Enhanced Distributed Simulation Interoperability and Algorithms Using Web Services

By

Khaldoon Al-Zoubi

A thesis submitted to the Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

(OCIECE)

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

May 2011

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis

Enhanced Distributed Simulation Interoperability and Algorithms Using Web Services

Submitted by Khaldoon Al-Zoubi

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical and Computer Engineering

_____

Thesis Supervisor

Dr. Gabriel A. Wainer

_____

External Examiner

Dr. Andreas Tolk

_____

Chair, Department of Systems and Computer Engineering

Dr. Howard M. Schwartz

Carleton University

May 2011

ABSTRACT

With the expansion of the Internet, the desire toward global cooperation in the distributed simulation technology has also been on the rise. As a result, much research has been devoted to develop middleware interoperability methods on the Web, particularly using purely SOAP-based WS or HLA with SOAP extension. However, such frameworks still have constraints in the structural rules that are placed on the middleware design methods. In particular, the way they exchange, structure, and use information is tied to programming, making it difficult to decouple systems implementations and design. In this work, these issues are addressed, enhancing the distributed simulation methods on the Internet using SOAP WS and RESTful WS. In particular, the main objective of the methods presented is to decouple systems design and implementations while allowing composition scalability and dynamicity. To do so, the proposed SOAP-based methods wrap each system in a single WS port while algorithms synchronize simulation via exchanging XML messages. However, the thesis shows how the objective can better achieved by the RESTful Interoperability Simulation Environment (RISE) middleware, which is the first existing RESTful simulation environment. RISE is a general middleware that serves as a *container* to hold different simulation environments without being specific to a certain environment. In RISE, all functionalities are hidden in resources (URIs) that connected to each other via constant uniform virtual channels where all synchronization messages are described in XML. To prove the concept of the general middleware container, we started by RISE-enabling a simulator called CD++, so that it can perform distributed simulation on the Web. This system performance tests have shown promising results comparing to the SOAP-enabled version. Additional

methods have also been defined such as algorithms that could be used as a basis for DEVS standardization, and workflow methods to enhance simulation experimentation automation and reusability via the Web.

ACKNOWLEDGMENT

First of all, I thank ALLAH, the almighty, for giving me the ability and the will to complete anything I want to do.

I would strongly like to thank my thesis supervisor Dr. Gabriel Wainer for all of his constant support, kindness, understanding and patience during this research. I was truly lucky to have Dr. Gabriel Wainer as my friend and supervisor during this research. I also greatly appreciate Dr. Andreas Tolk, Dr. Azzedine Boukerche, Dr. Chris Herdman, Dr. Michel Gaulin and Dr. Babak Esfandiari for being on my defense committee and for their valuable inputs. I would also like to thank Dr. Azzedine Boukerche, Dr. Shikharesh Majumdar, and Dr. Greg Franks for their greatly appreciated inputs during my thesis proposal defense.

I am always in debt for my parents, my uncle, Abraham, my two sisters and my five brothers for their infinite support at all times.

Finally, I greatly thank my wife Sahar and my son Tariq for their endless patience and support in anything I want to do, particularly in this research. I *dedicate* this work for them.

CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

API	Application Programming Interface
CI	Confidence Interval
DCD++	Distributed CD++
DEVS	Discrete EVent Simulation
DS	Distributed Simulation
HLA	High Level Architecture
HTTP	Hyper Text Transfer Protocol
IPC	Inter-Process Communication
M&S	Modeling and Simulation
OS	Operating System
P-DEVS	Parallel-DEVS
REST	Representational State Transfer WS
RISE	RESTful Interoperability Simulation Environment
RPC	Remote Procedure Call
RTI	Run-Time Infrastructure
SOAP	Simple Object Access Protocol
URL	Uniform Reference Locator
URI	Uniform Reference Identifier
WADL	Web Application Description Language
WS	Web-Services
WSDL	Web Services Description Language
XML	Extensible Markup Language
YAWL	Yet Another Workflow Language

CHAPTER 1: INTRODUCTION

Modeling and Simulation (M&S) has evolved to become a discipline that has its own knowledge, formalisms, and methodologies [11]. Indeed, the technology has advanced in the last 30 years to be applied in nearly every aspect of life [11][109][140]. At the heart of the M&S, technology is the *model* concept: a representation of a system with the purpose to promote understanding of that system. The idea is that we abstract what we learned about the system into a model that represents the system under study. This abstraction implies a loss of information, but it allows us to describe the behavior of the system, analyze it, and prove properties of the proposed model [139]. The second concept is *simulation*, which refers to the execution of those models with particular sets of data using a computing device [139]. This approach allowed solving problems with a level of complexity that traditional methods could not answer. Computer-simulated models also have additional benefits: they can be executed safely, and experiments can be easily repeated in a cost effective, risk-free environment and are thus well suited for training purposes [139].

As simulated systems become increasingly sophisticated, the simulation software becomes larger and more complex. In these cases, the resources provided by a single-processor machine often become insufficient to execute these systems. Parallel And Distributed Simulation (PADS) is a technique that deals with these issues by executing simulations over multiple processors [137][138][139]. Distributed simulation is distinguished from parallel simulation by their physical architecture, communication network and latency [58]. Parallel simulation systems usually exist in a machine room

connecting homogeneous simulation partitions with a latency measured of a few microseconds, while distributed simulation can expand from a single building to global networks usually interoperating heterogeneous processors (and software) [58]. A focal point of distributed simulation software has been on how to achieve model reuse via interoperation of different simulation components. Other benefits include [138] connecting geographically distributed simulation components (without relocating people/equipment to other locations), interoperating different vendor simulation components (allowing M&S solutions reuse), providing fault tolerance, and information hiding—including the protection of intellectual property rights [13][125], and simulating larger problems via exploiting more available distributed computer resources (e.g. memory).

As those benefits point out, *interoperability* is indeed the major function of most existing simulation middleware [138][124][125]. Interoperability enables two or more different software systems to interface and use each service correctly [130][138]. In the case of distributed simulation, various simulation partitions (components) interoperate with each other to execute the same simulation run throughout the network. Such interoperability must be achieved at the semantic and syntactic levels: syntactic is the standardized rules to exchange and structure information while semantics are the meaning of the exchanged simulation information [130][154]. As distributed simulation technology increasingly needs to interoperate heterogeneous systems on a global scale (particularly over the Internet), we need to use mature interoperability standards. Consequently, in most cases Web-Services are the technology of choice (the latest widely accepted interoperability standards over the Web, which has been proven to achieve such

global interoperability between diverse systems of different disciplines). Indeed, Web-services have already been successfully used for achieving model and simulation interoperability on the Web [140][154], which can be defined as the ability to deploy services by a machine and consumed by a different machine via the Web [115][140]. At present, the two most popular classes are the SOAP-based Web-services and the RESTful Web-services [115][138]. Both types aim on reusing existing solutions on the Web, motivated by enhancing collaboration to increase products quality with reduced cost. On the other hand, they are different in the way they use to reach this goal, in particular, in the way the services API is designed, orchestrated, and consumed at the software level.

1.1 Motivations, Objectives, and Assumptions

As already mentioned, because of the expansion of the Internet, the way modern systems are being built has taken a historical shift. Nowadays, for example, the customers can have a say at early stages of a product development cycle as the Web can virtually put them inside the factory walls, hence shifting away from the traditional company-centric development methods [118]. Indeed, data sharing and reuse is becoming a necessity for modern research, scientific and engineering organizations as the expansion of the Internet and the advancement of XML and Web services technologies have transformed the way modern systems are built [154]. Certainly, it is a powerful concept is that to be able to assemble a number of existing Web-enabled services from around the globe to form a new novel system. This notion starts gaining momentum in the M&S field as countless of scattered solutions are waiting to be Web-enabled where they can be accessed and reused by other systems, which leads to the need for a widely accepted

interoperability framework to overcome such systems heterogeneity issues [140]. Indeed, while interoperability was in the past as an issue specific for a standard or a system, it is now becoming a globalized notion that goes beyond the boundaries of the M&S field. For example, the US department of defense is in the transition of moving all of its operations into the Global Information Grid (GIG) [154]: a high-speed version of the Internet where globally interconnected information entities, systems, and personnel store, share and distribute information on demand to war fighters, policy makers, and support personnel. Therefore, the road is currently open for the distributed simulation to incorporate different simulation assets around the globe, motivated by reducing products cost and enhancing solving problems rapidly. These types of motivations have affected the course of distributed simulation technology and its requirements.

At present, the trend of distributed simulation (as discussed in Section 2.3) is going toward cooperation at globalization level, information hiding and achieving a general container middleware that is able of supporting different simulation components [12][125]. Thus, the middleware is in the heart of advancing distributed simulation use and applications. This is because the middleware is responsible of exposing all simulation components that reside on a machine [124][138]. Thus, the middleware does not only affect the information flow from and to those components, but also the boundaries of the audience circle that can access those components. Further, the middleware interoperability methods play a major role in affecting the overall distributed simulation synchronization algorithms and semantics [143]. This is because the distributed simulation partitions need both communication and synchronization to work correctly, which both of these are under the middleware control. These issues are clearly recognized

in the literature, for instance most published surveys (e.g. [12][13][125]) of experts of different backgrounds require the distributed simulation middleware to solve most of the technology current challenges, as discussed in Chapter 2. Therefore, Web Services allow building distributed simulation middleware that can interoperate diverse simulators, assemble new systems from existing ones, and put anything within a simulation loop (regardless of systems platforms, implementations, architectures or geographical boundaries). We envision, for example, a user that pipelines different solutions outputs (collected from around the world) as inputs to other components by simple drag-and-drop operations to invent a new system that might have taken years to build without such capability. However, such vision will never become reality until enough research is conducted to study and experience the basic distributed simulation interoperability at the software level (particularly by using Web-services over the Internet).

Based on the above, our objective is to develop an all-purpose Web-services based distributed simulation middleware with features similar to the following:

- To find a framework that eases distributed simulation interoperability protocols between different independent-developed simulation systems. In this case, systems heterogeneity (which resides in implementation) must be hidden in the way information is exchanged, accessed and described, hence at the software levels of the API and semantics.
- The middleware should serve as *container* of any type of simulation environment. Therefore, the middleware must be independent of any specific implementation, synchronizations algorithms and semantics, or formalisms. This means that the

simulation environments should operate at a different layer above the middleware, allowing the middleware to be extended to support various environments.

- Composition scalability: the middleware must ensure composition scalability in the distributed simulation environment regardless of the number of *partitions*.
- Each simulation environment (component) type should be able to support multiple interoperability distributed simulation synchronization protocols. This includes the way models partitions are coupled, the way simulation is advanced and the way simulation messages semantics are structured and processed. Such protocols should also be flexible enough for future enhancements in particular integrating performance related techniques.
- Design scalability: The middleware structure must scale up when adding/removing a simulation environment type. This also includes the middleware address space (URIs).
- Experiment-oriented framework:
    - Modeler-oriented: Experiment instances resources (URIs) are created and named by modelers. They can be created for any model, of any settings of any simulation environment.
    - Define an experiment with a semantic Web interface. In this case, an experiment instance is spread over a number of URIs, similar to any other attached URIs to the Web. This interface enables those simulation experiments to be manipulated externally via the Web.
    - Define a blueprint experiment pattern that is satiable for workflows use, allowing experiments sharing, automation, repeatability, and reusability.

- Distributed simulation experiments partitions are configured and distributed on participant machines by the middleware.
- All experiment resources (URIs) must be preserved indefinitely unless deleted by their owner (i.e. the authorized user).

The research first attempted to achieve these objectives using the SOAP-based WS, which was successful to some extent in hiding implementations in components and reducing synchronization protocols programming dependency, as discussed in Chapter 4. However, since interoperability is realized at the software level via their systems API links and the exchanged information through those links, it makes it difficult for a middleware to achieve certain objectives if an applied technology (i.e. SOAP-based WS in this case) places implementation constraints on those software interoperability joints (see Chapter 2). These problems were better solved via basing the middleware on RESTful WS, mainly because REST separates systems implementations from their API and the exchanged information semantics, which is a fundamental concept that affects most of the middleware interoperability methods. The RESTful Interoperability Simulation Environment (RISE) middleware we developed is the first existing middleware to achieve these goals via the use of RESTful WS.

The RISE middleware serves as general container that is capable of encapsulating different simulation environments. This general concept led to layered interoperability, which includes the *Middleware Layer*, the *Simulation Layer*, and the *Modeling Layer*. The Middleware Layer provides all means to exchange all information (on behalf of the supported simulation environments) and to encapsulate supported solutions. The

Simulation Layer deploys different simulation environment types (e.g., DCD++, discussed in Chapter 6) where the Modeling Layer operates on the top of a simulation environment. These three layers cover all interoperability aspects of distributed simulation interoperability and modeling. Thus, it is beyond a single research to face all of simulation and modeling related issues at once. Hence, this research focuses on some of these issues, opening the door for future researches to carry on further (see Chapter 9). Consequently, this thesis makes the following assumptions:

- As the RISE middleware serves as a general container for different simulation environments, the simulation *time* and *data distribution* management should be a part of the simulation environment responsibilities (at the simulation layer). This approach allows the middleware to be open for extensions, in particular for supporting different types of simulation environments.
- Model representations are compatible with the simulation environments in their partitions. Thus, the conceptual alignment is provided by the model designer, and not by the middleware. In other words, RISE assumes that the distributed model interrelations comes from the modeler (e.g. in form of XML file) as part of setting up an experiment. However, the middleware should allow the modeler to manipulate this information at anytime.

It is worth to note that the RISE environment layers: *Middleware*, *simulation*, and *modeling* still match other existing interoperability conceptual frameworks, particularly the Level of Conceptual Interoperability Model (LCIM) framework [130]. The LCIM generally divides interoperability into three general layers (Figure 1): *Integratability* (which deals with networks issues and connectivity), *Interoperability* (which deals with

the software implementations of interoperations, including simulation and middleware), and *Composability* (deals with the alignment issues of the models).

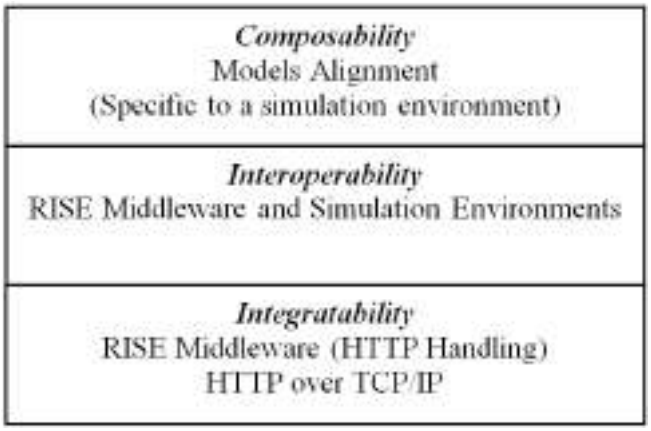

Figure 1: RISE Layers within LCIM Framework

In the scope of this thesis, the focus is on the LCIM *Interoperability* layer (Figure 1). In RISE, this layer contains the middleware and supported simulation environments (which can be extended with simulation environments). The RISE middleware is involved in the *Integratability* layer for handling the HTTP envelope and TCP connections. However, in the LCIM *Composability* layer, the RISE middleware assumes that the models conceptual alignment is provided by the designer as previously stated in the Assumptions above. Note that in this thesis context, the *composition* term deals with interoperating various distributed simulation partitions. In this case, a *partition* is number of URIs (within the middleware) that encapsulates a portion of a distributed model along with a simulation environment that is capable of executing that portion of the model. Our use of this term is in conformance of the Web service literature [115]. In this case, a partition is viewed as a WS component. However, in the M&S field, as in the case of the LCIM conceptual framework and other existing works (e.g. [45][106][113]),

composability in some cases deals with the issues of constructing a model correctly to achieve a solution.

The RISE middleware is designed as a *resource-oriented* architecture, which means that all functionalities (resources) are spread in a parent-child relationship structure. Resources are addressed via URI templates (blueprints). Thus, resources are classes of services whose URI instances can be named and created at runtime. Therefore, RISE deploys simulation services in classes so that modelers can create instances of those services classes as part of their experiments. This provides modelers with an experiment blueprint since an experiment instance can be created for any simulation environment with settings manipulated via a few URIs throughout its life cycle. To enhance composition scalability, each resource (URI) is connected via few virtual software *channels* (i.e. HTTP methods in RISE case), hence a *uniform-interface* mechanism to exchange information between resources instances (i.e. experiments URIs). This means that each resource is connected with the same number of virtual links regardless of the number of simulation partitions. Further, all synchronization simulation messages semantics are described in XML, a *message-oriented* method describing the syntax of the information exchanged via those uniform channels. The uniform interface and the message-oriented concepts allow RISE to hide system implementations in components, which is where systems heterogeneity lives. Further, the use of XML messages allows each simulation service to support multiple synchronization algorithms and protocols to accommodate different interoperability domains, which allow systems to evolve independently.

1.2 Thesis Contributions

As discussed in the previous section, the central theme of this thesis is to improve distributed simulation interoperability and synchronization algorithms at the software level. Our objective is to develop an all-purpose Web-services based distributed simulation middleware. The interoperability methods are introduced by studying the interoperability structural and syntactic rules that form the foundations of the design methodologies of those methods. Those methods were initially developed using the SOAP WS framework. However, due to the numerous problems posed by SOAP WS, we explored the use of the RESTful WS. This thesis presents the first existing effort on distributed simulation interoperability methods using the RESTful WS principles according to the major Web standards: HTTP, URI, and XML. The key contributions are summarized as the following:

- The research first extended an existing SOAP-based DCD++ with a new WS component to enhance hiding the CD++ internal implementation as much as possible. This component consists of a few RPCs and supports synchronization algorithms that describe exchanged messages between partitions in form of XML messages (i.e. as SOAP attachments).
- The design and development of the RISE middleware, which is the first existing middleware to be fully based on RESTful WS principles according to the Web standards. The RISE key features are summarized as the following:
  - It is a general middleware where it serves as a container to hold different software components without being specific to any implementation.

- It organizes the resources in a scalable hierarchical relationship. Resources are exposed by the middleware as URI templates. This means that those resources instances can be created and named by modelers at runtime. This leads to the concept of general layered interoperability where different simulation resources (URIs) organized at a separate layer above the middleware.
- Because RISE experiments are seen as URIs, URI templates allow RISE to provide experiments as blueprint patterns. This means that various experiments of different types and URIs can be created at runtime, and that the usual steps that are usually performed manually to create and manipulate experiments can be automated, as in the case of simulation workflows.
- RISE maintains all URIs in a database (unless they are deleted by authorized users). This database allows RISE to maintain all URIs similar to a typical HTTP server, allowing all experiments related data such as models and simulation results to be accessed via their URIs on the Web.
- The resources (URIs) *access channels* are designed in RISE as virtual software channels, and are realized outside the resources internal implementations. Further, by having standardized channels for each resource, RISE enhances composition scalability and improves dynamicity in distributed simulation (since the channels of each resource automatically exist upon that resource creation). Furthermore, because of the uniform interface, all messages are transmitted uniformly regardless of the number of destinations.
- The exchanged *data syntax* is designed as XML messages used by the distributed simulation synchronization algorithms. This further enhances the idea of supporting

multiple synchronization methods, since each algorithm can have its own set of XML messages.

- In RISE, since the *exchanged data (XML messages)* and the *access channels* form the resources APIs, thus, simulation systems APIs are realized outside interoperated systems implementations. In contrast, current approaches (e.g. HLA, SOAP WS) systems APIs are embedded in implementations (Chapter 2).
- The RISE middleware is fully multithreaded. In this case, each message is transmitted in its own thread, and each received message to a URI is processed within its own thread. This mechanism allowed RISE to balance load distribution well under increasing workload pressure.
- The Authentication and Authorization scheme is required on all channels that might modify a resource while not required, by default, on the read-only channels.
- API XML Description (based on XML WADL standards) is used to publish the API in XML machine processing format. This WADL document is retrieved on demand by users via the HTTP OPTION channel (see Appendix-B).
- RISE supports two types of deployments as a standalone HTTP server or as a Servlet runs inside any HTTP server containers (see Appendix-A).

- An interface is developed between CD++ and the RISE middleware so that various instances of CD++ engines can cooperate with each other to simulate the same distributed simulation session. In this case, the distributed CD++ (DCD++) simulation is performed within the RISE middleware experiment framework. This means that experiment URIs can be created, named, and manipulated at runtime. During simulation, algorithms synchronize simulation activities by exchanging XML

messages between partitions URIs. Further, algorithms use an aggregation scheme to group multiple simulation messages in single XML messages to reduce the number of remote transmissions via the Web. These algorithms (via RISE) showed a substantial performance improvement when compared to the current SOAP-based DCD++ system.

- The design of a workflow component (on the client side) that could be used to automate the steps been taken by modelers to create and manipulate experiments (which can be easily implemented on the RISE middleware). Such component would serve as means for automation, repeatability, controlling processes and management.
- This research has also contributed in the process of DEVS standardization to interoperate different DEVS-based implementations by organizing and evaluating different DEVS groups recent interoperability methods (published in [140][141][142][143]). As part of this process, the research proposed distributed simulation algorithms to synchronize simulation activities via the Web by exchanging XML messages wrapped in HTTP envelopes. The main RISE methods have also been incorporated into those algorithms such as hiding systems implementations, composition scalability, dynamicity and message aggregation in XML. A dynamic extension has also been proposed to handle the idea of having systems join/disjoin simulation session during runtime. The main contribution of such algorithms is that they are presented without dictating their software implementation methods.

1.3 Research Publications

Parts of this thesis have appeared in the following publications:

Journal Papers:

- Wainer, G.; Madhoun, R.; Al-Zoubi, K. “Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web-Services”. Simulation Modelling Practice and Theory 16 (2008), pp. 1266-1292. Elsevier.

Book Chapters:

- Al-Zoubi K.; Wainer, G.; “Distributed Simulation Using RESTful Interoperability Simulation Environment (RISE) Middleware”. Chapter 6 in “Handbook on Intelligence-based Systems Engineering”. Andreas Tolk and Lakhmi Jain Editors. Springer-Verlag (expected publication in 2011).
- Wainer, G. and Al-Zoubi, K. "An Introduction to Distributed Simulation". Chapter 11 in book “Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains”. Catherine Banks, John Sokolowski Editors. Wiley. New Jersey, 2010.
- Wainer G., K. Al-Zoubi, S. Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “DEVS Standardization: Foundations and Trends”. Chapter 15, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.
- Wainer G., K. Al-Zoubi, S. Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “An Introduction to DEVS Standardization”. Chapter 16, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.
- Wainer G., K. Al-Zoubi, S. Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “Standardizing DEVS Model Representation”. Chapter 17, “Discrete-Event

Modeling and Simulation: Theory and Applications." G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

- Wainer G., K. Al-Zoubi, S. Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. "Standardizing DEVS Simulation Middleware". Chapter 18, "Discrete-Event Modeling and Simulation: Theory and Applications." G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

Conference Papers:

- Al-Zoubi K.; Wainer, G.; "Managing Simulation Workflow Patterns using Dynamic Service-Oriented". Proceedings of the Winter Simulation Conference (WSC 2010). Baltimore, Maryland, USA. 2010.
- Al-Zoubi K.; Wainer, G. "RISE: REST-ing Heterogeneous Simulation Interoperability". Proceedings of the Winter Simulation Conference (WSC 2010). Baltimore, Maryland, USA. 2010.
- Al-Zoubi K.; Wainer, G. "Using REST Web-Services Architecture for Distributed Simulation". Proceedings of the 23rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2009). Lake Placid, New York, USA. June 2009.
- Al-Zoubi K.; Wainer, G. "Performing Distributed Simulation with RESTful Web-Services Approach". Proceedings of the Winter Simulation Conference (WSC 2009). Austin, TX, USA. 2009.
- Al-Zoubi K.; Wainer, G. "Interfacing and Coordination for a DEVS Simulation Protocol Standard". Proceedings of the 2008 12th IEEE/ACM International

Symposium on Distributed Simulation and Real-Time Applications (DS-RT '08). Vancouver, BC, Canada. October 2008.

1.4 Thesis Organization

The rest of the chapters are organized as follows: **Chapter 2** provides background information about the distributed simulation state-of-the-art and Web-services technologies. It further discusses the trends and challenges of distributed simulation and current approaches limitations. **Chapter 3** discusses the main argument of the thesis. It summarizes the major existing problems in current distributed simulation interoperability (previously detailed in Chapter 2) and shows how these issues are solved based on both SOAP-based (detailed in Chapter 4) and RESTful WS (detailed in Chapter 5). **Chapter 4** discusses the proposed design methodologies and implementation based on the SOAP-based WS structural rules and standards. **Chapter 5** discusses the RESTful Interoperability Simulation Environment (RISE) middleware design methodologies. **Chapter 6** discusses the design methodologies of interfacing CD++ into RISE so that it can perform distributed simulation on the Web. It also describes the distributed CD++ (DCD++) algorithms that enable various CD++ instances to synchronize the distributed simulation via RISE. **Chapter 7** analyzes the performance of the RISE middleware in two parts. The first part studies the middleware sensitivity under increasing concurrent workload pressure. The second part studies the distributed simulation performance by comparing the SOAP-based DCD++ against the RISE-based DCD++. In this case, both systems are placed in a similar testing environment while simulation performance is compared over executing various CD++ models. **Chapter 8** describes additional RISE-

based methods. It first presents additional distributed simulation algorithms built with RISE (which could be used as a feasible proposal for DEVS standardization). It then shows how RISE could improve simulation experimentation via the use of workflows. **Chapter 9** summarizes the thesis topics and presents suggestions for possible future work. **Appendix-A** presents the RISE middleware implementation, RISE deployment types, and the implementation of interfacing the CD++ engine into RISE. **Appendix-B** summarizes the RISE API with more focus on the simulation blueprint experiment related API, since they form the URI templates for all simulation experiments.

CHAPTER 2: BACKGROUND

Distributed simulation technologies employ multiple distributed processors, connected via communication networks, to execute the same simulation run over a geographic area correctly [58][138]. Correctness means that the simulation should produce the same results as if it was executed sequentially using a single processor [58]. Distributed computers can expand from a single building to global networks, often employing heterogeneous processors (and software), and communication latency is measured with hundreds of microseconds to seconds. The simulation is divided spatially (or temporally) and mapped to participating processors [58].

A focal point of distributed simulation software has been on how to achieve model reuse via interoperation of different simulation components. Indeed, interoperability is the major function of most existing simulation middleware [138][124][125]. Such interoperability must be achieved at the semantic and syntactic levels [130]. Syntactic is the standardized rules to exchange and structure information while semantics are the meaning of exchanged simulation information [130].

In this chapter, we present an extensive survey of different applied distributed simulation approaches. In terms of where they come from, their current state, and where they are heading. These approaches have advanced distributed simulation by exploiting other technologies advances in software, hardware and standards. In most cases, the Web-services technology is highly leveraged, which has proven useful in achieving model and simulation interoperability [140][154]. Web-services (WS) provide general interoperability syntactic standards, enabling deployment of services on a machine and

consumed by another via the Web. They fall into two popular classes: SOAP-based WS and RESTful WS [115].

2.1 A Brief History of Distributed Simulation

Simulations have been used for war games by the U.S. Department of Defense (DoD) since the 1950s. However, until the 1980s, simulators were developed as stand-alone and with a single-task purpose (such as landing on the deck of an aircraft carrier). Those standalone simulators were extremely expensive compared with the systems that they were supposed to mimic. For example, the cost of a tank simulator in the 1970s was $18 million, while the cost of an advanced aircraft was around $18 million (and a tank was significantly less) [91].

The defense sector started such distributed simulations with the SIMulator NETworking (SIMNET) project in 1983 [31][91][128][129]. The success of SIMNET led to developing standards for distributed interactive simulation (DIS) during the 1990s [75][76][77][78]. Indeed, DIS can be viewed as the standardized version of SIMNET. SIMNET and DIS use an approach in which a single virtual environment is created by a number of interacting simulations, each of which controls the local objects and communicates its state to other simulations. This approach led to new methods for integrating existing simulations into a single environment, and during the 1990s, the Aggregate Level Simulation Protocol (ALSP) was born. ALSP was designed to allow legacy military simulations to interact with each other over LANs and WANs. For example, ALSP enabled Army, Air Force, and Navy war game simulations to be integrated in a single exercise [6][56][9]. The next major progress in the defense

simulation community occurred in 1996 with the development of the High Level Architecture (HLA) standards [72][73][74][82]. HLA was a major improvement because it combined both analytic simulations with virtual environment technologies in a single framework [58]. The HLA replaced SIMNET and DIS, and all simulations in US Department of Defense (DoD) are required to be HLA compliant since 1999 [58].

The distributed simulation advancement in the defense community along with the popularity of the Internet in the early 1990s led to the emergence of nonmilitary distributed virtual environments, for instance, the distributed interactive virtual environment (DIVE) (which is still in use since 1991). DIVE allows a number of users to interact with each other in a virtual world [55]. The central feature in DIVE is the shared, distributed database where all interactions occur through this common medium. Another environment that became popular during the 1990s was the Common Object Request Broker Architecture (CORBA) [67]. CORBA introduced new interoperability improvements since it was independent of the programming language used. On the other hand, CORBA use had sharply declined in new projects since the year 2000 [66]. Examples of CORBA based simulation are described in [37][88][152]. Some reflect this for being very complicated to developers or by the process by which the CORBA standard was created (e.g., the process did not require a reference implementation for a standard before being adopted [66]). Further, Web services emerged as an alternative approach to achieve interoperability among heterogeneous applications (which also contributed to CORBA's decline). Web services standards were fully finalized in the year 2000 (i.e. SOAP [25][83] and WSDL [39][40]). However, other WS needed standards particularly the TCP/IP, HTTP, and XML [27][150] standards had already matured

during the 1990s. SOAP WS and CORBA programming styles are similar, but with different standards (e.g. WSDL vs. IDL). This similarity perhaps allowed CORBA-based legacy systems to migrate to SOAP WS with less difficulty. This point can be seen when CORBA-based simulation systems are compared against their SOAP WS counterpart systems design (e.g. [37][46][152]). Further, Web-services technology fever has recently spread back to the defense sector. For example, the new HLA IEEE 1516-2007 [74] standard is extended with a Web-service interface.

It is worth to note that the Web-based simulation concept goes beyond having several computers performing distributed simulation on the Web. In addition to Web-based distributed simulation, other Web-based simulation categories have been used [29][44][53][107]. Examples of such categories are local visualization (to download a simulation engine with visualization capabilities), remote visualization (to use a browser to access a remote simulation engine with visualization capabilities), hybrid visualization (to execute simulation remotely and view results locally), and models repositories (to store models representations to enhance models reusability).

2.2 A Survey of Distributed Simulation Approaches

Distributed simulation can be divided into two parts: algorithms and middleware [138]. Distributed simulation middleware is responsible for interoperability and composability. On the other hand, algorithms are concerned with synchronizing the overall simulation correctly. They can be classified in two types [58]: the conservative type (e.g. [28][34][43]) which executes safe events to avoid local causality errors while the optimistic type (e.g. [80][97][117]) advances simulation optimistically, but fix those

errors once detected [58]. The common implementation of the existing distributed simulation systems is summarized as follows (e.g. [19][36][152]): **(1)** A time-coordinator component requests the minimum time from all partitions. **(2)** The time-coordinator calculates the global minimum time, broadcasts it to all partitions, and waits for their replies. **(3)** The time-coordinator instructs all the partitions to execute the events with minimum global time, waits for all partitions replies, and starts again with step #1.

2.2.1 HLA-based Simulation and Limitations

The HLA [87] was developed to provide a general architecture for simulation interoperability and reuse [72][73][74]. Examples of HLA-based simulations are described in [14][69][135]. Figure 2 shows the overall HLA simulation interaction architecture. The HLA simulation entities are called *Federates*. In this case, multiple federates (called a *Federation*) interact with each other to synchronize the overall simulation using the *Run-Time Infrastructure* (RTI).

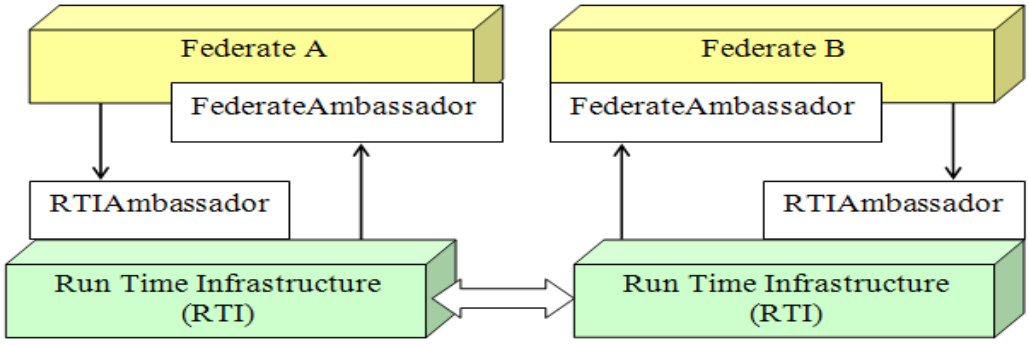

Figure 2: HLA Interaction Overview

The HLA standards consist of three parts: **(1)** the *Object Model Template* (OMT) [73], **(2)** the *HLA Rules* [74]**,** and **(3)** the *HLA Interface Specification* [74]. The *Object Model Template* (OMT) [73] provides standards for documenting HLA Object Model

information. This ensures detailed documentation (in a common format) for all visible objects and interactions managed by all federates. The *HLA Rules* [74] describe the federates' obligations and their interactions with the RTI. The *HLA interface specification* standardize the API between federates and RTI services [74]. The specification defines RTI services and the required callback functions that must be supported by all federates. Many contemporary RTI implementations conform to the IEEE 1516 and HLA 1.3 API specifications such as Pitch pRTI™ (C++/Java) [111], CAE RTI (C++) [30], MÄK High Performance RTI (C++/Java) [94], and poRTIco (C++) [110]. However, the RTI component itself is not part of the standards.

The HLA standard is mainly involves in interfacing federates with local RTIs in a specific programming language. RTI is the important component in HLA, since it connects the whole federation. Federates use the *RTIambassador* method to invoke RTI services while the RTI uses the *FederateAmbassador* method to pass information to a federate in a callback function style (Figure 2). A callback function is a function passed to another function in the form of reference (e.g., C++ function pointer). These callback functions are called *interactions* in HLA.

