the name/parameter handles of the interaction subscribed/published. Since RTI interactions are delivered with regions when DDM is used, we also incorporate a hashtable to maintain existing Regions. New Region objects will be stored in the hashtable so that no future Region creation is required.

The SimKernel also has string attributes for both the federation name and the federate's literal name. To facilitate federate migration in the future, SimKernel uses *fedStatus* to indicate the current execution status. Valid value for *fedStatus* includes `running`, `restarting`, `collected`, `suspended`, `restoring`, and `terminating`. The states are explained later in Chapter 4.

Other attributes of SimKernel are related to the RTI time management. Both regulating and constrained flags are used to enable time management service.

Table 3.1: **SimKernel Class APIs - User APIs**

protected void init()	This *abstract* method allows modeler to specify user-level initialization information.
protected String getAlias()	This method will return the literal representation of the current federate.
protected String getFedStatus()	This method will return the current simulation status of the federate.
protected void setFedStatus(String aStatus)	This method will set the *fedStatus* to *aStatus*.
protected double getCurrentTime()	This method will return the current simulation time of the federate.
protected double getLookahead()	This method will return the *Lookahead* value of the federate.
public void setLookahead(int theLA)	This method allows the modeler set the federate's *lookahead* value, which has a default value of 1.0 in logical time.
public void setTimeLimit(long limit)	Modelers are allowed to specify a simulation termination condition. This method specifies a termination condition based on the simulation time upper bound. When the federate's simulation time is greater than the given *limit*, the federate will finish simulation and resign from the federation afterwards.

Lookahead, CurrentTime and TimeAdvanceOutstanding define the variables used to manage the correct simulation time.

The end user should not modify the SimKernel attributes directly. The set of operations supplied to programmer are listed in Table 3.1 and other system APIs are provided in Appendix A.

The SimEvent class will be used intensively by the users. The user may obtain the simulation time and lookahead value of the residing federate using the *getLookahead()* and *getCurrentTime()* methods. Events can be sent by SimEvent using either of the two APIs, the *sendEvent(String ename, String param, double ttime, String destList)* method or *sendEvent(SimEvent se, String destList)*. The

Table 3.2: **SimEvent Class APIs**

public double getCurrentTime()	This method will return the current simulation time at the federate where this SimEvent object is being processed.
public double getLookahead()	This method will return the *lookahead* value at the federate where this SimEvent object is being processed.
public void encode()	This abstract method allows the modeler to specify how the event's parameters will be encoded into a single String payload.
public void decode()	This abstract method allows the modeler to specify how to interpret the received single String data and decode it into the event's parameter fields.
public void consume()	This abstract method allows the modeler to specify how the event will be processed in the receiving federate.
public void sendEvent(String ename, String param, double ttime, String destList)	This method forwards the request to the SimKernel instance. SimKernel reference is used to invoke the sendEvent() method defined in the SimKernel.
public void sendEvent(SimEvent se, String destList)	This method will allow the user to send the SimEvent object *se* to the destination *destList*. Similarly, sendEvent() method defined in the SimKernel will be invoked.
public Object getLPAttribute(String attriName)	This method enables the modeler to obtain the value of the named attribute at the LP where this SimEvent object is being processed. The return value is an Object. The modeler must perform necessary type conversion using the Java API.
public synchronized void updateLPAttribute(String attriName, Object value)	This method allows the modeler to update the named attribute value of the LP where this SimEvent is being processed. If attriName does not exist in the LP's attribute table, this call has no effect. Otherwise, the corresponding attribute's value will be updated accordingly.

first API requires the modeler to specify the event name, the encoded event parameter, the time to send and destination list. The second API requires the modeler to construct a SimEvent object before invoking the API. These methods are different from the *sendEventToRTI()* method in SimKernel class. They will forward the invocation to the corresponding *sendEvent()* method defined in the SimKernel, which will transform the user-specified event into Interaction object if destination is foreign federates or SimEvent object if the destination is itself. The reference to the SimKernel is hidden from the users. Description of the SimEvent class APIs is shown in Table 3.2.

The other classes, namely SimEventQueue, MyFederateAmbassador, Interaction, and InteractionQueue, are designed to assist the operation of SimKernel. These classes are hidden from the user, and should not be modified by the user. The complete APIs and descriptions of these classes can be found in Appendix A.

3.6.2 Implementation

A federate under the framework extends the SimKernel. It has a MyFederateAmbassador object, a SimEventQueue object named *inQ* and an InteractionQueue object named *outQ*. Each type of event transmitted between two LPs corresponds to a SimEvent subclass. The MyFederateAmbassador object, instantiated from the predefined library, will create SimEvent subclass objects using a dynamic class loader based on the received interaction handle, source federate and destination federate. The new event created is then pushed into the *inQ*. Figure 3.7 illustrates the class hierarchy of the final federate code. Note that only Federate, MySimEvent and Interaction classes will be created by the code generator, whereas other classes are provided by the library.

3.7 Code Generator

The GUI allows modelers to specify the simulation model at a higher abstraction level and transforms the modeler's input files into final executables. What the code generator requires is a set of input java files of event handling routines, federate initialization routines and a specification file named *LPConf.txt*. The format of the LPConf file is described below. It is also shown how the LPConf works for Figure 3.5. An extensive use case is illustrated in Appendix B.

3.7.1 LPConf Syntax

The GUI design will be represented by an LPConf text file named *LPConf.txt*, and the LPConf file will be the basis of generating the FED file and federate code. Thus, all information entered from the GUI must be contained in the LPConf file. Hence, LPConf will include the following information:

LP Information This section includes information of all LPs entered by the modeler, including LP classes, local attributes and assigned instance literal names of each LP class.

Event Information This section includes not only the type of events and all classes of events, but also the source-destination of each event class, and the handling details.

Based on these requirements, a proposed file format will include the following sections:

- **Federation information** This section will include federation information, such as the federation name.

- **LP section** This section includes the names of all LP classes. The attributes of each class are also specified here, followed by the literal name of each instance. Each literal name is unique in the whole simulation. The literal names will be used to specify the source or destination of an event instance at the next section.

- **Event class section** This section includes all the information of all the event classes. The information of each event class includes two subsections, namely the parameter subsection and mapping subsection. The parameter subsection is self-explanatory; it simply states the parameters that the event type can accommodate. The mapping subsection describes the literal name of source LP and destination LP and where the event handling routine is stored temporarily. We propose the following format:

  - **sourceLP: destLP classHandlerFile** This clause specifies an event class that will be sent to destLP from sourceLP, and at the destLP, this event will be handled in the way as specified in the classHandlerFile.

3.7.2 How LPConf Works

Figure 3.8 illustrates the LPConf file generated for Figure 3.5. The simulation has a federation name called `TestCase`, with four LPs, namely *LP1*, *LP2*, *LP3*, and *LP4*, from the same LP class called `LP`. The `LP` class defines a local attribute `messageOfTheDay` in `String` format. An event class *e* is defined with a parameter named `myMessage`, also in `String` format. The way how message *e* is handled by each LP is defined in a placeholder with the name eventName_SourceLP_DestinationLP.java. In this case, they are e_LP1_LP4.java, etc.

```
Federation TestCase
startLP
        LP (String messageOfTheDay) (LP1 LP2 LP3 LP4)
endLP

startEvent
        event e
                parameter String myMessage
                mapping
                        LP1: LP4 e_LP1_LP4.java
                        LP2: LP1 e_LP2_LP1.java
                        LP2: LP3 e_LP2_LP3.java
                        LP3: LP1 e_LP3_LP1.java
                        LP4: LP3 e_LP4_LP3.java
                endmapping
endEvent
```

Figure 3.8: **LPConf File for Figure 3.5**

3.8 Summary

In a nutshell, SimKernel framework is designed with the following characteristics so that federate development and migration is made easy:

- Simulation design is allowed to be specified at Logical Process (LP) level. This saves the modeler's effort to code the simulation in RTI-compliant way. The automatic code generator will produce the final executables based on modeler's input.

- Each federate is abstracted to a main simulation loop with two event queues, namely *inQ* and *outQ*, holding incoming events and outgoing events respectively. This feature makes checkpointing and state saving modular and simple. Federates only differ from each other by the queue contents and local attributes. State saving is reduced to save the queue contents and local variables. Efforts to analyze state information is greatly reduced.

- All federates adopt the same execution pattern. This results in an application-independent SimKernel. Thus, the SimKernel code library can be pre-uploaded to the destination, saving the data size transferred at migration time.
- Event interest of a particular federate is specified in a configuration file. The configuration file customizes the federate to represent a particular federate.
- Event processing detail is defined in each event's user-defined *consume()* routine. Thus, processing of event is achieved by dynamically loading the event object and invoking the processing routine.
- Each federate is identified by a unique literal name at the LP level. Instead of routing the interactions using regions directly, SimKernel allows modelers to specify the destination in a more intuitive approach. SimKernel will transform the literal name to RTI region and perform the message delivery.

These features facilitate easy federate migration at a higher abstraction level. As the SimKernel is application independent, information transferred at migration time is tremendously reduced. Since the standard library of the SimKernel framework can be pre-uploaded to the destination hosts, only the events in the inQ, the local attributes and the LP's event interest specifications need to be transferred. During federate migration, the migrating federate can be dynamically reconstructed at the destination with the LP specifications and the saved states.

Chapter 4

Load Management

The SimKernel framework only provides the basic features to support easy load management. We aim to deploy the SimKernel-based simulation onto the computing Grid to benefit from the vast computing resources and shorten the simulation execution time. However, load level at different computing nodes may not be evenly distributed, which may cause some federate processing at a lower speed. Consequently, simulation execution time may not be always optimized. Migrating federates from heavily-loaded host to less-loaded ones could overcome this problem and make the less-loaded resources better utilized. In this chapter, the migration support architecture is first presented, followed by a variety of federate migration protocols which take advantage of the SimKernel framework and result in less migration overhead. A simple protocol may achieve migration with minimal interference to the simulation at the cost of potential message loss in heavily congested network. Three improved protocols are therefore introduced to solve the potential message loss problem using a counter mechanism.

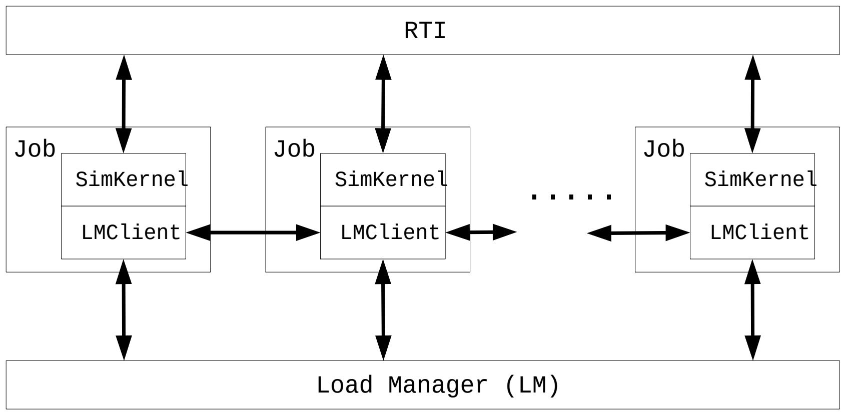


Figure 4.1: **Migration Architecture**

4.1 Migration Architecture

Our simulation execution support system consists of two subsystems, namely simulation subsystem and load management subsystem. Each simulation job is a combination of components of the two subsystems. A SimKernel component performs the simulation activity using the HLA/RTI and the Load Manager (LM), with the LMClients at each individual hosts, performs the load management activity (Figure 4.1). The SimKernel component and the LMClient component are implemented as two separate threads running concurrently and interacting with each other through shared objects.

The LM determines the destination host for the federate to be migrated. The LMClients at the source and destination hosts will communicate until a successful migration is achieved.

The LMClient at each host performs three major tasks. First, it monitors the load level at the host and reports the information to the LM. The information will be used by the LM to determine the federate and hosts involved in migration. Second, on receiving a migration request from the LM, the LMClient will migrate the selected federate using the protocol described in the next subsection. Third,

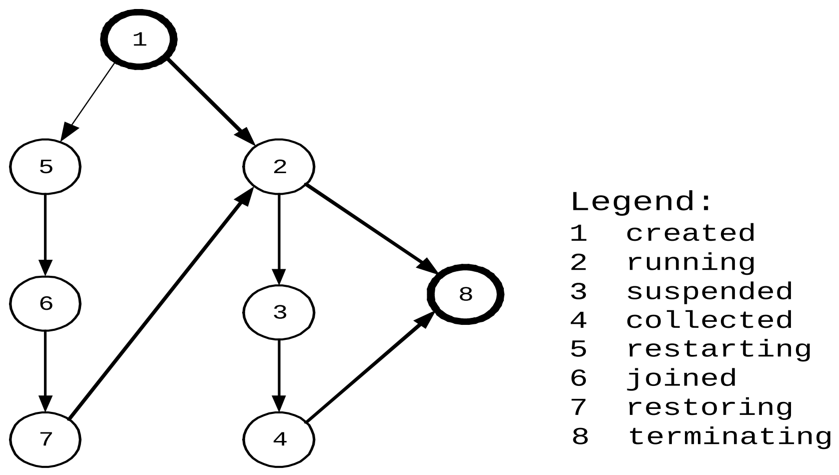


Figure 4.2: **Federate States**

the LMClient will create, suspend, communicate with, and destroy the federate when necessary.

To support migration, the SimKernel main simulation loop is adapted to a state-based process model (Algorithm 2). A set of valid states is illustrated in Figure 4.2. Most state names are self-explanatory. The state is set to "created" when a normal federate is created. The federate sets its state to "running" after it finishes its interaction subscription/publication and simulation execution will proceed. If the federate is created at the migration destination, the newly created federate, hereafter referred to as restarting federate, will have its state set to "restarting". After the restarting federate finishes interaction subscription/publication, its state transits to "joined", which means the federate is now a part of the simulation federation and starts to receive interactions. This information is critical to the simple migration protocol and is discussed in detail in Section 4.2. A "joined" federate will switch to "restoring" state when it receives federate execution state from the migration source and starts restoration. The "restoring" federate will reach the "running" state when the restoration process is completed. A "running" federate will go to "suspended" state when it is selected as the victim in a migration operation and is referred to as the *migrating* federate in the following text. Here, "suspended" means that the federate will not progress to the next simulation loop. So, it will not process any event in its *inQ* with simulation time greater than its current simulation time. But, it can still execute other tasks such as the FederateAmbassador callbacks and LMClient requests. State "collected" means that the execution state information of the migrating federate is already extracted and transferred to the destination. The state "restarting" is not shown in Algorithm 2 because the restarting federate's state is set to "joined" prior to the execution of the main simulation loop.

4.2 Simple Migration Protocol

The SimKernel framework defines a set of application-neutral base classes. Before a federate is migrated to the destination, the base classes and federate codes, together with the federate's specification file are uploaded to the destination.