The RTI provides a number of different services groups. Two of the major groups are the *Time Management* and the *Data Distribution Management* (DDM) services. DDM services distribute data in the federation (i.e. data routing) while the *Time management* services ensure events delivery and logical time advancement. These two groups usually get the most research attention because of their direct affect on the RTI performance and its quality of service (QoS). Some of the RTI performance research efforts (e.g. [20][21][99][157]) focused on using techniques such as multi-threading and load-

balancing. Other efforts (e.g. [16][22][23][127]) focused on improving DDM methods of shipping attribute regions between RTIs. The main purpose of such methods is to avoid unnecessary information transmission across the network. Further, some research efforts focused on providing RTI formal design methods. For example, work described in [17] uses DEVS formalism [153] as the formal method to design a Real-time RTI.

On another line of work, a new WSDL API has been added to the HLA IEEE 1516-2007 standard [74]. Examples of such efforts are described in [18][84][102][156][158][159]. In this way, federates can connect with the Web Service Provider RTI component (WSPRC) using SOAP WS (Figure 3). However, this solution still does not solve interoperation of different WSPRC vendors, since the standards do not cover this part. Further, it does not provide a scalable solution, since many simulation components are still managed by a single component.

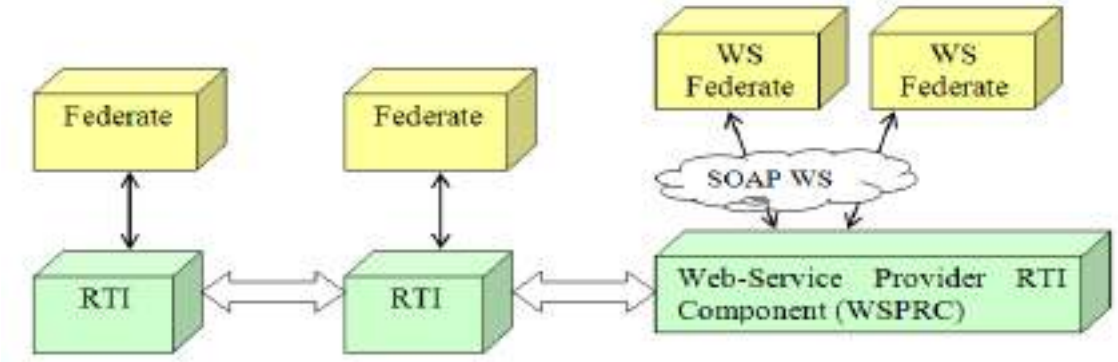

Figure 3: Interfacing RTI with Web Services

The HLA presents a number of shortcomings that can be summarized as follows:

- The HLA only standardizes the programming interface (functions) for integrating federates into local RTI, but RTI-to-RTI interoperability is not standardized, hence vendor specific.

- The HLA RTI does not scale up and acts as a shared bus for all federates.
- It can be difficult to integrate federates with a local RTI if the federates are not implemented with the same language of that RTI. This even gets worst if federates are implemented in a language other than C++ or Java, since existing RTIs mostly implemented in those programming languages.
- The RTIs maintain simulation information as attributes in regions. In this case, an RTI ships an entire region to other remote RTIs once that region is updated. Thus, the RTIs may transmit redundant information remotely. To solve this problem, the HLA DDM services require optimization via complex algorithms to deliver the information needed as accurate as possible (e.g. [15][21][22][113]).
- Interfacing federates with RTIs can vary from a standard to another. For example, it is a strong selling point for commercial RTIs to be able to use the old HLA-1.3 based federates with the new HLA-1516 based RTI implementations. This is because the federate-RTI interface is programming design dependent. Thus, it may not be trivial issue to migrate those functions to the new standard design.
- The standards are complex and tied to programming languages.

2.2.2 SOAP-based WS Simulation and Limitations

SOAP-based WS simulations [49][108] use a client-server model where clients initiate requests to consume services from remote servers. The SOAP-based WS framework is discussed in details in section 2.4.1. In this case, the simulation components (e.g. [89][101][137]) act as both client and server. A simulator becomes a client when it sends a message, while the receiver simulator becomes the server. This mechanism is shown in Figure 4. In this method, the sending component passes all simulation

information as parameters into the subject procedure (stub) and makes a procedure call. Consequently, a procedure (service) is called in the receiver simulation software where the passed-in information is processed. This is exactly like invoking a procedure directly at the receiving simulation software (Figure 4). Therefore, all different simulation components become a virtual single implementation coupled by those procedures. It is worth to note that XML is not applied at the simulation synchronization level, in spite of being used at the underlying Web-service level (discussed in Section 2.4.1). On the other hand, the XML standards have widely been recognized as the way to arrange information [154].

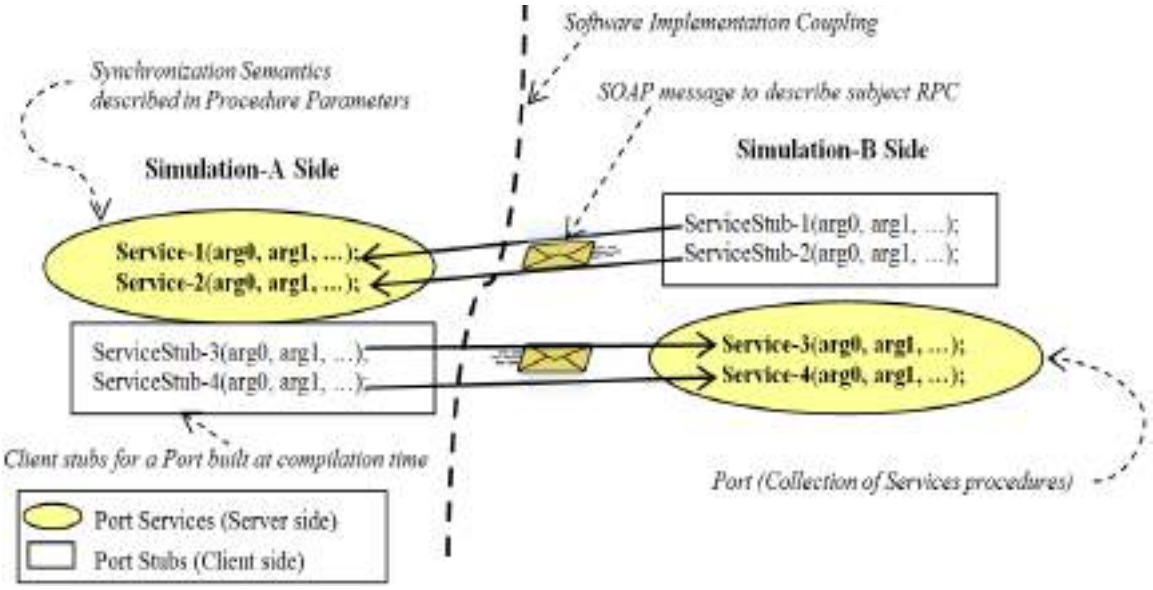

Figure 4: Interoperating SOAP-based WS Simulations

Many simulation systems have evolved and been developed independently. Thus, interoperating those systems via the RPC style API can be a complex task to accomplish. This task even gets more complicated when attempting to interoperate legacy independent-developed systems. This is because all existing heterogeneous APIs for all

systems need to be homogenized. The RPC style is a heterogeneous interface because procedures are invented by different programmers to fit their systems specific needs. In this type of interoperability, simulation systems are usually best to be designed according to a preexisting common RPCs interface. For example, the IDSim federation simulators [57] use the Globus Toolkit [52] implementation of the Grid Services Infrastructure (OGSI) [134] specifications to provide distributed simulation on the Web. The OGSI defines SOAP-based WS interoperability specifications with defined set of RPCs. In this case, the IDSim software design architecture (as described in [57]) has clearly been extended from the Globus implementation. This is because the SOAP WS sneak into software design and implementations. This may work well for the IDSim system, but it may not necessary work as well for a different system (in particular, if the latter system had already been designed without taking OGSI interface in consideration).

Further, this type of API leads to other challenges particularly in the areas of systems composition scalability and dynamic interoperability. The composition scalability problem arises because a procedure stub is needed at the client side for every unique service at the server side (Figure 4). Further, because these stubs are programming procedures, they need to be written and compiled with the client software. This is a static approach and can be a problem for simulation systems that need to apply dynamic interoperability. For example, the Ad Hoc distribution simulation based systems require that the number of simulators is not known in advance and can be changed during simulation runtime [59]. Furthermore, passing simulation information in a form of procedure parameters complicates semantic standards. This is because implementation issues can easily sneak into those standards.

A performance issue worth mentioning is that RPCs in existing SOAP-based simulation implementations (e.g. [89][101][137]) are implemented as blocking calls. This means that the sending simulation needs to be blocked until the RPC is completed.

To summarize RPC-style issues:

- RPCs split implementation between different systems, since procedures in a system implementation directly point to other procedures in other systems implementations.
- Heterogeneous interface, since RPCs were invented by programmers to fit their systems specific needs.
- Making interoperability sensitive to changes since any slight change would break interoperability with other systems even at compiling time.
- Static approach since software compiling is required in advance before interoperating with new systems.
- Make it difficult to support multiple interoperability protocols. This is because exposing different RPC-based APIs still need to map to the same internal calls.
- Make it difficult to develop semantic standards. This is because semantics need to be expressed as procedure parameters, which usually sneak into internal systems implementations, hence standardizing implementation.
- Services composition does not scale well since a stub is required at the client side for each unique service at the server side.
- Tools convert WSDL into empty stubs, leaving programmers to write those stub functionalities, which would slow services compositions.

- Tools treat changes to existing RPCs as new services, causing programmers to verify and test each new RPC against already existing services.

2.2.3 Additional Simulation Approaches

We already covered a number of approaches when we discussed the history of distributed simulation such as DIS, ALSP, CORBA, and DIVE. However, more approaches were also used.

Distributed simulation has also been used in the grid environment. For example, DEVS/Grid [121] implements a grid-enabled DEVS simulator following a layered approach. Another grid example is described in vGrid [86], which divides the model into components that can be grouped together to form a virtual computational unit.

Other distributed simulation systems made a use of the JXTA standards [81][147]. JXTA is an open peer-to-peer (P2P) [147] standards developed by Sun Microsystems. Examples of such efforts are described in [19][36]. In the P2P systems, simulation messages may go through a number of intermediate machines before reaching their destinations.

DEVS/RMI [155] is a distributed DEVS simulator based on Java Remote Method Invocation (RMI). RMI is similar to CORBA and SOAP WS at runtime, but works only with Java. ISEE [100] is distributed simulation example that is based on MySQL database [48]. In this case, all of the simulation components synchronize via the MySQL database unit, which keeps tracks of all states and activities across the network. Performing distributed simulation directly on top of TCP was also used. For example, the work described in [96] performs real-time distributed simulation using TCP socket communications.

2.3 Trends and Challenges of Distributed Simulation

The defense sector is currently the largest user of distributed simulation technology, but limited elsewhere. In recent years, there have been some studies conducted in the form of surveys of experts from different backgrounds such as the ones described in [12][13][124][125]. Those studies aimed on analyzing a number of issues concerning the current distributed simulation state-of-the-art and research challenges that must be resolved with the purpose of advancing these technologies use particularly outside the defense sector. It has been predicted that in the coming years, the sectors that will drive future advancement in distributed simulation are not only the defense sector, but also the gaming industry, the high-tech industry (e.g., auto, manufacturing, and training), emergency, and security management [125]. The studies clearly pointed out for the need to enhance cooperation across geographical areas at different levels, as follows:

- Cooperation at globalization level, which is driven by economic incentives to form industrial clusters [125]. The Internet came as the natural choice to globalize this cooperation.
- Hiding information in components where they conduct distributed simulation together, which might become a necessity when extending the product development beyond factory walls [60][138]. Further, because of the cooperation at global scale, the desire to protect intellectual property right (IPR) for different products is also increased; hence, information and design need to be hidden [125].
- Packaging functionalities as commercial off-the-shelf (COTS) simulation packages with the ability to select sub-components in the final product [12][13][125]. At

present, the COTS simulation packages do not usually support distributed simulation due to the cost/benefit issue [12][13].

Based on the above, those surveys also have acknowledged that the most research challenge that needs to be resolved in order to advance distributed simulation technology is to ease interoperability via the middleware. Consequently, a number of middleware challenges that need to be resolved have been identified:

- It must ease the use of distributed simulation algorithms. This means that the middleware interoperability method must not place constraints on improving synchronization algorithms.
- It must be based on widely accepted syntactic standards.
- It must be accessed by any device from anywhere. This also should allow human/information in simulation loop from anywhere.
- It must ease semantic interoperability. This includes easing semantic standards to allow interoperating different systems and standards. This further means that different semantic standards may be developed by different distributed simulation communities.
- It needs to map the use of existing simulation solutions in the WWW into a Semantic Web method. The “Semantic Web” term can be defined differently by different people. However, the W3C defines it as the ability of a software system to realize the meaning of information on the WWW [149].
- It must ease integrating COTS simulation packages to reuse the same local middleware. The studies do not define the type of packaged functionalities in the

COTS components. However, if a component is a simulation model, thus this model still needs a simulation engine to execute it.

The studies have also recognized the difficulty for resolving the above issues using the HLA or the DIS approaches mainly due to the reasons that have previously been discussed in the previous sections. This implicitly indicates the desire in the industry to uncover other approaches. On the other hand, there are some works (e.g. [132][133]) attempt to use HLA as the standardized framework for plugging COTS simulation packages. In this case, the COTS components are HLA federates.

Based on the above, new interoperability standards are definitely needed to interoperate various components that contain heterogeneous models and simulation engines. Interoperability is the main function in most existing simulation middleware [138]. It enables two or more different software systems to interface and use each other services correctly [130][138]. However, developing interoperability standards is a complex process. The involved parities will only support standards if the benefits are worth it and the cost is low. Further, the interoperability problem at the software level is not a trivial task to solve. This is because software components accessed via their API and the exchanged information flow through those APIs. Further, those software components are developed independently by different programmers. Thus, having syntactic and semantic standards that are not tied to implementation issues can help easing such software integration. Note that all existing interoperability technologies, including HLA, CORBA and SOAP-based WS, provide interoperability at the syntactic level, leaving semantics to developers. No distributed simulation semantics standards have been yet finalized. However, developing semantic standards has already started in

different simulation communities such as the DEVS community [140][141][142][143], and the HLA community [132][133].

To this end, distributed simulation is going toward globalizing interoperability with the desire to hide information. This direction also involves putting “anything” into the simulation loop regardless of geographical locations. Of course, semantic standards are needed to interoperate heterogeneous simulators developed by different teams. Further, the desire to have a middleware to be shared locally by different components is certainly there. On the other hand, current distributed simulation interoperability approaches provide a number of challenges that need to be examined and improved.

All existing systems perform distributed simulation according to their specific designed semantics, which are tied to systems internal implementations, as discussed in previous sections. In practice, interoperability is realized at the software level. Thus, coupling existing systems interface and semantics to specific software implementations can introduce challenging interoperability issues. For example, this makes different software APIs extremely difficult to combine, since each was developed with a specific implementation in mind. In practice, this even applies to simulation environments that implement the same formalism as we learned throughout the DEVS standardization process [143]. Therefore, interoperating independent-developed systems that are based on different formalisms or tools would probably multiply those problems. Further, implementation-coupled API and semantics make it more difficult for systems to support multiple algorithms and semantics. For example, a system of type *A* may interoperate with a system of type *B* using certain API while interoperating with a system of type *C* using different API. This could happen for many reasons in the real world when different

teams favor to evolve independently. However, multiple-semantic protocols support becomes extremely difficult when an API is coupled with internal implementation [137]. In reality, when a system exposes different APIs to the external world, these APIs still need to map to the same internal calls. This is because the received requests are still need to be handled by the same internal software. This is not impossible to do, but it gets more difficult to implement every time a new protocol is added.

Further, some existing middleware systems only provide modularity at the modeling level, but not at the simulation engine level. This is because existing middleware approaches do not support local deployment of different types of simulation environments (engines). This completely goes against the modularity concept. In contrast, a new simulation environment should be able to be added to reuse the same local middleware without affecting other types of simulation services.

2.4 Web-services Framework and Standards

Web-services can be defined as the ability to deploy services by a machine and consumed by a different machine via the Web. At present, the two most popular classes are SOAP-based Web-services and RESTful Web-services [115]. Both types aim on reusing existing solutions via the Web. On the other hand, they are different in the way they use to reach this goal, in particular, in the way the services API is designed, orchestrated, and consumed at the software level.

2.4.1 SOAP-based Web-services

The SOAP-based WS (or Big Web Services) [49][108] interoperability model follows the RPC-style approach. In this case, the server exposes services in groups where

each group is deployed in a port. Each service is implemented as an RPC where information is exchanged in form of procedure parameters. Thus, clients need to have those services as procedure stubs compiled with their software.

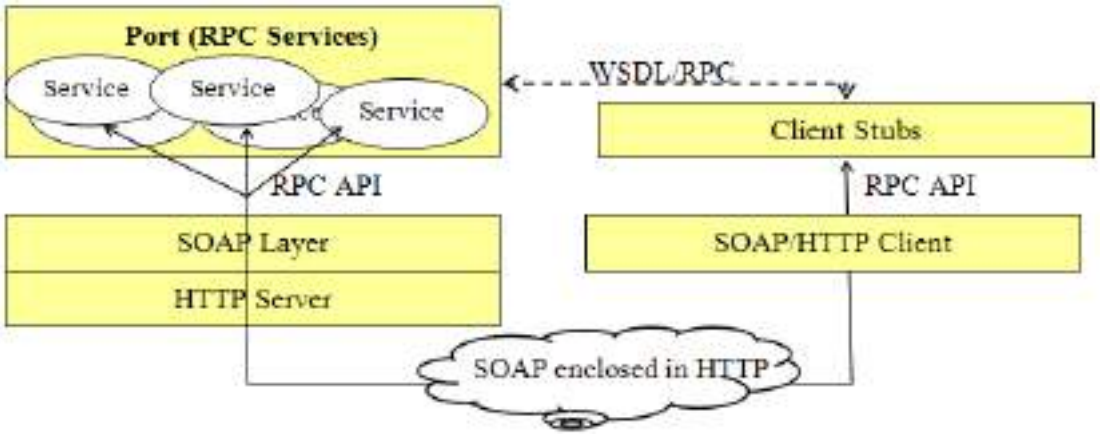

Figure 5: SOAP-based Web Service Interoperability Architecture

At runtime, clients consume a service by invoking its stub as follows (Figure 5):

1. The SOAP engine (e.g. Apache AXIS [145]) converts the RPC into a SOAP message and wraps it in an HTTP request envelope.
2. Acting as an HTTP client, the client sends this HTTP message to the URI of the destination port, using HTTP POST method.
3. Once received by the server, the server extracts the SOAP message and hands it to the SOAP engine.
4. The SOAP engine selects the appropriate port (object) based on the received message URI.
5. The SOAP engine converts the received SOAP message to a procedure call on that port.

6. Once the procedure is completed, the SOAP layer sends results back to the client as a SOAP message wrapped in HTTP response message.

```
<wsdl:message name="stopSimulationRequest">
  <wsdl:part name="in0" type="xsd:int"/>
</wsdl:message>

<wsdl:message name="stopSimulationResponse">
  <wsdl:part name="stopSimulationReturn" type="xsd:boolean"/>
</wsdl:message>

<wsdl:portType name="CDppPortType">
 <wsdl:operation name="stopSimulation" parameterOrder="in0">
  <wsdl:input message="impl:stopSimulationRequest"
             name="stopSimulationRequest"/>
  <wsdl:output message="impl:stopSimulationResponse"
             name="stopSimulationResponse"/>
</wsdl:operation>

</wsdl:portType>

<wsdl:binding name="CDppPortTypeSoapBinding"
                     type="impl:CDppPortType">
  <wsdlsoap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="stopSimulation">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="stopSimulationRequest">
        <wsdlsoap:body encodingStyle="http://.../"
               namespace="http://..." use="encoded"/>
     </wsdl:input>

     <wsdl:output name="stopSimulationResponse">
        <wsdlsoap:body encodingStyle="http://.../"
              namespace="http://..." use="encoded"/>
     </wsdl:output>
   </wsdl:operation>
</wsdl:binding>
```

Figure 6: Excerpt of WSDL Document Example

To demonstrate the role of SOAP and WSDL in an example, suppose that a simulation port exposes a *stopSimulation* service that takes an integer parameter to specify the simulation session number, and returns the operation status. The service

signature is as follows: "`boolean stopSimulation (int in0)`". In this case, the service provider needs to publish this service as a WSDL document so that clients can build their programming stubs (as shown in Figure 6). Tools usually help with constructing the communication skeleton of the client side from a WSDL document. However, a developer needs to write the contents of this stub and compile it with the overall client software. This can get more difficult if WSDL is being converted on the top of previously existing stubs. This is because existing tools treat any slight change of a service as a new one. Thus, developers need to verify new stubs against existing ones.

In our example, the SOAP message shown in Figure 7, describes the *stopSimulation* RPC at runtime. The *stopSimulation* RPC is mapped into lines 6–8 in Figure 7. In this example, Line #6 indicates the port URI that this service belongs to. This enables the destination server to discover the intended port object (e.g. Java/C++ objects). Further, Line #7 indicates that the value of the input parameter (i.e. session number) is 1000. For instance, the service can then be invoked as follows: `status = Port.stopSimulation(1000)`. As a result, the *status* value is then shipped back to the client as a SOAP message wrapped within the HTTP response message, as shown in Figure 7.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <SOAP-ENV:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
4    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
5  <SOAP-ENV:Body>
6    <ns1: stopSimulation xmlns:ns1="http://WS-Port-URI/">
7      <in0 xsi:type="xsd:int">1000</in0>
8    </ns1: stopSimulation>
9  </SOAP-ENV:Body>
10 </SOAP-ENV:Envelope>
```

Figure 7: SOAP Message Request Example

2.4.2 RESTful Web-services

The Representational State Transfer (REST) Web-services [50][115] provide interoperability by imitating the Web architectural style [79] and principles. RESTful Web-services [115] are gaining increased attention with the advent of Web 2.0 [105] and the concept of mashups [71]. A mashup groups various services from different providers and presents them as a bundle in order to provide single integrated service. For example, IBM enterprise mashup solutions [70] aim on integrating Web 2.0 functions as rapid as possible. Those functions usually called "widgets". Nowadays, RESTful Web-services are supported, in conjunction with SOAP-based Web-services, in leading companies' tools such as IBM [70] and Sun Microsystems (e.g. NetBeans IDE [103]). It is worth to note that the WSDL 2.0 standards [41] have now full support for describing RESTful services [95]. This is because WSDL is shifting toward component-based type of services rather than RPC-based service type [112]. WSDL 2.0 is the recommended standard since year 2007 [41].

Figure 8: Resources State Transfer Concept

REST exposes services as "resources" (which are named with unique URIs similar to Web sites) and manipulated with a uniform interface, usually HTTP methods [51]: GET (to read a resource), PUT (to create/update a resource), POST (to append to a resource), and DELETE (to remove a resource). For example, a Web browser makes a

request to a URI via an HTTP GET method to read that URI representation. In return, the URI sends the representation (e.g. HTML) to the Web browser, as shown in Figure 8. It is worth to note that REST applications need to be designed as resource oriented to get the benefits of this approach [115].

The REST architecture separates the software interface from its internal implementation; hence, the services can be exposed while the software internal implementation is hidden from the consumers. In this case, the consumers and the providers need to conform to the service agreement, which comes in the form of messages (e.g., XML). This type of design has the potential to achieve dynamic and improved interoperability. For example, using standardized protocols, a consumer may search, locate, and consume a service at runtime. These protocols are usually XML-based format such as Atom [8] and RSS [116]. For example, the Web 2.0 [105] usually uses such XML-based protocols via REST WS to achieve machine-based interactions on the Web [42]. In contrast, other RPC-style form of interfacing requires a programmer to build the interface stubs and recompile the application software before being able to use. Therefore, the API design matters when connecting diverse software together.

REST has been used in many applications such as IBM enterprise mashup solutions, Yahoo, Google Maps, Flicker, and Amazon S3. It is also used in distributed systems such as National Aeronautics and Space Administration (NASA) SensorWeb (which uses REST to support interoperability across Sensor Web systems that can be used for disaster management) [32]. Another example of using REST to achieve plug-and-play interoperability heterogeneous in sensor and actuator networks was described in [126]. Example of REST usage in Business Process Management (BPM) was described

in [90], which showed different methods and tools to automate, manage, and optimize business processes. REST has also been used for modeling and managing mobile commerce spaces as described in [98].

REST is sometimes confused with HTTP, since REST is normally implemented using HTTP protocol. First, to provide "Web-services" on the Web, you need typically to use the Web major protocol, which is HTTP [24]. This is also the case of SOAP WS. In fact, one can quickly know the need of HTTP in SOAP-based WS when reading the SOAP standards [63]. However, REST is easily implemented using HTTP protocol, because REST is based on the Web principles where HTTP is the major Web protocol. On the other hand, conforming to the HTTP standards does not necessary lead to a RESTful application. For example, SOAP-based WS uses the POST HTTP method to send all SOAP messages wrapped in HTTP envelopes. This is perfectly allowed according to HTTP standards, since HTTP allows overloading the POST method. However, this forbidden in REST because overloading the POST method converts the uniform interface into a heterogeneous interface. In short, REST is not HTTP, but it uses HTTP standards to realize its principles.

2.5 DEVS Formalism

Discrete Event System Specification (DEVS) [153] is M&S specification that is aimed to study discrete event systems. As in any discrete-event simulation, the models change their state only at discrete points in time, upon the occurrence of an event. The models consist of components connected together through external input/output ports where events are exchanged among models.

The Parallel-DEVS (P-DEVS) formalism [38] expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. The basic building block of DEVS models is the *atomic DEVS model*. A P-DEVS atomic model is formally defined as:

$$M = <X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta>$$

Where:

X is the set of input values;

S is the set of states;

Y is the set of output values;

δ_{int}: $S \rightarrow S$ is the internal transition function;

δ_{ext}: $Q \times X^b \rightarrow S$ is the external transition function, where

X^b is a set of bags over elements in X, and $\delta_{ext}(s, e, \varphi) = (s, e)$;

$Q = \{(s, e) \mid s \in S, 0 \leq e \leq ta(s)$ is the total state set;

e is the time elapsed since last transition;

$\delta_{conf:}$ $S \times X^b \rightarrow S$ is the confluent transition function;

λ: $S \rightarrow Y^b$ is the output function;

ta: $S \rightarrow R^{0 \rightarrow \infty}$.

At any given time, an atomic model is in some state $s \in S$. It stays in state s for the time period specified by the state time advance function ta(s). When the atomic model lifetime expires, the model an outputs a value $\lambda(s) \in Y$, and changes its state as indicated by the internal transition function $\delta_{int}(s)$. A P-DEVS model uses bag of inputs (X^b) to exploit parallelism in the system, hence execute multiple concurrent events simultaneously. Nevertheless, the model also changes its state as defined by the external

transition function $\delta_{ext}(s, e, X^b)$, if the atomic model receives one or more external events $x \in X$ before the expiration of ta(s). A confluent transition function (δ_{con}) is used to resolve collisions when receiving external events and internal transitions simultaneously.

The physical system model is created by integrating different DEVS models together through their input and output ports; resulting in a *coupled DEVS model*. A coupled DEVS model consists of atomic and/or other coupled models connected together. A P-DEVS coupled model is formally defined as:

$$N = <X, Y, D, \{M_d \mid d \in D\}, EIC, EOC, IC>$$

Both X and Y define the sets of input and output events respectively. D is an index of the components of a coupled model and, for each $d \in D$, M_d is a basic P-DEVS model (atomic or coupled). The External Input Coupling set (EIC) specifies the connections between external and component inputs, while the External Output Coupling set (EOC) describes the connections between component and external outputs. The connections between the components themselves are defined by the Internal Coupling set (IC). Thanks to the property of closure under coupling, a coupled model can be reduced to a behaviorally equivalent atomic model, and thus be treated as a basic component in construction of more complicated hierarchical models.

Cell-DEVS [139] is an extension to DEVS that defines each cell in a cellular model as an atomic DEVS model. Cell-DEVS describes n-dimensional cell spaces as discrete-event DEVS coupled models, where each cell is represented as a DEVS atomic model. Furthermore, it defines timing constructions rules for each cell, allowing explicit timing delays, asynchronous model execution, and integration with other DEVS models. A Cell-DEVS atomic model is formally defined as:

$$C = <X, Y, I, S, \theta, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D>$$

Each cell has a modular interface (I) that is composed of a number of ports connected to its neighboring cells or to other DEVS models. The future state of a cell is computed by the local transition function (τ) based on the cell's current state and input values. State changes are spread only after a delay given by the delay function (d). Each cell also has the computing apparatus (δ_{int}, δ_{ext}, and λ) as defined in P-DEVS atomic models. Two types of delays can be used: *transport* delays transmit every input received. *Inertial* delays can preempt a scheduled state change if there is a change before the consumption of the delay.

Cells are coupled by the neighborhood relationship to form a cell space, which can then be integrated with other DEVS and Cell-DEVS models. A cell space is formally defined as a Cell-DEVS coupled model:

$$GCC = <X_{list}, Y_{list}, I, X, Y, \eta, \{t_1, \ldots, t_n\}, N, C, B, Z>$$

The cell space (C) consists of a fixed-sized n-dimensional array of cells, and the relative position between each individual cell and its surrounding neighbors is defined by the neighborhood set (N). B specifies the border of the cell space, which can be wrapped (i.e., all cells have exactly the same behavior) or non-wrapped (i.e., the border cells have a different behavior from others in the cell space). The translation function (Z) defines the input/output coupling between the cells.

2.6 CD++

CD++ [139] is an object-oriented modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models. For each DEVS atomic model, users need to

implement the various functions as required by the DEVS formalism in a C++ class, which is then integrated into the modeling hierarchy during compilation. On the other hand, for DEVS coupled models and Cell-DEVS models, users can specify the coupling information as well as other attributes of cell spaces in a model configuration file using a built-in script specification language.

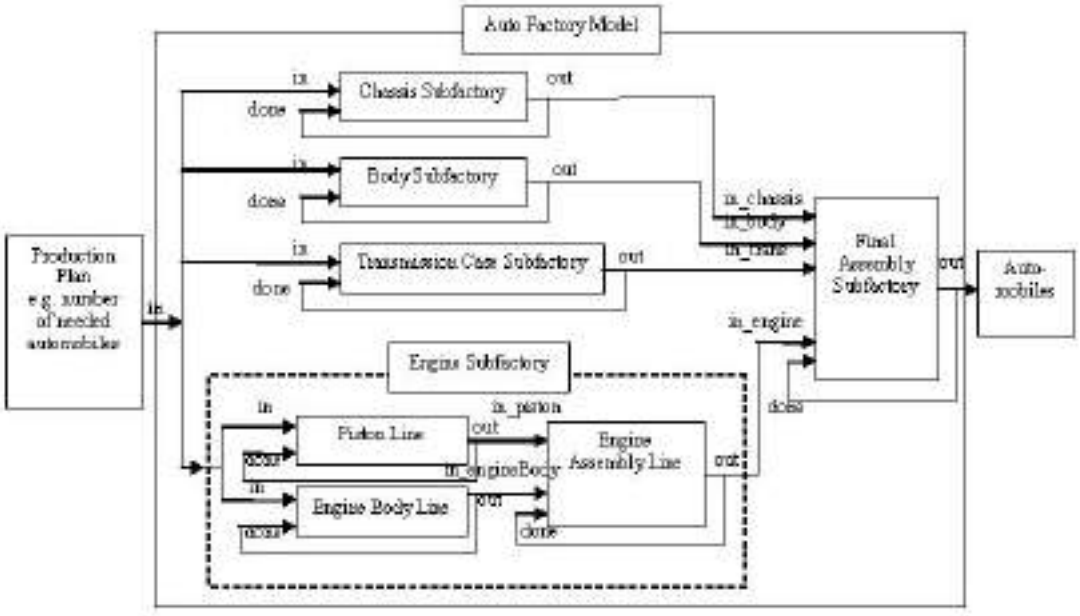

Figure 9: The auto factory coupled model example

Figure 9 shows an example of a DEVS coupled model that represents an auto factory, which coordinates different warehouses and assembly lines to make sure their productivity levels are suitable [137][33]. The factory only manufactures one type of car and each sub-factory manufactures only one type of auto part. Each sub-factory sends its completed component to the *Final Assembly Subfactory* where the automobile is assembled. Four sub-factories devoted to manufacture different parts of a car (*Chassis*, *Body*, *Transmission Case* and *Engine*), and a warehouse devoted to the *Final Assembly Subfactory*. Only one of each component is needed to make an automobile (i.e. 1 *Chassis*

+ 1 *Body* + 1 *Transmission Case* + 1 *Engine*). We need four Pistons and one Engine Body to make an *Engine* (i.e. 4 *Piston Line* + 1 *Engine Body Line* = 1 *Engine Subfactory*).