The simple migration protocol (see Figure 4.3) begins with the LM issuing a migration request to the LMClients at both source and destination hosts. The LMClient at the source host will set the federate state to "suspended". After the event(s) with timestamp less than or equal to the federate's current time, if any, is processed, the migrating federate sends out all outgoing events in its *outQ*. Then, the federate waits for the LMClient to set the *CollectInfo* flag and starts to extract its execution state after the flag is set. The LMClient will set the flag only when its peer at the migration destination sends a "requestInformation" request. The LMClient at the destination host will create a new instance (i.e., restartLP in Figure 4.3) of the federate with state "restarting" upon receiving the migration request from the LM. The *restarting* federate will proceed to join the federation execution

Algorithm 2 Adapted SimKernel Main Simulation Loop

```
while (notEndOfSimulation()){
    switch(fedState) {
      case running:
            processEvent(); // identical to the loop body in Algorithm 1
            if (flagSet("Suspended"))
              setFedState("Suspended");
            break;
      case suspended:
            sendOutgoingEvents();
            waitForFlag("CollectInfo");
            flushQueueRequest();
            saveState();
            setFlag("InfoReady");
            setFedState("collected");
            break;
      case collected:
            waitForFlag("Terminate");
            setFedState("terminating");
            break;
      case joined:
            waitForFlag("Restore");
            setFedState("restoring");
            break;
      case restoring:
            flushQueueRequest();
            restoreState();
            setFlag("Resume");
            setFedState("running");
            break;
      case terminating:
            break;
      default:
            System.out.println("invalid state.");
    }
}
```

and subscribe and publish any event of its interests with the same configuration of the original federate. After the restarting federate successfully completes the event subscription (i.e., "joined"), the LMClient at the host will be notified and subse-

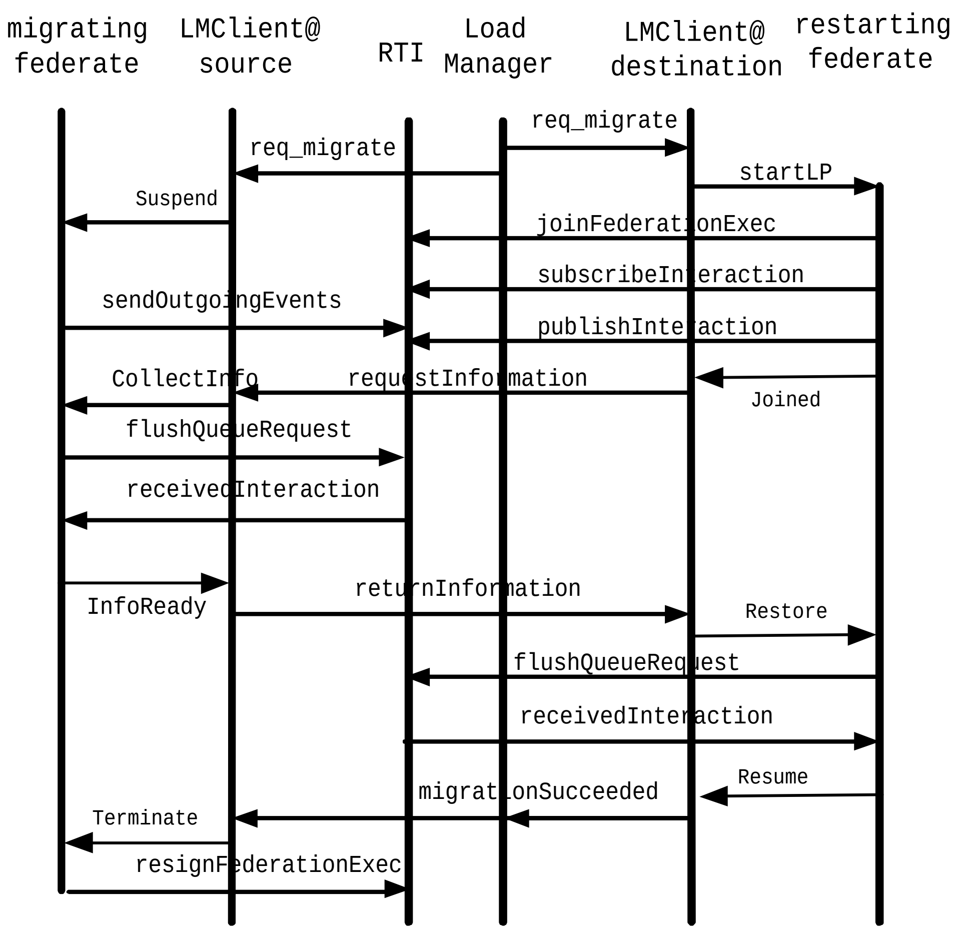


Figure 4.3: **Simple Federate Migration Protocol**

quently send "requestInformation" to its peer at the source host. the restarting federate also starts to receive messages of its interest after it reaches the "joined" state. Note that the new federate is identical to the original one. After the restarting federate subscribes to the events, both federates will receive the same set of messages from other federates.

When the migrating federate is instructed to collect execution state, it first invokes *flushQueueRequest()*, causing the RTI to deliver all messages by calling its *receivedInteraction()* callback regardless of time management constraints. Received events will be stored in the federate's *inQ*.

Upon the completion of flush queue request, the migrating federate encodes its current time, lookahead, events in the *inQ* and local attributes in the attribute table into a formatted string and in the meantime, sets the `infoReady` flag to indicate that execution state data is ready for migration. The federate state is also set to "collected". Event publication/subscription information is not included because it is specified in a separate text file and is transferred with the migration request to the migration host[4]. The LMClient at the source host waits until the federate sets the `InfoReady` flag and gets the execution state. It then starts to transfer the information to its peer at the destination host. After the LMClient at the destination host receives the information, it will notify the LM and its peer at the source that the federate is successfully migrated. The "collected" migrating federate's state will be set to "terminating" by the LMClient after it receives a "migrationSucceeded" message. The migrating federate will exit the main loop and resign from the federation. The information transferred to the destination host is restored by the *restarting* federate.

The LMClient at the destination host will set the federate state to "restoring". Subsequently, the federate begins restoration. A dynamic class loading technique [26] is used to reconstruct event objects from the string specification encoded by the migrating federate. Reconstructed event objects are inserted to *inQ*. The *restarting* federate then invokes *flushQueueRequest()* with its current logical time to obtain all events sent by RTI since it registered its event interests. When the restarting federate restores the received information, duplicates are removed. Since both migrating and restarting federates subscribe to the same set of interaction

[4]We use text file to specify the federate's event interests because the SimKernel framework only support static event publication/subscription at this stage.

classes, there may exist interactions that reach both the migrating federate and the restarting federates.

After the restarting federate has successfully restored its execution state, normal simulation execution resumes and the LMClient at the host will be notified. The LMClient will subsequently inform its peer at the migration source host with a "migrationSucceeded" message.

Note that the LMClient at each host is regularly updating the load level information to the LM. When an LMClient fails to do so, the host is assumed inactive and not eligible for migration. If the selected destination host is down after the migration decision is made, no socket channel to the host can be successfully created and the LM has to select another destination for migration.

4.3 Improved Migration Protocol

Although the simple protocol achieves migration with minimal interference to non-migrat-
ing federates, there is a potential possibility of message loss which will invalidate the state consistency constraint when the network is heavily congested (or in a very high latency network). In this section, we first present the problem, followed by three improved protocols that use a counter mechanism to solve the problem and to ensure message integrity.

4.3.1 Message Loss

Consistency is an important requirement for any migration design. In our design, we must ensure that the restarting federate completely mirrors the migrating

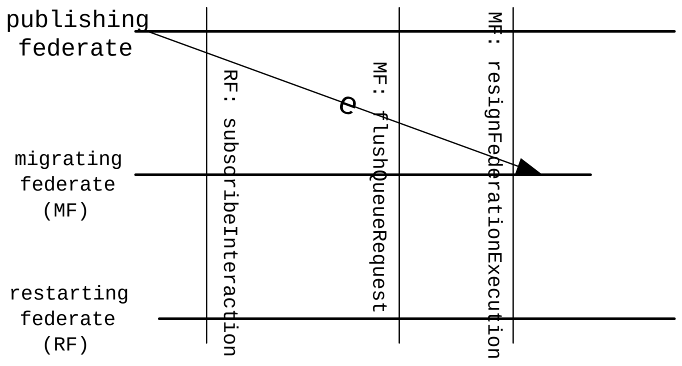


Figure 4.4: **Potential Message Loss Problem**

federate. Further, there is no message loss in transit during federate migration.

In a low latency network, messages are mostly delivered to the destination federate immediately. Hence, we assume that our simple protocol can comply to message integrity constraint most of the time. However, in a high latency network, such as the Wide Area Network (WAN), there is a high tendency that a message may take quite a long time to reach the migrating federate (See Figure 4.4).

Figure 4.4 illustrates how the problem could happen. When the publishing federate sends out an interaction e before the restarting federate subscribes that particular interaction class, the publishing federate will only deliver the message to the migrating federate. However, the interaction e may not arrive at the migrating federate before it resigns from the federation, due to high network latency. Thus, the message e will not be encoded as the migrating federate's execution state. Since the interaction e is not delivered to the restarting federate either, it is lost in transit during the federate migration.

4.3.2 Counter Mechanism

To overcome the problem identified above, an additional component is incorporated into each federate. We propose to attach an index for each interaction sent and for each individual interaction class, maintain counters for interactions sent and received at the publishing and subscribing federate respectively. The counter values will then be used to determine whether or not there is a message loss.

An object class, named *myOutCounter*, is defined in the FOM. It has a set of counter attributes, one for each interaction class defined in the FOM. After a federate joined the federation execution, it first checks whether or not an instance of *myOutCounter* is created in the federation. If not, it will create an instance[5]. Each federate will acquire ownership of attributes corresponding to its published interaction classes. They will publish and update the value of the acquired counter attributes. A counter attribute value is initialized to -1 and is only incremented by 1 when an instance of the corresponding interaction is sent out by the federate. The value -1 is used to identify that the federate has not sent any instance of the interaction class yet. Thus, interaction instances sent are labeled from 0 onwards. The counter value is also attached to the tag field as the index when an interaction is sent. The *myOutCounter* object class and its attributes are automatically generated by the code generator and are put into the FED file.

Each federate also maintains a *myInCounter* variable for each subscribed interaction class. If reliable communication is assumed, *myInCounter* variable can contain only the largest index value of the corresponding interaction instances received. In case of "best effort" communication, the *myInCounter* could maintain the intervals of indices of received interactions.

[5]Note that only one instance of *myOutCounter* needs to be created for each federation.

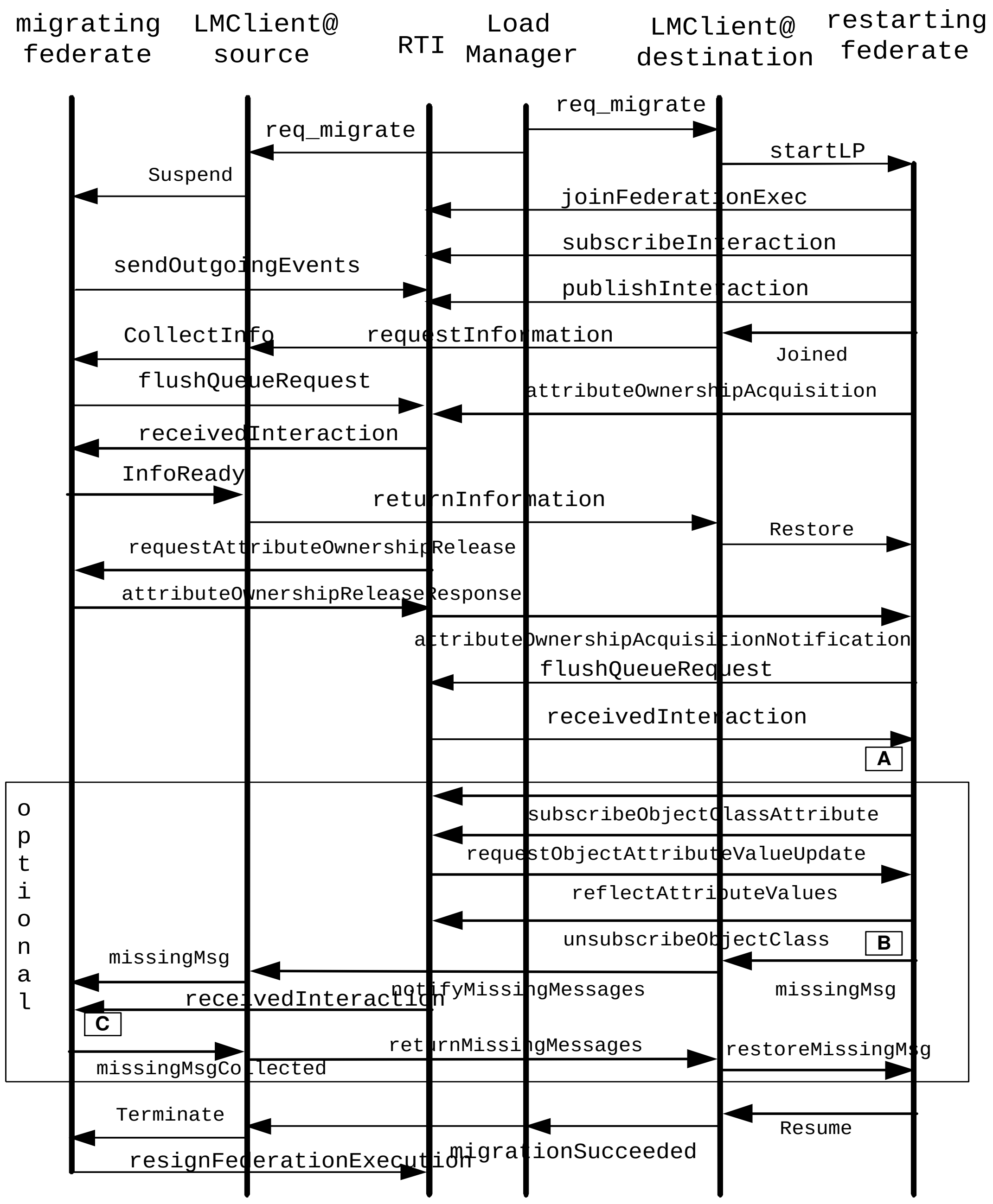


Figure 4.5: **Improved Federate Migration Protocol**

Thus, for each interaction class, a *myOutCounter* object attribute, maintained at the publishing federate, contains the index value of the latest interaction sent, and *myInCounter* value, maintained by a subscribing federate, contains the index value of the latest interaction received. Message in transit for an interaction class i can be detected by comparing the values of the corresponding *myOutCounter* attribute and the *myInCounter* variable as follows. If $myInCounter_i < myOutCounter_i$ is true, there is a message in transit.

For example, if a publishing federate has updated a *myOutCounter* attribute value to 10, and the subscribing federate only have a value 9 in its corresponding *myInCounter* variable, this means that the message with index 10 has not reached the subscribing federate yet, and it is still in transit.

4.3.3 Protocol Description

The counter mechanism is incorporated into the improved protocol to prevent message loss, and Figure 4.5 shows the details. Federates can be classified into three groups according to the role played during a migration operation:

1. Federates that does not participate in the migration.
2. The federate that is to be migrated, i.e., the migrating federate.
3. The restarting federate on the migration destination host.

For non-migrating federates, during the migration process, they behave as if no migration is occurring. That is, those federates continue to send, receive and process interactions. To ensure there is no message loss, the non-migrating federate may be requested to update some of its published *myOutCounter* attributes.

When a federate receives an interaction from the RTI, it needs to identify the class of the interaction and update the corresponding *myInCounter* variable for the specific interaction class.

The protocol is identical to the simple protocol except for the optional part shown in Figure 4.5. When the restarting federate receives the execution state information from the migrating federate, it first performs restoration. Immediately after restoration, the restarting federate sends a flush queue request to the RTI to receive interactions if there is any. Once the callback is received, the restarting federate starts to analyze whether or not the state is consistent (point A in the protocol). If there are interactions received for a particular interaction class, the interaction's index is checked against the restored *myInCounter* value of the specific interaction class. If the *myInCounter* value and the indices of the received interactions make a continuous series, or overlap, no message is lost during the migration process for the interaction class. The above procedure can be performed for all interaction classes subscribed. The migration is successful only if there is no missing message for any of the interaction classes subscribed.

However, the restarting federate may not yet receive any message for a particular interaction class. In this case, *myOutCounter* object needs to be used to verify whether or not there is a message loss for the interaction class. For an interaction class that the restarting federate subscribes to but has not yet received any interaction, it performs the following activities:

1. Subscribe to the attribute of *myOutCounter* object that corresponds to the interaction class with the RTI service *subscribeObjectClassAttributes()*.

2. Request the value of the attribute by RTI service *requestObjectAttributeValueUpdate()*.

3. Receive the value of the attribute by the *reflectAttributeValue()* RTI callback.

The reflected attribute values are compared with the corresponding restored *myInCounter* values. The verification process (point B in the protocol) is the same as that described for point A.

It is noted that this attribute update request/reflect process is optional, depending on the result of the operation at point A. If at point A, the restarting federate has already received interactions for all the interaction classes it subscribes to, this process in unnecessary.

The migrating federate will resign from the federation after a successful migration is achieved. If the restarting federate finds that there exists missing message(s) for an interaction class, it has to wait until all missing messages (point C in Figure 4.5) are received and forwarded to the restarting federate.

4.3.4 Ownership Management for *myOutCounter* Object

As described in the previous subsection, each federate only owns the ownership of *myOutCounter* attributes it publishes. Thus, each federate will acquire the ownership of the corresponding attributes at startup time.

The ownership of the object *myOutCounter* needs to be transferred if the migrating federate owns some of the attributes and/or the privilege to destroy the object. The restarting federate will acquire the corresponding attributes' ownership intrusively. The standard intrusive pull protocol of HLA ownership management is used here [22].

The restarting federate will request to acquire attribute ownership after it subscribes all events from the RTI. Since the restarting federate and the migrating federate are publishing the same set of interaction classes, the attributes acquired will be always from the migrating federate. If, after the restarting federate has acquired the attributes, but before it resumes normal simulation execution other federate requests for the update of the corresponding attribute values, the restarting federate will postpone the update until it resumes normal execution.

4.3.5 Alternative Procedure

There are two alternatives to place the *requestObjectAttributeValuesUpdate()* call: either immediately after restarting federate subscribes/publishes interactions (Figure 4.6) or as that shown in Figure 4.5.

In the first case (i.e., Figure 4.6), the restarting federate needs to request the latest *myOutCounter* attribute values corresponding to all the interaction classes it subscribes to. Since the restarting federate performs this request after it registers its event interests with the RTI, it is guaranteed that index value of future received interactions will be larger than the corresponding attribute values reflected by the *reflectAttributeValue()* callback. Those values are sent to the migrating federate with the *requestInformtion()* call. The migrating federate will send *flushQueueRequest()* to the RTI and wait until all the interactions with index less than or equal to the corresponding *myOutCounter* attribute value received. The migrating federate then saves the state and transmits them to the restarting federate.

This approach needs minimal modification of the simple protocol (Figure 4.3) and requires only one round of communication between the migrating federate and the restarting federate. However, the restarting federate needs to request object attribute value update for all the interaction classes that it subscribes to no matter whether or not it has received interactions.

In the second case (i.e., Figure 4.5), the restarting federate only requests object attribute value update for those interactions of which it has not received any interactions. This may reduce the overhead caused by object attribute value up-

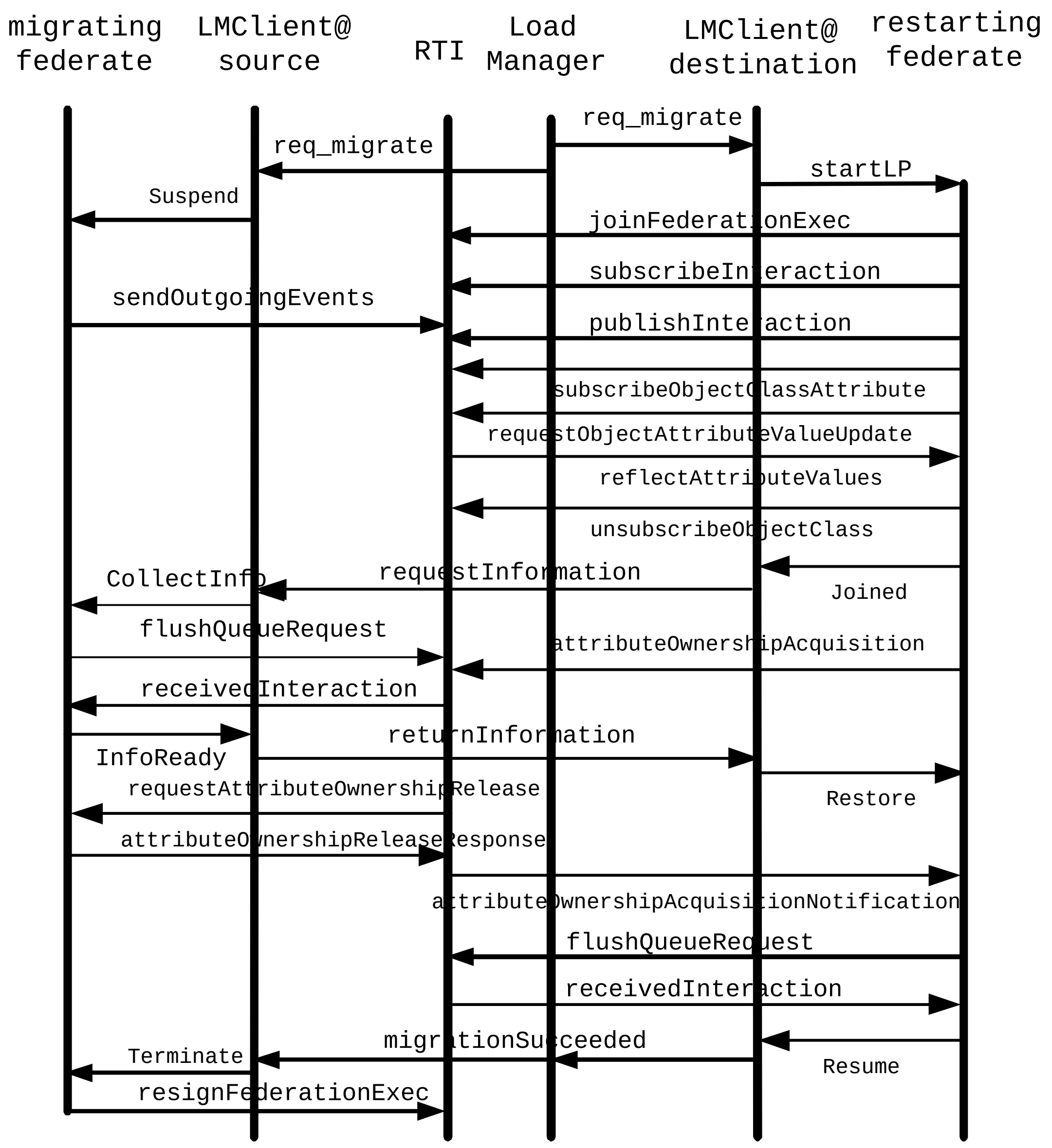


Figure 4.6: **Alternative Protocol**

date. Depending on the scenarios, the optional part in Figure 4.5 (or part of it) may not be executed at all. However, there is still a likelyhood that two rounds of communication may be required between the migrating federate and the restarting federate: one for transmitting saved state information; the other for handling missing messages.

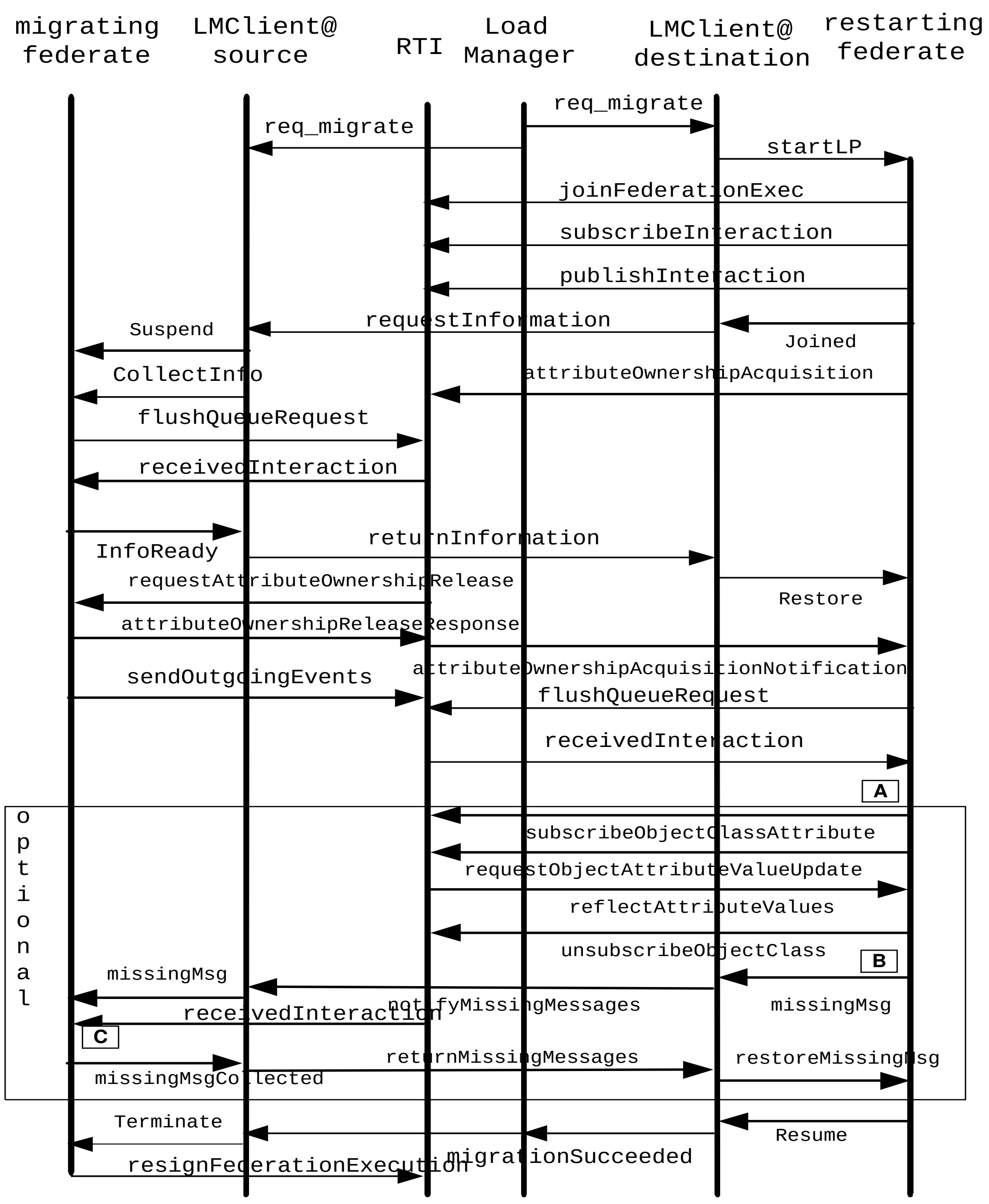


Figure 4.7: **Further Improvement**

4.3.6 Further Improvement

Further study of HLA/RTI activity shows that the process of joining the federation execution may contribute a very large portion to the migration overhead [47] because the restarting federate needs to establish TCP connections to all the existing federates. Hence, protocols that reduce the time for a federate to join the federation would potentially further reduce the migration overhead.

This can be achieved by modifying on our improved protocol (i.e., Figure 4.5). Instead of setting the migrating federate's state to "suspended" after the LMClient at the source host receives migration request, we allow the migrating federate continue simulation execution till the LMClient receives the "requestInformation" call from the migration destination. All other procedures remain the same as the improved protocol (see Figure 4.7). So, the process of restarting federate joining the federation execution will be carried out concurrently with the simulation execution of the migrating federate, and thus can be excluded from the migration overhead.

4.4 Summary

In this chapter, we first present our migration support system based on the SimKernel framework, followed by a simple migration protocol that has a low migration overhead at the expense of potential message loss in high latency network. The message loss problem is discussed in detail and an improved protocol using a counter mechanism to overcome the problem is then presented. An alternative to the improved protocol and a further improvement to reduce migration overhead are also presented. The performance of these protocols will be studied in the next Chapter.

Chapter 5

Experiments and Performance Analysis

This chapter first describes the benchmark and the test environment. Results of all the migration protocols are presented and discussion on each protocol follows. A study of each HLA activity is also presented, which further explains the performance difference between migration protocols.

5.1 Experiment Setup

In the improved protocols, to verify message integrity, the restarting federate is required to communicate with federates that sends interactions to it. Therefore, we used a bidirectional, completely-connected graph with number of nodes varied from 2 to 15 as test benchmark. Each federate sends messages to, and receives messages from, all federates in the simulation, including itself. Figure 5.1 shows the experimental setup for the case with three federates. To avoid the message numbers growing exponentially, each LP will not send out messages to other LPs if

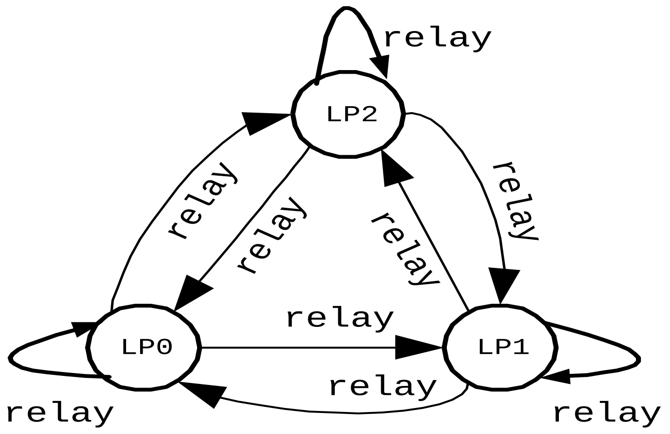


Figure 5.1: **Experimental Setup**

the message being processed is not sent by itself. In other words, an LP will send a message to *all* federates only when the source and destination of the message being processed are identical.

The various migration protocols are tested on a Linux cluster connected by both Ethernet and Myrinet. Figure 5.2 illustrates the architecture of the cluster and lists each node's specification. Load Manager (LM) runs on Surya, which deploys federates to the modes inside the cluster. In the experiments, there is only one federate running on each node.