```
[top]
components : chassis@Chassis body@Body trans@Trans
finalAssem@FinalAssem engineSubFact
out : out
in : in
Link : in in@chassis
Link : in in@body
Link : in in@trans
Link : in in@engineSubFact
Link : out@finalAssem out
Link : out@finalAssem done@finalAssem
Link : out@chassis in_chassis@finalAssem
Link : out@chassis done@chassis
Link : out@body in_body@finalAssem
Link : out@body done@body
Link : out@trans in_trans@finalAssem
Link : out@trans done@trans
Link : out@engineSubFact in_engine@finalAssem

[engineSubFact]
components : piston@Piston engineBody@EngineBody
engineAssem@EngineAssem
out : out
in : in
Link : in in@piston
Link : in in@engineBody
Link : out@piston in_piston@engineAssem
Link : out@piston done@piston
Link : out@engineBody in_engineBody@engineAssem
Link : out@engineBody done@engineBody
Link : out@engineAssem out
Link : out@engineAssem done@engineAssem
```

Figure 10: The auto factory coupled model CD++ definition

Afterward, a CD++ atomic model is implemented (e.g. C++) for each component in Figure 9. For example, the external transition (δ_{ext}) of the atomic model (i.e. this function determines what to do with the incoming parts) is implemented as follows: If a piston is received, it is stocked until the number of pistons needed is available. Once every component's atomic model is implemented, the CD++ coupled model for the

structure is defined, as shown in Figure 10, which describes the structure of Figure 9 in terms of the components names and their links connections.

In CD++, each atomic model is executed by a "Simulator" Processor type, and each coupled model is executed by a "Coordinator" processor type (see Figure 11).

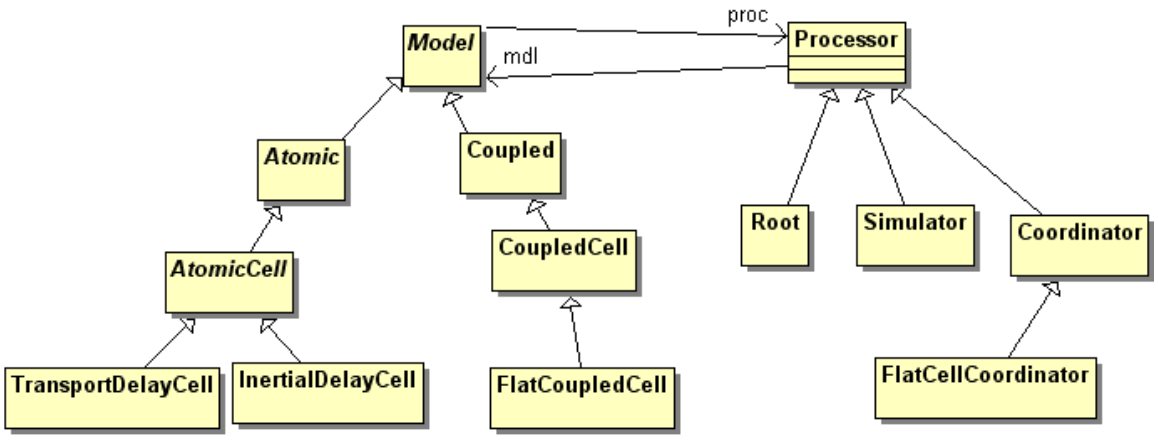

Figure 11: CD++ Model and Processor Hierarchies

The simulation is driven by the *Root* processor, which is responsible for (1) starting and stopping the simulation, (2) connecting the simulator with the environment, and (3) advancing the simulation clock. The *Simulator* class executes an atomic DEVS model by (1) reacting to the different types of received messages via invoking the appropriate function within the *Atomic* DEVS class. The *Coordinator* class manages a DEVS coupled model through routing messages between its children and its *parent-coordinator*. The *CellCoordinator* is responsible for message routing among the cells within a coupled Cell-DEVS model.

CHAPTER 3: THESIS ARGUMENT

In chapter 2, we showed that the way current distributed simulation approaches exchange, structure, and use information, is tied to programming and implementations, exposing systems *heterogeneity*. This path usually leads to the need for homogenizing different implementations, which is usually a complex problem to resolve. We particularly focused on HLA and SOAP-based WS simulations, since this is the state-of-the-art. The HLA performs distributed simulation via interfacing a number of federates into the RTI. This is done in the form of programming functions (called *interactions*); hence, data is exchanged at the federate level via those functions parameters. Thus, the data exchanged between federates is described in programming parameters while the data channels are realized as programming procedures. Further, RTIs themselves are implementation-specific, which makes it difficult to interoperate different vendors' RTIs. At the RTI level, simulation data is usually exchanged as attributes according to an RTI specific implementation. SOAP-based WS simulations group services as procedures in WS ports (addressed by a single URI). Thus, simulation data is exchanged and described in the form of procedure parameters while the data channels are described as procedures. In this case, XML SOAP messages (describing RPCs) are not exchanged at the simulation level, but at the Web service technology layer.

Based on the above state of the art, our work can be compared to other approaches in two ways. The first is comparing it to other distributed simulation interoperability approaches (Table 1). This is important because all existing systems design methods are influenced by the structural rules that they have to follow. The second way is comparing

the proposed approach to other Web-based simulation approaches (Table 2). This is important because we can compare capabilities to other existing solutions. Note that the work presented here provides simulation differently from the HLA standard, but both frameworks are comparable in their interoperability style (Table 1).

Table 1: Comparing RISE Middleware to Current Interoperability Approaches

Approach	Simulation Synchronization Description	Simulation Information Channels	Middleware to Middleware Interoperability	Services Addressing	Details
HLA	Procedure Parameters (Interaction data fields between the RTI and the federates)	Interactions (callback functions between the RTI and the federates)	RTI Implementation specific. RTIs exchange information as regions of attributes (programming variables in objects)	RTI Implementation specific	Section 2.2.1
SOAP WS Approaches	Procedure Parameters	RPCs (each set is grouped in a port)	RPC converted to SOAP over HTTP	URI instance per Port (port contains a set of services/RPCs)	Section 2.2.2
CORBA Approaches	Procedure Parameters	RPCs (each set is grouped in an object)	Parameters marshalling and unmarshalling	CORBA reference per Object	Section 2.1, Section 2.2.3
RISE Middleware (based on RESTful WS)	XML Messages (message-oriented)	Four Uniform Software Channels (HTTP methods)	XML over HTTP	URI template per resource (service type). Instances (URIs) are created at runtime	Chapter 5

Table 1 compares the proposed RISE middleware to current distributed simulation interoperability methods. In this case, existing interoperability approaches are tied to internal software design and implementations. This not only complicates interoperability methods, but also causes dynamicity and scalability issues. Therefore, the RISE middleware approach provides (comparing to existing approaches) a novel approach to decouple interoperability from simulation systems implementations and internal software design issues. This can be classified in the way simulation the synchronization messages are described, in the way simulation messages are exchanged, and in the way simulation services are accessed, structured, and addressed (as shown in Table 1).

Table 2 compares the proposed RISE framework with selected existing Web-based simulation systems. Note that plenty of web-based simulation software is published in the literature, which makes it difficult to list all of them. However, we can safely state that the selected references in the table reflect the most recent Web-based simulation works. Note further that HLA systems with SOAP WS extension are not shown in the table. This is because WS is only used to access the RTI, hence simulation is still of an HLA type, which is covered in Table 1.

Table 2: Comparing Current Web-based Simulation Approaches

Characteristic	RISE Approach	[57]	[44] [53]	[35]	[49][89] [101][137]	[64] [65]
General Middleware (ability to hold different simulation environments)	Yes (Interfaced DCD++ as a proof of concept)	No	No	No	No	No
Simulation Semantic Web Access	Yes	No	Yes	No	No	Yes
Distributed Simulation (several computers performs the same simulation on the Web)	Yes (as in the case of DCD++, but it depends on the simulation environment)	Yes	No (access to a single simulation engine)	No (access to a single simulation engine)	Yes	No (access to single simulation engine)
Distributed Simulation Composition Scalability	Yes (all partitions connected with the same virtual channels)	No	No	No	No	No
Standardized Distributed Simulation Information Channels	Yes (designed as HTTP methods)	Yes (according to OGSI [134])	No	No	No (RPC specific to each system)	No
Distributed Simulation Synchronization Messages Description	XML	Programming Parameters	No	No	Programming Parameters	No
Experimental framework (create different experiments)	Yes	Yes	Yes	Yes	Yes	Yes
Experiments blueprints (life cycle follow certain pattern, allowing the use of workflows)	Yes	No	No	No	No	No

Characteristic	RISE Approach	[57]	[44] [53]	[35]	[49][89] [101][137]	[64] [65]
Experiments Direct Access on the Web (Experiments created and wrapped as URI)	Yes	No	Yes	No	No	No
Experiments named with modelers choice of URIs	Yes (services deployed as URI templates)	No	No	No	No	No
Experiments URIs created at runtime	Yes	No	No	No	No	No
Experiments settings maintained in a database	Yes	Yes	Yes	No	No	Yes
Authentication and Authorization	Yes	Yes	Yes	Yes	No (within a distributed sim. session)	Yes
API XML Description	Yes (WADL; can be done in WSDL 2.0)	Yes (WSDL 1.1)	No	Yes (WSDL 1.1)	Yes (WSDL 1.1)	No
Models representation	Specific to the simulation environment	Specific to system	Specific to system	Specific to system	Specific to system	Specific to system
Visualization	Extendable	No	Yes	Yes	No	Yes
Models repository	Can be interfaced with existing repositories		Yes	Yes	No	No
Workflows	Yes (Can be done at Client side)		No	No	No	No
Interoperable with Web 2.0	Yes (since they apply REST)		Yes	No	No	No

As shown in Table 2, most Web-based simulation efforts focus on the modeling layer. This is mainly by providing Web access to models repositories. These models are usually represented in a text format, mainly as XML (with a visualization support on certain tools). However, existing Web-based distributed simulation are usually specific to a simulation environment. In this case, SOAP-based WS ports are usually connected with specific designed remote procedures. In contrast, the RISE middleware is designed as a general container that can encapsulate different simulation environments. This leaves the

door open for additional extensions. It further, provides more flexible semantic-web experimental frameworks that can be created and named by designers.

In Chapter 2 we also discussed a number of surveys [12][13][124][125] that pointed out that distributed simulation technology is going toward cooperation at globalization level, hiding internal implementations, and packaging functionalities in reusable components. They further pointed out that the middleware needs to overcome certain challenges to meet such objectives. For instance, the interoperability method should not place constraints on synchronization algorithms, interoperability syntax must be based on widely accepted standards, existing simulation solutions should be ready to be combined with Semantic Web, etc. It is worth to note that those studies have recognized the difficulty for resolving the above issues using the HLA or the DIS approaches. As discussed in Chapter 2, those studies state that the middleware interoperability methods should interoperate the simulation components via the Web while hiding internal implementations in the components (without placing any constraints on the synchronization algorithms). This objective matches our goals, discussed in Chapter 1. In addition, we want design scalability (in the way simulation services are externally exposed), composition scalability (in the way distributed simulation partitions connected to each other), a method that allows dynamicity (i.e., systems that join/disjoin the simulation at runtime), and a general experimental framework (that can be created, named and manipulated at runtime).

We initially focused on meeting these objectives using a SOAP-based WS framework (Chapter 4). However, we realized that the WS structural rules place restrictions in to designing the software. For example, we cannot decouple interoperated

systems implementations if the data channels (procedures) and the way data is described (programming parameters) are part of implementation itself. Further, composition scalability is complex if every service (implemented as procedure) at the server side requires a stub on the client side. Although we limited our procedures to a few (implemented in a single WS port), those channels are still embedded (and compiled) with the internal implementation. To reduce exposing the internal implementation, we exchanged and described the simulation synchronization messages as XML messages. In this case, the entire XML message is sent as a single SOAP attachment.

The major lesson learned in the above solution is that the syntactic and structural interoperability rules characterize the level of freedom of a software designer when defining the methods for middleware interoperability. Therefore, the structural and syntactic rules that govern the overall distributed structure have a direct effect on the middleware design methodologies and synchronization algorithms, since those methods need to conform to those rules. Thus, to have freedom for the software design, we need to find the syntactic and structural rules that do not place restrictions on how to design the software while meeting our previously stated objectives. In other words, interoperability methods should not try to answer the “how to implement” question for programmers. To do so, the systems API should be *decoupled* from internal implementation, since this is how software systems interact (that is, how to communicate data and how to synchronize actions). In fact, almost every issue with current approaches can be traced back to some programming and implementation issue around the systems API.

Consequently, in order to meet our previously stated objectives, we first need to solve the problem of hiding systems heterogeneity (that resides in implementation) in

components while allowing composition scalability and dynamicity. The main motivation of solving this problem is that distributed simulation systems are complex and have diverse implementations. Thus, hiding these internal details makes sense toward easing interoperability. As discussed earlier, none of the existing approaches is suitable to solve this problem.

Based on these ideas, we defined, designed and developed the RISE middleware, which is based on RESTful WS principles according to the Web standards (HTTP, URI, and XML). RISE is a general middleware that serves as a container to hold different software components without being specific to any implementation. The RISE middleware is a *resource-oriented* design architecture, which means that all functionalities (resources) have a parent-child relationship. Resources are addressed via URI templates (blueprints). Thus, resources are classes of services whose URI instances can be named and created at runtime. This applies to simulation experiments that exist as URIs which are created at runtime. During distributed simulation, algorithms communicate and synchronize activities by exchanging XML messages between simulation partitions URIs.

RISE has solved the above stated problems as follows: (1) it strictly conforms to the widely accepted Web standards of XML, URI, and HTTP. (2) It hides all software implementations (heterogeneity) in resources, which can be created at runtime. (3) All resources are automatically connected to each other via the same constant uniform data channels. These channels are software virtual channels realized in the exchanged messages themselves. (4) Exchanged simulation messages are described in XML documents.

Thus, RISE solves the problem of hiding systems heterogeneity in components while allowing composition scalability and dynamicity. Resources can be created and named with URIs at runtime. Further, regardless of the number of URIs (resources) in the distributed environment, they will always be automatically connected with the same HTTP methods (channels). Furthermore, because channels are realized outside implementations and resources synchronize their activities in XML messages, resources APIs are then decoupled from implementations. Thus, systems heterogeneity is hidden.

In Chapter 4, we discuss our SOAP-based design methodologies for meeting our objectives. The SOAP WS structural rules proved difficult for completely solving the research problem. For example, the WS ports that contain software implementations cannot be created at runtime. Further, since data channels must be implemented as none-standardized procedures, they must be compiled with internal implementations. Furthermore, each service procedure must have a stub built in the user software, which causes composition scalability difficulty. In Chapter 5, we discuss the RISE design methodologies and their use for meeting our objectives. In Chapter 6, we discuss a proof-of-concept implementation of the CD++ engine into RISE to perform distributed simulations. We show how the simulation URIs are built dynamically, and we discuss the algorithms to synchronize activities via exchanging XML messages between partitions. In Chapter 7, we show that synchronization via XML messages can help performance because of the flexibility it provides, and message aggregation to reduce the number of remote message transmissions. In Chapter 8, we show that distributed synchronization algorithms can be decoupled from software design specifics. In this case, we show algorithms designed as the synchronization rules and the messages that they require to

communicate without dictating the way systems have to implement them in software. We further show in Chapter 8 that the improvement of creating experiments (URIs) dynamically by software can be used in workflows to automate the simulation experimentation process.

CHAPTER 4: SOAP-BASED DESIGN METHODOLOGIES AND ALGORITHMS

As discussed in Chapter 3, the elements of the syntactic and structural rules form the basis of the middleware design methodologies. Those elements define all interoperability aspects such as services access, exchanging and describing messages, and the degree of decoupling systems from each other. Because the design decisions are restricted by those elements, they then define the degree of success of achieving our objectives stated in Chapter 1, such as building a general middleware that can support any simulation environment, decouple systems implementations, distributed simulation composition scalability, etc. As previously mentioned, the research first selected the SOAP-based WS architecture to design new algorithms that mainly aim on interoperating independent-developed simulation systems, as part of the DEVS standardization process [143]. Thus, the design methodologies presented in this chapter conform to the SOAP-based WS structural rules, which mainly expose services as RPCs where the simulation information flows from those procedures in form of programming parameters.

Since DCD++ had already been used for distributed simulation (using SOAP-based WS), we started by making use of this SOAP-based architecture. The original SOAP-based DCD++ [137] places each CD++ instance behind a single WS port (i.e. wrapper component) that handles all client interfaces and synchronization information that flow to/from other CD++ instance components. Each component defines many RPCs where all information flows through those procedure parameters. On the other hand, this interface exposes the internal CD++ implementation. This means that the RPCs

input/output parameters are tied to the way CD++ structures its internal C++ classes and their operations. Thus, using DCD++ specific interface to interoperate with other systems is not practical, since it would require implementation changes in those systems. This is even more complicated since other SOAP-based DEVS systems interfaces are also tied to their internal implementations. To bring those interfaces together and ease interoperability at the software level, this research targeted the two major syntactic and structural elements of the SOAP WS: RPCs and the information that flow through those RPCs. To do so, this research extended the overall SOAP-based DCD++ architecture with a new WS component with few RPCs, and described the synchronization messages in XML (i.e. as SOAP attachments). This interface simplified the synchronization between different systems, and increased their decoupling from each other. In this architecture, DCD++ still uses its original WS component to interoperate various distributed CD++ instances, while using the new designed component to interoperate with other systems (Figure 12).

The rest of the chapter discusses the new WS component (Figure 12) while the original CD++ WS component is discussed in [137].

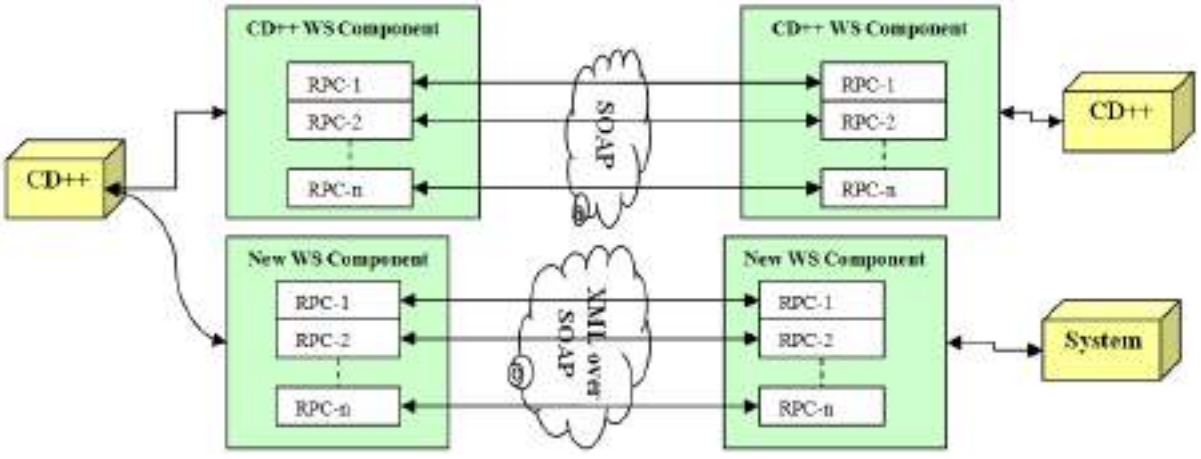

Figure 12: DCD++ SOAP-based Architecture Overview

4.1 Design methodology

The basic design principle employed was to hide systems implementations behind a single WS component with few RPCs, describe all synchronization information in XML, and enclosing all partitions inside one DEVS coupled model within an experimental framework. These are discussed in the following subsections.

4.1.1 Single-Port Wrapper Component

The main purpose of single a coordination entry (called *DEVS wrapper*) is to minimize the number of RPCs and to simplify the coordination between different systems via simplifying those RPCs. The RPCs define all the interface channels and the exchanged information through those channels; hence, they play a major role in decoupling systems implementations. The DEVS-WS Wrapper component (Figure 12) is expected to perform the following tasks:

1. To translate incoming common XML messages from other partitions (domains) to specific simulation messages of a partition (domain) and vice versa, and
2. To route incoming simulation messages to the correct models/ports within its partition.

These RPCs define the operations necessary to setup an experiment and to execute the simulation on that experiment. In this case, the simulation is started by a modeler from the main domain, which synchronizes the simulation on all other support domains. These operations are summarized as follows (Figure 13) [1]: **(1)** *retrieveResultFile* (Line #5) is used to retrieve the simulation result file from a support DEVS domain. **(2)**

startSimulation (Line #6) starts a simulation on a support DEVS domain. In this case, the simulation engine starts and waits for the first simulation message from the main DEVS domain. **(3)** *isSimRunning* (Line #7) checks if simulation is running on a DEVS domain. **(4)** *StopSimulation* (Line #8) stops the simulation normally on a support DEVS domain. **(5)** *setDEVSXML* (Line #9 and #10) sends an XML document to a DEVS domain. This XML document is either a configuration file (see Section 4.1.2) or a simulation message (see Section 4.1.3). **(6)** *deleteSession* (Line #11) deletes a simulation session on a support DEVS domain. **(7)** *createSupportSession* (Line #12) creates a simulation session on a support DEVS domain.

```
1 import javax.activation.DataHandler;
2
3 public interface DEVSWrapperType extends java.rmi.Remote {
4
5   public DataHandler retrieveResultFile(int SupportiveSession);
6   public boolean startSimulation(int SupportiveSession);
7   public boolean isSimRunning(int session);
8   public boolean StopSimulation(int session);
9   public boolean setDEVSXML(int session, String filename,
10              DataHandler file);
11  public boolean deleteSession(int SupportiveSession);
12  public int createSupportiveSession(int MainSession);
13 }
```

Figure 13: DEVS-Wrapper RPCs Services

The DEVS wrapper is the component (i.e. the software layer) that sits between a simulation system and the Web-service layers. Thus, all information is transmitted through SOAP/HTTP layers, wrapped within SOAP and HTTP envelopes, as shown in Figure 14. In this stack, the DEVS wrapper invokes an RPC via the SOAP engine layer (e.g. AXIS [145]), which in turn converts the call into a SOAP message and passes it to the HTTP server (e.g. Tomcat [7]), which wraps it in an HTTP envelope and sends it as

an HTTP message request. These software components are often available commercially or as freeware.

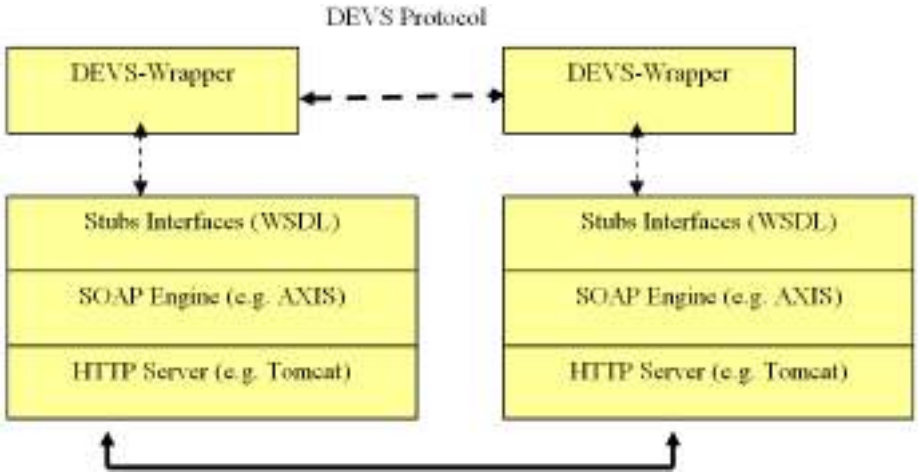

Figure 14: Connecting Domains using SOAP-based Web-Services

4.1.2 XML Message Synchronization

Our approach here is to describe the simulation synchronization information as XML messages instead of programming parameters. This is different from all existing SOAP-based simulation systems (see Section 2.2.2). Describing simulation messages as XML increases the level of decoupling system implementations from the synchronization algorithms and from each other. Making changes to an XML document is much more flexible than changing programming parameters to procedures. To do so, the actual XML message needs to be sent as a parameter to a procedure, since the interface is still RPC-style. This means that the XML message needs to be sent as a file embedded in the SOAP message; this is usually called a *SOAP attachment*. The operation *setDEVSXML* (Lines #9-10 in Figure 13) sends all XML messages as attachments. As shown in Figure 14, the actual work is performed by the SOAP engine layer. Therefore, the operation depends on that engine's capability of supporting such feature. In most cases, there are commercial or

open source SOAP engines that implement the SOAP standards available. Thus, all existing SOAP-based systems make a use of such available components, which reduce development time. However, the simulation systems become restricted to those components limitations (such as the type of programming parameters they support and their implemented programming languages). For example, since the WS components available are often implemented in Java, non-Java systems need to find a solution to interface the SOAP engine Java code with their programming language. In our case, we used the Java-based AXIS SOAP-engine [145] because it supports sending XML documents as SOAP attachments, and the Java Native Interface (JNI) [92] to interface Java and C++ programs.

Table 3: Message Elements in XML Simulation Message

Element	Format	Allowed Values	Comments
MessageType	Character	I, @, D, X, Y, *	I = INIT, @ = Collect, D = Done, X = External, Y = Output, * = Internal.
Time	String HH:MM:SS:MS	Numbers separated by colon (":")	Example: 08:50:00:00
SrcModel	String	Known Model Name	Source Model
DestModel	String	Known Model Name	Destination Model
Port	String	Known Port Name	Destination Port
Value	C++/Java double	N/A	Mandatory only for X and Y messages
NextChange	See Time element	See Time element	Next Change Time(mandatory only for D messages)
IsFromProxy	Java boolean	True or False	Mandatory for D messages if Head/ Proxy Algorithm is used. Allows the Head to synchronize its Proxies.

Table 3 shows all fields in a simulation message (the XML description of this information is shown in Figure 15). These fields define all the information needed to process a simulation event such as the message type, event time, source model, source port, destination model, destination port, next time change (used to calculate next internal event), and event value. There are six types of simulation messages (Table 3) for

synchronizing the entire simulation and exchanging simulation events (the simulation synchronization will be discussed shortly in Section 4.1.3):

- The *Init* (I) message is sent at the start of the simulation to initialize all models in the DEVS hierarchy.
- The *Collect* (@) message is used to collect all models output events in the collection phase.
- The *Internal* (*) message is used to execute collected message and any internal events.
- The *Done* (D) message is used to advance simulation from a phase to another.
- The *External* (X) message is used to send to input event to a model.
- The *Output* (Y) message is used to generate an event by a model.

```
<Message ver="1.0">
   <MessageType>I</MessageType>
   <Time>00:00:00:00</Time>
   < SrcModel>Coupled0</SrcModel>
   < DestModel>Coupled2</DestModel>
   <Port>IN</Port>
   <Value>1.0</Value>
   <NextChange>00:00:00:00</NextChange>
   <IsFromProxy>false</IsFromProxy>
</Message>
```

Figure 15: XML Simulation Message Example

4.1.3 Experimental framework

To perform a simulation session, an experimental framework needs to be setup. This means that the model partitions need to be placed on different simulation environments (typically, each is placed on a machine) where each simulation engine knows how the partitions ports are connected to each other.

The main principle was to enclose all various DEVS-domain heterogeneous models within a single coupled model. This simplifies the overall simulation since these models can be heterogeneous, hence expected to be simulated by different simulation engines. For example, in Figure 16, coupled 1 and coupled 2 can belong to different simulation environments. In this scheme, both of these models are then enclosed in Coupled 0. In this case, the main DEVS domain will own the Root coordinator and will be in charge of coupled 0, which encloses both heterogeneous models, giving the impression of simulating a single homogeneous DEVS model. As shown in Figure 16, both coupled models are interfaced without worrying about how each simulation engine does the simulation internally. Therefore, the coupled models are viewed as black boxes with input/output ports.

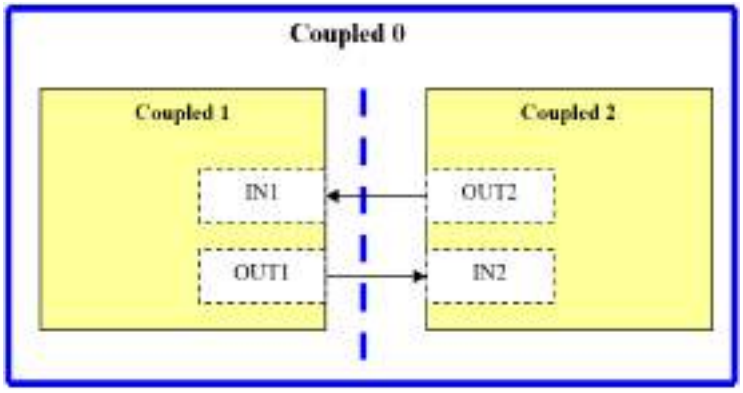

Figure 16: Coupled model partitioned across DEVS Domains

The structure shown in Figure 16 can be described in XML as in the document shown in Figure 17: Line #1 defines the XML document version. Lines 2-4 define the synchronization algorithm used (in this case, head/proxy). Lines 5-47 define all models partitions. In this example, there are three models: Lines 6-27, Lines 28-36, and Lines 36-47 define the first, second and third model respectively. For example, the first model is defined as follows: line #6 indicates the model type (a coupled model), Line 7 indicates

the name of the model (Coupled0), and Lines 8-10 define the model internal components (two coupled models components, Coupled1 and Coupled2). Line #12 defines the Web-service port URI that wraps the Coupled0 model. Lines 13-25 define the ports connections of all internal components. For example, Lines 14-23 defines the connection *from* port OUT1 port of Coupled1 *to* port IN2 of Coupled2. Further, from this XML document (Figure 17), the main machine can identify the participant support domains. This document originally comes from the modeler to the main DEVS domain, which in turn passes it to other domains via invoking the method *setDEVSXML*.

```
1 <MODEL_STRUCTURE ver="1.0">
2  <COUPLED_SYNC>
3     <scheme ver="1.0">HeadProxy</scheme>
4  </COUPLED_SYNC>
5  <Models>
6    <Model Type="Coupled">
7      <Name> Coupled0 </Name>
8        <Components>
9         <Name Type="Coupled">Coupled1</Name>
10        <Name Type="Coupled">Coupled2</Name>
11       </Components>
12     <URI>http://… </URI>
13     <LINKS>
14        <LINK>
15          <FROM>
16            <Component>Coupled1</Component>
17           <Port>OUT1</Port>
18          </FROM>
19          <TO>
20            <Component>Coupled2</Component>
21           <Port>IN2</Port>
22          </TO>
23        </LINK>
24        …
25     </LINKS>
26     …
27   </Model>
28   <Model Type="Coupled">
29     <Name> Coupled1 </Name>
30     <Ports>
31       <Port Type="in">IN1</Port>
32       <Port Type="out">OUT1</Port>
33     </Ports>
34     <URI>http://… </URI>
```

```
35      …
36    </Model>
37    <Model Type="Coupled">
38      <Name> Coupled2 </Name>
39      <Ports>
40        <Port Type="in">IN2</Port>
41        <Port Type="out">OUT2</Port>
42      </Ports>
43      <URI>http://… </URI>
44      …
45    </Model>
46
47  </Models>
48      …
49</MODEL_STRUCTURE>
```

Figure 17: XML Model Structure Document Example

Once all models are partitioned and placed in their proper locations, the simulation can then be started. The modeler software starts the simulation via the operation *startSimulation* on the main DEVS domain, which in turn opens a session with all the relevant support domains (using the RPC *createSupportSession*). Once the main domain opens and collects the session numbers from all support domains, it broadcasts this information to the support domains in one XML document (using the RPC *setDEVSXML*), as shown in Figure 18. The simulation session document (Figure 18) contains the main domain session number, and the support URIs paired with their session numbers as follows: Line #1 specifies the message version. Lines 2-5 defines the first partition session as follows: Line #2 indicates that this session belongs to the main partition. Line #3 specifies the session number on the main machine. Lines #4 specifies the main partition Web-service port URI. Similarly, Lines 6-9 define the second partition session. However, this session belongs to a support machine.

```
1 <Sessions ver="1.0">
2    <Session Type="Main">
3       <Number>123</Number>
4       <URI>http://…</URI>
5    </Session>
```

```
6     <Session Type="Supportive">
7        <Number>1000</Number>
8        <URI>http://…</URI>
9     </Session>
10     …
11 </Sessions>
```

Figure 18: Domain-Simulation Sessions XML Binding Document Example

After this XML message (Figure 18) is received by all domains, each partition structures the models coordinators as shown in Figure 19, where each partition IS able to send/receive XML synchronization messages to/from other partitions.

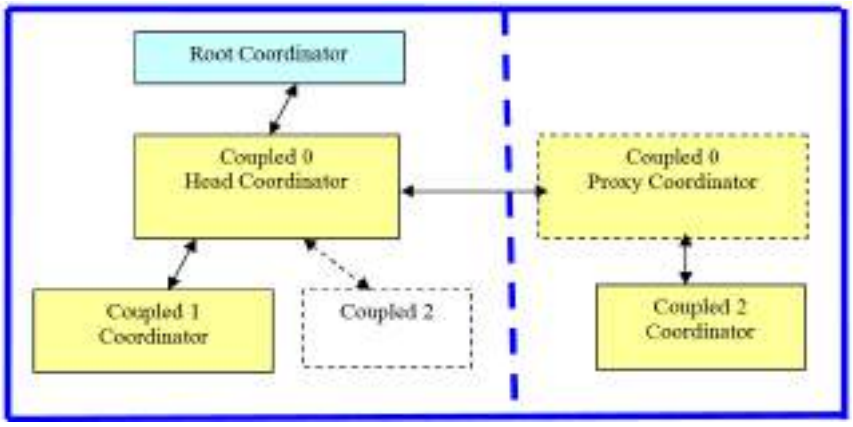

Figure 19: Coupled #0 Split between two DEVS Domains

The coordinators structure in Figure 19 represents the model partitions shown in Figure 16. Only one main DEVS domain will be in charge of driving the overall simulation. This domain creates and owns the *Root* coordinator while other DEVS domains become support and only react to messages from the main or support domains. The main domain is the one selected by the user to initialize and start the simulation session. The simulation engine uses a called Head/Proxy [1] architecture, which aims on reducing remote messages via routing them locally on the same partition, if possible. The DEVS coordinator concept is extended in two ways: **(1)** the Head Coordinator is in charge of simulating the entire coupled model. It coordinates the internal models that

exist in its domain and other remote models (via Proxy Coordinators). **(2)** The Proxy Coordinator acts as an agent on behalf of the Head Coordinator to simulate the internal sub-models of a coupled model that exist in its domain.

The coordinators in Figure 19 advance the simulation in phases via exchanging the XML simulation messages between each other (see Section 4.1.2). The first phase (initialization) only is triggered at the beginning of the simulation while the other two (collection and transition) simulate simultaneous events at a specific simulation time. These phases are based on the simulation phases of the CD++ engine [139] and they operate as in the following:

- *Initialization*: it starts when the topmost coupled model receives an *Init* (I) message. This message propagates in the model hierarchy until it executes every initialization method of every atomic model. In response, a *Done* (D) message propagates upwards in the model hierarchy, where each coordinator calculates the minimum next change of its children and passes it in a D message to its parent. Once all D messages propagate up to the top coupled model, the one with smallest time passes to the Root Coordinator, which updates the simulation clock and starts the Collection phase.
- *Collection*: The Root Coordinator sends a *Collect* (@) message to the top coupled model, which, in turn, passes it to all of its children. In this phase, all the output functions are triggered and *Output* (Y) messages may be passed by internal coordinators to their destination as *External* (X) messages (inserted in message bags). This phase ends when the Root Coordinator receives a DONE message from the top model.

- *Transition*: The Root Coordinator sends an *Internal* message (*) to the top coupled model, which in turn, passes it to all of its children. All the external messages collected in the message bags are passed downward in the model hierarchy. Once an atomic model is reached, the appropriate atomic operations are executed by its simulator, based on an *internal* event or *external* messages.

The coordinators exchange information and synchronize the above simulation phases by sending simulation messages to each other. In this case, all messages in a partition are queued in the same simulation event list to be processed. Processing those events means forwarding them to their intended coordinators where, eventually, they are executed. Therefore, there are two types of messages: **(1)** messages with known destinations; they are heading to coordinators within this partition. In this case, it is up to this domain engine to forward those messages. **(2)** Messages with unknown destinations; they are heading to coordinators belonging to other domains partitions. In this case, the messages are forwarded to the Web-service component, which converts them to XML messages (Section 4.1.2) and transmit them to the appropriate WS components of the other remote partitions. A possible algorithm to implement of such mechanism is shown in Figure 20.