5.2 Experimental Results and Discussion

The protocols tested was:

- Simple migration protocol described in Figure 4.3 (referred in Figure 5.3 as "Simple")
- Improved migration protocol that performs message integrity checking at

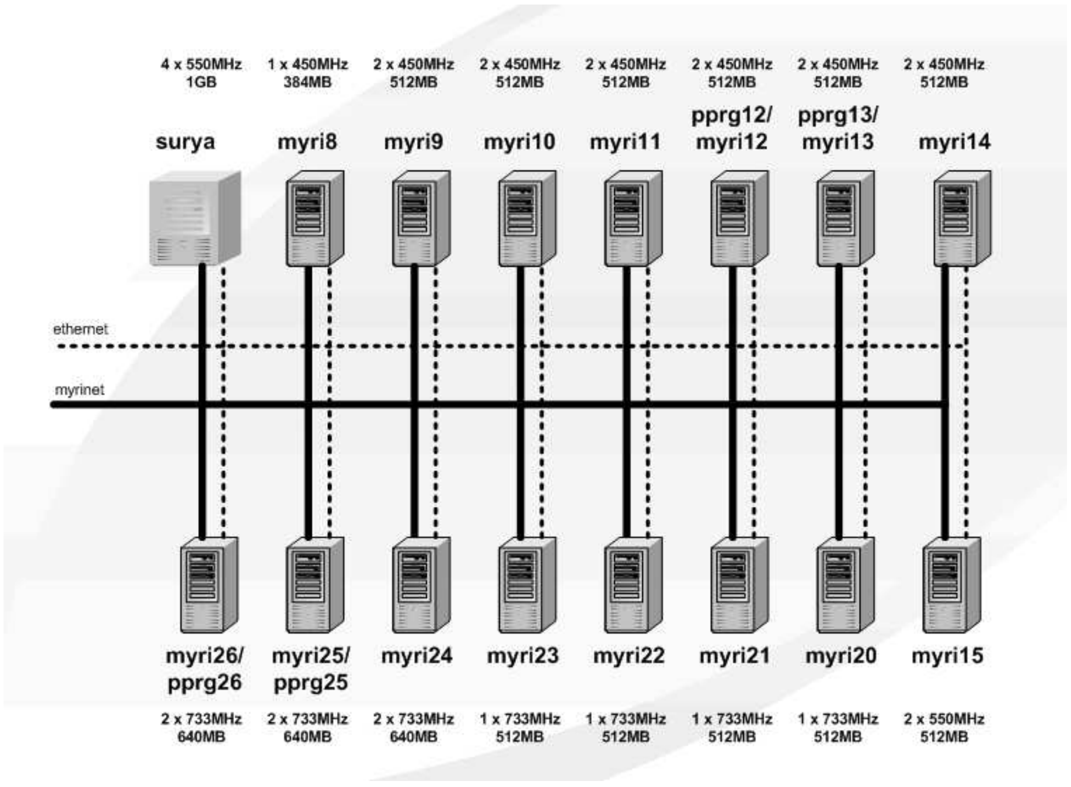


Figure 5.2: **Test-bed Cluster Surya Specification**

the restarting federate as described in Figure 4.5 (referred in Figure 5.3 as "Improved-1")

- Improved migration protocol that performs message integrity checking at the migrating federate as described in Figure 4.6 (referred in Figure 5.3 as "Improved-2")

- Improved migration protocol that reduces federate join time as described in Figure 4.7 (referred in Figure 5.3 as "Improved-3")

The federates are submitted to each host manually and it is ensured that each nost only runs one federate at a time. The LMClients are built to communicate in TCP/IP sockets. A Load Manager randomly selects a federate as the migration

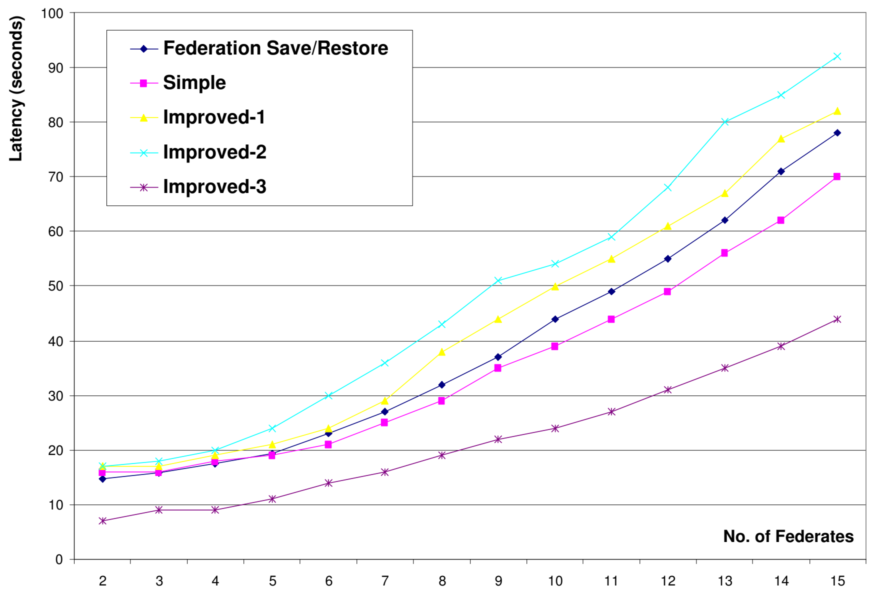


Figure 5.3: **Migration Protocols Performance Comparison**

victim, indicating the federate to migrate to another host which has LMClient but no federate is running. For each protocol, the test case is executed three times and the migration overhead is averaged and plotted in Figure 5.3 in comparison to the federation save/restore approach provided by the standard HLA/RTI interface.

5.2.1 Simple Migration Protocol

Comparing to the federation save/restore approach, the migration overhead incurred using the simple protocol is generally much smaller. Migration overhead spans from the time when the migrating federate is suspended to the time when the restarting federate resumes normal execution. The simple protocol results in reduced migration overhead mainly due to the following factors:

No explicit federation-wide synchronization is required. Since parallel and distributed simulations generally impose constraint on time management, halting a federate could potentially prevent other federate from advancing. Clearly, this is inevitable. However, when a HLA federation save or restore is performed, all federates in the federation are explicitly forced to participate. It effectively synchronizes the whole federation until the operation is achieved. Thus, all federates are directly affected during the migration while most of them are not migrating. Hence, performance is degraded. Federate migration that employs federation-wide synchronization suffers from poor performance since federates not involved in the migration are forced to be synchronized.

No communication with third party is required. While most migration implementations use a third party, such as a FTP server, to store the saved state, additional overhead is introduced since communicating with third party is more time consuming. In the simple protocol, migrating a federate requires only peer to peer communication between the source and the destination hosts. This greatly reduces the migration time.

Only the migrating federate is affected during state restoration. Some implementations that perform event history logging require the restarting federate to request event history from all federates interacting with it and carry out restoration when necessary. When the number of federates grows, the overhead grows proportionately. Using the simple protocol, the only communication required is between the migrating federate and its restarting copy, other federates need not take part in the restoration.

In addition, the simple migration protocol also has the following advantages:

Checkpointing is transparent to the modeler. The SimKernel framework is completely transparent to modelers. Modelers only need to specify the LPs in the simulation at LP level, the events between LPs and the processing details of each event. The design is translated into Java codes by automatic code generator. Checkpointing/state-saving is carried out at the SimKernel level, which does not require modelers to provide the code to explicitly save state at the application level. This also applies to the improved protocols.

Migration is transparent to non-migrating federates. This protocol further benefits the non-migrating federates with complete migration transparency. During the entire migration period, non-migrating federates that interact with the migrating federate continue to send and receive events. These federates have no knowledge whether the federate is processing a message or migrating to another host.

5.2.2 Improved Protocols

Results show that two protocols (Improved-1, and Improved-2) incurs larger overhead than the simple protocol due to additional activities to handle missing messages. It is noted that the migration overhead of the improved protocols increases faster than that of the simple protocol when the number of federates increases. This can be explained by two main factors: the complexity handling missing messages and the significance of the join time compared to other operations, as shown in a breakdown study of each HLA/RTI activity (see next Section).

For both Improved-1 and Improved-2, by the time the restarting federate receives the state information from the migrating federate, the restarting federate may have received messages for some of the interaction classes it subscribes to. In Improved-1, the federate will not request object value update from the RTI for those interaction classes, and this reduces the overhead. However, with Improved-2, the received message information is not used, and a larger overhead is observed.

As our benchmark uses bidirectional fully-connected graph, when Improved-2 is used for migration, it is noted that the migration overhead is higher than that of federation save/restore. This is because the restarting federate has to communicate with all other federates for counter value update.

It is also noted that the result of the further-improved protocol (Improved-3) shows that the overhead incurred is greatly reduced. Furthermore, the overhead incurred increases in a slower pace than other protocols. The reason lies in two aspects. First, in the further improved protocol, the join federation operation of the restarting federate overlaps with the continuous simulation execution of the migrating federate. This effectively excludes the join federation operation being counted in the migration overhead. Second, since the migrating federate continues simulation execution, it is likely that the migrating federate might have received all interactions/messages in transit when requested to collect state information. This will avoid the execution of the optional part of the protocol that handles the missing messages.

5.3 Breakdown Study of HLA Activity

A breakdown study of each HLA/RTI activity is also carried out to investigate the impact of each operation. Results are shown in Table 5.1. Results shows that creating and joining federation (createFE and joinFE) consumes tremendous amount of time compared to other HLA/RTI initialization activities, such as object/interaction publication/subscription (objPubSub and intPubSub). This is due to the fact that when a new federate joins the federation execution, it is required to establish TCP connection to all the rest federates, even though the new federate does not have any event subscription/publication relationship with the federate. This also explains why the Simple protocol's migration latency increases when the number of federates increases.

Table 5.1: **Breakdown Study of HLA/RTI Activity (in Seconds)**

FederateNo	2	3	4	5	6	7	8	9	10	11	12	13	14	15
createFE	2.9	3.1	3.3	3.7	3.8	3.9	4.0	4.3	4.6	5.6	5.7	6.3	6.9	7.5
joinFE	1.9	1.8	3.3	4.6	5.7	6.6	9.4	10.8	12.1	12.3	12.3	13.8	13.9	15.4
objPubSub	0.15	0.14	0.15	0.14	0.16	0.16	0.17	0.18	0.18	0.21	0.23	0.25	0.26	0.27
intPubSub	0.17	0.18	0.16	0.16	0.20	0.21	0.22	0.22	0.23	0.22	0.23	0.25	0.27	0.28

5.4 Summary

In this chapter, we presented the test environment and test results for both the simple migration protocol and the improved alternatives. The simple protocol achieves migration without federation-wide synchronization. No communication with third-party program/server is required and a better performance is achieved.

However, the simple protocol may cause potential message loss problem in high latency network. We solve the problem by using a counter mechanism. The improved protocol achieves consistency-guaranteed migration at a higher overhead. Study shows that creating and joining federation contribute a large amount to the migration overhead. The further improved protocol takes this into consideration, which achieves consistency in migration with reduced overhead.

Chapter 6

Conclusions and Future Work

High Level Architecture provides a software platform with simulation interoperability and reusability. However, developing simulation that conforms to the HLA specification requires tremendous effort. Hence, in this project, a SimKernel framework that allows modelers to focus on the design of the simulation rather than HLA/RTI details was developed.

The SimKernel framework includes:

- A GUI which allows the modelers to specify the simulation from the logical process level;
- A SimKernel code library which implements the HLA requirements and provides the modeler a set of intuitive APIs;
- An automatic code generator which takes the modeler input, links the SimKernel code library, and produces the final executables.

The SimKernel, the heart of the whole framework, was built on a queue-based message processing model: all incoming messages are stored in an incoming queue (*inQ*) and all outgoing messages are stored in an outgoing queue (*outQ*). It iteratively takes messages from the *inQ* and processes according to the modeler specification. Outgoing events generated during message processing are pushed to the *outQ* and sent out afterwards. Local attributes are stored in hashtable. In addition, each SimKernel is given a unique literal name, which can be used as destination of an outgoing message. These features provide modelers great convenience for the simulation design. Furthermore, state saving is also made easy and application-independent. The SimKernel's dynamic execution state contains the messages in its *inQ*/*ouQ*, and its local attributes, and state saving can be done in a transparent and modular manner.

When simulation is executing over a distributed network, there may be scenario that some host becomes heavily-loaded. Subsequently, the federate deployed on the host may progress slowly and cause the slowdown of the entire simulation. Migrating the federate to a less-loaded host could solve this problem. However, traditional HLA-based simulation requires tremendous effort to be migration-enabled, mainly due to the effort required to save the execution state. With SimKernel framework, migration can be achieved easily because the framework incorporates built-in support for state-saving.

Protocol plays a very important role in migration. Most existing migration protocols employ the federation save/restore approach or with the help of a third-party FTP server. These protocols incur unnecessary synchronization or communication, which affects the performance. In this project, we proposed some migration protocols which do not require federation wide synchronization or communication with a third party. A simple protocol was first developed, which achieves better performance than the protocol that uses federation wide synchronization. However, it may result in message loss during migration. A counter mechanism was then proposed to solve the problem. With the counter mechanism, although there is a higher migration overhead, message consistency is guaranteed. Study shows that the action of joining federation contributes most of the overhead, thus, the protocol was further improved to avoid the federation join operation being counted in the migration overhead.

In summary, the SimKernel framework not only provides the modeler an easy-to-use interface for creating parallel and distributed simulation that conforms to the HLA/RTI standard, but also provides features to facilitate easy federate migration. Aspects that affect migration performance were carefully studied and protocols were developed accordingly. Experimental results show that the final improved protocol achieves better performance than the protocol using federation save/restore. It further reduces the migration overhead in comparison to the simple protocol and ins the meantime it guarantees that there is no message loss during migration.

Appendix A

SimKernel Code Library API

This Appendix includes the programmer's APIs of SimEventQueue class, Interaction class, InteractionQueue class, MyFederateAmbassador class and the system APIs of the SimKernel class.

Table A.1: **SimEventQueue Class APIs**

public synchronized SimEvent getTopEvent()	This method will pop the topmost SimEvent object in the queue if any.
public synchronized void insertEvent(SimEvent se)	This method will insert the SimEvent object *se* to the queue according to the timestamp in ascending order.
public synchronized int size()	This method will return the size of the queue.
public synchronized double getTimeAtIndex(int index)	This method will return the timestamp of the SimEvent object at queue index *index*.

Table A.2: **Interaction Class APIs**

public int getHandle()	This method will return the handle of the Interaction object.
public double getTime()	This method will return the timestamp at which the interaction should be delivered.
public Region getDest()	This method will return the Region where the current Interaction will be sent to.
public hla.rti13.java1.SuppliedParameters getParams()	This method will return the SuppliedParameters to be sent to RTI.

Table A.3: **InteractionQueue Class APIs**

public synchronized Interaction pop()	This method will pop the topmost Interaction object in the queue if any.
public synchronized void insert(Interaction ip)	This method will insert the Interaction object *ip* to the queue according to the timestamp in ascending order.
public synchronized int size()	This method will return the size of the queue.
public double getTimeAtIndex(int index)	This method will return the timestamp of the Interaction object at queue index *index*.

Table A.4: **MyFederateAmbassador Class APIs**

public void turnInteractionsOn(int theHandle)	This callback advises the federate of the presence of active subscribers for an interaction class published by the federate. Parameter *theHandle* specifies the interaction class.
public void turnInteractionsOff(int theHandle)	This callback advises the federate of the absence of active subscribers for an interaction class published by the federate. Parameter *theHandle* specifies the interaction class.
public void receiveInteraction(int theInteraction, hla.rti13.java1.ReceivedInteraction theParams, byte[ ] theTime, String theTag, EventRetractionHandle theHandle)	This callback informs the federate that an interaction in the federation relevant to the federate's current subscription interests. This method will convert the ReceivedInteraction into a SimEvent object and insert it to the SimKernel's incoming event queue (i.e., *inQ*).
public void timeAdvanceGrant(byte[ ] theFederateTime)	This callback informs the federate that a previous *nextEventRequestAvailable()* request has been completed. The granted logical time is *theFederateTime*.
public void timeRegulationEnabled(byte[ ] theFederateTime)	This callback advises the federate that time regulation has been enabled with effect from *theFederateTime* onwards, as response to a previous *enableTimeRegulation()* service invocation.
public void timeConstrainedEnabled(byte[ ] theFederateTime)	This callback advises the federate that time constraint has been enabled with effect from *theFederateTime* onwards, as response to a previous *enableTimeConstrained()* service invocation.

Table A.5: **SimKernel Class APIs - System APIs**

private void createAndJoinFederation()	This method will create and join the federation as named in the constructor.
private void resignAndDestroyFederation()	This method will resign the federate from the federation execution. If the federate is the only executing member of the federation, it will also destroy the federation execution.
private void loadFederateList()	This method initializes the *fedList* attribute used for region extent creation.
private void initializeTimeManagement()	This method will initialize the time management scheme for the federation execution. By default, all federate are both **regulating** and **constrained**.
private void parseLPConfSpec(String fileName)	This method will read in the configuration information for the current LP (federate) and do the event (interaction) publication and subscription work.
private void publishInteraction(String iName)	This method will publish an event (interaction) as specified in *iName*.
private void subscribeInteraction(String iName)	This method will subscribe the event (interaction) *iName* with the region of itself.
private Region createRegion(String destList)	This method will create a multi-extent region for destination LP list *destList*.
private void sendEventToRTI()	This function will check the *outQ* and send the interaction stored in the queue to RTI.
protected void sendEvent(String ename, String param, double ttime, String destList)	This method will cause the federate send an event *ename* with parameters encoded into a single payload *param* to the destination LPs specified in *destList* at simulation time *ttime*. If *destLP* is itself, event will be converted to `SimEvent` object and put into *inQ*; otherwise, it will be converted to `Interaction` object and put into *ouQ*. Note that SimEvent class has an API of exactly the same contract as this API. The SimEvent API will invoke this API to send the message through RTI interaction.
private void run()	This is the main simulation loop. It iteratively checks the *outQ*, sends interaction to RTI, advances the simulation time, and processes the received interactions.

Appendix B

SimKernel Framework Use Case

There are two approaches for the modeler to utilize the SimKernel framework: using the GUI to specify the design or writing the necessary files directly. In this Chapter, we build a simple use case and illustrate how to generate the simulation using each approach.

B.1 Use Case

The use case implements a simple one-way super-ping [44] with a conditional multicast by relaying each federates local realtime to the following federate in the simulation.

In the use case, All LPs belong to the same *LP* class, with name of LPa, LPb, LPc and LPd respectively. Each LP has a local String variable named strLocalTime, which contains, in String format, the LP's current realtime returned by the System.currentTimeMill-
is() method. All LPs start at simulation time 0.0 and have a *lookahead* value of 1.0 (simulation time). An event class "relay" is defined to contain a single parameter theTime, which is of the Java String type. LPa will relay its current real time to LPb, and LPb will further relay its current wall clock time to LPc. LPc will also

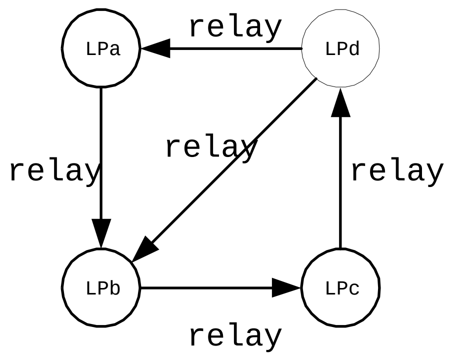


Figure B.1: **Use Case: One-way Super-ping**

relay to LPd. However, when LPd receives LPc's time, it will multicast its current wall clock time to both LPa and LPb if the received time value's least significant digit is 5. When LPb receives the conditional multicast from LPd, it simply sleep for 1 second (wall clock time). Each LP has an initial event which is sent to the corresponding destination.

B.2 Using the GUI

When the GUI is used to specify the simulation design, the modeler must follow the following steps:

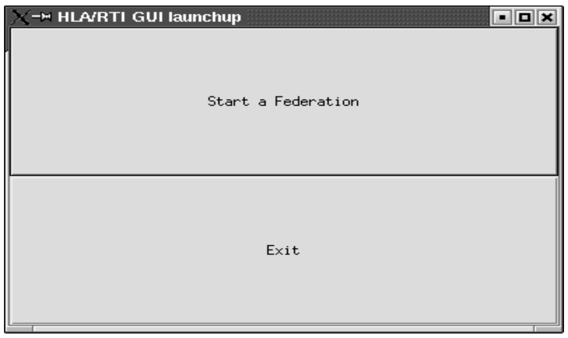


Figure B.2: **GUI Launchup**

Step 1: Start the GUI using the following command:

```
$ java GUI/showgui
```

Then the GUI launch-up window will appear (Figure B.2).

Step 2: Click on "Start a Federation", the design environment is popped up. Select "Design View" (Figure B.3), and the GUI shows a tree view of the LP classes and message classes, a design area, and control pane.

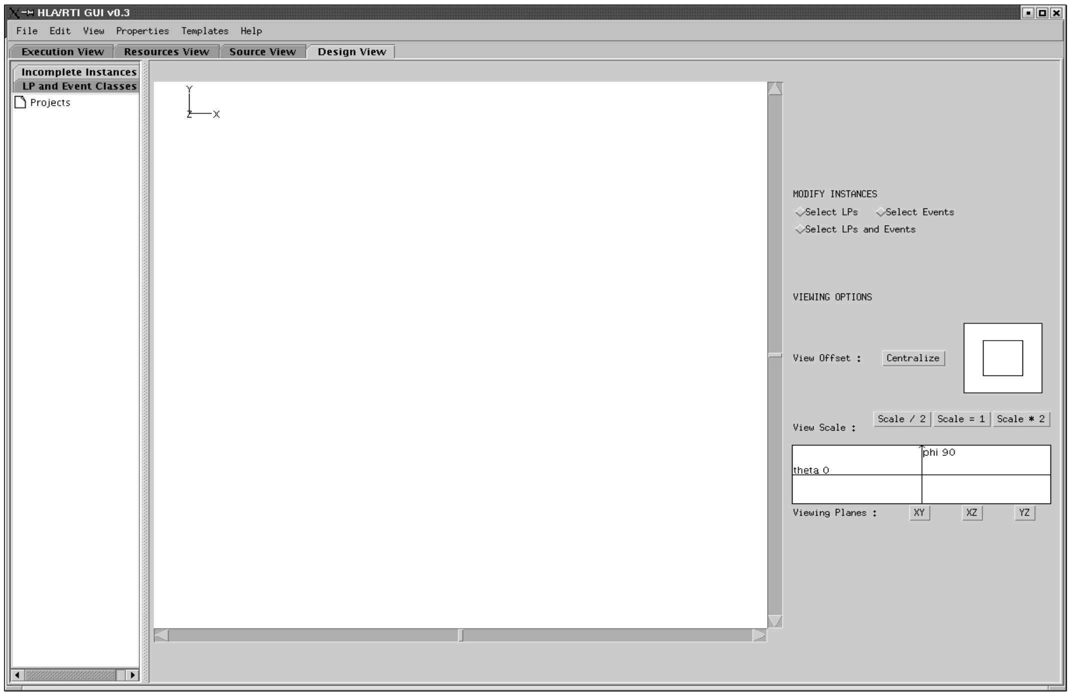
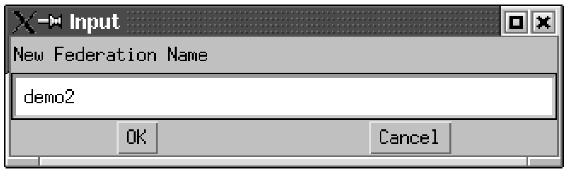


Figure B.3: **GUI Design Environment**

Step 3: Start a simulation design by starting the federation from menu `File`, then click on "Create New Federation". A dialog box will ask the modeler to specify the simulation name. In this case, we name the federation as "demo2" (Figure B.4). Then click `OK`.

Step 4: Before the modeler specify the simulation, LP classes and Event classes must be created. This is done through the Template menu. Click on "Create New LP Class".



New Federation Name

demo2

Figure B.4: **Create New Federation Name**

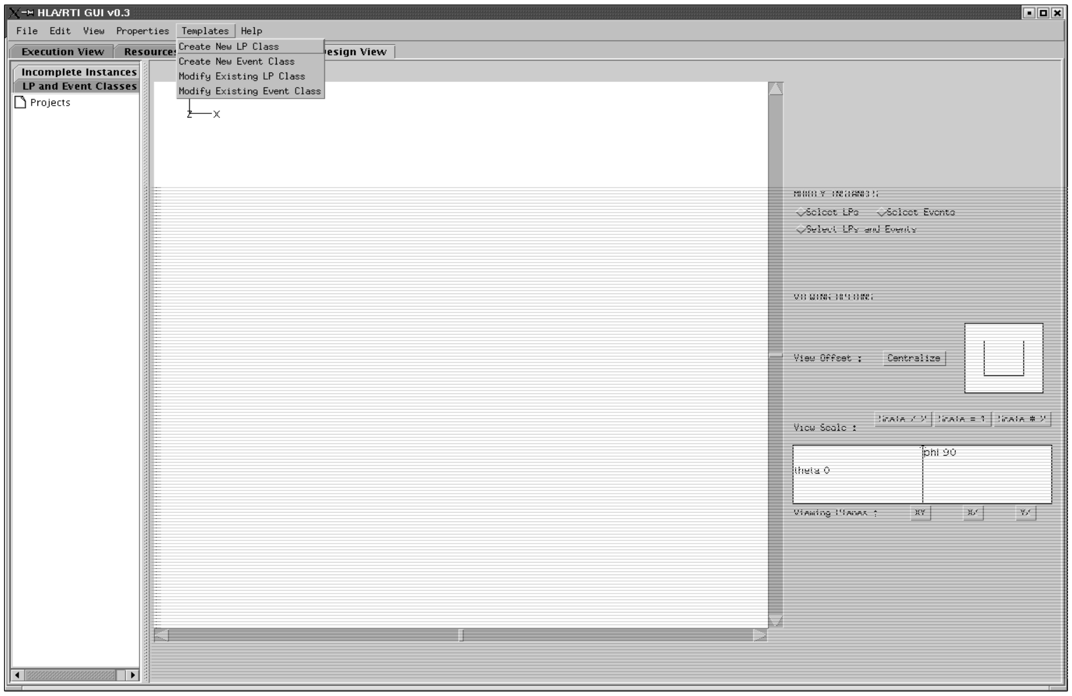
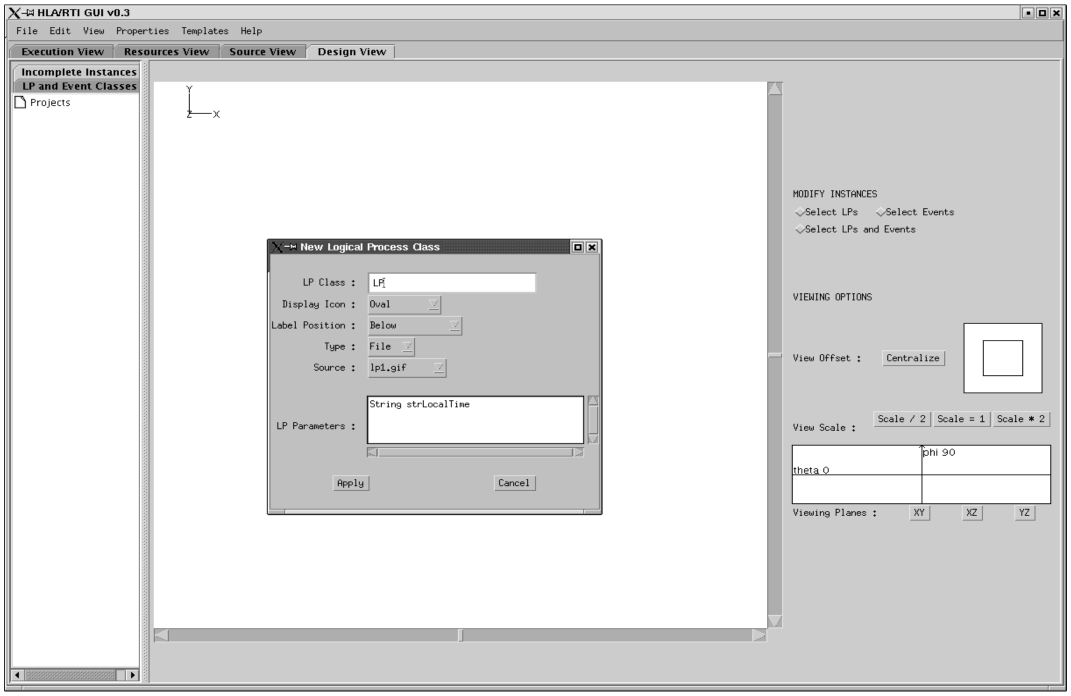


Figure B.5: **Create new LP Class**

Step 5: The "New Logical Process Class" dialog box will allow the modeler to define the LP class and specify the LP class' attributes (Figure B.6). In this case, we only have one LP class, and we name it "LP", and the class has one attribute, `String strLocalTime`. If multiple attributes are defined, they will be delimited by coma, like ***`Type1 var1, Type2 var2, Type3 var3`***. The LP class' display properties can also be changed.

Step 6: After the LP class is defined, click on `File` -> `Save`, followed by `View` -> `Refresh-all`, the newly defined LP class "LP" will appear in the project tree view under the `LPs` branch (Figure B.7).



New Logical Process Class

LP Class : LP
Display Icon : Oval
Label Position : Below
Type : File
Source : lp1.gif
LP Parameters : String strLocalTime

Figure B.6: **Define LP Class' Properties**

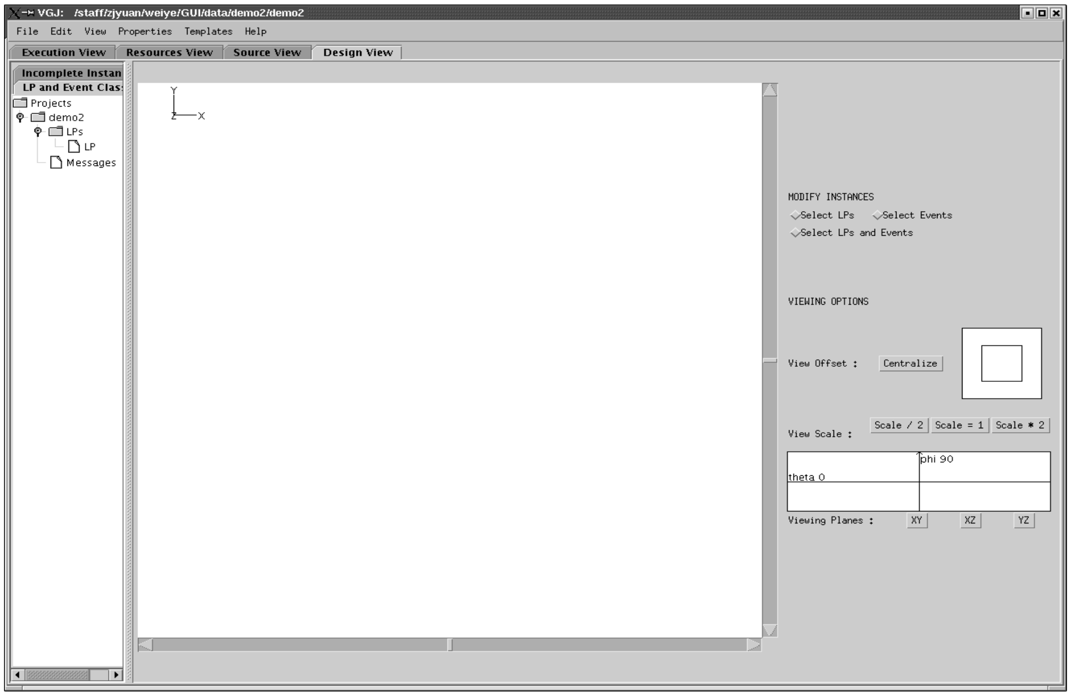
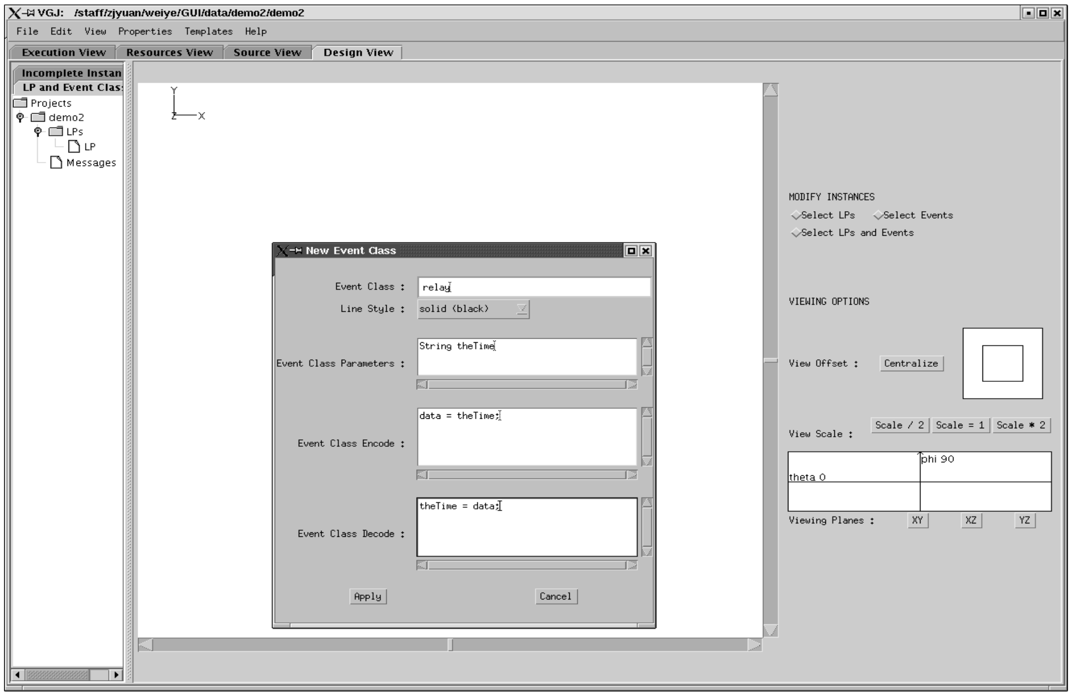


Figure B.7: **Tree View Refresh for LP Class**



Event Class : relay
Line Style : solid (black)
Event Class Parameters : String theTime
Event Class Encode : data = theTime;
Event Class Decode : theTime = data;

Figure B.8: **Define New Event Class**

Step 7: Event Class is defined in a similar way to the LP class. For the use case, we define the event name as "relay", and parameter as "String theTime" (Figure B.8). Notice that the SimKernel framework defines the SimEvent class with a single trunk, named as "data" of String type, of parameter(s) for simplicity purpose, the modeler must specify how each individual parameter is encoded to and decoded from the "data" field. In this case, the "relay" event class only has one parameter, which is "theTime" of String type. After the event class' details are defined, click `OK`.