```
While (simulation is running)
   If (unprocessed messages exist in queue) {
    Get first message from queue;
    If (message belongs to my DEVS domain)
       // Destination is either Root
       // Coordinator, coupled Coordinator
       // or atomic simulator
       Send message to its Destination;
    Else   // going to another DEVS domain
       Send message to my DEVS-Wrapper port;
  }
```

Figure 20: Algorithm for Simulation Message Processing

Based on the simulation loop shown in Figure 20, the Root Coordinator (which exists only in the main DEVS domain) receives simulation messages like any other coordinator. The main function of the Root Coordinator is performed when it receives the D message, as follows: **(1)** It advances the simulation clock (carried by the D message), (**2)** It starts the collection/transition phase or **(3)** It stops the simulation. The Root Coordinator will receive its first D message to indicate the end of the initialization phase. Figure 21 shows a sample algorithm showing a possible implementation for the Root Coordinator.

```
Root Coordinator::ReceiveDoneMessage () {
If (Next Phase == Transition)  {
      // Start transition phase
       Next Phase = Collect;
       Send Internal Msg to highest model;
} Else if (next Time <= STOP_TIME) {
       Send Stop to all;
} Else {
      While (envExternal == NextEventTime){
           Send environment external event;
       }
       If (Next Event is NOT external) {
           // Start the Collect Phase
           Next Phase = Transition;
           Send Collect Msg to highest model;
       } Else {   // Start transition phase
           Next Phase = Collect;
           Send Internal Msg to top model;
       }
   }
} // end root
```

Figure 21: Done Message Processing by Root Coordinator

4.2 Web-Service Component Implementation

The Web Service components are implemented in Java. As seen in Figure 22, they fall into the following categories:

1. Web Service wrapper (*JavaWrapper* class in Figure 22): this is the backbone of the web service components since and the actual management of a simulation session within the AXIS SOAP-engine [145] for a single session. For example, the client side makes its stubs calls via the *DEVSPortTypeSoapBindingStub* class. However, once received at the server side, the message is handled by the class *DEVSPortTypeSoapBindingImpl*. The message is then forwarded to the appropriate operation in the JavaWrapper class to fulfill the request.
2. SOAP engine Interface (*DEVSPortType* interface in Figure 22): the interface operations in the stubs on the client side are implemented in the class *DEVSPortTypeSoapBindingStub*, while the server services implementation are implemented in the class *DEVSPortTypeSoapBindingImpl*.

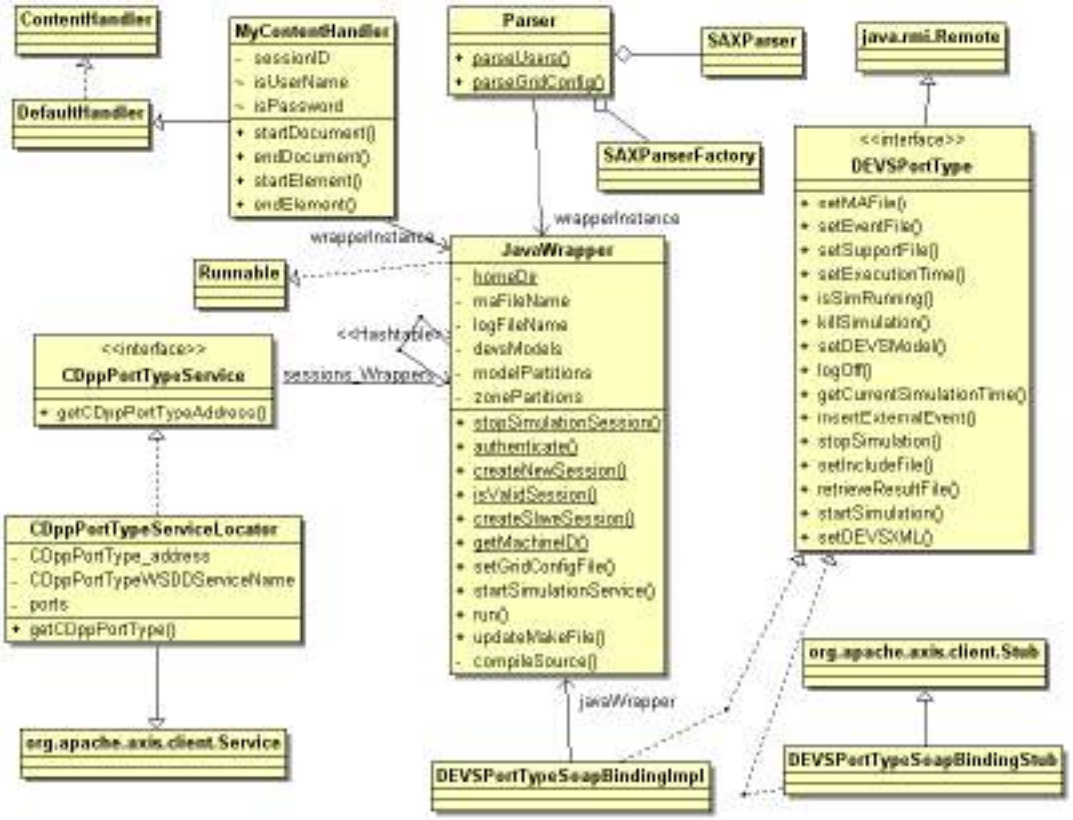

Figure 22: Web-Service Component Design

The Web service operations are provided by the *DEVSPortType* interface in Figure 22. We have previously described the major operations when discussed Figure 13 (in Section 4.1.1): *retrieveResultFile, startSimulation, isSimRunning, StopSimulation, setDEVSXML deleteSession,* and *createSupportSession*. Further, there are more services mainly provided to setup the CD++ simulation, summarized as follows:

- *Login /logout*: to log current user in/out.
- *setMAFile*: to set CD++ model definition file (.ma).
- *setDEVSModel*: to set DEVS model (C++ header and implementation files).
- *setEventFile*: to define the external events file (.ev).
- *setExecutionTime*: to choose maximum execution time of the simulation.

4.3 SOAP-based Approach Design Challenges

The approach presented in the previous section, which is based on the SOAP-based WS structural rules, showed a number of difficulties in achieving our objectives of developing an all-purpose simulation middleware as previously defined in Chapter 1. Examples of the presented approach difficulties are listed below:

- Decoupling heterogeneous independent-developed implementations is enhanced via describing synchronization information in XML messages. However, these XML messages still need to be passed as file parameters via RPCs (i.e., as SOAP attachments). Thus, the systems are still linked via those RPCs that need to be compiled with systems software. Nonetheless, the proposed approach uses a single component with few RPCs to simplify the interface. In spite of this, other systems

may find difficulties mapping those RPCs to their internal implementations as we did with the DCD++ system in this chapter.

- The main purpose of adding a second WS component in the presented architecture is to allow DCD++ to support new synchronization protocol in addition to its original. To do so, we finished up with a complete new WS component, re-implementing the Web-service definition. This means injecting new software code to an existing stable system, but we had a little choice since the original interface is specific to the DCD++ implementation. Based on this, adding new simulation services to reuse the same local middleware becomes even more complicated, which makes it more difficult to achieve a general middleware that can support different simulation services.
- Composition scalability is a desired capability in distributed simulation. However, a system implementation needs to have a procedure stub with every unique remote service. This is because RPCs are the channels that systems use to exchange information, which need to be constructed at all systems (even at compiling time).
- To allow multiple distributed experiments to execute simultaneously, each simulation partition is identified by a session number. This is because each machine always has the same Web-service port (i.e. addressed by a URI instance) for all experiments. Thus, based on a specific session number, the incoming simulation messages are routed to the appropriate simulation partition. This makes all the experiments on a machine to be addressed by the same URI instance, which makes it difficult to design a Web semantic for each experiment since they do not have unique URIs (e.g. an experiment cannot be reached by a Web browser because it is not directly attached to the Web). Further, it is difficult to provide experiment blueprint patterns where

multiple experiment instances may be created with their own URI instances at runtime. This is because the Web-service port is addressed by a single URI instance rather than a URI template where multiple URI instances of that template can be created at runtime.

The above examples of such difficulties are mainly because that the SOAP-based structural and syntactic rules limit designers to interface systems with RPC style and pass information as parameters. The designers can then reduce programming dependency and hide the implementation to some extent, as we did in this chapter. However, this may not fit well with other systems that are not under their control. In practice, interoperability at the software level is extremely difficult to achieve unless systems are completely decoupled from each other. Otherwise, they would be homogenizing their systems implementations. This means to achieve interoperability between simulation systems, their interfaces must be homogenized (in advance) to avoid major implementation changes later in the integration process. Thus, various parties need to agree on some implementation issues, which can be slow and difficult process. However, most systems are usually developed independently, which become more complex to integrate their interfaces after they have been developed.

4.4 Chapter Summary

In this chapter, we showed the structural and syntactic rules of the SOAP-based WS architecture to design new algorithms that mainly aim on interoperating independent-developed simulation systems, as part of DEVS standardization process. The SOAP-

based DCD++ introduced here places each CD++ instance behind a single WS port (i.e., a wrapper component) that handles all client interface and synchronizations information flow with other CD++ instances components. Each component defines many RPCs where all information flows through those procedure parameters. This interface heavily exposes the internal CD++ implementation as in the case of other SOAP-based systems (see Section 2.2.2). Thus, using DCD++ specific interface to interoperate with other systems is not practical since it would require major implementation changes in those systems.

We extended the overall architecture of the SOAP-based DCD++ with a new WS component with fewer RPCs, and described synchronization messages in XML. This simplified the interface, and increased decoupling interoperating systems implementations with the purpose of reducing changes to legacy systems implementations. In the new proposed architecture, DCD++ still has its original WS component to interoperate various CD++ instances while has the new component to interoperate with other systems.

This new WS component provides the RPCs interface of each domain. To enable the WS component to handle multiple experiments at the same time, the WS component assigns a session number to each experiment partition (which resides on its machine). Before the simulation takes place, the modeler needs to setup the experiment. This is mainly done via partitioning the entire model over the participant machines. In this case, all partitions are viewed as coupled models wrapped within a single distributed DEVS coupled model. This configuration (e.g. models ports connections) is described in an XML document, allowing models repartition if needed. During simulation, all the synchronization messages are exchanged as XML instead of procedure parameters

between WS components. The XML messages are sent as file parameters via the SOAP engine (this mechanism usually called a *SOAP attachment*). However, the ability to do so depends on the SOAP engine used. Upon XML messages arrival, the WS component at the destination routes the information received to the CD++ engine associated with that experiment partition. The CD++ engine advances the simulation in three phases: initialization (to initialize all models), collection (to collect models output events), and transition (to execute simultaneous collected and internal events). During these phases, the messages heading to remote partitions are routed via the WS wrapper component of the local partition (as XML messages).

The SOAP-based approach showed a number of difficulties in achieving a number of objectives such as fully decoupling heterogeneous implementations, distributed simulation composition scalability, experiment blueprints, easing supporting new synchronization protocols, etc. This is mainly because SOAP-based structural and syntactic rules limit designers to interface systems with RPC style and pass information as parameters. The designers can then reduce programming dependency and hide implementation to some extent (as we did in this chapter), however this may not fit well with other systems that not under their control. In practice, interoperability at the software level is extremely difficult unless systems are completely decoupled from each other (otherwise, they would be homogenizing their systems implementations). Further, a WS component at a machine is addressed by a single URI instance, which is the only address to reach all experiments on that machine. This makes it difficult to design experiment blueprints, allowing experiments instances to be created with their own URIs at runtime (i.e. attached directly to the Web).

CHAPTER 5: RISE MIDDLEWARE DESIGN METHODOLOGIES

The SOAP-based WS structural and syntactic rules (described in Section 2.4.1) provided the foundations of our applied design methodologies presented in Chapter 4. Nevertheless, the structural rules in Chapter 4 presented some difficulties in achieving some of our objectives (stated in Chapter 1) such as decoupling heterogeneous implementations, composition scalability in distributed simulation, etc. In this chapter, we discuss a different method that can be used to achieve our goals: the RESTful Interoperability Simulation Environment (RISE) middleware. The idea is to provide a design methodology that uses RESTful WS structural rules (described in Section 2.4.2). We believe that these rules allow us more freedom with making better design decisions in order to achieve our objectives (stated in Chapter 1). This is because as argued in Chapter 3 that the syntactic and structural rules that govern the overall distributed structure have a direct effect on the middleware design methodologies and synchronization algorithms, since those methods need to conform to those rules.

Those syntactic and structural rules elements clearly distinguish the RISE approach from all other existing approaches (previously discussed in Chapter 2). These differences can be classified in the way simulation the synchronization messages are described, in the way simulation messages are exchanged, and in the way simulation services are accessed, structured, and addressed (see Table 1 in Chapter 3). In this case, the RISE approach hides all simulation services in resources that are named by URI templates. Thus, resources themselves become services templates (i.e. types) whose

instances (URIs) can be created at runtime. The ability to create services of any type on the middleware and name them by URIs chosen by users' at runtime is an important element for achieving a flexible and generic middleware. Further, these resources are automatically connected upon creation to each other via constant uniform virtual software channels to exchange all information (in our case, HTTP methods), which is described in XML messages.

The RISE structural rules form the basis of the design methodologies discussed in this chapter. RISE uses the RESTful WS structural rules to spread services over a number of resources (resource-oriented), and resources exchange synchronization information in form of XML messages (message-oriented) via predefined uniform channels (uniform interface). Each of these concepts is discussed in the next sections (while the RISE middleware implementation is presented in Appendix-A).

5.1 Resource-Oriented Architecture

One of RISE design objectives is to allow different simulation services types to share the same local middleware, allowing modelers to setup different experiments based on different selections via the same server. Thus, we want to design a general middleware that can support different simulation services simultaneously. For example, Chapter 6 will present a proof-of-concept implementation of DCD++ simulation services via RISE. However, additional services (besides DCD++ services) could be supported by the same server on the same machine. The next subsections focus on achieving such general middleware, as follows:

- Section 5.1.1: it discusses how resources are organized in a hierarchical structure. These resources are exposed as URI templates whose instances can be created at runtime. This leads to a concept of general layered interoperability where different simulation resources (URIs) organized at a separate layer above the middleware.
- Section 5.1.2: because resources instances (URIs) are a part of experiment instances, this allows RISE to provide experiments blueprints, which can be used in applications such as workflows (a concept to be discussed in Chapter 8).
- Section 5.1.3: one of the RISE objectives is to maintain experiment instances unless deleted by authorized users (Chapter 1). This section discusses the design of the database that maintains those experiments resources (URIs).

5.1.1 Resources Hierarchical Design

A resource on the Web is conceptually intended to capture a target of a hypertext reference [50]. In this case, a resource is named with a URI and can be used to find other resources, similar to typical Web browsers hyper links. This concept is applied in RISE, but with one difference: resources are types whose instances are created at runtime. To do this, RISE applies the concept of URI templates to deploy resources types. URI Templates [62] are URIs with variables (placed between braces '{}') which can be substituted with the appropriate values to obtain the actual URI instances. For example, *"username"* in URI template *"users/{username}"* can be substituted with any string to obtain the actual URI instance such as *"users/Bob"*. URI templates lead to a general middleware organization, since URIs can be created and named at runtime to wrap concrete services (Figure 23).

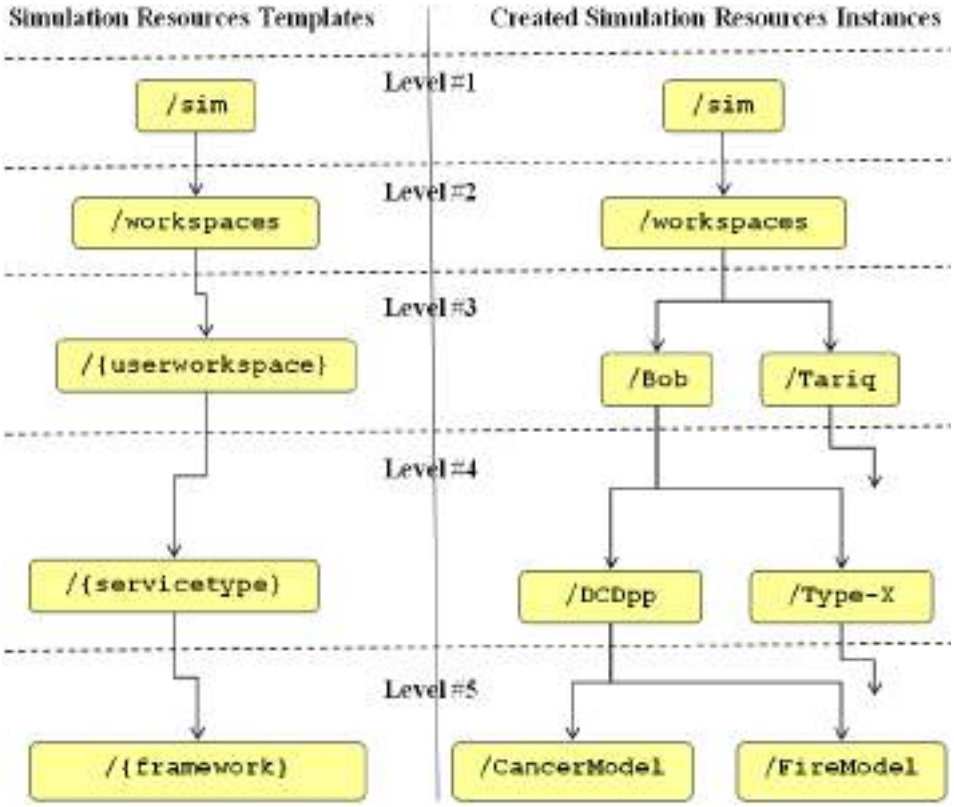

Figure 23: Excerpt of RISE Resources Templates

Figure 23 shows how resources are organized in a hierarchical structure, where multiple instances of each template may be created simultaneously (the API is described in Appendix-B). For example, the *"{userworkspace}"* template at Level #3 allows any number of clients' workspaces to be created, separating modelers' experiments from each other. Level #4 allows each client workspace to select a simulation service type. For instance, setting the *"{servicetype}"* template to *"DCDpp"* selects the DCD++ simulation environment. This allows modelers to create experiments based on different environments. It also allows middleware developers to add additional services types without affecting other existing services types.

Level #5 indicates that the modelers may create any number of experiment frameworks with any environment type (the experiment blueprint is shortly discussed in

Section 5.1.2). As shown in Figure 23, these resources can be used to find each other similar to browsing a Web site. For example, the *"{servicetype}"* template (at Level #5) does not only hold a simulation service type, but also serves as a structural resource for its children. For instance, typing URI *".../Bob/DCDpp/"* in a Web browser address bar returns all of Bob's DCD++-based experiments URIs.

The concept of providing services as general resources led to layered interoperability, which includes the *Middleware Layer*, the *Simulation Layer*, and the *Modeling Layer* (Figure 24). The Middleware Layer provides all means to exchange all information. The Simulation Layer deploys different simulation environment types (e.g. DCD++) where the Modeling Layer operates on the top of a simulation environment. In this case, the simulation environment (e.g., DCD++ described in Chapter 6) provides all simulation management such as time management while the Modeling Layer is specific to each simulation environment type.

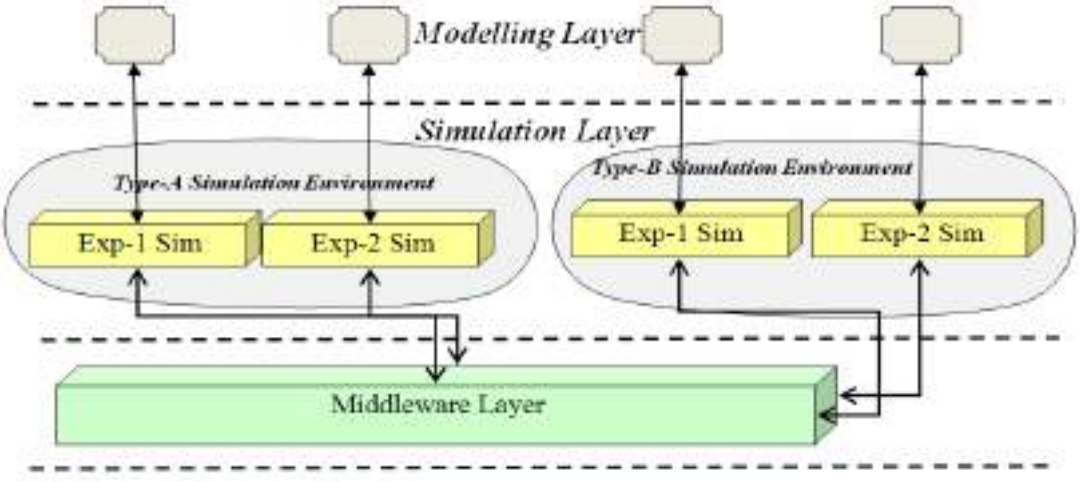

Figure 24: RISE Middleware General Simulation Container

Using this architecture, any number of experiments of any type may be conducted at the simulation layer. For example, Figure 24 shows experiments with two *simulation*

environments: *Type-A* and *Type-B*. In this example, the URI *<User-1/Type-A/Exp-1>* corresponds to the experiment *"Exp-1"* of simulation system *"Type-A"* (which is owned by modeler *"User-1"*). Likewise, URI *<User-1/Type-B/Exp-1>* corresponds to the experiment *"Exp-1"* of simulation system *"Type-B"* (owned by modeler *"User-1"*).

5.1.2 Simulation Experiment Blueprint

As discussed in the previous section, the simulation resources (i.e., the second layer in Figure 24) are typically part of experiments instances. Because the resources are defined as templates, this makes experiments *blueprints* whose instances can be created for any user of any service type. The experiment blueprint is shown in Figure 25.

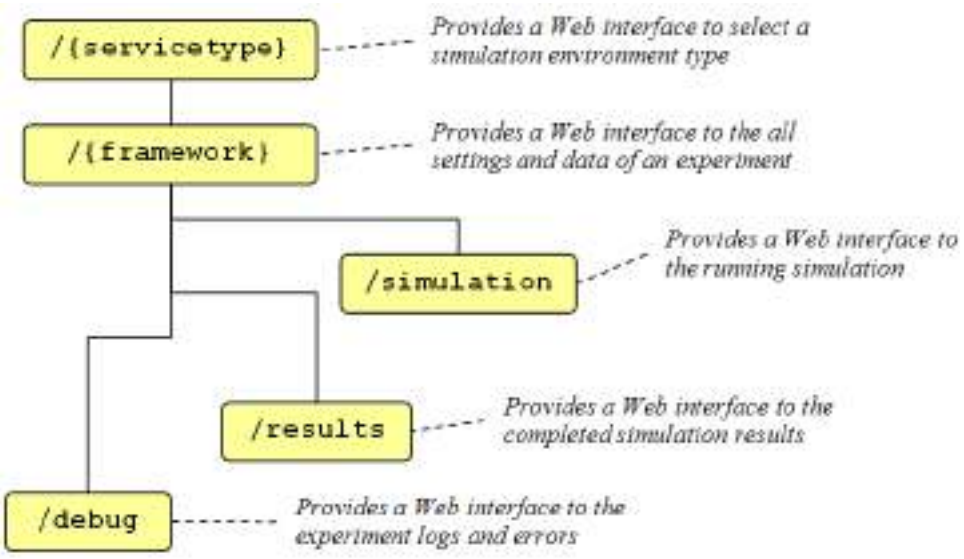

Figure 25: Simulation Experiment Resources (URIs)

The main resource available is the *"{framework}"* template, which holds the experiment name, serves as the parent resource for other resources in the experiment, and it is used to interface and manipulate the experiment setup. The children resources provide a Web interface for the experiment, depending on the experiment states. These

resources can exist at certain states and disappear at others. In this case, the resource *"{framework}/simulation"* is used to wrap an active simulation, and used to manipulate the active simulation in progress. Note that this resource is used to communicate synchronization XML messages between the various distributed simulation partitions. The resource *"{framework}/results"* is used to store simulation results once a simulation is completed. The resource *"{framework}/debug"* is used to store any faults related information regard the subject model under simulation.

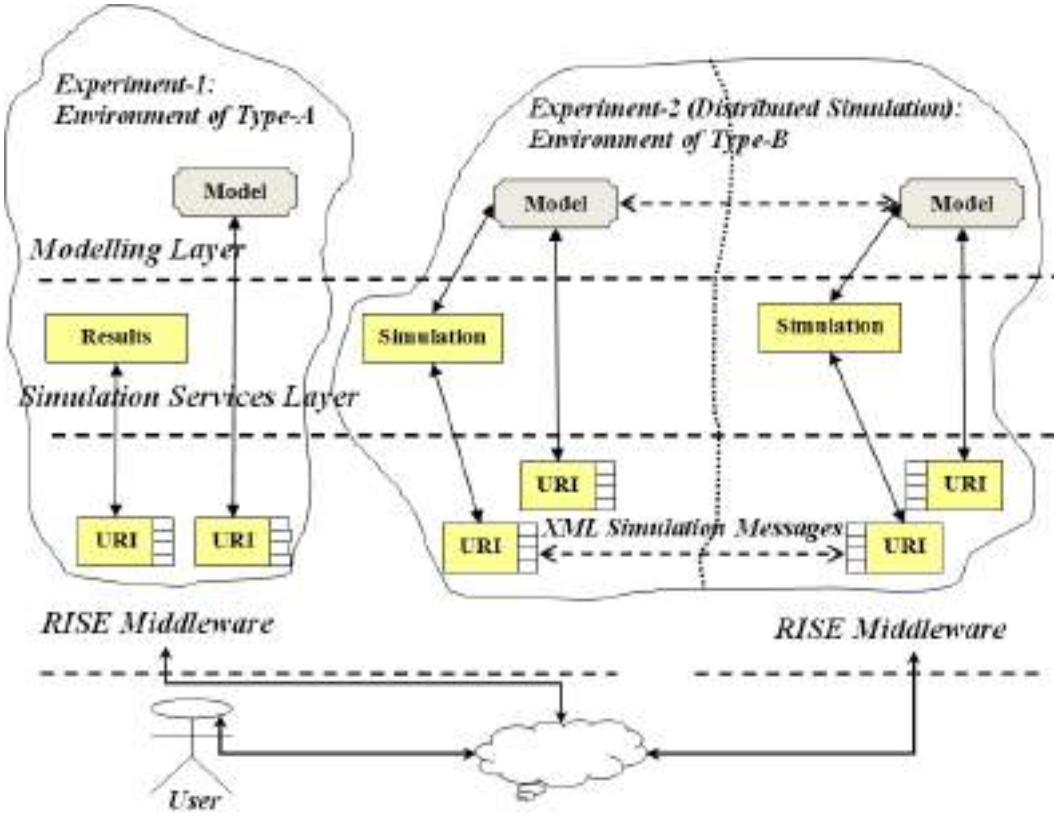

Figure 26: Simulation Experiment Resources (URIs)

Each of the experiment resources instances are attached to the Web (since they are URIs) where all information flow to/from those resources through a set of uniform channels (typically via XML messages; the channels provide a uniform interface and are discussed in Section 5.2). In fact, the URIs work as wrappers to concrete simulation

services and data. For example, Figure 26 shows two experiment instances where experiment-1 is of Type-A simulation environment while experiment-2 is of Type-B environment. In this example, experiment-1 holds results from a previous simulation run (the completed state) while experiment-2 is currently executing a distributed simulation over two computers (active simulation state). In distributed simulation experiments, similar to experiment-2, the algorithms in each partition synchronize their execution among each other via exchanging XML messages with URI "{framework}/simulation". This URI wraps all software components that execute the simulation in each partition, including simulation engines. It is worth to note that each resource in RISE API is described in terms of its URI, supported channels (see Section 5.2), messages exchanged via those channels to execute a certain function (see Section 5.3), and the type of errors responses that may be generated. Appendix-B describes the RISE resources specifications API.

Each of the experiments instances follows a pattern where it moves from a state to another, as shown in Figure 27 (the API details are described in Appendix-B). The figure shows that the modeler (for the first time) needs to *create* an *Experiment Framework* on the main RISE middleware and submit all of the necessary files and configuration settings to it. The experiment framework creation is performed via the PUT channel where the experiment settings (e.g. model partitioning) may optionally submitted in form of an XML message. If the XML message is received to an existing experiment URI, the experiment URI is updated; otherwise, the URI it is created. The *"{framework}"* URI is named by the modeler upon creation (e.g. *.../FireModelWithRain*).

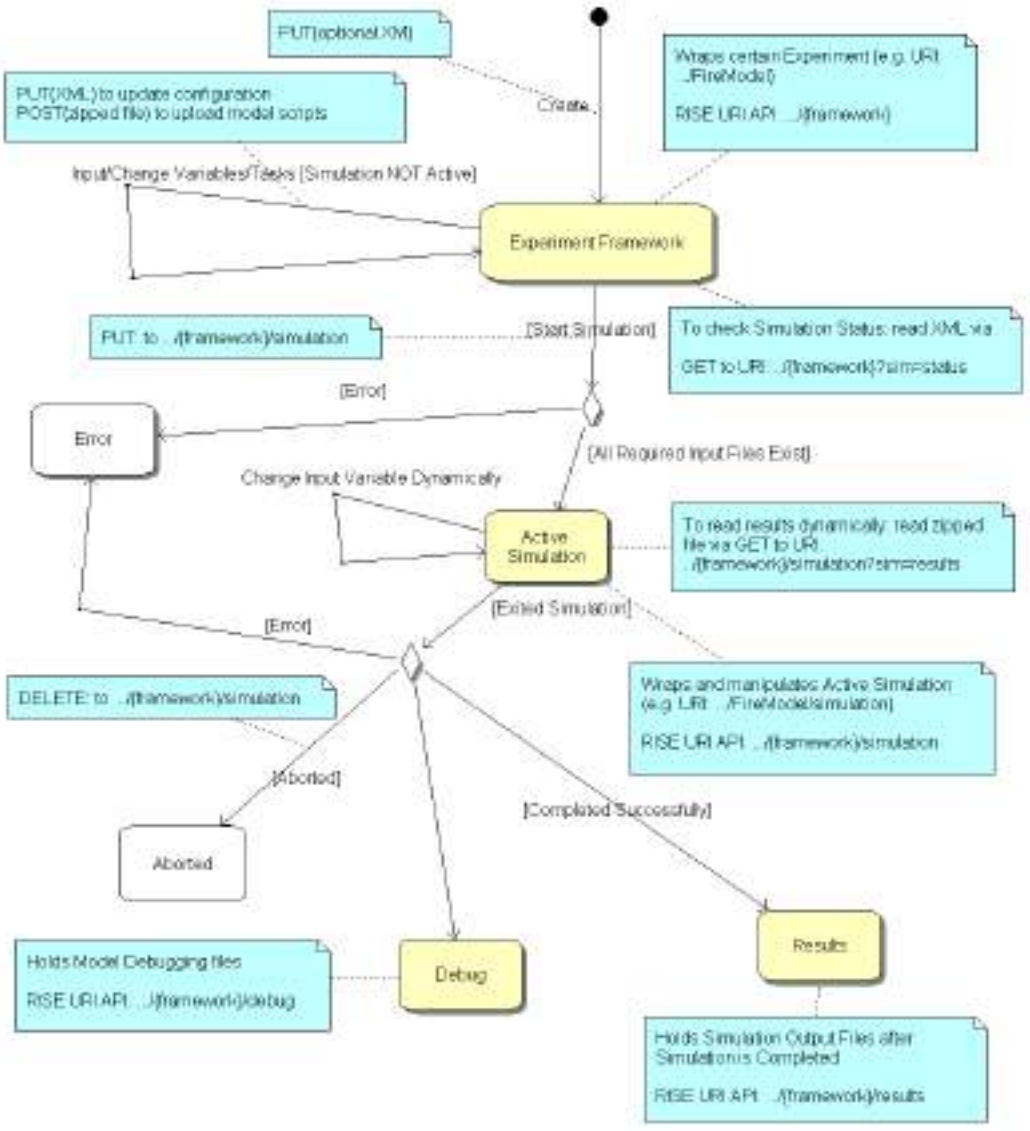

Figure 27: Simulation Experiment Pattern Context

After the framework (Figure 27) is created, the modeler can update the existing data via this URI. For instance, the models scripts can be submitted to this URI as a zipped file via the POST channel. Furthermore, the experiment settings may be updated sending new XML messages via the PUT channel. Note that these changes are only allowed if a simulation is not running the experiment. The main experiment URI can be used to check the simulation status via the GET channel to URI (*.../{framework}?sim=status*). In this case, the middleware responds with an XML

message containing one of the following states: IDLE (the simulation never run), INIT (the simulation is being initialized), RUNNING (the simulation is being executed), ABORTED (the simulation was stopped by the modeler), ERROR (the simulation stopped due to an error), STOPPING (the simulation is finishing), and DONE (the simulation completed correctly).

Once the experiment is setup (Figure 27), the simulation can be started by creating the *Active Simulation* URI (e.g. *.../FireModelWithRain/simulation*). However, before creating this URI, the middleware verifies that the experiment has been set correctly. This, for example, includes all of the model script files and configurations like the model partitioning scheme. This *Active Simulation* URI is used to manipulate simulation in an experiment during execution such as sending distributed simulation synchronization messages (by sending those messages via POST channel), inserting external events (by sending events via POST channel), reading simulation results (via GET channel to URI *.../simulation?sim=results*), etc. Once the Active Simulation URI is correctly created, the necessary components are created in each partition to manage and execute the simulation. At this point, the simulation exits on one of the following states: ERROR (e.g., a problem in the model scripts), ABORTED (the modeler sent a DELETE request to remove the simulation URI), or DONE (the simulation completed successfully). When the simulation is successfully completed, a *results* resource (e.g. *.../FireModelWithRain/results*) is created to hold all simulation output files. For example, these results can be downloaded at anytime to replay a simulation's visualization without executing the simulation again. However, if the model simulation execution aborted, the errors are stored in the debugging resource (e.g. *.../FireModelWithRain/debug*).

Note that the workflow component discussed in Chapter 8 uses this blueprint to create and manipulate different simulation experiment instances (executing workflow patterns), thus enhancing experiments automation, repeatability, management and reusability. For example, the settings of an experiment may be saved for future reuse without changing its input variables, since a modeler can create any number of experiment frameworks. In this case, a modeler may have different repeatable scenarios without being forced to change a certain experiment settings. For instance, a modeler may have an environment for model *FireModelWithRain* to simulate a forest fire when rain is present while creating another instance to simulate a forest fire without rain, say *FireModelWithoutRain*. In this case, the modeler has two scenarios to run simultaneously.

5.1.3 Resources Database

As stated in Chapter 1, one of RISE objectives is to preserve all experiments instances unless deleted by an authorized user. This leads to the need of a database to maintain all resources instances and settings, since an experiment uses a number of resources.

The database is divided into sections where each section belongs to a user (i.e., a username account). This allows multiple messages from different users to modify the database without blocking each other. This is because each incoming message is processed in its own thread (as shortly discussed in Section 5.2), and a single thread is only allowed to modify a data object at a time, but different threads are allowed to manipulate different objects simultaneously. The database is transactional, which means that a transaction is only allowed to enter an object in the database when the previous one

to the same object is completed. The database stores its objects in a file where each object is brought into the memory cache upon its first access.

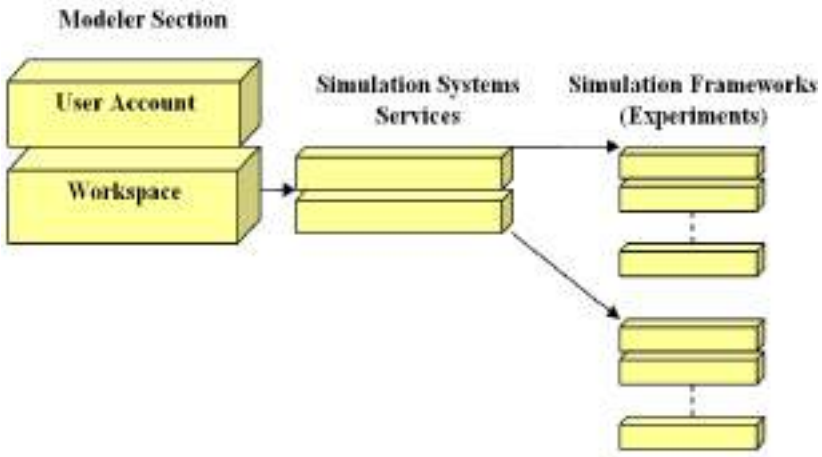

Figure 28: User Section in the Database

As seen on Figure 28, each user's section contains an account object (i.e. username, password, etc.) and a workspace object. The workspace contains the list of the simulation services (e.g. DCD++) that are currently used in this user's experiments. Each service object contains the list of the experiments objects that have been created by the subject user (the RISE implementation in Appendix-A provides more details). The database main issues are summarized as follows:

- RISE divides the database into sections, each section belonging to one username account. This minimizes the number of threads that need to manipulate the same data objects simultaneously.
- The database (stored in the file system) and the objects in memory have to be synchronized at all times (without degrading performance) because server reboots or failures may happen at any time. We should notice that the database and cache synchronization does not affect distributed simulation performance. This is because in RISE, simulation in an experiment always needs to be restarted if the

server fails during simulation (e.g. power failure). However, the RISE database keeps the door open to mark a simulation progress so that it can later be resumed due to such unexpected failures.

5.2 Uniform-Interface Mechanism

In the previous section, we focused on how resources (URIs) are organized, created and named to provide a general middleware and experiments blueprints. In this section, our focus is on how these resources are connected to each other and how information flows from/to those resources.

RISE advocates the concept of uniform interface of each resource (URI). This means that all resources are connected with the same software channels that are used to exchange all information between resources. The concept of software channels is usually realized (at the software level) by setting a field in the header of a message to specify the used channel of that message, hence providing a software multiplexing method for the messages exchanged. Since RISE already uses the HTTP envelopes to wrap all the transferred information, HTTP methods are then the ideal choice to realize those channels. Thus, RISE uses those HTTP methods and treats them as software virtual channels as follows (Figure 29):

- The GET channel is used to read information from resources such as simulation status and results.
- The PUT channel is to create a resource or update an existing data in a resource such as experiment settings.

- The POST channel is used to append new information to an existing resource such as sending an XML synchronization message in a distributed simulation session.
- The DELETE channel is used to remove a resource from RISE such as deleting an experiment URI.
- The OPTIONS channel is used to retrieve an XML description of all RISE API to provide a machine processing API based on the WADL XML standards [144] (see Appendix-B).

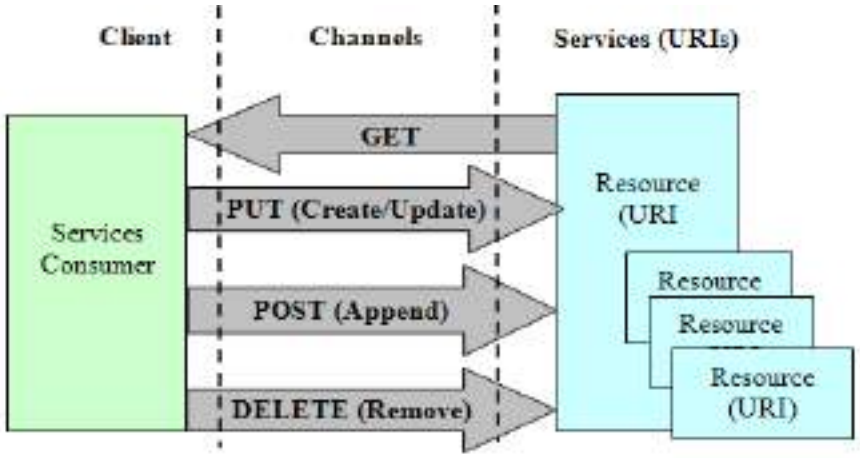

Figure 29: Uniform Channels for RISE Resources

Having predefined standardized channels for each resource, we can achieve composition scalability and improve dynamicity in distributed simulation. Since the channels of each resource automatically exist upon that resource creation, therefore, the dynamic interoperability foundations already exist. In chapter 8, we provide suggestions for extending the presented algorithms in that chapter to allow systems to join/abandon simulation at runtime. This would have been extremely difficult if, for instance, each system needed to compile other systems API stubs before interoperability can take place (see simulation systems API in existing approaches in Table 1 in Chapter 3). Further, because each resource is connected with the same number of virtual channels (regardless

of the number of remote resources); composition scalability is therefore automatically achieved, as shown Figure 30. Thus, a system always sends messages in a uniform way and accessed in a uniform way. The rest of this section discusses these two concepts: The uniform message transmission and the uniform access of resources.

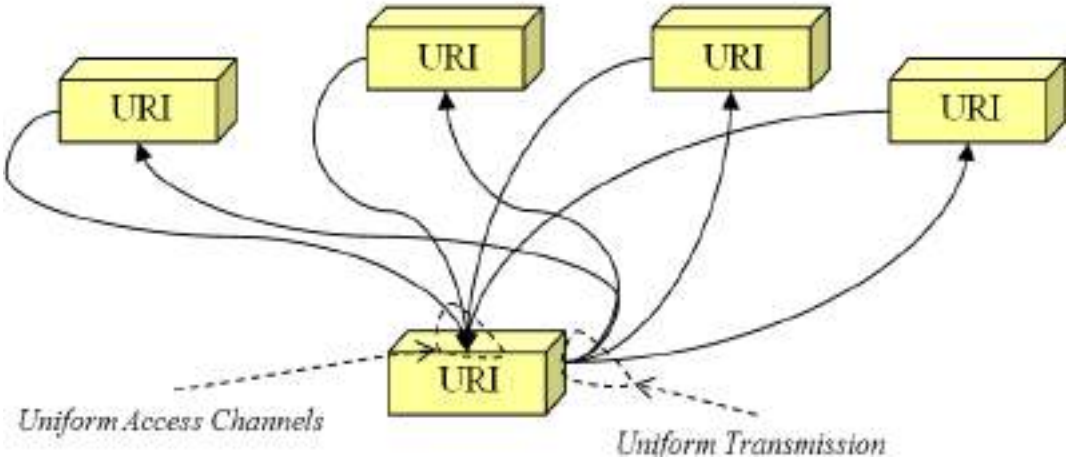

Figure 30: RISE-based Interoperability Channels Overview

Because of the uniform interface, all messages are transmitted uniformly regardless of the number of destinations (Figure 30). In this case, four parameters are needed to transmit a message: the destination URI, the channel name, the message syntactic format (e.g., defined in XML), and the actual message data. This means that the message transmission mechanism can be implemented in a single programming procedure (see RISE implementation in Appendix-A). However, RISE starts a separate thread (from a thread pool) for each message transmission. It further reuses the same TCP connections to transmit multiple messages over the same connections. This avoids establishing a TCP connection with every message transmission, which can be expensive.

Figure 31 shows an example of the message transmission process in RISE. In this figure, the simulation system sends external simulation messages to its simulation

manager component. The simulation manager interfaces the middleware with a specific simulation component. Note that simulation systems can extend the simulation manager implementation, if specific issues are not handled by the RISE general simulation manager (whose implementation is discussed in Appendix-A). The simulation manager converts a system specific simulation message to XML, which usually aggregates several simulation messages in a single message (XML messages are discussed in Section 5.3). It then hands the XML message to the middleware to be sent remotely. At this point, a simulation component assumes the information is sent correctly unless the middleware reports a transmission error. Finally, RISE starts a thread to handle the message transmission from the thread pool, wraps it with an HTTP envelop, and sends it to the destination URI.

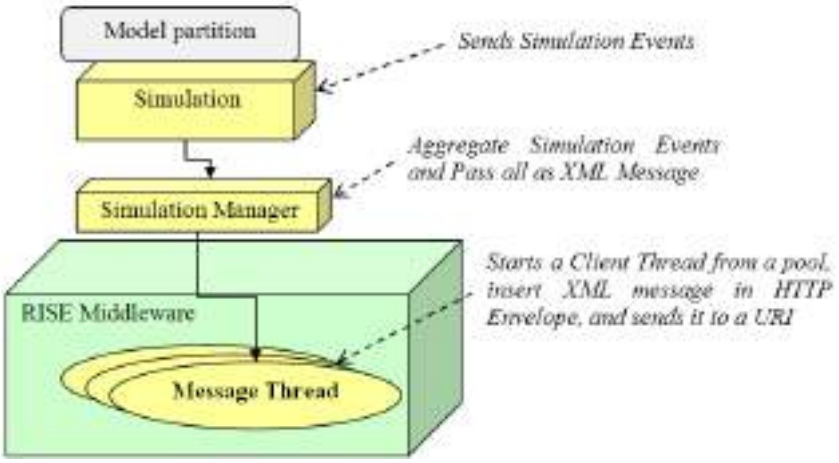

Figure 31: RISE-based Simulation Messages Transmission

Once the message arrives at its destination, RISE needs to route it to the appropriate resource (URI). RISE internally routes incoming messages to a URI instance based on the best-case match to a URI template. If a match is found, it assigns a thread from a pool to handle each incoming request. This processing mechanism is performed in three steps (Figure 32):

1. **Step 1:** the Router (i.e. thread in RISE) checks if the URI matches one of the URI templates in the server. If so, it starts a thread (from the threads pool) and initializes it with the HTTP request along with an instance of the Java class that is associated with the subject URI template (see implementation in appendix-A). Note that the thread owns the HTTP message and the Java object; hence, data contention by other threads is not possible in this case.
2. **Step 2:** The proper operation (of the Java object) is invoked based on the message channel. At this point, the enclosed message in the HTTP envelope is processed (e.g. converting received XML message and sending it to a simulation engine instance). Thus, a resource is implemented as a Java class where channels are operations in that class. These operations take HTTP requests as inputs and produce an HTTP response.
3. **Step 3:** The HTTP response is then generated and the message thread is terminated.

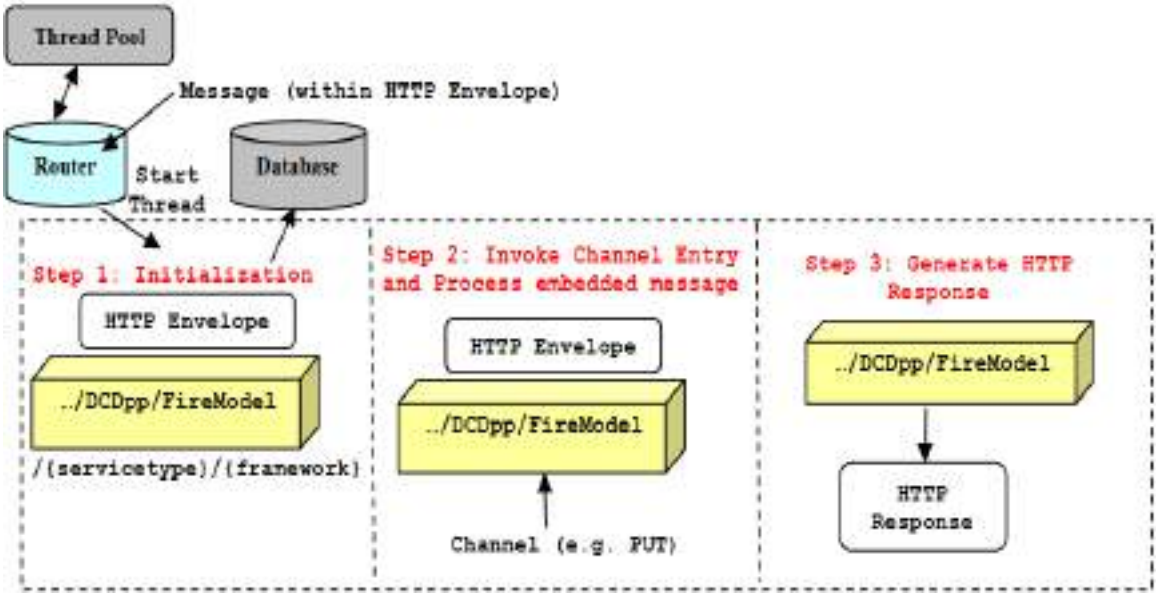

Figure 32: Processing Received Message in RISE Middleware

Based on the second step of the message processing in Figure 32, the message always enters a resource through a specific channel. RISE uses this characteristic to filter

all incoming messages via those channels based on the authentication and authorization scheme. The idea is that the GET channel (i.e. read data) does not change resources while the others do (i.e. write data). Therefore, in RISE, resources are created (by default) as read-only to everyone and read-write to the resource sole owner. However, the owner of a resource can change settings to block other users from retrieving information from that resource, if needed. RISE realizes the access mechanism by protecting every resource with a filter, as shown in Figure 33. The filter performs the following two steps upon receiving a request: **(1)** *Authentication* which verifies the username and password in the received request, and if authentication passes, it performs **(2)** *Authorization*, which verifies that the received request belongs to the owner of that resource. The filter responds with the Unauthorized (HTTP code 401) error, if either authentication or authorization fails. Otherwise, the received request is processed as expected.

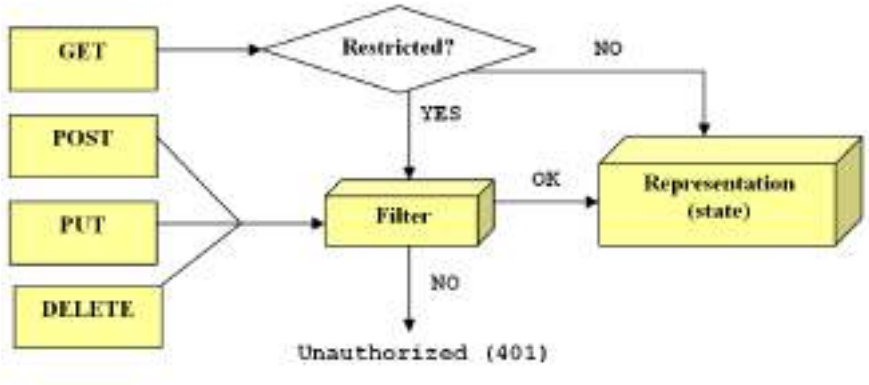

Figure 33: Resources Authorization Process

RISE authenticates each message, including synchronization messages in distributed simulation. The RISE middleware is capable of supporting the Hypertext Transfer Protocol Secure (HTTPS), if needed. In this case, the entire HTTP message contents (header and body) are encrypted to achieve higher security [115]. On the other hand, we assume here the use of HTTP protocol, since it is the typical use of the RISE

middleware. In this case, RISE applies the HTTP Basic authentication method [54]. In this method, the client combines and encodes the username and password into a single string with base 64 encoding, and inserts it in the HTTP header envelope. RISE uses this method for the following reasons:

1. It does not affect performance since it does not add new extra overhead for handling messages, particularly for distributed simulation synchronization messages.
2. It is supported by every Web-browser [115] so that users can still retrieve information from restricted resources via regular Web browsers.
3. It is widely supported by Web programming languages such as JavaScript, Java, etc.

5.3 Message-Oriented Information Description

In this section, our focus turned on the actual information that flow between resources through the virtual uniform channels discusses earlier. Since the RISE middleware transfers all information in HTTP envelopes and realizes the resources channels by HTTP methods, any data format type supported on the Web can be transferred between resources. For example, simulation output files can be downloaded from RISE as zipped files (see API in Appendix-B). However, our focus here is on the distributed simulation synchronization information, since they directly affect synchronization algorithms design and decoupling systems from each other. In RISE, all synchronization simulation messages semantics are described in XML (as in the case of the algorithms to be presented next in Chapter 6 and Chapter 8). One of the purposes of describing simulation information in XML is to enhance decoupling different systems implementations in a number of ways such as:

- Design decisions for handling simulation messages in a system become internal issues that do not need to conform to other systems implementations.
- Changes applied to existing simulation messages can be done without modifying systems software. In contrast, for example, programming parameters changes in existing approaches break software even at compiling time.
- Synchronization algorithms can be proposed on a higher level from programming details, independent of programming languages and implementations. For instance, the algorithms presented in Chapter 8 do not discuss software implementation issues. This is because the main concern of such message-oriented protocols is on what XML messages can be sent and what messages are expected in return. On the other hand, how these messages are handled in a system implementation is out of the protocol scope. In other words, one does not need to be a programmer to design such synchronization algorithms.