Step 8: Event class can be reflected in the project tree view by the same procedure as described in Step 6 (Figure B.9).

Step 9: After LP classes and Event classes are defined, the modeler can specify the design by adding LP instances and event definitions between LP instances. LP instance is added by first selecting the LP class in the tree view, this makes

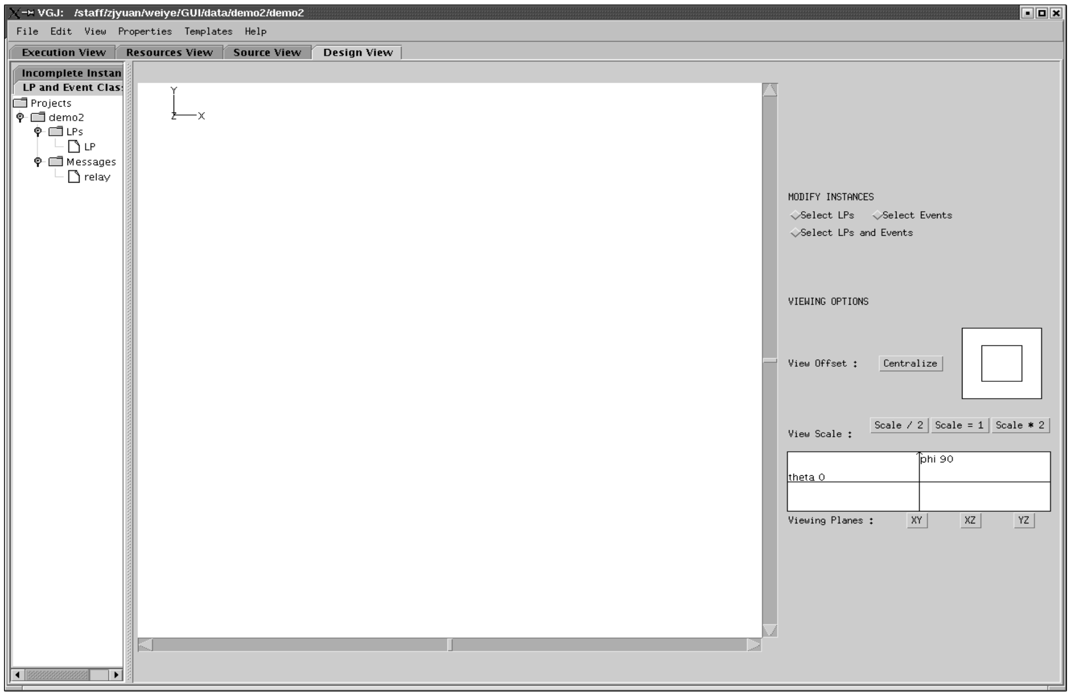


Figure B.9: **Reflect the Event Class on Project Tree View**

subsequent operations in the design *pane*[6] recognized as "create LP instance" operation. Click on the design pane, new LP instance will be added to the design. All new instances are named "default" (Figure B.10) and the modeler must fill in the required details.

Step 10: Each LP instance's properties can be changed. The modeler must click the "select LP" option from the control pane on the right. Double clicking on an LP instance, the instance's properties will be displayed (Figure B.11). The modeler can change the LP instance's name and initialization routine (Figure B.12). After modification is performed, click on `Apply`.

Step 11: More LP instances can be added to the simulation design by repeating Step 9 and Step 10 (Figure B.13).

[6]In Swing, the Java graphics library, a pane is piece of a frame and is used to track the various graphics components in the frame.

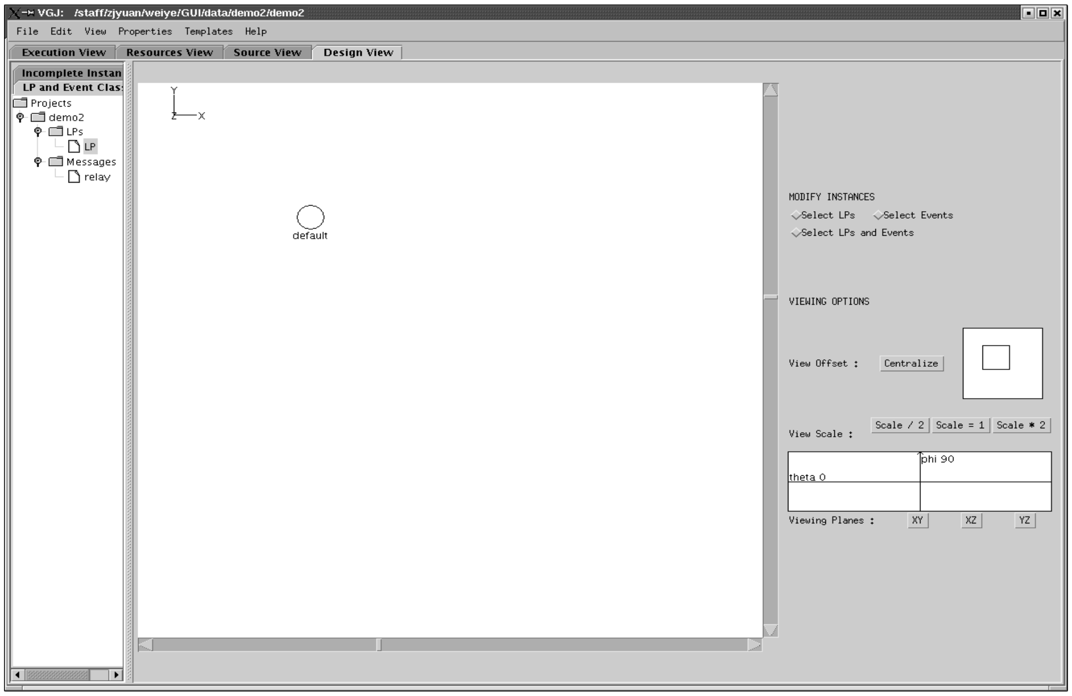
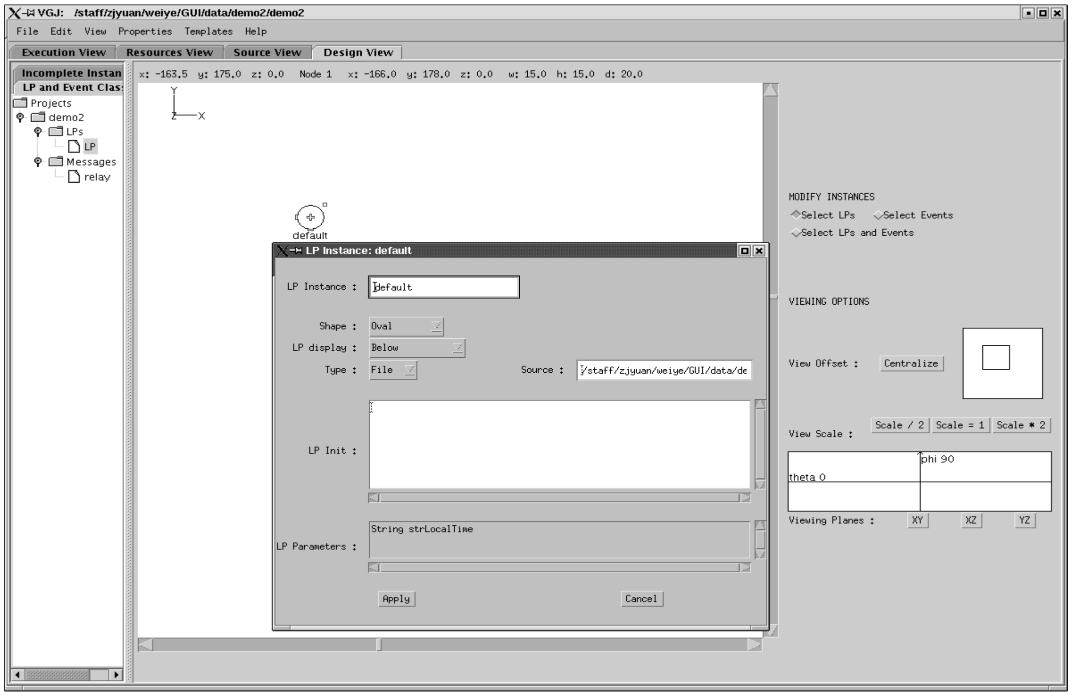


Figure B.10: **Add New LP Instance to Simulation Design**



x: -163.5 y: 175.0 z: 0.0 Node 1 x: -166.0 y: 178.0 z: 0.0 w: 15.0 h: 15.0 d: 20.0

LP Instance : default

Shape : Oval

LP display : Below

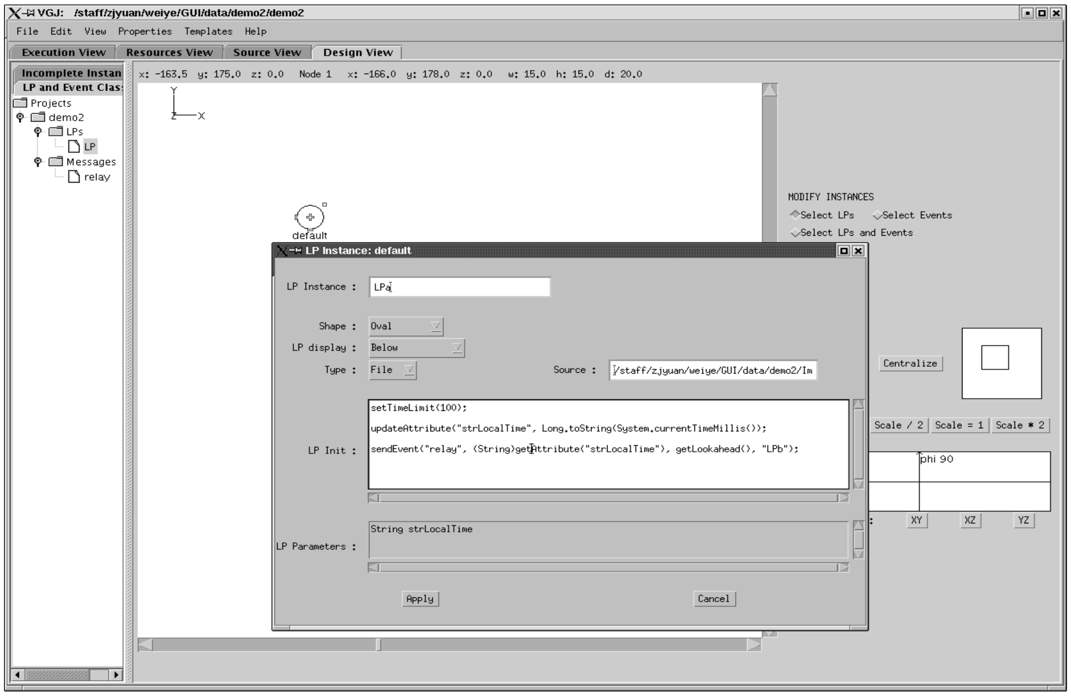
Type : File

Source : /staff/zjyuan/weiye/GUI/data/de

LP Init :

LP Parameters : String strLocalTime

Figure B.11: **Default LP Instance Properties**



```
x: -163.5  y: 175.0  z: 0.0   Node 1   x: -166.0  y: 178.0  z: 0.0   w: 15.0  h: 15.0  d: 20.0
```

LP Instance : LPa

Shape : Oval

LP display : Below

Type : File

Source : /staff/zjyuan/weiye/GUI/data/demo2/Im

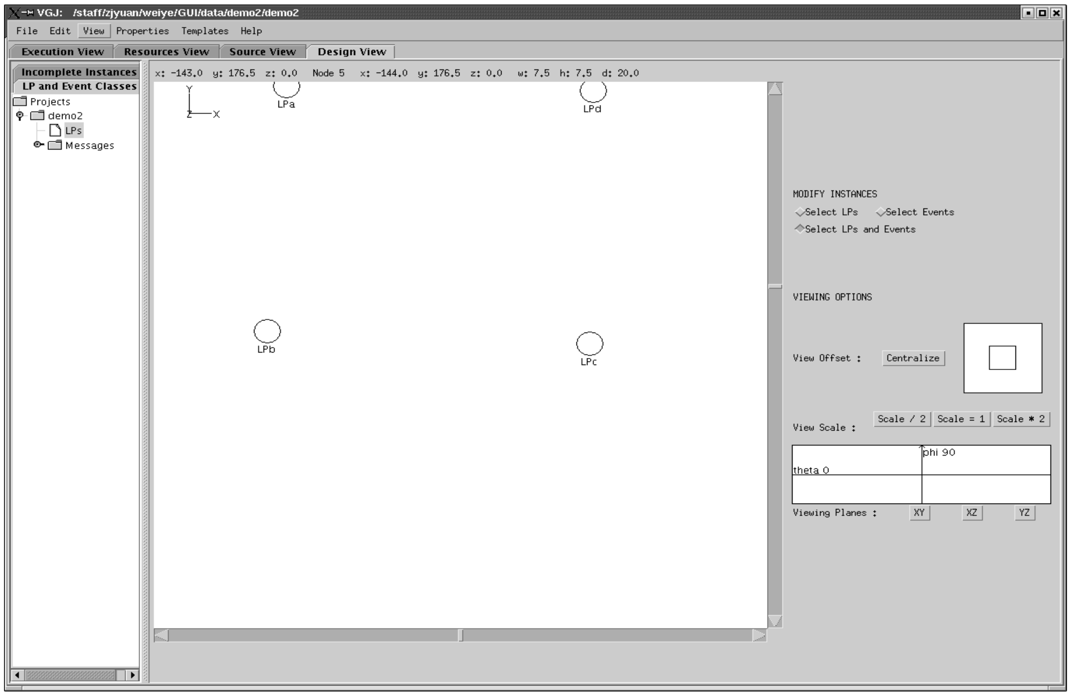
LP Init :

```
setTimeLimit(100);
updateAttribute("strLocalTime", Long.toString(System.currentTimeMillis());
sendEvent("relay", (String)getAttribute("strLocalTime"), getLookahead(), "LPb");
```

LP Parameters :

```
String strLocalTime
```

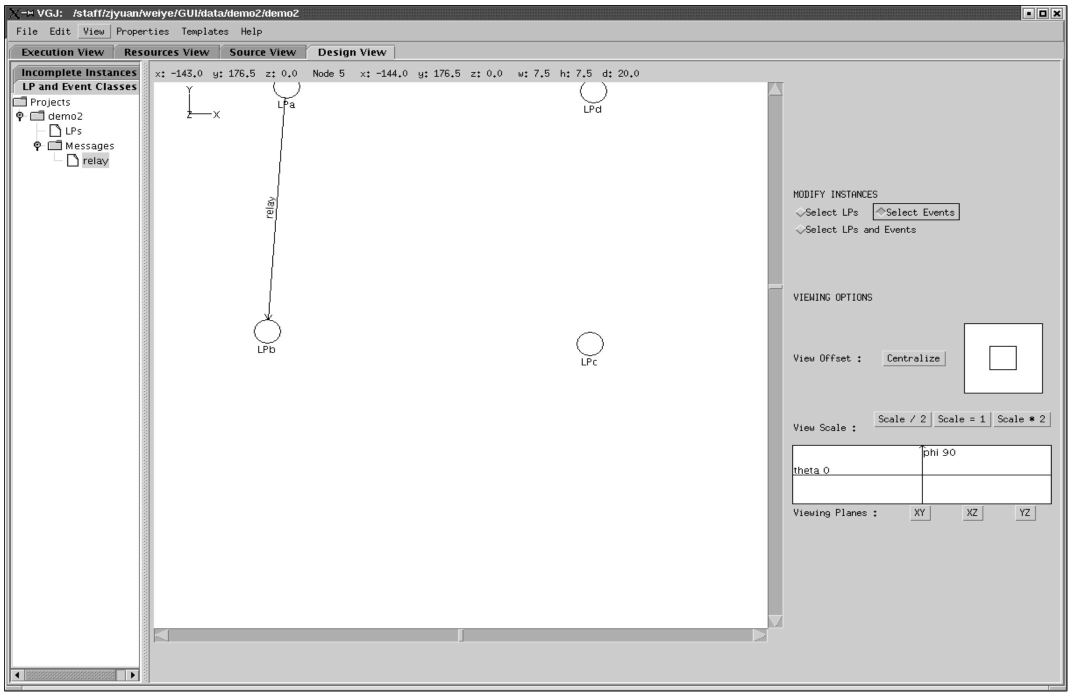
Figure B.12: **Modify LP Instance's Properties**



```
x: -143.0  y: 176.5  z: 0.0   Node 5   x: -144.0  y: 176.5  z: 0.0   w: 7.5  h: 7.5  d: 20.0
```

Figure B.13: **Create More LP Instances**

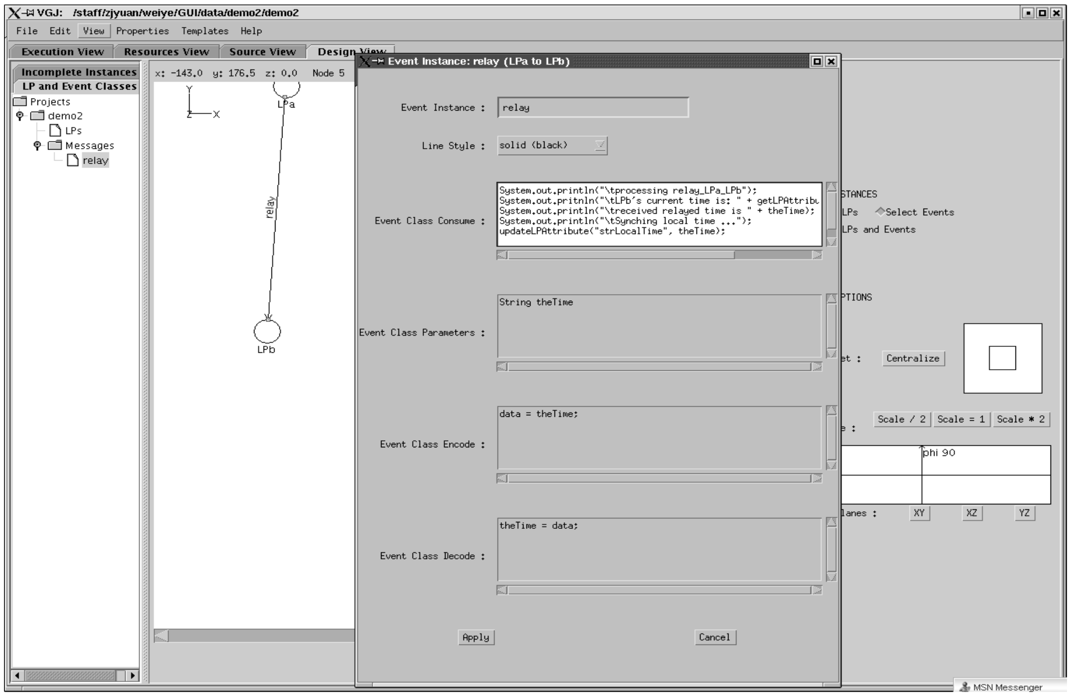
Step 12: Event is added to the design similar to adding LP instances. After the appropriate Event class from the tree view is selected, all subsequent operations in the design field will be recognized as "create event" operations. Each event is defined by a directed link from one source LP to one destination LP. When both source and destination LPs are selected, the link will show with an arrow and the event class name is shown beside the link. In the example, an event from LPa to LPb is specified, and the event is of type "relay" (Figure B.14).



```
x: -143.0  y: 176.5  z: 0.0   Node 5   x: -144.0  y: 176.5  z: 0.0   w: 7.5  h: 7.5  d: 20.0
```

Figure B.14: **Create Event Between LPs**

Step 13: Event defined in Step 12 has empty processing detail and the modeler must explicitly specify how the receiving LP will process the event. Click "Select Events" from the control pane on the right, then double click on the event, a dialog box showing the event's properties will pop up. The event class' parameters, encoding and decoding functions are shown in grey text. The editable textbox allow the modeler specify how exactly the receiving LP will handle this event in the *consume()* method (Figure B.15). In our example, upon receiving a "relay" event from LPa, LPb prints its local realtime on screen, followed by showing the realtime relayed from LPa. It then updates its local realtime through the *updateLPAttribute()* API, obtains the residing machine's local time and relays the time to LPc by invoking the *sendEvent()* API. Available APIs are shown in Table 3.2.



Event Instance : relay

Line Style : solid (black)

Event Class Consume :

```
System.out.println("\tprocessing relay_LPa_LPb");
System.out.pritnln("\tLPb's current time is: " + getLPAttribu
System.out.println("\treceived relayed time is " + theTime);
System.out.println("\tSynching local time ...");
updateLPAttribute("strLocalTime", theTime);
```

Event Class Parameters :

```
String theTime
```

Event Class Encode :

```
data = theTime;
```

Event Class Decode :

```
theTime = data;
```

Figure B.15: **Define Event Processing Detail**

Step 14: More Events can be defined by repeating Step 12 and Step 13 until all events are specified (Figure B.16). The LPs can also be moved to a convenient place in the design pane using the control pane on the right. Click inside the white rectangle and hold the mouse button down, move the mouse and the design will also move. Until a suitable position is reached, release the mouse button.

Step 15: When the simulation design is completed, the modeler can start code generation by select "Execution View", followed by click on "Generate Federation File". A dialog box will appear for the modeler to confirm file generation (Figure

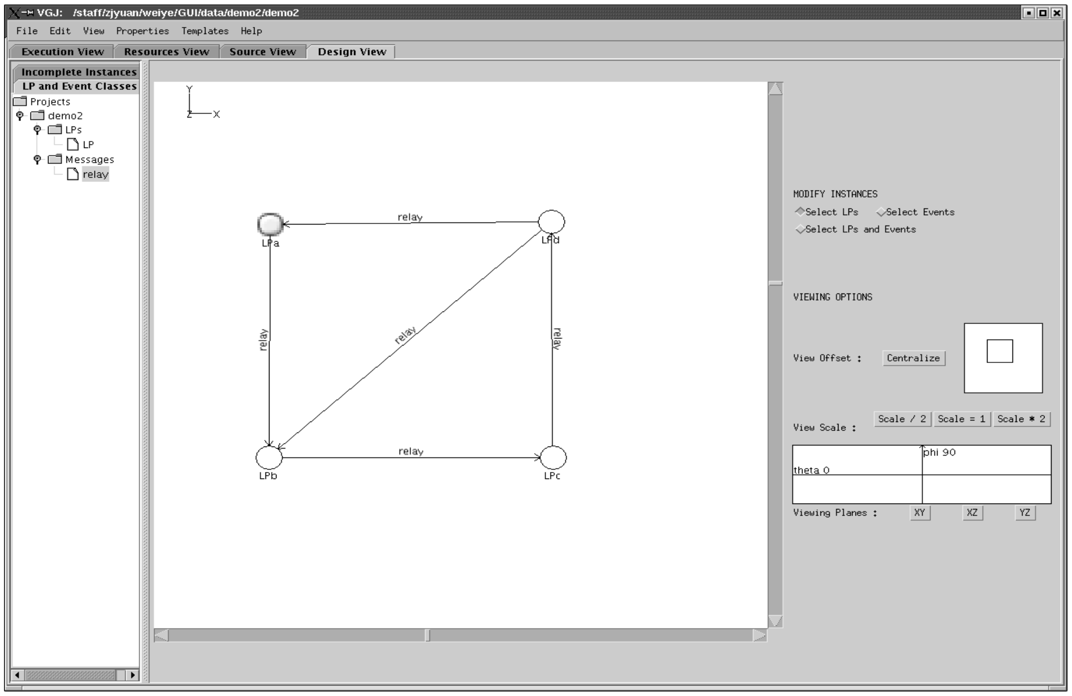
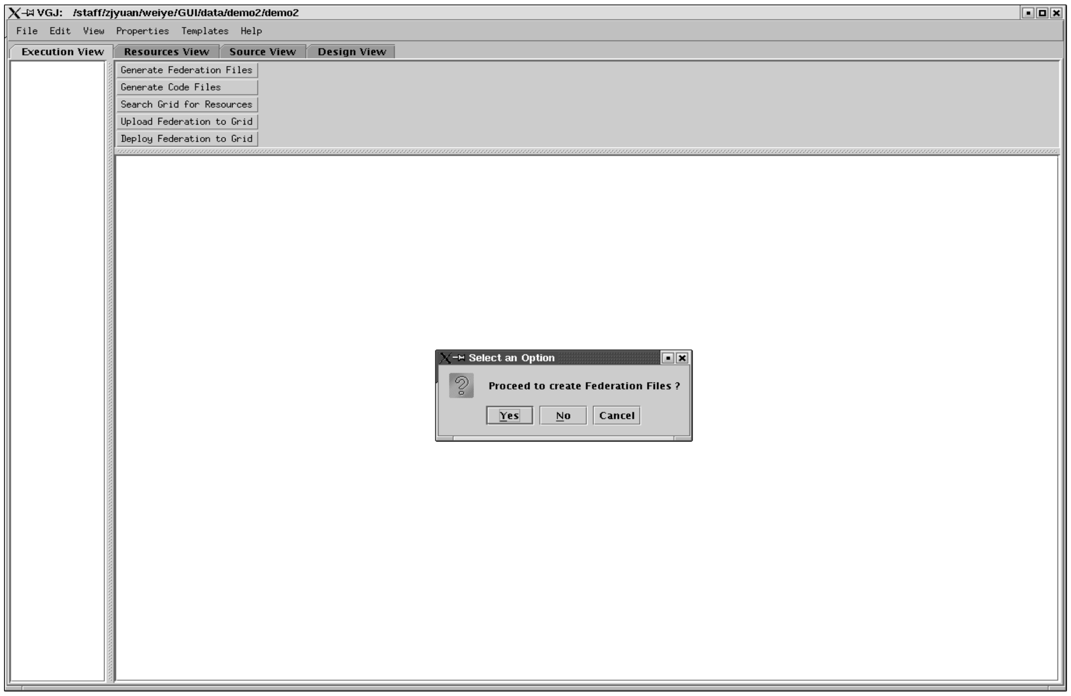


Figure B.16: **Complete Simulation Design**

B.17). Click `Yes`, and the GUI will write the modeler's design in the GUI to an intermediate file recognized by the code generator. When files are generated, a message indicating the operation is successful will show up (Figure B.18).

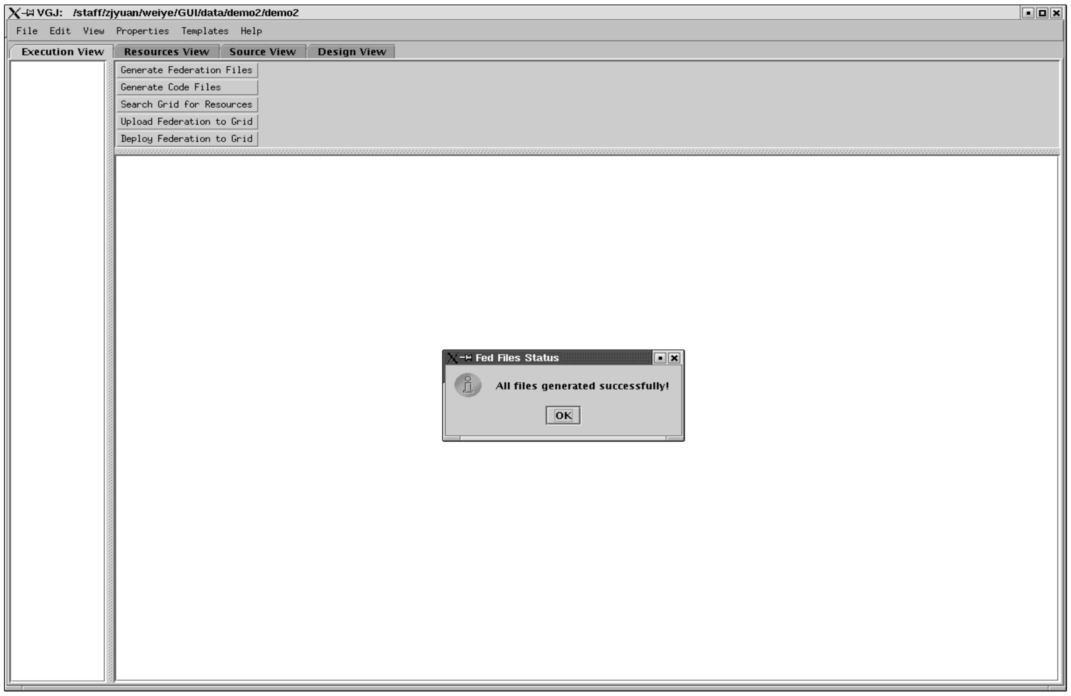
Step 16: The final executable code can be generated from the intermediate files by clicking on "Generate Code Files". The code generator will take the intermediate files as input, link with the SimKernel code library, and produce the final code for all LP instances and event classes. The code generator also compiles the code to executables (Figure B.19).