```
<Messages>
 <MessagesCount>3</MessagesCount>
 <Message>
   <MessageType>X</MessageType>
   ...
 </Message>
<Message>
   <MessageType>X</MessageType>
    ...
 </Message>
<Message>
   <MessageType>D</MessageType>
   ...
 </Message>
</Messages>
```

Figure 34: XML Simulation Message Aggregation Example

XML messages can also enhance synchronization protocols performance, particularly via aggregating several simulation messages in single XML message, as

shown in Figure 34. The DCD++ algorithms and the common algorithms aggregate simultaneous events in single XML messages. The performance results to be presented in Chapter 7 showed that aggregating remote messages could improve distributed simulation performance. This flexibility can go beyond improving performance to being able to support multiple synchronization protocols. This can accommodate different interoperability domains, which allow systems to evolve independently at different directions. For example, DCD++ (to be discussed in Chapter 6) can also support other algorithms (in addition to its own), like those presented in Chapter 8. This is achieved mainly because of two reasons: the messages uniform transmission, and the use of XML. In the uniform transmission, to transmit a message four pieces of information are always needed: the destination URI, the channel name, the message syntactic format, and the actual message data. Thus, to accommodate different protocols, the software needs to pack the XML messages according to different synchronization protocol semantics. In this case, sending different types of XML messages is the same from the sending routine viewpoint. It can be as simple as of the following:

```
If (Protocol-A) then {     //protocol A
   Pack_XML_Message_TypeA(Msg);
} else If (Protocol-B) then {   //protocol B
  Pack_XML_Message_TypeB(Msg);
}
Send(URI, Channel, XML_Type, Msg); //Send the message
```

At the receiving system, the XML messages are processed similar to the above transmission principle. Of course, the packed information in an XML message still needs to be mapped to the local software implementation. On the other hand, these types of issues are irrelevant to the other systems implementations and synchronization protocols,

which is a fundamental factor for decoupling systems implementations and easing interoperability.

5.4 Chapter Summary

This chapter provides a design methodology that uses RESTful WS structural rules. These rules allow more freedom fir making better design decisions in order to achieve the objectives stated in Chapter 1. RISE uses the RESTful WS structural rules to spread services over a number of resources (resource-oriented), and resources exchange synchronization information in form of XML messages (message-oriented) via predefined uniform channels (uniform interface).

The resource-oriented design organizes resources in a scalable hierarchical structure. Those resources are exposed by the middleware as URI templates. This means that those resources instances can be created and named by modelers at runtime. This leads to the concept of general layered interoperability where different simulation resources (URIs) are organized at a separate layer above the middleware. Further, because RISE experiments are externally seen on the Web as URIs, URI templates allow RISE to provide experiments as blueprints patterns. This means that various experiments of different types and URIs can be created at runtime, and the steps usually performed to create and manipulate experiments can be automated (as in the case of the workflow component to be discussed in Chapter 8). RISE also maintains all resources in a database (unless deleted by authorized users), which allows RISE to maintain all URIs similarly to a typical HTTP server.

RISE advocates the concept of uniform interface of each resource (URI). This means that all resources are connected with the same software channels that are used to exchange all information between resources. The concept of software channels is usually realized (at the software level) by setting a field in the header of a message to specify the used channel of that message, hence providing a software multiplexing method for the messages exchanged. Those channels are based on the HTTP methods (conforming to universally accepted standards): GET, PUT, POST and DELETE. By having predefined standard channels for each resource, we can achieve composition scalability and improve dynamicity in distributed simulation (since the channels of each resource exist upon that resource creation, the dynamic interoperability foundations already exist). Further, because of the uniform interface, all messages are transmitted uniformly regardless of the number of destinations. In this case, four parameters are needed to transmit a message: the destination URI, the channel name, the message syntax (e.g., defined in XML), and the actual message data. Thus, a single programming procedure is sufficient to transmit all messages to any destination. Furthermore, because of the uniform interface, all incoming messages to a resource access from known predefined gates. RISE uses this characteristic to filter all incoming messages via those channels based on the authentication and authorization scheme. The idea is that the GET channel (i.e. read data) does not change resources while the others do (i.e. write data).

In RISE, all synchronization simulation messages semantics are described in XML (message-oriented). One of the purposes of describing simulation information in XML is to enhance the decoupling of different systems implementations. This is because implementing those messages in a system becomes an internal issue that is irrelevant to

other systems. It further enhances the idea of supporting multiple synchronization algorithms. This is because the idea behind synchronization algorithms is that the software implements a set of rules and coordinates the internal activities with other systems via messages. Thus, to accommodate different protocols, the software needs to pack the XML messages according to different synchronization protocol semantics.

CHAPTER 6: DISTRIBUTED CD++ (DCD++) SIMULATION

RISE-based Distributed CD++ (DCD++) performs distributed simulation between different CD++ engines where each is placed in a partition and is in charge of simulating a portion of the entire CD++ model. As discussed in Chapter 5, experiments in RISE are seen on the Web as URIs built by modelers with names of their choice according to the RISE URI templates. Therefore, for DCD++ simulations, one must build an experimental framework in RISE. In this case, modelers start the simulation by building URI <.../*{framework}/simulation*> where *{framework}* is the experiment name. Once the simulation starts, all of the software components needed to execute the simulation are built on each partition (in our case, the URI <.../*{framework}/simulation*> wraps the software components in each partition during active simulation). Thus, this URI is externally used to reach and communicate with the active simulation.

As also discussed in Chapter 5, RISE is designed as a software layer supporting different services in upper layers. The assumption is that those services can be built and executed as separate Operating System (OS) process. In other words, concert services need to be packaged in standalone software components. At runtime, those components are started as separate OS processes outside the RISE layer and all communication is performed via the OS Inter-Process Communication (IPC) queues. This applies to the CD++ engine once started within an experiment partition, since CD++ is a standalone application. The advantage of this design is to separate RISE implementation from other software components. This design has worked, based on our testing, with different

software components (other than CD++) as long as those components can be started as a standalone OS process. This means that this component can be started by a defined command (e.g. *startComponent*) so that RISE can use this command to start the component on the local machine. The components that cannot be started on the local machine (for reasons such as operating system compatibility) were not investigated in our testing. We further added a thread to each component to handle sending/receiving messages to/from RISE through the OS IPC (while the component is running). This design was tested by components written by us (and obtained by available open source software from the Internet). These components implemented various functions such as a simple calculator, monitoring hardware devices such as atmospheric pressure sensors, and device drivers to manipulate local hardware, etc. In conclusion, software components were different because they were written by different programmers, thus interfacing such components varied from one to another.

This chapter main contribution is the DCD++ architecture and the enhancement of the simulation synchronization algorithms. In this case, the CD++ engine, which is based on Parallel DEVS (P-DEVS) [38], is interfaced to the Simulation Manager component (on the RISE middleware side). In this case, within each partition, the CD++ performs the time management while the simulation manager performs the data distribution on behalf of the CD++ instance. Note that the P-DEVS algorithms have already been proven to work correctly within the DEVS community since their first introduction in 1994 by Chow and Zeigler [38].

The RISE-based DCD++ architecture is described (Section 6.1) in terms of the software components built during simulation and their roles. The synchronization

algorithms are discussed (Section 6.2) in terms of simulation messages, simulation progress phases, and remote messages aggregation (which remote messages and ensures accurate order arrival). Note that implementation is discussed in Appendix-A.

6.1 DCD++ Simulation Architecture

Distributed CD++ (DCD++) splits the model hierarchy presented in Chapter 2 between several CD++ engines to be executed across the distributed environment. In this case, each CD++ is responsible of simulating a portion of the model hierarchy and of coordinating with other CD++ engines to execute the simulation correctly. However, as discussed in Chapter 5, before simulation can be started, the experiment framework needs to be created and setup on RISE. In this case, the experiment is created by creating the URI *.../{framework}* where all required setup files are submitted to. Afterward, the simulation is started via creating URI *.../{framework}/simulation* (see API in Appendix-B). Note that the RISE middleware that the modeler uses to create the experiment will be the main experiment partition. The main partition owns the Root coordinator, which drives the simulation on all other remote partitions (on other machines). Model partitioning is discussed shortly in this section.

Once simulation is started, all of the necessary software components are created in all partitions. All of these components, built upon starting the simulation, are deleted when the simulation is completed or aborted. The overall architecture is shown in Figure 35.

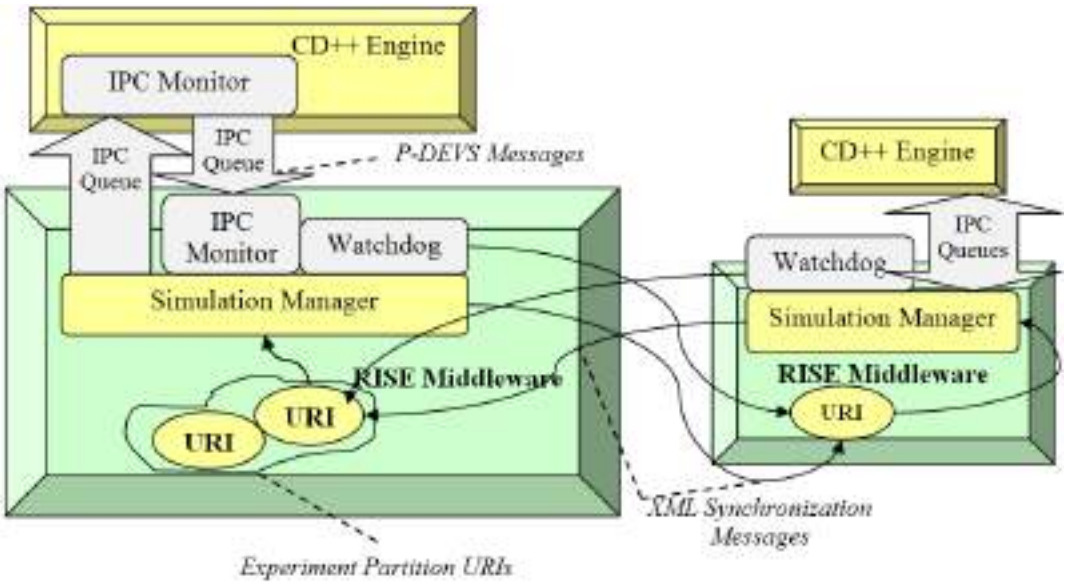

Figure 35: DCD++ Experiment with Two Partitions during Simulation

Figure 35 shows an example of a RISE-based distributed CD++ (DCD++) experiment with two active partitions during simulation. Each partition contains an instance of the CD++ engine [139] to simulate the DEVS hierarchy models that belongs to this partition. The CD++ engine used here is based on the SOAP-based DCD++ system described in [137], which was originally based on P-DEVS algorithms [38]. The simulation manager is the component that manages the distributed activities on behalf of the CD++ engine with remote partitions. Note that the CD++ (which is based on P-DEVS) performs the simulation time management while the simulation manager performs the simulation data distribution on behalf of its CD++ instance. Both Simulation Manager and the CD++ engine communicate P-DEVS messages through the OS IPC queues.

The idea is that the CD++ executes by sending messages to their destination processor (coordinator/simulator). However, if the destination processor does not exist locally, it is sent to the simulation manager (on the RISE side), which knows how to

handle it. This allows CD++ to execute the simulation locally as if it was running on a single processor, while allowing the simulation manager to handle the distributed activities.

The IPC Monitors are threads found at both ends of the IPC queues, which are in charge of processing P-DEVS received from the other. Once the CD++ IPC Monitor thread receives a message (from the simulation manager), it inserts it in the main CD++ external event list. This relieves the main thread (within CD++) of continually checking the IPC queue. Further, once the IPC Monitor thread (at the Simulation Manager side) receives a message from the CD++ engine, it buffers it at the Simulation Manager according to its partition destination. This allows the Simulation Manager to aggregate those remote messages in XML and transmit them together (via RISE). This not only reduces the number of remote messages through the Internet, but also avoids causality errors. This incorrect simulation could occur because P-DEVS messages may arrive in the incorrect order of their transmissions at the distant CD++ (since those messages are transmitted concurrently via the Internet). Thus, these messages may violate the local causality constraint in the distant CD++, starting the wrong simulation phase, as discussed in the next section.

The final component shown in Figure 35 is the watchdog component. A watchdog is a thread that sends periodic messages to other simulation URIs to check their presence. If a partition is present, it responds back with the XML message *<simulation>ALIVE</simulation>*. Otherwise, it responds with HTTP error 401 (Not Found). The watchdog on the main partition watches all other partitions existence, while the watchdog threads on other partitions watch the main partition. Watchdog threads are

necessary because a CD++ engine on a partition may fail during simulation, leading to inaccurate simulation or deadlocks. For example, assume a CD++ in a partition fails while the CD++ Root coordinator (in the main partition) is waiting for a "Done" message from that dead partition. In this case, the Root coordinator would wait forever without being able to advance the simulation (and without being aware of the problem). Note that this periodic message value (two minutes by default) is configurable by the modeler. It only indicates the time it takes the modeler to know of a hanged simulation session (RISE, in this case, would abort simulation and change the experiment state to ERROR). Otherwise, it does not interfere with the actual simulation progress or correctness.

It is worth to note that the SOAP-based DCD++ [137] interfaces different simulation sessions to a single Web-service Wrapper. This wrapper assigns a number to each simulation session, and builds a CD++ engine for each session. However, this design does not scale well performance wise (all CD++ engines of different sessions on the same machine are interfaced to a single WS wrapper using the same IPC queues). Thus, the more sessions are added the more communication overhead pressure is added on those IPC queues and to the WS wrapper (which handles all sessions' requests sequentially). Further, sessions that simulate models with low remote communication overhead performance can be degraded if they share those queues with other sessions with heavy communication overhead. In contrast, the architecture in Figure 35 separates different experiments from each other. Experiments managed by RISE on the same machine will have their own CD++ engines, threads, IPC queues, and simulation managers.

As previously discussed, once the simulation is started, the RISE middleware builds the simulation manager, which in turns builds all components shown in Figure 35. Afterward, the model needs to be loaded by the CD++ engine in each partition. In this case, CD++ only needs to simulate a portion of that model. Thus, the simulation manager stores the model partitioning requirements, allowing the CD++ to use this information during model loading. To do so, CD++ in a partition initializes the model in two steps:

1. Load the model by parsing the CD++ model coupled model file (CD++ assumes that it will simulate the entire model; hence, it parses the entire model).
2. Build the processors that will simulate all models in the DEVS models hierarchy. Before building each *simulator*, CD++ requires permission from the simulation manager to do so (because it knows how the model is partitioned, as discussed shortly). If permission is granted, the *simulator* is built and assigned a unique ID. However, if permission is not granted, the *simulator* is not built, but the ID is still generated (and forwarded to the simulation manager). This is important because DCD++ maintains a unique ID for each processor in all partitions. Note that *simulators* only exist in one partition, since they simulate atomic models. However, *coordinators* can span over multiple partitions, since they simulate coupled models.

The CD++ model is structured across the network based on the model partitioning requirements (which helps the simulation manager to perform data distribution on behalf of its CD++ instance). These requirements originally come from the modeler as part of setting up the experiment. Therefore, the assumption here is that the modeler is aware of how the model partitions influence each other before the simulation starts. For example,

the XML document in Figure 36 splits a 30x30 Cell-DEVS based fire model into two partitions