In summary, the GUI allows the modeler to specify the simulation from the GUI at a higher abstraction level. What is required is the essential simulation information, such as the LPs, events and their associated properties. The modeler specifies how each event is handled by specifying the consume() routine from the GUI, and our code generator will automatically transform the LP-level simulation


VGJ: /staff/zjyuan/weiye/GUI/data/demo2/demo2
File Edit View Properties Templates Help
Execution View
Resources View
Source View
Design View
Generate Federation Files
Generate Code Files
Search Grid for Resources
Upload Federation to Grid
Deploy Federation to Grid
Select an Option
Yes
No
Cancel

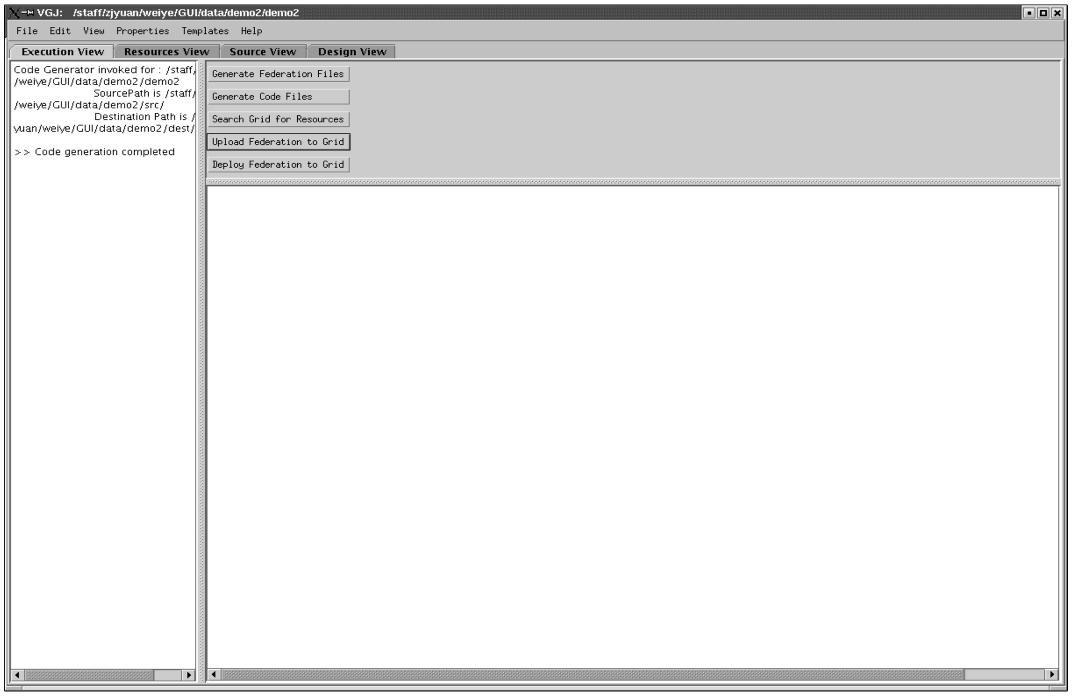
Proceed to create Federation Files ?

Figure B.17: **Generate Federation File**


VGJ: /staff/zjyuan/weiye/GUI/data/demo2/demo2
File Edit View Properties Templates Help
Execution View
Resources View
Source View
Design View
Generate Federation Files
Generate Code Files
Search Grid for Resources
Upload Federation to Grid
Deploy Federation to Grid
Fed Files Status
OK

All files generated successfully!

Figure B.18: **Federation File Generation Completed**


VGJ: /staff/zjyuan/weiye/GUI/data/demo2/demo2
File
Edit
View
Properties
Templates
Help
Execution View
Resources View
Source View
Design View
Generate Federation Files
Generate Code Files
Search Grid for Resources
Upload Federation to Grid
Deploy Federation to Grid

```
Code Generator invoked for : /staff/
/weiye/GUI/data/demo2/demo2
                    SourcePath is /staff/
/weiye/GUI/data/demo2/src/
                    Destination Path is /
yuan/weiye/GUI/data/demo2/dest/

>> Code generation completed
```

Figure B.19: **Code Generation Completed**

design to HLA-compliant code ready for execution. The next section will explain the format of intermediate files and how the code generator works using the same use case.

The GUI was implemented by a Final Year Project (FYP) student, and more details regarding its implementation can be found in [39].

B.3 Manual Setup

The modeler may also create the simulation by manually setup all the required files and run the code generator from command line to achieve exactly the same result as using the GUI. The modeler needs to provide the following inputs:

- **LPConf.txt file** The LPConf.txt file specifies the LP level simulation specifications. This file must conform to the format described in Section 3.7.1.
- **LPInit files** These are the LP's initialization routines that set up each LP's characteristics.
- **Event Encoding/Decoding Definition File** This type of files contains the event parameter encoding/decoding definition for the event class.
- **Event Processing Definition Files** Each event's processing detail definition is defined in a file using naming convention *eventName_srcLP_destLP.java*.

All files should be located at `project/src` folder where ***`project`*** is the simulation project directory. For example, in this use case, all files will be in the directory `demo2/src`. The modeler must create the LPConf.txt file exactly as shown in Figure B.20.

All LP instances' initialization routines are defined in the corresponding LPInstNameInit.java file. For example, LPaInit.java contains the initialization routine for LPa, which is shown in Figure B.21.

For the event class, the encoding/decoding details are defined in a Java file named as event.java. In this case, relay.java will do the job and is shown in Figure B.22.

```
Federation demo2
startLP
        LP1 (String strLocalTime) (LPa LPb LPc LPd)
endLP

startEvent
        event relay
                parameter String theTime
                mapping
                        LPa: LPb relay_LPa_LPb.java
                        LPb: LPc relay_LPb_LPc.java
                        LPc: LPd relay_LPc_LPd.java
                        LPd: LPa relay_LPd_LPa.java
                        LPd: LPb relay_LPd_LPb.java
                endmapping
endEvent
```

Figure B.20: **LPConf.txt File for Use Case Figure B.1**

```
// LPaInit.java

public void init(){
        setTimeLimit(100);

        updateAttribute("strLocalTime",
                Long.toString(System.currentTimeMillis()));

        sendEvent("relay", (String)getAttribute("strLocalTime"),
                getLookahead(), "LPb");
}
```

Figure B.21: **LPaInit.java for LPa in Use Case Figure B.1**

For each event subclass, defined by a source and a destination LP, the event processing behavior is defined in eventName_srcLP_destLP.java file. Figure B.23 shows the *consume()* method for event subclass relay_LPc_LPd.

After all files are created, the modeler can generate the final code by invoking the CodeGenerator with the project directory as parameter from the command line:

```
// relay.java

public void encode(){
        data = theTime;
}

public void decode(){
        theTime = data;
}
```

Figure B.22: **Event Class relay Encoding/Decoding Definition**

```
// relay_LPc_LPd.java

public void consume(){
        System.out.println("start processing relay_LPc_LPd");
        System.out.println("received time is: " + theTime);
        System.out.println("LPd's local time is (before sync): "
                + getLPAttribute("strLocalTime"));

        System.out.println("synching time with remote time ...");
        updateLPAttribute("strLocalTime", theTime);
        System.out.println("done!");

        System.out.println("LPd's local time is (after sync): "
                + getLPAttribute("strLocalTime"));

        String dest;
        if(theTime.endsWidth("5"))
              dest = "LPa LPb";
        else
              dest = "LPa";
        System.out.println("relaying LPd's local time.");
        sendEvent("relay", Long.toString(System.currentTimeMillis()),
                getCurrentTime() + getLookahead(), dest);
}
```

Figure B.23: **Event Processing Definition for relay_LPc_LPd**

```
$ java CodeGenerator demo2
```

The CodeGenerator will look into and take input from the `src` subdirectory of the `demo2` directory, link with the SimKernel code library, and produce the final HLA-compliant `.java` files in the `demo2/dest` directory and compile them to `.class` files. The modeler could start the federate by the following:

```
$ cd demo2
$ java LP_Name
```

Bibliography

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer USENIX Conference*, pages 93–112, 1986.

[2] Roberto Baldoni, Jean-Michel Hélary, Achour Mostefaoui, and Michel Raynal. Consistent Checkpointing in Message Passing Distributed Systems. Technical Report 92, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, France, 1995. PI–925, available online at `ftp://ftp.irisa.fr:/techreports/1995/PI-925.ps.gz`.

[3] B. Beebe, C. Bouwens, W. Braudaway, S. Harkrider, J. Ogren, D. Paterson, R. Richardson, and P. Zimmerman. Building HLA Interfaces for FOM Flexibility: Five Case Studies. In *Proceedings of the 1997 Fall Simulation Interoperability Workshop*, 1997. 97F-SIW-172.

[4] Fran Berman, Geoffry Fox, and Tony Hey. The Grid: Past, Present, Future. *Grid Computing - Making the Global Infrastructure a Reality*, pages 9–50, 2003. John Wiley & Sons Ltd., England.

[5] Azzedine Boukerche and Sajal K. Das. Scalabilty of a Load Balancing Algorithm, and Its Implementation On an Intel Paragon. In *1999 International*

Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99), pages 274–281, Fremantle, Australia, 1999.

[6] M. Brasse and O. Stroosma. A Component Architecture for Federate Development Description. In *Proceedings of the 1999 Fall Simulation Interoperability Workshop*, 1999. 99F-SIW-025.

[7] Carl Byers and Lily Lam. TRAXX: An Extensible Federate Development Framework to Support HLA Compliant Federation Implementation. In *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997. 97S-SIW-043.

[8] Wentong Cai, Stephen J. Turner, and Hanfeng Zhao. A Load Management System for Running HLA-based Distributed Simulations over the Grid. In *Proceedings of the Sixth IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '02)*, pages 7–14, 2002.

[9] Kasidit Chanchio and Xian-He Sun. Data Collection and Restoration for Heterogeneous Process Migration. *Software – Practice and Experience*, 32(9):845–871, 2002.

[10] D. Cohen and A. Kemkes. DDM Scenarios. In *1997 Fall Simulation Interoperability Workshop*, 1997. 97F-SIW-057.

[11] V. S. Dobbs. Managing a Federation Object Model with Rational Rose: Bridging the Gap Between Specification and Implementation. In *Proceedings of the 2000 Fall Simulation Interoperability Workshop*, 2000. 00F-SIW-010.

[12] Ian Foster and Carl Kesselman. The Globus Project: A Status Report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[13] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[14] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems.* Wiley-Interscience, 2000.

[15] J. Graham, J. Foscue, and D. Cutts. HLA Object Models as Software Object Models: An Approach to Automatic Software Generation from HLA Object Models. In *Proceeding of the 1998 Spring Simulation Interoperability Workshop*, 1998. 98S-SIW-043.

[16] Andrew S. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[17] George T. Heineman and William T. Councill. *Component-based Software Engineering: Putting the Pieces Together.* Addison-Wesley Longman Publishing Co., Inc., 2001.

[18] E. Heymann, F. Tinetti, and E. Luque. Preserving Message Integrity in Dynamic Process Migration. In *Proceedings of Euromicro Workshop on Parallel and Distributed Processing (PDP-98)*, pages 373–381, 1998.

[19] J. Huang, Y. Du, and C. Wang. Design of the Server Cluster to Support Avatar Migration. In *Proceedings of The IEEE Virtual Reality 2003 Conference (IEEE-VR2003)*, pages 7–14, Los Angeles, USA, March 2003.

[20] W. Huiskamp. Laguna Beach: HLA on Baywatch? In *Proceedings of the 1999 Fall Simulation Interoperability Workshop*, 1999. 99F-SIW-027.

[21] Ken Hunt, Judith Dahmann, Robert Lutz, and Jack Sheehan. Planning for the Evolution of Automated Tools in HLA. In *Proceedings of the 1997 Spring Simulation Interoperability Workshop*, 1997. 97S-SIW-067.

[22] IEEE. P 1516, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - IEEE Framework and Rules, September 2000.

[23] IEEE. P 1516.1, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -Federate Interface Specification, April 2000.

[24] IEEE. P 1516.2, Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) -HLA Object Model Template (OMT), February 2000.

[25] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-Assisted Full Checkpointing. *Software–Practice and Experience*, 24(10):871–886, October 1994.

[26] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 36–44, 1998.

[27] M. Litzkow. Remote UNIX-Turning Idle Work-stations into Cycle Servers. In *Proceedings of the Summer USENIX Conference*, pages 381–384, 1987.

[28] Johannes Lüthi and Steffen Großmann. The Resource Sharing System: Dynamic Federate Mapping for HLA-based Distributed Simulation. In *Proceedings of Parallel and Distributed Simulation*, pages 91–98. IEEE, 2001.

[29] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, September 2000.

[30] Hyo-Chang Nam, Jong Kim, Sung Je Hong, and Sunggu Lee. Probabilistic Checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 48–57, 1997.

[31] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, pages 23–26, 1988.

[32] S. Parr, A. Radeski, R. Keith-Magee, and J. Wharington. Component-Based Development Extensions to HLA. In *Proceedings of the 2002 Spring Simulation Interoperability Workshop*, 2002. 02S-SIW-046.

[33] Shawn Parr, Alex Radeski, and Rob Whitney. The Application of Tools Support in HLA. In *The 11th Annual Simulation Technology and Trainning Conference*, Melbourne, 2002.

[34] Ellard Thomas Roush. The Freeze Free Algorithm for Process Migration. Technical Report UIUCDCS-R-95-1924, UIUC, 1995. Available online at `http://www.cs.uiuc.edu/Dienst/UI/2.0/Describe/ncstrl.uiuc_cs/UIUCDCS-R-%95-1924`.

[35] J. Cracknell Rule, Nick. Lessons Learned Implementing and Using a General-purpose Federation Management, Prototyping and Debugging Tool. In *Proceedings of the 1997 Fall Simulation Interoperability Workshop*, 1997. 97F-SIW-019.

[36] Luis Moura Silva and Joao Gabriel Silva. System-Level Versus User-Defined Checkpointing. In *Symposium on Reliable Distributed Systems*, pages 68–74, 1998.

[37] F. G. Smith, A. W. Dunstan, and G. H. Lindquist. Information Architecture Based Tools for Interoperable Simulation Within the HLA. In *Proceedings of the 1997 Fall Simulation Interoperability Workshop*, 1997. 97F-SIW-069.

[38] Hussam M. Soliman and Adel Said Elmaghraby. An Analytical Model for Hybrid Checkpointing in Time Warp Distributed Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, October 1998.

[39] Wei Yeh Sun. GUI for HLA/RTI Simulation Modeling. Final Year Report, Nanyang Technological University, 2004. SCE03-333.

[40] Xian-He Sun, Vijay K. Naik, and Kasidit Chanchio. A Coordinated Approach for Process Migration in Heterogeneous Environments. In *1999 SIAM Parallel Processing Conference*, March 1999.

[41] Sun Grid Engine. http://www.sun.com/software/gridware.

[42] Clemens Syzperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[43] Yong Meng Teo and R. Ayani. Comparison of Load Balancing Strategies on Cluster-based Web Servers. *Simulation, The Journal of the Society for Modelling and Simulation International*, 77(5-6):185–195, November-December 2001.

[44] Yong-Meng Teo and Seng-Chuan Tay. Performance and Granularity Control in the SPaDES Parallel Simulation System. In *1999 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99)*, pages 94–99, Fremantle, Australia, 1999.

[45] Ming Q. Xu. Effective Metacomputing Using LSF MultiCluster. In *Procs. of 2001 International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, pages 100–105, Brisbane, Australia, May 2001.

[46] Zijing Yuan, Wentong Cai, and Malcolm Yoke Hean Low. A Framework for Executing Parallel Simulation using RTI. In *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '03)*, pages 12–19, Delft, The Netherlands, October 2003.

[47] Katarzyna Zając. A Framework for HLA-based Interactive Simulations on the Grid. Technical report, Kraków University.

[48] Katarzyna Zając, Marian Bubak, Maciej Malawski, and Peter Sloot. Towards a Grid Management System for HLA-based Interactive Simulations. In *Proceedings of the Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT '03)*, pages 4–11, Delft, The Netherlands, October 2003.

[49] Songnian Zhou. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Trans. on Software Engineering*, 14(9):1327–1341, September 1988.

[50] Songnian Zhou, Xiaohu Zheng, Jingwen Wang, and Pierre Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software-Practice and Experience*, 23(12):1305–1336, 1994.