```
1 <DCDpp>
2      <Partitions>
3         <Partition IP="10.0.40.175" PORT="8080">
4           <ZONE>fire(0,0)..(14,29)</ZONE>
5         </Partition>
6         <Partition IP="10.0.40.162" PORT="8080">
7           <ZONE>fire(15,0)..(29,29)</ZONE>
8         </Partition>
9      </Partitions>
10 </DCDpp>
```

Figure 36: XML Model Partitioning Example

Lines 2-4 in Figure 36 describe the first partition; Lines 6-8 describe the second partition. Each partition specifies the RISE middleware identification (IP address and the port number) and the atomic models belonging to this partition. IP addresses and port numbers enable the machines to calculate each other base URIs. For example, <http://10.0.40.175:8080/cdpp> is the base URI of the RISE running the first partition (Line #2). The above XML document also places the cells zone (0,0) to (14,29) on the first partition (Line #4) and zone (15,0)..(29,29) on the second partition (Line #7). This partitioning information is mapped to the DCD++ model hierarchy shown in Figure 37.

As shown in Figure 37, the CD++ in Partition 0 (on the left) builds the Root coordinator to manage the overall simulation on partitions (the time management and synchronization according to P-DEVS algorithms). Note that the “Local communication” indicates that all messages are indirectly sent via the local CD++ Event List, as described shortly. This is the case, between the Head Coordinator and its children Simulators, between the Proxy Coordinator and its children Simulators, and between the Head Coordinator and the Root Coordinator. However, the “Communication via RISE” indicates that messages are sent through the Internet (which usually aggregated in XML

messages as described in next section). This is the case between the Proxy and the Head Coordinators.

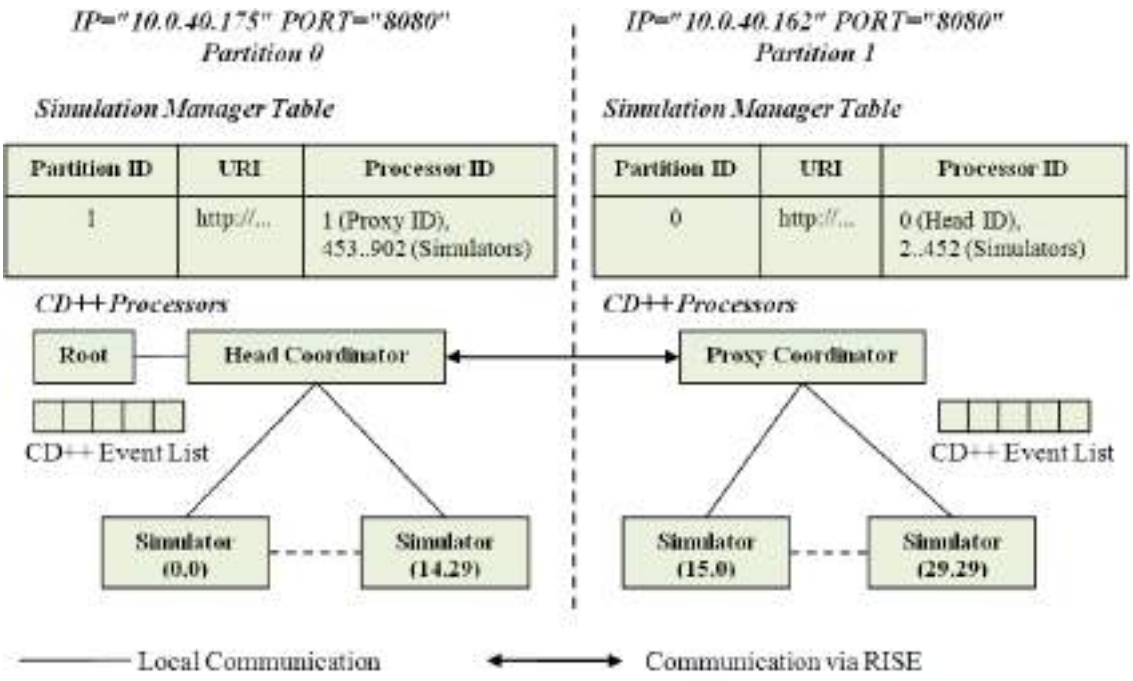

Figure 37: DCD++ Processors Hierarchy Partitioning Example

As discussed in the next section, the Root starts a new simulation phase by locally sending one message to the highest coordinator, which then propagates in the hierarchy until it reaches the Simulators. Once Simulators complete their work, they respond with DONE messages to their coordinators where each coordinator forwards a single message to its parent until it reaches the Root coordinator. Thus, the Root coordinator is similar to any other intermediate DEVS coordinator in the hierarchy. In this case, the time it takes a simulation phase of a coordinator to complete is the time it takes all of its children segments to complete. Most of these segments are concentrated locally (Figure 37), however, these local communications still need to be performed via the same CD++ Event List. Note that the Simulation Manager Tables (Figure 37) allow the simulation

manager component to distribute data to their appropriate destinations during simulation execution.

In this example (Figure 37), this Root coordinator becomes the parent of the Head coordinator with ID 0, which simulates cells zone (0,0) locally to (14,29). Since CD++ simulates each cell as an atomic model, CD++ will then run 450 simulators (with IDs 2..452). In the same way, the CD++ in Partition 1 (in the right) starts the Proxy coordinator with ID 1. This Proxy coordinator locally simulates cells zone (15,0)..(29,29) on behalf of the Head coordinator in the other partition. In this case, the CD++ in Partition 1 runs 450 simulators with IDs 453..902. The CD++ processors pass to each other the P-DEVS messages by first inserting those messages in the CD++ External Event List (to be executed later by the *Administrator*). The *Administrator* picks the message at the front of the CD++ Event List and checks the destination of the message; if the destination is a local processor, the message is directly delivered to that processor. However, if the destination processor does not locally exist, the message is then sent the Simulation Manager, which maintains the necessary partitions information to be able to transmit remote messages. Specifically, it stores the remote partitions URIs and the CD++ processors IDs on those partitions.

As shown Figure 37, DCD++ extends the concept of coordinators into a head/proxy structure [137]. The idea of the head/proxy depends on using two kinds of coordinators for each coupled model: **(1)** Head Coordinator: is responsible for synchronizing the coupled model execution, interacting with upper level coordinators and message routing among the local and remote processors. **(2)** Proxy Coordinator: is responsible for message routing among the local processors. The advantage of using

proxy coordinators is to avoid remote message transmission between local processors. For example, assume that Simulator (15,0) needs to send Simulator (29,29) a message. In this case, this message will be routed through the Proxy coordinator. On the other hand, if the Proxy coordinator was not used, the message would be first sent to the Head coordinator in Partition 0, which would then routed back to Partition 1.

6.2 DCD++ Simulation Synchronization

During simulation, the CD++ processors discussed in the previous section send each other P-DEVS messages via inserting them first in the local CD++ External event list. The CD++ administrator processes messages in the list by dropping them directly in the local CD++ processors or by sending them to the simulation manager (to be forwarded to remote processors). P-DEVS messages can be categorized as follows:

1. Content messages represent events generated by a model. Content simulation messages include External messages (X) and Output messages (Y). Y messages are usually converted to X messages when they arrive at destinations. X and Y messages that are exchanged within a simulation phase are simultaneous, since they are stamped with the same simulation time.
2. Synchronization messages cause the simulation to move into another simulation phase (those phases discussed next). In other words, synchronization messages mark the beginning and the end of each simulation phase. Synchronization messages include Initialize message (I) (to start initialization phase), Internal message (*) (to start transition phase), Collect message (@) (to start a collection phase), and Done message (D) (to end a phase). In this case, the I, *, and @ messages are sent from the

parent coordinator downward throughout the model hierarchy. On the other hand, the D messages are generated from simulators upward throughout the model hierarchy (Figure 38).

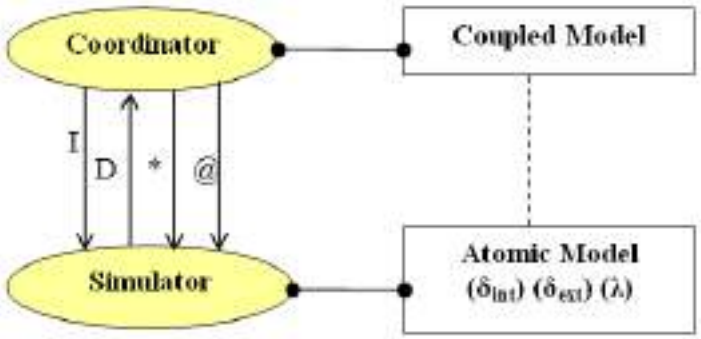

Figure 38: Message exchange during a simulation cycle

The simulation phases are divided into three phases:

1. The Initialization phase initializes all models in the hierarchy. It is initiated by the Root coordinator by sending an I message to the topmost coordinator. It propagates downward in the hierarchy until it reaches the simulators. In response, a D message propagates upward in the model hierarchy where each Coordinator calculates the minimum next change of its children and passes it in a D message to its parent. Eventually, Root receives a D message with smallest time, which updates the simulation clock and starts the Collection phase.
2. The Collection phase is initiated when the Root coordinator sends the @ message downward the hierarchy. The Y messages generated in this phase are collected to ensure their execution at the same time with internal events (in the transition phase). The collection phase is ended once the Root coordinator receives a D message.
3. The Transition phase is initiated by the Root coordinator by sending the * message downward the hierarchy. In the Transition phase, all the Y messages collected in the

previous phase are converted into X messages. In this case, all X messages are executed alongside the simultaneous internal events (by simulators).

Once the initialization phase is completed by the receipt of D message at the Root coordinator, the Root coordinator immediately starts the transition phase. From now one, simulation phases are executed by the Root coordinator (upon the receipt of D message) according to the shown algorithm in Figure 39.

```
// At first call Next Phase is initialized to Transition
Root Coordinator::ReceiveDoneMessage (DoneMessage msg) {

   If (Next Phase == Transition) {
      // Start transition phase
      Next Phase = Collect;
      Send Internal (*) Msg to highest model;
   } Else {
      Time = Current Time + Next Change in msg;
      If (Time <= STOP_TIME) {
         Send Stop to all;
      } Else {
         While (environment event == Time) {
            Send environment external event to highest model;
         }
         If (Next Event is NOT external) {
             // Start the Collect Phase
             Next Phase = Transition;
             Send Collect (@) Msg to highest model;
         } Else {   // Start transition phase
            Next Phase = Collect;
            Send Internal (*) Msg to top model;
         }
      }
   }
}
```

Figure 39: Root Coordinator Handling DONE Message Algorithm

Therefore, as shown in Figure 40, at each virtual time (t), there is at least one mandatory Transition phase and an optional Collection phase. This means that multiple Transition phases may be executed multiple times (since additional internal events may continually be generated). However, the initialization phase only exists at t_0.

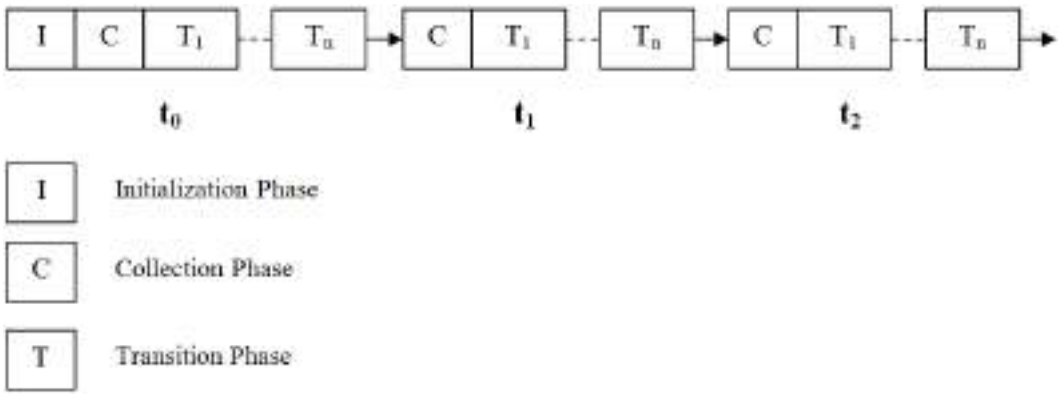

Figure 40: Root Coordinator Simulation Phases

Several key characteristics of the above simulation phases can be summarized as follows (which underlies some of the needs behind aggregating remote P-DEVS messages as discussed later):

1. Once a phase is initiated by the Root coordinator, it keeps moving downward in the processors hierarchy. In other words, each coordinator initiates the simulation phase for all of its children.
2. Each phase is started by a message (I, @, or *) and ends with a D message. This means that the messages must be received at the destination processor in the correct order to ensure correct simulation progress.
3. Content messages (Y and X) are sent to the destination processors (collection phase) so that they can be executed (by simulators) with the internal events that need to be simulated at the same time.

The above issues can cause incorrect simulation in the RISE-based DCD++ if one applies the original algorithms used by the SOAP-based DCD++, because all the remote messages in RISE are transmitted concurrently (for obvious performance reasons). In this case, each message is transmitted within its own thread independently from other

messages. Therefore, one can never guarantee that independent messages transmitted via the Internet would arrive to the destination in the same order of their transmission and in the correct simulation phase. This is not a problem in the SOAP-based DCD++ because messages are transmitted sequentially and the transmitter is blocked until the message is acknowledged by the receiver (i.e. the RPCs blocks the caller software until is completed). In order to overcome this problem, we aggregate simultaneous remote P-DEVS messages and send together. This solution applies parallelism during the message transmission, ensures the correct order of P-DEVS messages at destination, and ensures messages execution in the correct simulation phase.

The aggregation also improves the original head/proxy algorithms by reducing the number of remote messages transmissions (as it will be seen in the performance tests introduced in Chapter 7). As previously mentioned, the original head/proxy only routes remote messages locally if destinations processors exist in the local CD++ engine. However, messages still need to be transmitted remotely if their destinations exist in remote partitions. In this case, aggregation can reduce multiple messages transmissions to a single message transmission cost.

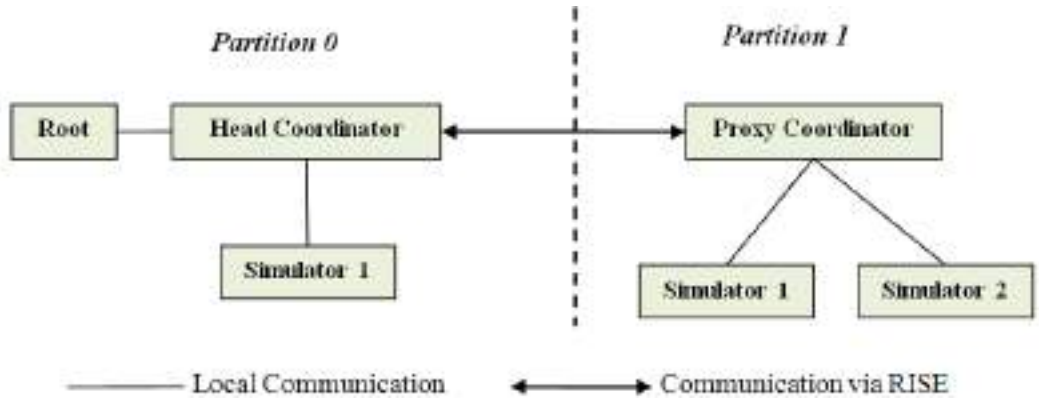

Figure 41: DCD++ Processors Hierarchy Example

To give an example of the simulation phases with message aggregation, let us consider the DCD++ processors hierarchy shown in Figure 41. Here, Partition 0 consists of the Root coordinators, Head coordinator and Simulator 1, while Partition 1 consists of the Proxy coordinator, Simulator 2 and Simulator 3. The *Local Communication* is performed directly by the local CD++ engine. However, the *Communication via RISE* indicates that the destination processor is in a remote partition. In this case, CD++ forwards remote messages to the simulation manager, which aggregates simultaneous messages in each phase and transmit them in an XML message.

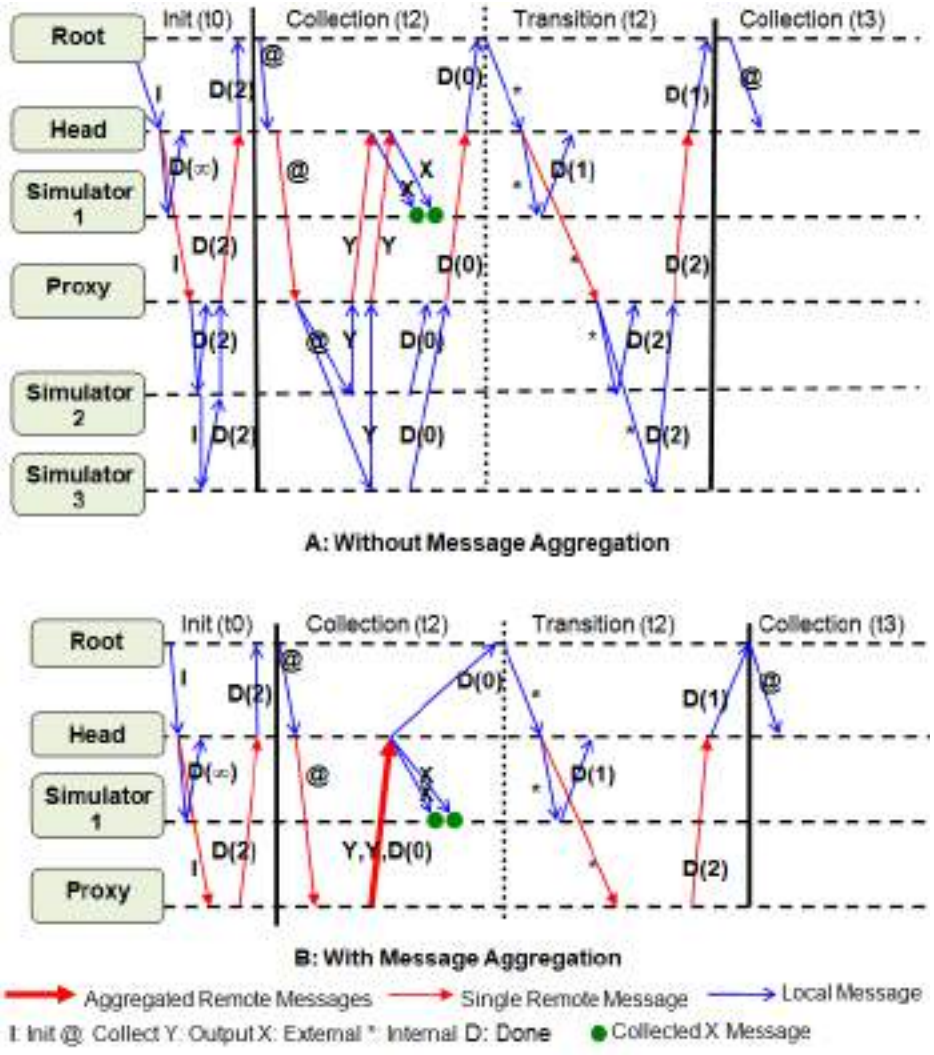

Figure 42: DCD++ Simulation Phases and Time Advancement

Figure 42 shows an example of the simulation progress phases based on the DCD++ processors hierarchy shown in Figure 41. Figure 42-A shows the simulation phases progress *without* message aggregation while Figure 42-B shows the simulation progress *with* message aggregation. Let us suppose that Simulators 2 and 3 outputs a job to simulator 1 every two time units while Simulator 1 takes one time unit to process each job regardless of the number of jobs are being processed.

The simulation phases without aggregation progress as the follows (Figure 42-A):

1. The first phase is Init(t0) (initialization phase at time 0). A Message I is sent to the Head coordinator, which passes it to Simulator 1 and the Proxy coordinator. Consequently, the Proxy routes message I to Simulator 2 and Simulator 3. Each of Simulator 2 and 3 reply with D(2) message. This means that both have scheduled an internal change in two time units from now. Further, Simulator 1 replies with D(∞). This means that Simulator 1 has no more events to execute.
2. The second phase is Collection(t2). In this phase, Root advances time to 2 and starts the collection phase by sending message @ to the Head coordinator, which only sends it to the proxy. This message is not sent to Simulator 1 because it did not schedule a change in the previous phase; hence, it becomes irrelevant in this phase. Consequently, the Proxy passes message @ to Simulator 2 and 3, which causes them to send two jobs (i.e. each sends a Y message) to Simulator 1 (via the Head and Proxy coordinators). The Head coordinator converts those Y messages to X messages and send them to Simulator 1. Simulator 1 then *collects* them to be executed in the next phase. Simulator 2 and Simulator 3 ends this phase by sending D(0) message upward in the hierarchy. This also means that they will be involved in the next phase.

3. The third phase is Transition (t2). The Root starts transition phase (by sending * message downward) causing Simulator 1 to execute the two previously collected X messages. It further schedules a change at one time unit from now. In addition, Simulators 2 and 3 schedule a change at two time units from now (when they will produce their next jobs as Y messages).

As shown in Figure 42-B, the only messages that were aggregated and sent together are in the Collection (t2) phase. These messages are the two Y messages and the D(0) sent from the proxy coordinator to the head coordinator. However, other remote messages were sent individually. This is because other remote messages must not be delayed to allow simulation to progress. For example, the @ message sent by Head coordinator to the Proxy coordinator in Collection (t2) phase must be transmitted to trigger the collection phase on the Proxy portion of the hierarchy. Otherwise, the simulation does not advance.

The basic idea behind aggregation is that content messages (Y and X) are sent to processors within a simulation phase. Thus, they are simultaneous messages (messages exchanged within the same simulation phase). On the other hand, synchronization messages (I, @, *, and D) are sent to start/end a phase. Therefore, content messages that are heading to same processor can be buffered until the first synchronization message is received to that processor. Further, messages that are heading to processors in the same remote partition can also be sent together. Therefore, dispatching and aggregating simulation messages are only performed to remote messages (which were originally forwarded from the CD++ to its simulation manager). This algorithm is listed in Figure 43 followed by an example described in Figure 44.

```
DispatchAndAggregateRemoteMessage (Msg) {

   Dispatch_XML_Msg = false;

   Find_Remote_Partition_ID (Msg.DestinationProcessorID);

   If (Remote Partition does not exist) {
      Start aggregation for this partition;
   }
   If (Remote Processor does not have a message bag) {
      Start Msg bag for Remote Processor;
   }

   Insert Msg in the Processor's bag;

   If (Msg is of Synchronized type) {// if D, @, *, or I
      If (All Partition Processors ends with Synchronized type) {
         Dispatch_XML_Msg = true;
      }
   }

   If (Dispatch_XML_Msg) {
      Start XML Document Builder;
      Pack partition Msgs count in XML Document;
      For (all messages Processors bags in this partition) {
        Pack Msg in XML Document;
      }
      Close XML Document;
      Release all partition bags memory;
      Send XML Document to partition simulation URI;
   }
}
```

Figure 43: Dispatching and Aggregating Simulation Messages in XML

Figure 44 shows an example of the aggregation scheme. In this scheme, the CD++ (in the local partition) maintains unprocessed events in a Least-Time-Stamp-First (LTSF) list. The CD++ Administrator executes the first event in the list by sending to a local processor or by sending it to the simulation manager. The Aggregator (in the simulation manager side), maintains the aggregation message queues. These queues are built and destroyed dynamically as needed. The aggregation queues are organized by their destination partitions and processors.

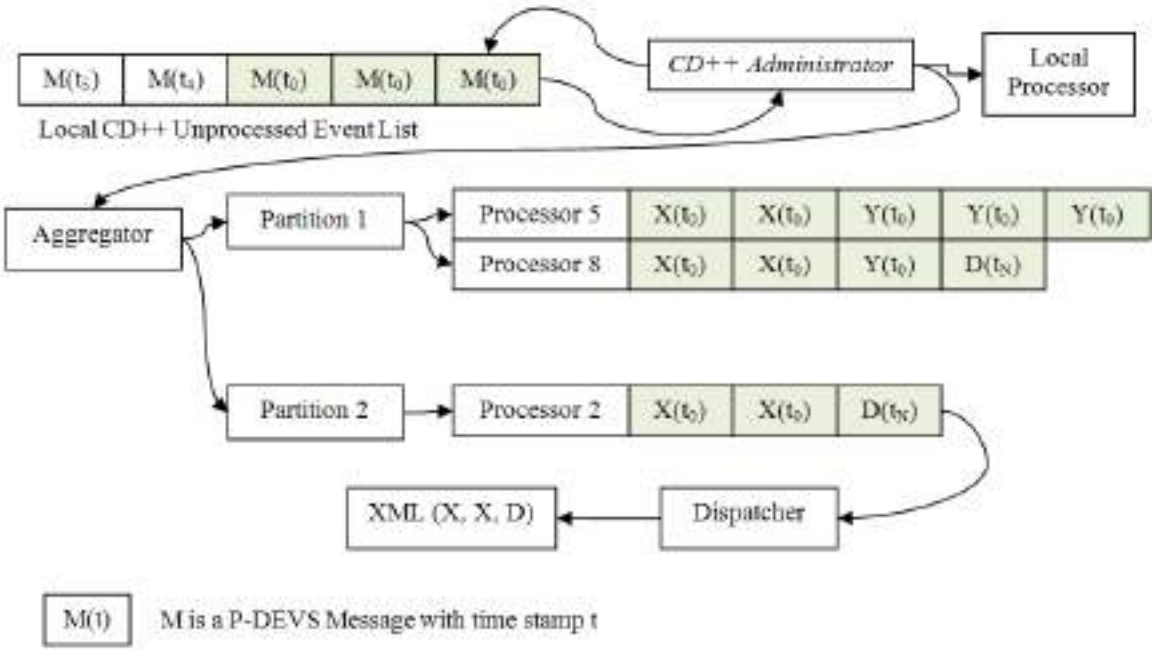

Figure 44: DCD++ Aggregation Message Queues

For example, the Aggregator (Figure 44) opened queues for Partition 1 and Partition 2. In this case, Partition 1 has two queues one for Processor 5 while the other for Processor 8. As shown in the figure, Processor 8 queue is complete since it has already received the synchronized message $D(t_N)$. This means that CD++ is not going to generate any messages to Processor 8. However, Processor 8 messages are not dispatched yet because Processor 5 still has more expected messages. On the other hand, the Dispatcher sends Processor 2 messages to Partition 2, since all of Partition 2 queues have been completed. Consequently, the Dispatcher walks over all messages and sends them in single XML message. Note that all aggregated messages are stamped with the same time since they are generated within the same simulation phase according to the P-DEVS algorithm. On the other hand, the aggregation scheme in Figure 44 needs to answer two possible situations:

1. The Dispatcher may send messages to a partition before other processors (belong to that partition) queues even started. In this case, new queues are built for the other processors and transmitted in the same way. The simulation phases still progress correctly even if multiple XML messages are sent to the same partition. This is because simulation phases are executed per processor rather than per partition.
2. The order of messages is only guaranteed per processor. This is because the Dispatcher packs them, in XML, as they were stored in a processor queue. This is the important part, since the simulation phases are executed per processor rather than per partition.

```
1 <Messages>
2    <MessagesCount>2</MessagesCount>
3     <Message>
4       <MessageType>X</MessageType>
5       <Time>08:50:00:00</Time>
6       <SrcProcId>2</SrcProcId>
7       <PortId>5</PortId>
8       <Value>9.0</Value>
9       <SenderModelId>3</SenderModelId>
10      <DestProcId>1</DestProcId>
11    </Message>
12    <Message>
13      <MessageType>D</MessageType>
14       <Time>08:50:00:00</Time>
15       <SrcProcId>2</SrcProcId>
16       <NextChange>00:00:00:00</NextChange>
17       <SenderModelId>3</SenderModelId>
18       <Proxy>True</Proxy>
19       <DestProcId>1</DestProcId>
20    </Message>
21 </Messages>
```

Figure 45: Aggregating Simultaneous Simulation Messages Together

Figure 45 shows an example of an XML message aggregating two messages. Line #2 specifies the number of the packed messages (2 in this case). Lines 3-11 define the X message while Lines 12-20 define the D message. Messages contain the following

information: Message type (Line #4), simulation time (Line #5), sender processor ID (Line #9), destination port ID (Line #7), content value (Line #8), next change time (Line #16), sender model Id (Line #9), and destination Processor Id (Line #10).

6.3 Chapter Summary

RISE-based Distributed CD++ (DCD++) performs distributed simulation between different CD++ engines where each is placed in a partition and is in charge of simulating a portion of the entire distributed CD++ model. The DCD++ simulation exists in RISE within an experimental framework. In this case, a CD++ engine is built in each partition of the experiment once simulation is started. RISE wraps the CD++ simulation in an experiment partition with a URI. This URI is externally used to reach and communicate with that CD++ engine.

RISE is designed to build local software components as separate OS processes where all communication is performed via OS IPC queues. This applies to the CD++ engine once started within an experiment partition. In this case, a CD++ in a partition forwards all remote messages to its associated simulation manager. This allows a CD++ engine in a partition to progress as if it was simulating the entire model locally, while leaving the distributed environment details to the simulation manager.

The presented synchronization algorithms extends the original Head/proxy algorithms by buffering simultaneous DCD++ messages in a simulation cycle and then transmit all of them in single XML messages. This aggregation reduces remote messages and ensures accurate order arrival since RISE transmits all XML messages concurrently.

CHAPTER 7: RISE PERFORMANCE

Although the main objective of this thesis is the improvement of simulation access and interoperability, performance still matters, particularly because the distributed nature of the simulations. In our case, the RISE middleware manages multiple experiments at the same time where various simulation sessions may be running simultaneously. This requires the RISE middleware to provide workload management for those sessions to manage the workload distributions in order to provide best practical performance for those sessions.

The focus of the tests presented here is to study the middleware sensitivity to concurrent workloads, bottlenecks, and distributed simulation performance. The workload sensitivity study aims on testing the system under pressure. In other words, how the middleware performance is affected when it is required to process increasing workloads. On the other hand, the distributed simulation performance study aims on comparing the performance of the RISE-based and the SOAP-based distributed simulations in similar environment settings. The aim is to compare the performance of the design methodologies based on these two different syntactic and structural interoperability rules.

All of the results presented in the following sections have been collected over a number of different runs to reach at least a 95% Confidence Interval (CI). The CI is calculated as the following [10]: the model follows the normal distribution with mean θ and variance σ^2 and the goal is to estimate θ. The natural estimator of θ is the overall mean of *R* independent replications (runs), that is:

$$\overline{Y} = \frac{1}{R}\sum_{i=1}^{R}\overline{Y_i}$$

Where $\overline{Y}$ is the sample mean (average). However, $\overline{Y}$ is not θ, but an estimate with error (μ**)**. Thus, the usual CI, which assumes the Y_i values are normally distributed, is:

$$\overline{Y} \pm \mu, \quad \text{where} \quad \mu = t_{\alpha/2,R-1}\frac{\sigma}{\sqrt{R}}$$

Here, σ is the standard deviation of all runs while $t_{\alpha/2,R-1}$ is the quantile of the *t* normal distribution with *R-1* degrees of freedom that cuts off α/2 of the area of each tail (with probability α). For example, suppose 120 runs have been repeated with $\overline{Y}$ of 5.80 and σ of 1.6. Since, our objective is a 95% of CI, thus, $t_{0.025, 119} = 1.98$ (this value is based on the normal distribution tables in [10]). Therefore, the CI is 5.80 ± 0.29, since $\mu = \pm 1.98\frac{1.6}{\sqrt{120}}$.

7.1 Concurrent Workload Testing

The results presented in this section aim on studying the middleware overall design sensitivity and any possible bottlenecks because of increasing workload pressure. The workload is the total number of concurrent requests being served by the middleware using DCD++ to execute the simulation tests.

As discussed in Chapter 5, the simulation experiment is exposed as a number of URIs. In this case, once a request (message) is sent to a URI, a thread is started to process that request and generate a response. Once a DCD++ simulation is started, RISE creates a URI to wrap the simulation on that partition. In this case, the CD++ engine is started as a

separate operating system process by RISE, and all communications to/from that process are performed via the operating system IPC queues, as shown in Figure 46.

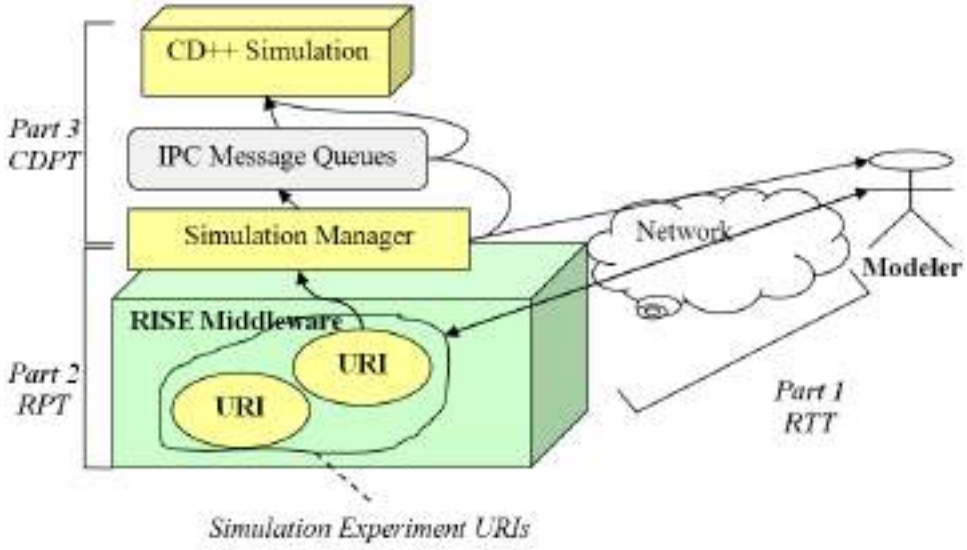

Figure 46: Workloads Performance Test Environment

Figure 46 shows the path of the request message from the Modeler (client) to a URI within a simulation partition. Part 1 indicates the time it takes the request message to travel back and forth through the network, which we call as the message Round Trip Time (RTT). Part 2 indicates the RISE processing time of the received request. Each request message is performed in a separate thread. We call this part as the RISE Processing Time (RPT). Part 3 indicates the time it takes a message to be processed by a simulation manager and sent via the IPC to the CD++ engine process. It further measures the time it takes the CD++ to respond back to the simulation manager, which generates a message back to the modeler. We call this as the CD++ Processing Time (CDPT). There are two types of request messages: messages that travels through Part 1 and Part 2 paths, and messages that travel through Part 1, Part 2, and Part 3.

The following tests view this path as a three-stage process (Figure 47). The tests cases use an increasing workload on these three stages on the path. We collect the RTT, RPT, and CDPT metrics just discussed.

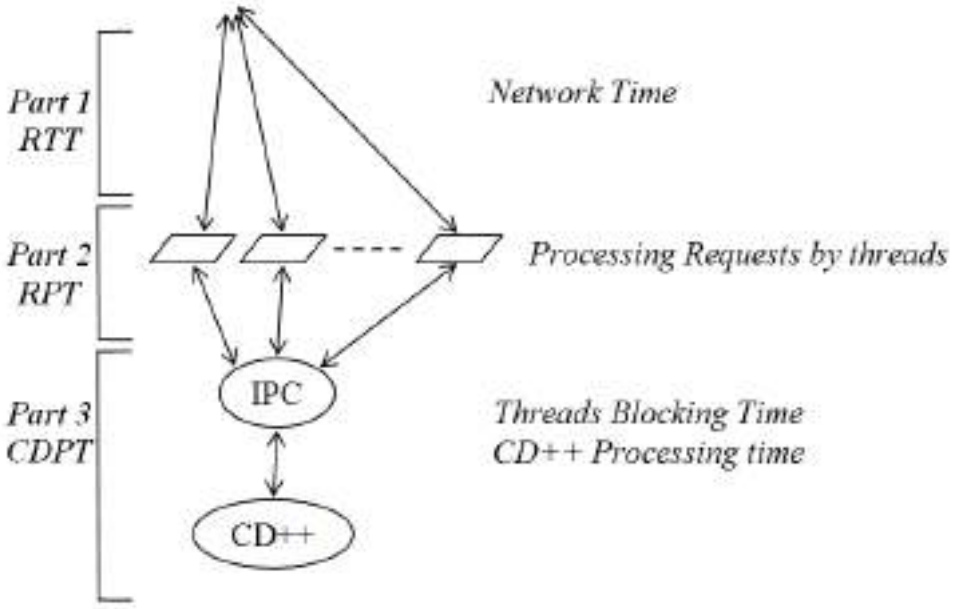

Figure 47: Messages Path Performance Metrics

The requests were sent (from separate computers) during active simulation to the CD++ URI (*DCDpp/{framework}/simulation*), allowing the requests to travel through all of the path parts shown in Figure 47. The tests were repeated using different workloads. The workload is the number of requests was concurrently handled by the middleware. Note that we considered requests are handled concurrently, if at least 99% of those requests (in a workload) arrived at the middleware within the duration of one second (i.e. the arrival rate is the number of requests in a workload per one second). This is because it is difficult in practice to transmit a number of messages through the network and ensures their simultaneous arrival (particularly on the Internet due to the usual variation in messages propagation delay). Further, each workload test was also repeated over a

number of runs (usually 50 or more) to achieve a Confidence Interval (CI) of at least 95%.

The tests used the following workloads: 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 500, and 5000 concurrent requests. It is worth to note that our objective is to study the middleware sensitivity when increasing the load pressure. In other words, our objective is to study the system behavior under heavy load, rather than making the decision of the system being fast or slow. However, because of the finite processing power of the hardware and software, there will be a point when the server performance becomes degraded. However, this type of decisions is usually based on monitoring the system over a long period during normal use operations. On the other hand, our workload testing here represents the worst-case scenario since the middleware is required to handle various requests at the same time (using certain physical machine processing power).

The presented tests have been conducted within two testing environments (This provides two different networks setups and machine processing power capabilities):

- In the first environment, the requests are sent from a computer on the same LAN as of the RISE machine. RISE is installed on a machine with dual processors at 2.33 GHz with RAM of 4.0 GB.
- In the second environment, both machines are on the Internet (within the Ottawa area). RISE is installed on a machine with dual processors at 2.66 GHz with RAM of 8.0 GB.

Table 4 shows the results of the three metrics with different workloads in the first environment. Each metric shows the average ($\overline{Y}$) of all repeated runs of a certain

workload. It also shows the standard deviation (σ) of all repeated runs, and the 95 % CI marginal error (μ).

Table 4: First Environment Message Processing Time Paths Results

Workloads	RPT (ms)			CDPT (ms)			RTT (ms)		
	$\bar{Y}$	σ	±μ	$\bar{Y}$	σ	±μ	$\bar{Y}$	σ	±μ
1	3	0.45	0.09	8	0.18	0.04	5.0	0.81	0.16
10	5	0.65	0.13	10	0.60	0.12	5.4	0.85	0.17
20	6	0.67	0.13	13	0.65	0.13	5.0	0.91	0.18
30	6	0.68	0.13	15	0.64	0.13	5.2	0.98	0.19
40	8	0.70	0.14	14	0.91	0.18	5.0	0.93	0.18
50	6	0.85	0.17	14	0.81	0.16	5.1	0.85	0.17
60	7	0.71	0.14	15	0.76	0.15	5.2	0.90	0.18
70	7	0.69	0.20	17	0.69	0.20	5.0	0.96	0.27
80	8	0.70	0.14	17	0.79	0.16	5.0	0.87	0.17
90	9	0.69	0.14	18	0.91	0.18	5.0	0.87	0.17
100	9	0.65	0.13	19	0.86	0.17	5.3	0.84	0.17
150	12	0.64	0.13	22	0.79	0.16	5.0	0.81	0.16
200	15	0.73	0.14	26	0.93	0.18	5.0	0.88	0.17
500	21	2.10	0.42	39	4.05	0.80	5.0	0.09	0.02
5000	54	9.51	1.88	81	5.13	1.02	5.0	0.07	0.01

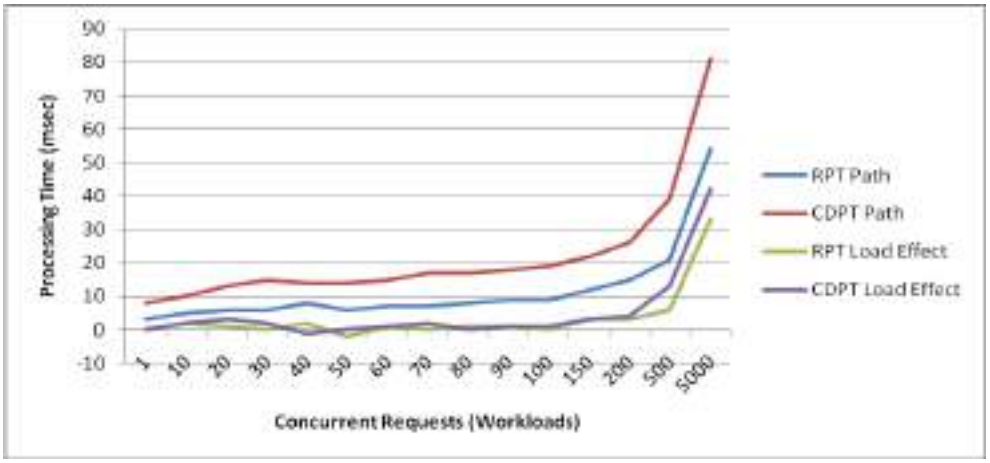

Figure 48: RPT and CDPT Averages for First Environment Setup

The RPT and CDPT metrics $\bar{Y}$ values are plotted in Figure 48. The figure shows that each request processing time increases as the workload increases (RPT Path and CDPT path graphs). This is expected since the machine is performing more work. The other two graphs (RPT Load Path and CDPT Load Effect) aim on measuring the

increased load effect. This effect is the difference between the current load and the previous load results.

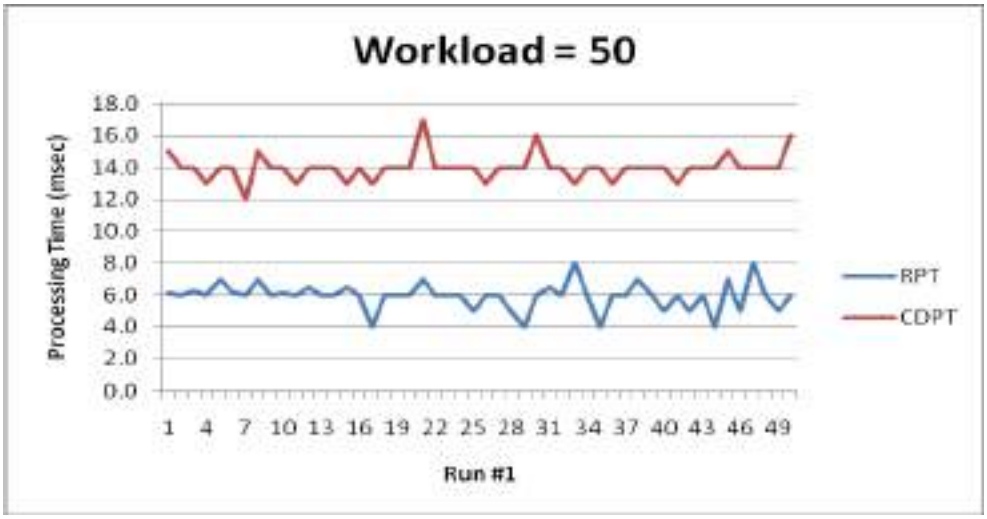

Figure 49: All Runs for the First Environment at Workload 50

As Figure 48 shows, these two graphs tend to stay constant, which indicates that the middleware is stable; and it supports increasing workloads. However, the RPT is increased by 2.57 times, and the CDPT is increased by 2.07 times when the workload is increased from 500 to 5000. Even though this is a sharp jump, it is still reasonable because the workload is increased by 10 times. Thus, those results show that increasing workloads lead to more time it takes to process requests. However, the middleware manage to make the best of the available processing power to serve all requests fairly. This observation is shown in Table 4, since σ is maintained low. This means that various runs results are served almost with the same processing time. To illustrate this point further, Figure 49 shows a detailed plot of all the 50 runs performed in the first environment. In this figure, the RPT ranges between 4 to 8 ms with CI = 6±0.17, while the CDPT ranges between 12 to 17 ms with CI = 14±0.16. However, the minimum and maximum values do not tell the whole picture. These tell us that there is a spike in the

graph, which is expected since the physical machine is also busy performing other computations such as the CD++ engine. However, the results plot in Figure 49 shows that the processing times maintained similar to each other (with few spikes).

Table 5: Second Environment Message Processing Time Paths Results

Workloads	RPT (ms)			CDPT (ms)			RTT (ms)		
	$\overline{Y}$	σ	±μ	$\overline{Y}$	σ	±μ	$\overline{Y}$	σ	±μ
1	2	0.75	0.21	4	0.81	0.23	101	7.10	1.41
10	4	0.95	0.27	3	0.84	0.24	103	7.50	1.49
20	5	0.65	0.18	4	0.68	0.19	105	7.90	1.56
30	4	0.79	0.22	6	1.01	0.29	98	10.10	2.00
40	5	0.29	0.08	6	0.89	0.25	99	7.40	1.47
50	5	0.28	0.08	5	0.65	0.18	100	13.70	2.71
60	6	0.39	0.11	8	0.89	0.25	100	6.50	1.29
70	6	0.55	0.16	9	0.76	0.21	101	7.10	2.01
80	6	0.61	0.17	11	1.80	0.51	102	8.70	1.72
90	7	0.50	0.14	11	1.02	0.29	103	12.30	2.44
100	7	0.48	0.14	12	1.86	0.53	109	7.60	1.50
150	9	0.39	0.11	14	2.01	0.57	101	7.90	1.56
200	11	0.51	0.14	16	2.12	0.60	104	6.40	1.27
500	15	0.21	0.06	21	3.01	0.85	102	5.01	0.99
5000	24	0.52	0.15	31	3.12	0.88	102	4.91	0.97

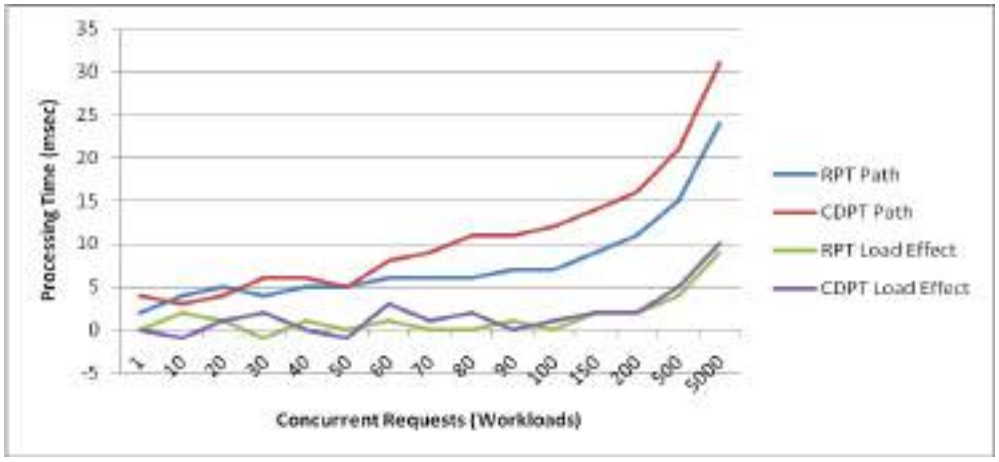

Figure 50: Second Environment Setup RPT and CDPT Metrics

The second environment tests are shown in Table 5. The RPT and CDPT metrics $\overline{Y}$ values are plotted in Figure 50. This second set of results show that the CDPT and the RPT values support an increasing workload as in the case of the first environment. The

results also show the high communication overhead represented with the RTT metric. This overhead indicates that most of the time message are in the network. Therefore, it is common sense to target the remote messages to improve performance, reducing the need to transmit many remote messages, and/or by using high-speed network lines.

7.2 Distributed Simulation Performance

This section aims on testing the distributed simulation performance via RISE. Our approach here is to compare the RISE-based DCD++ (discussed in Chapter 6) against the SOAP-based DCD++ (discussed in [137]) over a number of different CD++ models using different experiment environment settings. In this case, the only difference between both systems is the design methodology, based on the RESTful and SOAP-based WS.

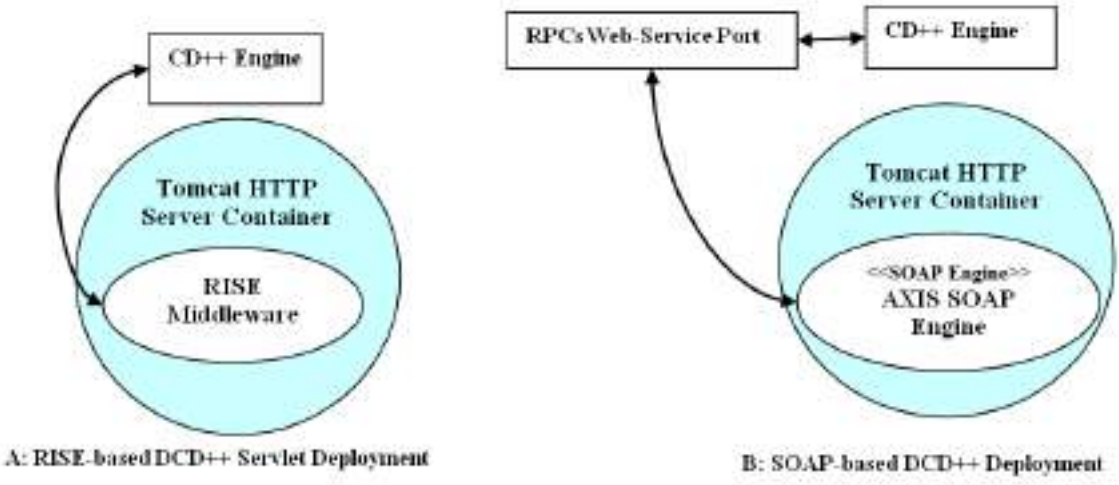

Figure 51: RISE-based and SOAP-based Deployment on a Machine

Both systems are shown in Figure 51. Figure 51-A shows the RISE middleware running as a Servlet (i.e. a program running within an HTTP container) inside the Apache Tomcat HTTP server container [7]. In this case, Tomcat forwards all of the HTTP messages to RISE while RISE processes the messages according to their destination

URIs. In the RISE-based distributed simulation experiments, those URIs interface a partition on a machine (where the CD++ engine is reached by one of those URIs). In these tests, the RISE middleware is deployed as a Servlet within a Tomcat HTTP server. Such deployment is similar to the SOAP-based DCD++, as shown in Figure 51. However, RISE can also be deployed as a standalone HTTP server (similar to Tomcat in Figure 51), as discussed in Appendix-A. This standalone deployment is beneficial, particularly when testing distributed systems. For example, many instances of RISE can be executed on the same physical machine (each having its own TCP port). In this case, the distributed simulation is partitioned and tested without the need of deploying RISE on many physical machines. Figure 51-B shows the Apache AXIS SOAP engine [145] running as a Servlet inside Apache Tomcat. The SOAP engines translate RPCs to SOAP messages and vice versa, hence they implement the SOAP standards. In this case, Tomcat forwards the HTTP messages to the AXIS engine, which translates the contained SOAP message into a local procedure call in the RPCs Web-service port, which then communicates the passed-in parameters to the CD++ engine.

The experiments in this section used the following five CD++ models. Note that each cell in a Cell-DEVS model represents an atomic DEVS model (see [33] for more details):

- *Barbershop* is a DEVS model simulates a retail barbershop store activities. In this model, customers arrive to the store and have their hair cut by the available barber in First Come First Serve (FCFS) order. The overall model consists of an atomic model (called *Reception*, which simulates the customers arrival and waiting at the receptionist desk), and a coupled model (called *Barber*, which simulates a barber

activity). The *Barber* model consists of two atomic models: *CheckHair* (to simulate the barber's job time estimate based on the customer request) and *CutHair* (to simulate the barber actual work of cutting the hair such as arms movement, holding scissors, etc). In this case, for example, once a customer leaves the *CutHair* model, it signals the *Reception* model. As a result, the *Reception* model sends a waiting customer to the *CheckHair* atomic model (within the Barber coupled model).

- *Fire* is a 2-D 30x30 Cell-DEVS model used to simulate forest fire. The simulation allows foreseeing the propagation and intensity of the fire. Three parameters are involved in the ratio of spread: (1) particles properties (amount of heat, minerals and density), (2) type of fuel (includes the size of the vegetation) and (3) values involved with the natural environment (wind speed, territory inclination and humidity). Each cell's rules only influence its neighborhood, which is defined as one cell in each direction (of each specific cell).
- *Ship* evacuation is a 2-D 49x27 Cell-DEVS model used to simulate the evacuation of a ship in an emergency. This model has two phases. In Phase 1, each cell calculates its shortest path toward the exit. In Phase 2, people run in their initial direction until they encounter another person or an obstacle (e.g. wall). In this model, a person usually tends to follow the direction of other people groups. The simulation is completed once all persons leave the ship. Each cell's rules only influence its neighborhood where the neighborhood is defined as 11 cells (i.e. a cell in each direction from each specific cell, but there are two cells in the upper and right directions).

- *Cancer* is a 2-D 20x20 Cell-DEVS model is used study cancer spreading on different types of tissue. In this model, cancer tumors invade normal tissues (replacing healthy cells with cancer cells) until cancer is spread over other parts of the body. Each cell's rules only influence its neighborhood, which is defined as two cells in each direction (of each specific cell).
- *Battlefield* is 3-D 10x10x6 Cell-DEVS model used to simulate a battle between two armies trying to capture the other flag. This model also simulates different activities such as soldiers' injuries, deaths, movements, and fight engagements. Each cell's rules only influence its neighborhood, which is defined as two to four cells in each direction (of each specific cell).

The presented results were conducted using three different distributed environment setups, as shown in Table 6. In each setup, each model is partitioned into two or more partitions where each partition is assigned to a machine; hence, each partition is simulated by a single CD++ engine on a machine. This allows each run of simulation experiments performance metrics to be collected independently. Note that the RISE-based DCD++ model partitioning mechanism is discussed in detail in Section 6.1 while the SOAP-based DCD++ partitioning is discussed in [137]. Note further that the CD++ models under simulation are partitioned in the same way for both systems in all test runs. This means that the same models portions are assigned to the same machines. This is part of the requirement to ensure fair comparison between the two systems. Therefore, the complexity of the models and the assignment of models portions to

machines are irrelevant in our tests. This is because our objective is to compare these two systems with the same experiments setup.

Table 6: Test Environments Settings

Environment	Machines Geographical Locations	No. of Partitions	Machines Physical Capabilities (A partition on each machine)
First	All machines attached to the same Ethernet.	2	• dual processors at 2.33 GHz with RAM of 4.0 GB • four processors at 2.66 GHz with RAM of 3.9 GB
		3	• dual processors at 2.33 GHz with RAM of 4.0 GB • four processors at 2.66 GHz with RAM of 3.9 GB • dual processors at 3.0 GHz with RAM of 3.0 GB
		4	• dual processors at 2.33 GHz with RAM of 4.0 GB • four processors at 2.66 GHz with RAM of 3.9 GB • dual processors at 3.0 GHz with RAM of 3.0 GB • dual processors at 3.0 GHz with RAM of 2.0 GB
Second	Placed Machines at different locations within the city of Ottawa, Canada	2	• dual processors at 2.33 GHz with RAM of 4.0 GB • dual processors at 2.40 GHz with RAM of 6.0 GB
		3	• dual processors at 2.33 GHz with RAM of 4.0 GB • dual processors at 2.40 GHz with RAM of 6.0 GB • dual processors at 2.13 GHz with RAM of 2.0 GB
Third	One machine in Ottawa, Canada; the other in Amman, Jordan.	2	• dual processors at 2.33 GHz with RAM of 4.0 GB • four processors at 2.66 GHz with RAM of 3.9 GB

Both systems were compared by setting up the same simulation experiments where simulation is repeated alternately between both systems over a number of runs.

The same experiment setup means that the CD++ model under simulation is partitioned in the same way over the same physical machines. The performance results were then collected based on the following metrics:

- Number of Remote Messages (NRM) exchanged in a simulation run. This counts the messages that travel through the network within HTTP envelopes. The NRM is the same over the same experiment setup using the same system, because the simulation always executes the same events (deterministic simulation). However, because the RISE-based DCD++ system aggregates remote messages in XML (Chapter 6), the NRM values are usually different between RISE-based and SOAP-based systems.
- Total Execution Time (TET) is the average time to complete the simulation of an experiment. The simulation is repeated over a number of runs (usually 25 or more) to achieve at least a 95% Confidence Interval (CI).
- Middleware Processing Time (MPT) is the average time it takes a received message at a machine to reach the simulation (with at least 95% CI). In the words, the time it takes a message to go through the different software components. As shown in Figure 52, the SOAP-based MPT is the time it takes a received HTTP message by Tomcat to travel through the AXIS engine (SOAP processing) until the subject procedure is invoked at the WS port. The RISE-based system MPT is the time it takes a received HTTP message by Tomcat to travel through the RISE middleware (XML processing) until the received simulation information is forwarded to CD++.

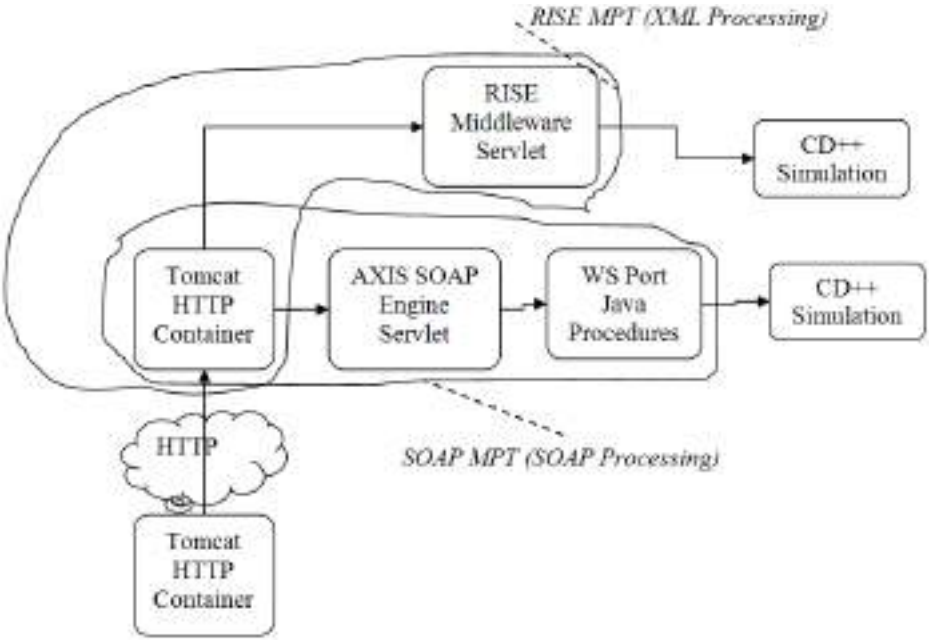

Figure 52: Middleware Processing Time (MPT) on a Partition

The Middleware Processing Time (MPT) results are shown in Table 7. Because the messages received in a partition follows the same processing path at all times, the MPT is calculated over all received messages of all experiments simulation runs. Note that the partition machine used to collect the results (in Table 7) is a dual-processor at 2.33 GHz with RAM of 4.0 GB. Many factors have contributed to these results such as RISE multithreading scheme and the XML parsing and handling. Further, the software implementations of the components chain in Figure 52 such as the AXIS handling of the received SOAP messages. Other contributing factor is the way information is received by the main partition during simulation. As discussed in Chapter 6, synchronization messages are exchanged during simulation cycles. In this case, if the partition receives few messages during a cycle, it then processes them quickly. On the other hand, if it needs to process many messages at the same time in certain cycles, it then needs more time to process them.

Table 7: Comparing MPT Results for All Testing Environments

Test Environment (Table 6)	RISE MPT (ms)		SOAP MPT (ms)	
	CI = $\overline{Y} \pm \mu$	σ	CI = $\overline{Y} \pm \mu$	σ
First Environment	3.02±0.03	0.091	9.46±0.04	0.127
Second Environment	2.95±0.02	0.085	6.27±0.03	0.115
Third Environment	3.2±0.02	0.088	10.76±0.03	0.114

Table 8 shows the *Number of Remote Messages* (NRM) to complete a simulation run for all models with different partitions. The table shows the number of partitions used in the experiment, the RISE NRM, the SOAP NRM, and the *RISE Aggregation Effect* (RAA). The RAA was calculated as follows: *RAA = (SOAP NRM ÷ RISE NRM)*. Thus, the RAA value measures how much remote messages have been reduced via aggregation. Note that the NRM stays the same of each simulation run for the same experiment setup. On the other hand, the NRM results show the effect of aggregating simultaneous events in XML messages by RISE. The RAA value in Table 8 varies from a model simulation to another or from a setup (e.g. partitions) to another. This is because the DCD++ synchronization algorithms only aggregate the simultaneous simulation events (i.e. events executed at the same simulation cycle), which vary from a simulation setup to another. For example, the aggregation is able to reduce remote messages by 1.3 and 1.95 times for the *Barbershop* model with 2 and 3 partitions respectively. On the other hand, the aggregation is able to reduce remote messages by 10.30, 11.53 and 14.89 times for the *Battlefield* model with 2, 3, and 4 partitions respectively. The presented results also show that the NRM values change when changing the number of partitions in the simulation environment for the same model. This is mainly because the more partitions, the more remote synchronization messages are required. For example, some of the local messages

in a partition might become remote messages when this partition is repartitioned further into more partitions.

Table 8: Comparing NRM Values between RISE-based and SOAP-based DCD++

Model	No. Machines (Partition per machine)	RISE NRM	SOAP NRM	RAA = (SOAP NRM ÷ RISE NRM)
Barbershop	**2**	**661**	**861**	**1.30**
	3	**745**	**1451**	**1.95**
Fire	**2**	**1676**	**1796**	**1.07**
	3	**1915**	**2540**	**1.33**
	4	**2198**	**3891**	**1.77**
Ship	**2**	**1266**	**3166**	**2.50**
	3	**1911**	**3861**	**2.02**
	4	**2172**	**4994**	**2.30**
Cancer	**2**	**12**	**192**	**16.00**
	3	**18**	**378**	**21.00**
	4	**58**	**656**	**11.31**
Battlefield	**2**	**86**	**886**	**10.30**
	3	**156**	**1798**	**11.53**
	4	**208**	**3098**	**14.89**

The following tables (Table 9, Table 10, and Table 11) show the simulation Total Execution Time (TET) results obtained for the first, second, and third environment settings respectively (see Table 6). The tables results show the model used in the simulation, the number of partitions applied to the model where each partition is assigned to a machine, and both RISE and SOAP TET averages of all repeated runs (in seconds). In this case, the CI (with 95%) is presented in addition to the standard deviation (σ) of all repeated runs. The tables also show the *RISE Speedup*, which is calculated as *SOAP TET ÷ RISE TET*. The tables further show the *New Partitions Effect* for both systems, which is calculated as *(Current Partition TET – Previous Partition TET) ÷ Current Partition TET*.

Table 9: First Environment Test Setting TET Results

Model	No. of Partitions	RISE TET (seconds)		SOAP TET (seconds)		RISE Speedup	New Partitions Effect	
		CI = $\overline{Y} \pm \mu$	σ	CI = $\overline{Y} \pm \mu$	σ		RISE	SOAP
Barbershop	2	10.21±0.44	1.07	13.41±0.65	1.57	1.31		
	3	31.47±0.72	1.74	49.10±0.79	1.91	1.56	2.08	2.66
Fire	2	10.89±0.33	0.81	20.06±0.87	2.11	1.84		
	3	33.30±0.70	1.70	147.01±3.13	7.60	4.42	2.05	6.33
	4	81.47±0.72	1.75	539.51±7.79	18.90	6.62	1.45	2.67
Ship	2	26.87±0.42	1.02	47.02±1.90	4.60	1.75		
	3	31.15±0.82	1.98	59.65±1.31	3.18	1.91	0.16	0.27
	4	71.23±0.31	0.75	247.12±2.51	6.10	3.47	1.29	3.14
Cancer	2	13.12±0.40	0.98	31.98±0.87	2.11	2.44		
	3	9.54±0.42	1.01	51.78±1.57	3.81	5.43	-0.27	0.62
	4	18.19±0.70	1.71	109.74±3.90	9.46	6.03	0.91	1.12
Battlefield	2	6.29±0.20	0.49	380.84±7.09	17.21	60.55		
	3	9.23±0.30	0.72	557.47±8.00	19.42	60.40	0.47	0.46
	4	13.11±0.39	0.95	854.17±8.90	21.60	65.15	0.42	0.53

Table 10: Second Environment Test Setting TET Results

Model	No. of Partitions	RISE TET (seconds)		SOAP TET (seconds)		RISE Speedup	New Partitions Effect	
		CI = $\overline{Y} \pm \mu$	σ	CI = $\overline{Y} \pm \mu$	σ		RISE	SOAP
Barbershop	2	13.54±0.50	1.21	30.70±0.98	2.37	2.27		
	3	37.76±1.19	2.89	86.12±2.02	4.91	2.28	1.79	1.81
Fire	2	16.10±0.50	1.21	29.45±1.36	3.31	1.83		
	3	41.27±0.82	1.98	194.05±5.52	13.41	4.70	1.56	5.59
Ship	2	34.34±0.72	1.75	73.00±1.75	4.24	2.13		
	3	46.79±0.83	2.01	144.10±4.69	11.39	3.08	0.36	0.97
Cancer	2	14.21±0.46	1.11	48.40±1.72	4.18	3.41		
	3	13.56±0.51	1.23	109.64±3.75	9.10	8.09	-0.05	1.27
Battlefield	2	09.12±0.43	1.05	532.45±11.95	29.01	58.38		
	3	15.87±0.59	1.42	941.78±13.93	33.80	59.34	0.74	0.77

Table 11: Third Environment Test Setting TET Results

Model	No. of Partitions	RISE TET (seconds)		SOAP TET (seconds)		RISE Speedup
		CI = $\overline{Y} \pm \mu$	σ	CI = $\overline{Y} \pm \mu$	σ	
Barbershop	**2**	**16.25±0.73**	**1.76**	**41.45±1.69**	**4.11**	**2.55**
Fire	**2**	**27.10±1.20**	**2.91**	**109.45±4.63**	**11.23**	**4.04**
Ship	**2**	**40.83±1.24**	**3.00**	**96.36±4.25**	**10.31**	**2.36**
Cancer	**2**	**16.35±0.70**	**1.70**	**70.47±3.63**	**8.80**	**4.31**
Battlefield	**2**	**10.72±0.46**	**1.11**	**805.06±29.35**	**71.24**	**75.10**

The TET results clearly show substantial performance improvement for DCD++ simulation via RISE comparing to using SOAP-based WS regardless of the used model or environment setup. For example, in the first environment settings results (Table 9), *RISE Speedup* ranges from 1.31 to 60.55 times (with 2 partitions setup), 1.56 to 60.40 times (with 3 partitions setup), and 6.62 to 65.15 times (with 4 partitions). Further, the second and third environments showed similar results as shown in Table 10 and Table 11 respectively. These results compare both systems implementations, including distributed simulation algorithms. As previously mentioned, we tried to make this compression as far as possible by deploying both systems in Tomcat, by using exactly the same CD++ engines, and the same physical machines. However, we are still using the AXIS engine to process the SOAP messages (not needed in RISE), since we need software to realize the SOAP standards. We further tried to introduce multithreading to the SOAP system by invoking each RPC stub within a thread. This not only proved to be difficult because of the way RPC stubs are structured in AXIS, but also because it requires rebuilding the entire DCD++ WS component. This new implementation would also need to aggregate simulation messages to ensure accurate simulation because of the reasons discussed in Chapter 6. Consequently, this message aggregation also needs to be sent via the RPC as

an array (which proved to be nontrivial to implement), or within an XML message (sent as an attachment). These solutions depend on the use of AXIS; therefore, replacing AXIS with a different vendor implementation would affect our WS solutions.

The above issues show one of the REST major contributions: REST does not place restrictions on implementations, allowing the programmers to introduce their ideas freely. In REST (which is the Web method), senders build a message and transmit to a URI on the Web, according to the Web standards (without restricting implementation to a certain style). On the other hand, the use of SOAP-based WS is tied to existing software implementations, which usually makes it difficult to introduce new improvements without major software changes.

The TET results also showed that adding more partitions into the simulation has slowed down the simulation in most cases, but the effect on RISE was much less. For example, adding a third partition to the *Ship* model simulation (for the second environment shown in Table 10) slowed down the simulation by 0.36 and 0.97 for the RISE and SOAP systems respectively. This simulation slowdown is mainly because the communication synchronization overhead is larger than the local computation overhead by machines. However, few cases showed that adding another partition (machine) sped up the simulation. For example, adding a third partition for the cancer model has speeded up the simulation a little bit in the RISE-based simulation as shown in Table 10 and Table 11. This means that the synchronization overhead is not large enough to overweigh the benefits of adding more computation power via splitting the model between more machines. Particularly that the Cancer model was able to reduce the NRM value via aggregation by a factor of 21 times as shown in Table 8. Thus, speeding up simulation is

possible in distributed simulation but it depends on the environment and the model under simulation characteristics.

Figure 53, Figure 54, and Figure 55 shows all repeated simulation runs for both systems using the first, second, and third testing environments respectively.

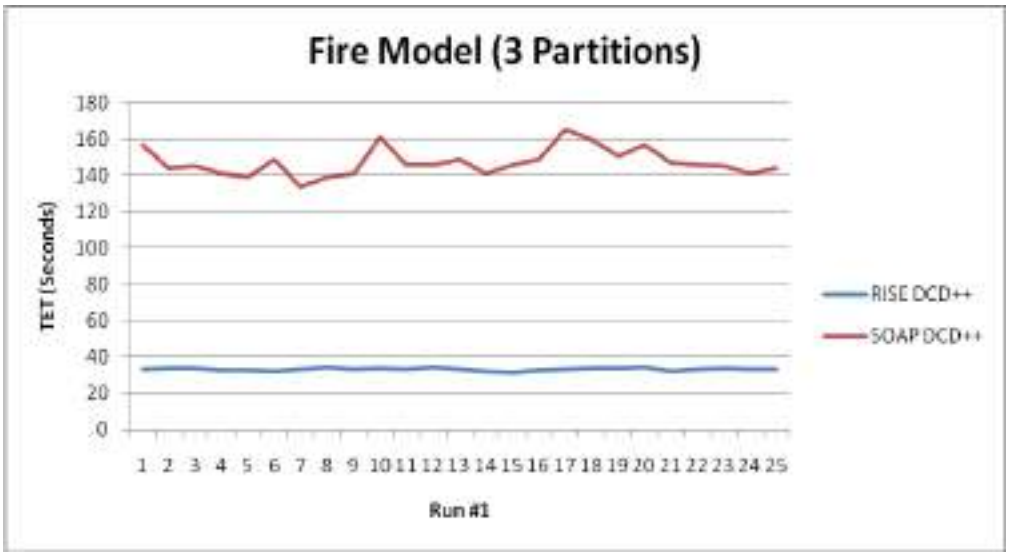

Figure 53: First Environment All Runs Example

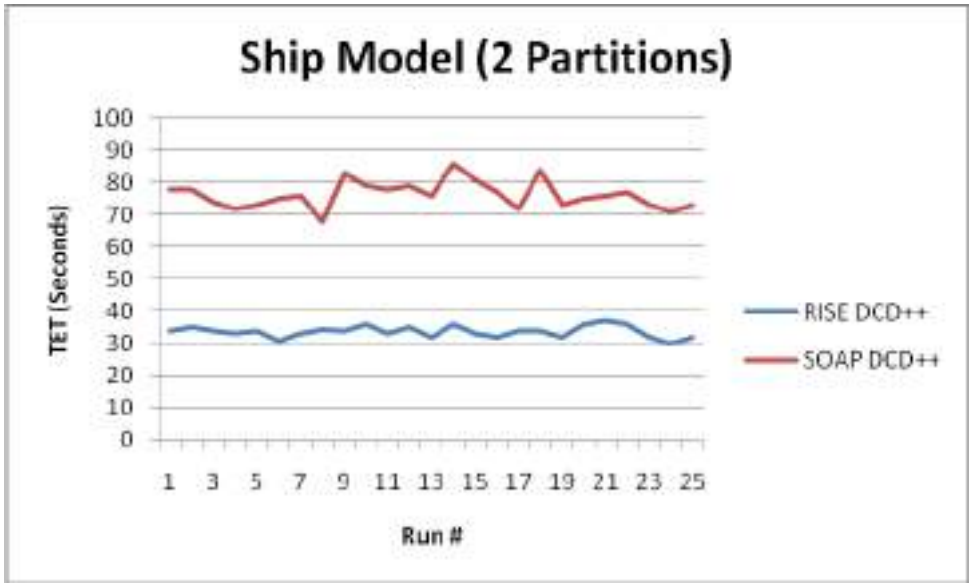

Figure 54: Second Environment All Runs Example

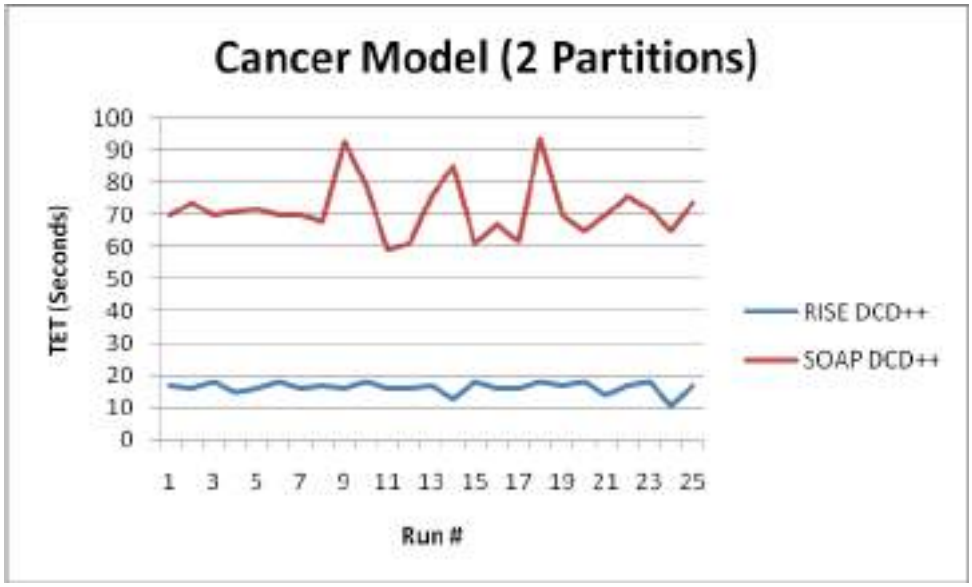

Figure 55: Third Environment All Runs Example

Figure 53 plots the TET results of the *Fire* model with 3 partitions setup. Figure 54 plots the *Ship* model with 2 partitions setup. The results show that obtained values vary from a run to another. This is mainly because of the status of the network during a specific simulation run.

The presented TET results so far obtained for simulating a CD++ model by itself on the participant machines. However, in practice this is usually not the case, since a modeler may run several models concurrently within an experiment alongside other experiments. To study such case, the results, shown in Figure 56, aim on studying concurrency on both systems. These tests were conducted within the first test environment with the two-partition setup (Table 6). The tests start by simulating one ship evacuation model, then two of them simultaneously, and so on. The results (Figure 56) show that the more simulated models are added to the simulation the larger the gap it gets between both systems. In particular when simulating six concurrent models, RISE executes each model with 39 seconds while the SOAP-based system executes it with 179

seconds. This gap is clearly widened comparing to executing a single model by itself. Note that all TET results are averaged over various runs to achieve a 95% CI.

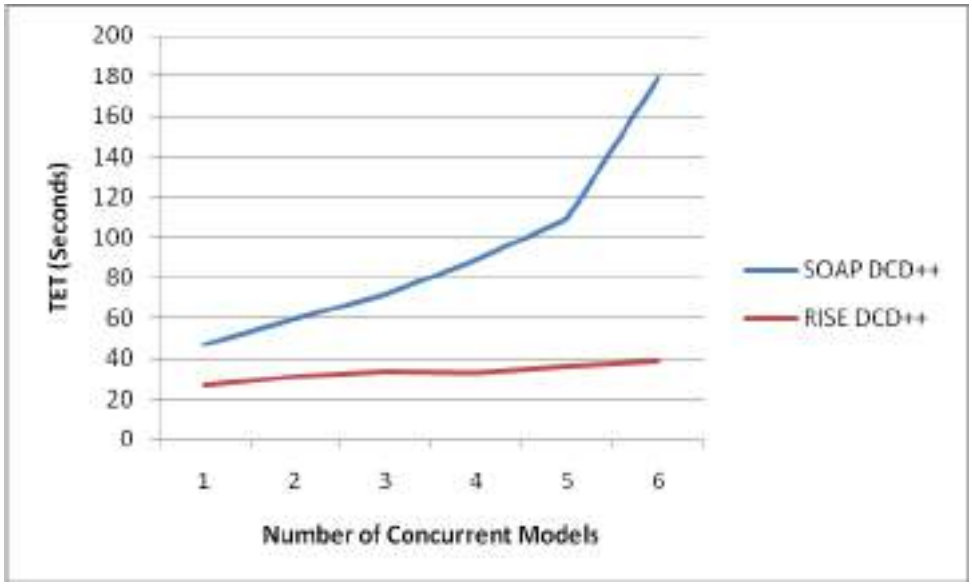

Figure 56: Executing Multiple Ship Models Simultaneously

Note that our presented results in [2][3] showed slight improvment of the RISE-based DCD++ simulation over the SOAP-based simulation. Those results were collected while deploying RISE as a standalone HTTP server (RISE types of deployments are described in Appendix-A). As part of our testing of the RISE middleware implementation, it was found that the Simple-framework [122], which is used by RISE standalone deployment to carryout HTTP connections, has some performance issues. To fix this problem, the RISE middleware was modified so that it can be deployed as a Servlet in an HTTP container.

7.3 Chapter Summary

The main theme of this thesis is to enhance simulation access and interoperability via the Web. However, performance still matters since distributed simulation is usually performed on the Web over broad geographical area. Thus, communication overhead forms a performance hot spot in synchronization algorithms that need to be design carefully. Further, the RISE middleware is accepted to manage different simulation experiments at the same time. Therefore, its ability to handle multiple simulation services workload simultaneously would affect the performance of each one of them.

The performance tests in this chapter have two objectives: (1) they aimed on testing the overall design sensitivity to increasing workload and possible bottleneck spots, and (2) they aimed on studying the distributed simulation performance via RISE.

The distributed simulation performance testing approach in this chapter is to compare the RISE-based DCD++ (discussed in Chapter 6) against the SOAP-based DCD++ (discussed in [137]) when put in a similar testing environments. Thus, the only difference of both systems is the use of the interoperability syntactic and structural rules, which led to different design methodologies, as argued in Chapter 3.

The workload tests was studied via break requests messages path into three parts: network time, RISE processing time, the CD++ processing time. Afterward, concurrent requests were pumped into this path at various workloads.

The distributed simulation performance testing approach in this chapter is to compare the RISE-based DCD++ (discussed in Chapter 6) against the SOAP-based DCD++ (discussed in [137]) when put in a similar testing environments. Thus, the only difference of both systems is the use of the interoperability syntactic and structural rules,

which led to different design methodologies, as argued in Chapter 3. Our used performance metrics in these tests are: **(1)** the Total Execution Time (TET) to complete simulation, **(2)** Number of Remote Messages (NRM) in simulation, and **(3)** the Middleware Processing Time (MPT), which is the time it takes a received message at a machine to reach the simulation engine.

CHAPTER 8: RISE MIDDLEWARE APPLICATIONS

This chapter shows how to apply the different methods introduced in previous chapters for other applications (besides DCD++ distributed simulation). To do so, this chapter first presents additional distributed simulation algorithms built with RISE (Section 8.1), and it then shows how RISE could improve simulation experimentation via the use of workflows (Section 8.2).

The algorithms presented in Section 8.1 aim on interoperating independent-developed simulation services (which could be used as a feasible proposal for DEVS standardization [140][141][142][143]). To do so, algorithms must be decoupled from software design and implementation. In this case, the algorithms place models in each partition as black boxes interconnected with other models via input/output ports. The simulation is executed in cycles where all exchanged synchronization messages are described in XML. These algorithms are also extended to handle dynamic simulation, where simulation partitions can join/disjoin during runtime.

This chapter also presents (in Section 8.2) the design of a workflow component (on the client side) that could be used to automate the steps been taken by modelers to create and manipulate experiments (which can be easily implemented on the RISE middleware). Such component would serve as means for automation, repeatability, controlling processes and management (i.e. be part of a formal Business Process Management (BPM) [146]).

8.1 Distributed Simulation Algorithms

This section describes distributed simulation algorithms in terms of interconnecting models in different domains and simulation synchronization. This proposal is part of the ongoing efforts to interoperate different DEVS-based simulation tools. Thus, our objective here is to keep algorithms as simple as possible without placing any implementation restrictions. This is because DEVS-based implementations are highly diverse, in spite of implementing the same DEVS formalism. In this case, programmers should answer the question “how to implement” the algorithms in their systems. On the other hand, the algorithms are expected systems to be able to exchange XML messages in HTTP envelopes.

On the other hand, the “how to implement” question should be an internal design issue. This is one of the principles behind the RISE middleware. Therefore, synchronization algorithms can be implemented as rules in software that coordinate their internal activities with remote systems via messages (XML in our case) and the software can implement different synchronization algorithms, triggering different XML messages accordingly. This type of separation makes algorithms enhancements evolve independently of each other.

This section is organized as follows: Section 8.1.1 describes the models interconnections via their input/output ports while Section 8.1.2 describes the simulation synchronization.

8.1.1 Interconnecting Models Partitions

Each simulation model needs to be placed in a simulation environment domain (partition) that is capable of executing that model. To do so, we treat all models as black boxes with input and output ports and the modeler needs to decide how those models need to influence each other via those ports. Specifically, modelers need to connect the appropriate input ports to the appropriate output ports. For example, in the setup shown in Figure 57, Model-2 influences Model-1 via Port-2 and Port-3. Model-3 influences both Model-1 and Model-2 via Port-1. This implies that those models output ports can generate external messages to their correspondent input ports during simulation, hence influence recipient models. Note that Port-1 and Port-3 of Model-3 and Port-4 of Model-1 are not connected to other models.

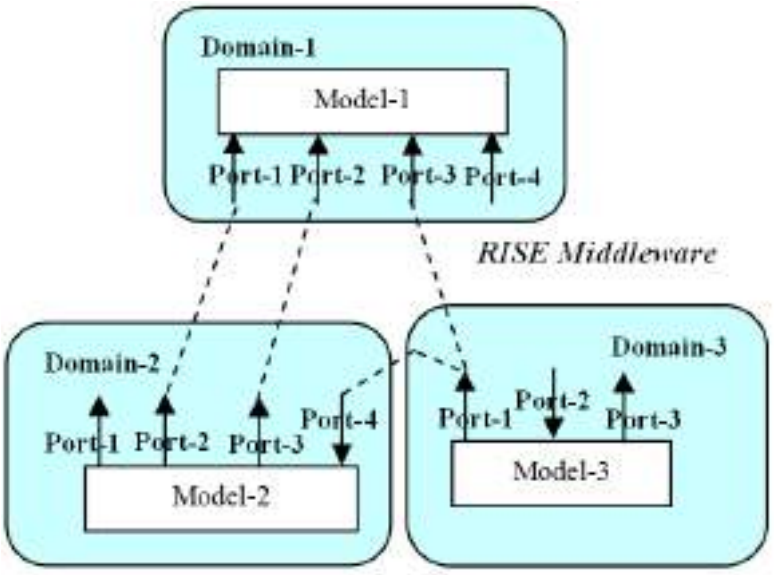

Figure 57: Models Partitions Interconnections

The ports interconnections in Figure 57 are described in the shown XML document in Figure 58.

```
1 <ConfigFramework>
2     …
3     <RISE Version="1.0" Type="P">
4      <Domains>
5       <Main><URI>…/Domain-1</URI></Main>
6        <Links>
7           <Link>
8            <From><Port>Port-2</Port>
9               <URI>…/Domain-2</URI></From>
10           <TO><Port>Port-1</Port>
11              <URI>…/Domain-1</URI></TO>
12          </Link>
13          <Link>
14           <From><Port>Port-3</Port>
15              <URI>…/Domain-2</URI></From>
16           <TO><Port>Port-2</Port>
17              <URI>…/Domain-1</URI></TO>
18          </Link>
19          <Link>
20           <From><Port>Port-1</Port>
21              <URI>…/Domain-3</URI></From>
22           <TO><Port>Port-4</Port>
23              <URI>…/Domain-2</URI></TO>
24          </Link>
25          <Link>
26           <From><Port>Port-1</Port>
27              <URI>…/Domain-3</URI></From>
28           <TO><Port>Port-3</Port>
29              <URI>…/Domain-1</URI></TO>
30           </Link>
31       </Links>
32      </Domains>
33    </RISE>
34    …
35 </ConfigFramework>
```

Figure 58: XML Configuration Example (see Setup in Figure 57)

In this XML document, the configuration must be enclosed within the element <*ConfigFramework*> block, shown in lines 1-35. This allows different implementations to specify other XML configuration outside this block, if desired. Line #3 specifies the synchronization algorithms version and type. In this case, setting attribute *Type* to value "P" indicates the P-DEVS based synchronization. Lines 4-32 define domains (i.e. models) configurations. Line #5 defines the *Main* domain URI, which is the main

partition in an experiment. Its major function is simulation synchronization, discussed in the next section. Lines 6-31 define models ports interconnections (i.e. links). For example, Lines 7-12 define the connection from Port-2 (i.e. Line #8) in URI <…/Domain-2> (i.e. Line #9) to Port-1 (i.e. Line #10) in URI <…/Domain-1> (i.e. Line #11). In this case, simulation messages generated at Port-2 are sent to URI <…/Domain-2>, which are then sent to Port-1. Other ports connections (i.e. links) are defined in a similar way in lines 13-18, 19-24, and 25-30. Upon receiving this XML document (Figure 58), the domains are expected to construct internal routing tables similar to the example shown in Table 12.

Table 12: Domains RISE Routing Tables (see Setup in Figure 57)

Domain	Source Port	Destination Port	Destination URI
Domain-2 Routing Table	Port-2	Port-1	…/Domain-1
	Port-3	Port-2	…/Domain-1
Domain-3 Routing Table	Port-1	Port-4	.../Domain-2
	Port-1	Port-3	…/Domain-1

This table shows the Domain-2 and the Domain-3 routing tables. For example, Domain-3 generates two external messages for each single output message appears on its Port-1: the first one is sent to URI <*.../Domain-2*> on Port-4, and the second one is sent to URI <*.../Domain-1*> on Port-3. Upon receipt, the message is forwarded to the appropriate port regardless of the sender.

8.1.2 Simulation Synchronization Interoperability

Before simulation take place, an experiment framework needs to be setup. In RISE case, this means creating the URI *.../{framework}* and submitting all necessary model scripts to it (see API in Appendix-B). Afterward, the simulation can be started via

the main domain (i.e. this means creating URI *{framework}/simulation*). Consequently, the main domain starts the simulation on all other partitions domains (as in the case of DCD++ described in Chapter 6).

The synchronization approach presented here places a central component to manage the overall simulation in all domains. This is usually the common used approach in the wide geographical distributed simulation systems such as [19][36][152]. However, the presented approach here is different in a sense that it reduces remote message transmissions via message aggregation in XML messages. Our preference of this approach is because it requires low remote synchronization overhead through the Internet. It further adapts the P-DEVS synchronization style, which is supported by all DEVS-based distributed simulation systems.

In this section, we discuss this P-DEVS based synchronization in terms of the simulation phases and the XML messages exchanged. We further discuss dynamic simulation algorithms, where simulation domains can join/disjoin simulation at runtime. Note that optimistic-based algorithms have been described in [5].

P-DEVS Based Synchronization

Upon simulation startup, the main domain creates the RISE Time Manager (RISE-TM), within its domain, to manage the entire distributed simulation (Figure 59). Note that RISE-TM is reached at the same URI of the main domain simulation. Thus, the relation between the main domain and RISE-TM in a system is internal implementation specific. However, we separate them here in our discussion for clarity.

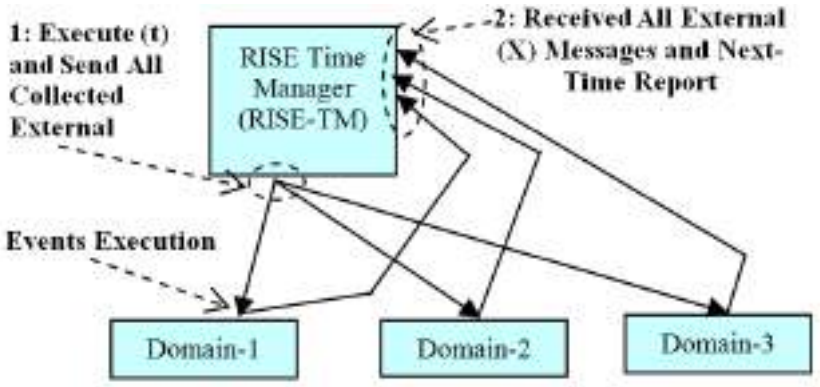

Figure 59: Simulation Cycle at Time t Example

The RISE-TM advances the simulation in cycles (phases). Each cycle is executed in the following two steps, shown in Figure 59:

1. RISE-TM requires all domains to execute all of their events at current (or newly calculated) RISE time (i.e. the time that simulation partitions are allowed to execute events at). This is done via sending in parallel an XML message to each relevant domain, containing the current RISE time along with all external messages generated in the previous cycle, if any. Once a domain partition executes all the internal events with the current RISE timestamp, it responds to RISE-TM with one XML message containing all external messages generated for other domains, if any. Note that all generated external messages must be stamped with the current RISE time (or larger). Further, this XML message also contains the next event time in the sender partition. The next time is the time of the next event in a partition larger than RISE time. If no more events exist, this value is then set to "-1", indicating infinity.
2. Once RISE-TM receives all replies from all relevant domains, it calculates the next RISE time. Further, RISE-TM merges all generated external messages and passes them to all relevant domains at the beginning of the next simulation cycle, as

described in step #1. If RISE-TM finds a new RISE time to be infinity, or receives a stop request from the modeler, it stops the simulation.

The above two steps require two types of XML messages: the first one is sent from RISE-TM to domains while the second comes as a reply to the first message (i.e. it is sent from domains to RISE-TM). Figure 60 shows an example of the first XML message while Figure 61 shows an example of the second XML message.

```
1 <RISE Version="1.0">
2     <Time>00:00:01:000</Time>
3     <XEvents>
4       <MessagesCount>2</MessagesCount>
5          <XEvent>
6             <Time>00:00:01:000</Time>
7             <Port>Port-1</Port>
8             <Value>9</Value>
9             <URI>…/Domain-1/simulation</URI>
10         </XEvent>
11         <XEvent>
12            <Time>00:00:02:000</Time>
13            <Port>Port-2</Port>
14            <Value>10</Value>
15            <URI>…/Domain-1/simulation</URI>
16         </XEvent>
17    </XEvents>
18 </RISE>
```

Figure 60: Example of RISE-TM to Domains Message

Figure 60 shows example of RISE-TM message to domains. This message must be sent to all relevant domains in this simulation cycle. Relevant domains are the partitions that have events to execute in this cycle. In Figure 60, Line #1 defines the protocol version. Line #2 specifies the RISE time; hence, every event with this time must be executed in this cycle. Lines 3-16 enclose all collected external messages in the previous cycles from all domains. In this case, there are two external messages as indicated by Line #4. For example, the first external message is defined by the *<XEvent>*

element block at lines 6-10: Line #6 defines the message time. Line #7 defines the model's destination port name, Port-1 in this case. Line #8 holds the message data, "9" in this case. Line #9 defines the destination URI. Lines 11-16 define the second external message in a similar way.

```
1 <RISE Version="1.0">
2     <URI>…/Domain-2/simulation</URI>
3      <XEvents>
4        <MessagesCount>2</MessagesCount>
5           <XEvent>
6              <Time>00:00:01:000</Time>
7              <Port>Port-1</Port>
8              <Value>9</Value>
9              <URI>…/Domain-1/simulation</URI>
10          </XEvent>
11          <XEvent>
12             <Time>00:00:02:000</Time>
13             <Port>Port-2</Port>
14             <Value>10</Value>
15             <URI>…/Domain-1/simulation</URI>
16          </XEvent>
17       <Time>00:00:01:000</Time>
18     </XEvents>
19     <Next>00:00:03:000</Next>
20   </RISE>
```

Figure 61: Example of Domains Reply to RISE-TM Message

Figure 61 shows an example of a domain response to RISE-TM: Line #1 defines the protocol version. Line #2 specifies the sender domain URI, in this case Domain-2. Note that the sender URI allows RISE-TM to recognize when it has received all the replies from all the relevant domains, so that it can start the new simulation cycle. Lines 3-18 list all external messages generated by Domain-2 in this cycle. In this case, there are two messages as indicated by line #4. The first external message is defined by the <*XEvent*> element block at lines 5-10. Line #6 defines the message time. Line #7 defines the model's destination port name, in this case Port-1. Line #8 holds the message data, in this case "9". Line #9 defines the destination URI. Lines 11-16 define the second external

message in a similar way. Line 17 specifies the minimum time of all the enclosed external messages. Note that RISE-TM includes this time when calculating the next RISE time. Line 19 specifies the time of the next event of the sender domain. Note that RISE-TM not only includes this time when calculating the next RISE time, but also uses it to decide relevant domains in the next cycles. This means that RISE-TM only needs to involve a domain in a certain simulation cycle if it has events to execute in that cycle.

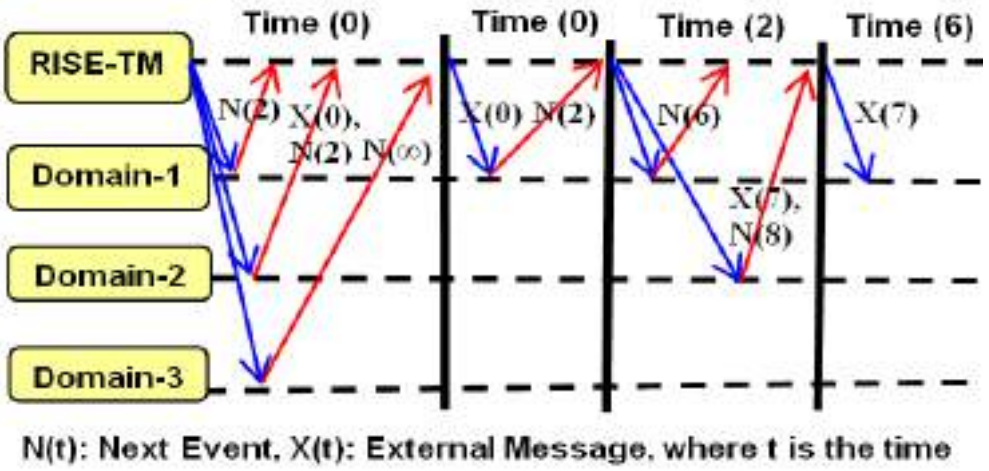

Figure 62: Simulation Cycles Example

To show how the simulation progresses, let us consider the example shown in Figure 62. Here, RISE-TM drives the simulation on three domains. In the first cycle, RISE-TM starts requiring all domains to execute all of their events at time 0 and waits for all domains replies. Domain-1 responds with N(2) (the next event has a timestamp 2). Domain-2 responds with X(0) (an external message with time 0) and N(2) (the next internal event is scheduled for time 2). Assume further that X(0) is intended to be sent to Domain-1. Finally, Domain-3 responds with N(∞) (the model passivates; the next internal event has time infinity). Upon receiving all replies, RISE-TM calculates the current RISE time to be 0. It further starts the second simulation cycle at time 0 and forwards the X (0) to Domain-1. Once Domain-1 executes X(0), it responds with N(2) (it

schedules itself at time 2). Upon this reply receipt, RISE-TM advances time to 2, and requires Domain-1 and Domain-2 to execute all events at time 2. In response, Domain-1 schedules itself at time 6, while Domain-2 schedules itself at time 8 and generates X(7) for Domain-1. In the fourth cycle, RISE-TM advances time to 6 and passes X(7) to Domain-1. Note that Domain-1 does not execute X(7) in this cycle, since the current RISE time is 6 in this cycle. Thus, it inserts X(7) in its events list (so that it can later be executed at time 7), and then responds to RISE-TM. Once RISE-TM receives this reply, it starts a new cycle.

Note that for simplicity, the example in Figure 62 ignores the fact that all of the exchanged messages are aggregated in XML messages as previously discussed. This aggregation enhances performance because remote messages transmissions are expensive and take in the range of milliseconds to seconds [58]. It further ensures accurate RISE time calculation without having complex time-calculation schemes. In our approach, all systems involved can only receive or send one XML message in each simulation cycle. In this case, RISE-TM cannot start a new simulation cycle until it receives that XML message from each involved domain in the current cycle. Otherwise, if simulation messages were transmitted individually, RISE-TM would need to take into account any possible messages that still in transit. This is because all messages are transmitted concurrently for performance reasons, which makes it impossible to know the time it takes a message to be transmitted. Simulation messages aggregation solves this problem. Note that *without* message aggregation, each sender must be blocked until the message arrival is acknowledged by the destination, in order to ensure all messages have arrived in the correct order in the correct simulation cycle. This prevents the sender of performing

local computation, and doubles the cost of each message (i.e. the sender is blocked for the duration of transmitting of the message and its acknowledgment).

Dynamic Simulation

This section provides design suggestions in performing dynamic simulations in which domains are removed or added at runtime. To start the process, the modeler sends the XML configuration document, shown in Figure 63, to the main domain URI.

```
1 </ConfigSimulation>
2     …
3     <RISE Version="1.0">
4      <Domains>
5       <Add><URI>…/Domain-4</URI></Add>
6       <Remove><URI>…/Domain-2</URI></Remove>
7        <Links>
8          …
9        </Links>
10     </Domains>
11    </RISE>
12    …
13 </ConfigSimulation>
```

Figure 63: RISE XML Dynamic Configuration

This XML document (Figure 63) instructs the main domain to remove or add certain domains from/to the simulation. It further describes the new models partitions ports connections. Note that for simplicity, we do not allow main domain removal, but modeler can still disconnect all of its ports. Line #5 defines the <*Add*> element block, which lists all new joined domains URIs, in this example <*…/Domain-4*>. Line #6 defines the <*Remove*> element block, which lists all new disjoined domains URIs, in this example, <*…/Domain-2*>. Lines 7-9 define the new ports connections similar to lines 6-31 in Figure 58, previously described in Section 8.1.1.

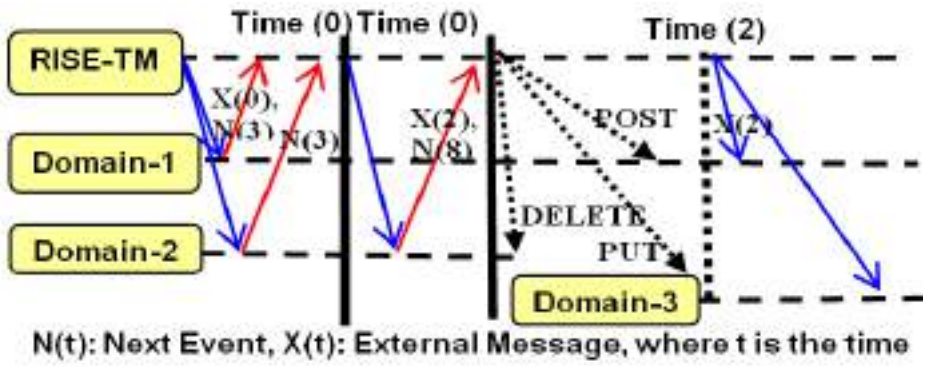

Figure 64: Dynamic Simulation Phases Example

In the P-DEVS based synchronization approach, RISE-TM performs the critical steps of the dynamic configuration before starting a new simulation cycle with a new RISE time, as illustrated in the example shown in Figure 64. Assume that RISE-TM receives the configuration request during the first phase at time zero. Assume further that RISE-TM is instructed to remove Domain-2 and to add Domain-3. In this case, RISE-TM performs the second phase because the time does not change. However, it starts the dynamic configuration before starting the third phase, as follows:

1. It deletes Domain-1 via the DELETE channel,
2. It creates Domain-3 (via the PUT channel) and supplies it with the new configuration along with the current RISE time, and
3. It sends the new XML configuration document to Domain-1 (via the POST channel), allowing new routing tables to be reconstructed. At this point, RISE-TM must wait until successful acknowledgements are received back. Subsequently, a normal simulation cycle is started at time 2 where all new created domains are being involved.

Note that all external messages produced by deleted domains (e.g. Domain-2 in Figure 64) are still processed since they are still part of the simulation. On the other hand,

all the X messages going to deleted domains are ignored, since RISE-TM does not forward simulation messages to dead domains. For example, if the removed Domain-1 was supposed to receive X messages at time 1 in Figure 64, those X messages are discarded, as Domain-1 died at the start of time 1. Further, Domain-3 started at time 2. Note that initialization phase for new added domains are assumed to start with the current RISE time.

8.2 Simulation Experiment Workflows

A *product workflow* is the set of required steps for developing a product until it gets into market [146]. They serve as means for automation, repeatability, controlling processes and management (i.e. be part of a formal Business Process Management (BPM) [146]). Nevertheless, there is still no workflow integration for the M&S software currently in use. Users are still forced to combine a number of different software products using general workflow tools for scientific processes (e.g. Kepler [85] and Trident [131]), scripting languages like Python [151] or combining different functionalities manually.

This section presents the use of RISE to design a workflow component (and various workflow examples). A workflow component [4] running on the client side would be useful to automate the manual steps that would have been taken by modelers to create and manipulate experiments on the RISE middleware. In the component presented here, workflows are designed graphically and stored as XML scripts. During execution, a workflow instance is created and executed based on its XML script. This execution means that a token is moved from a computation step to another. The computation step could be for example, creating an experiment URI on RISE, running simulation on that

experiment, checking a simulation status on that experiment, etc. Thus, many experiments can be created automatically and manipulated simultaneously.

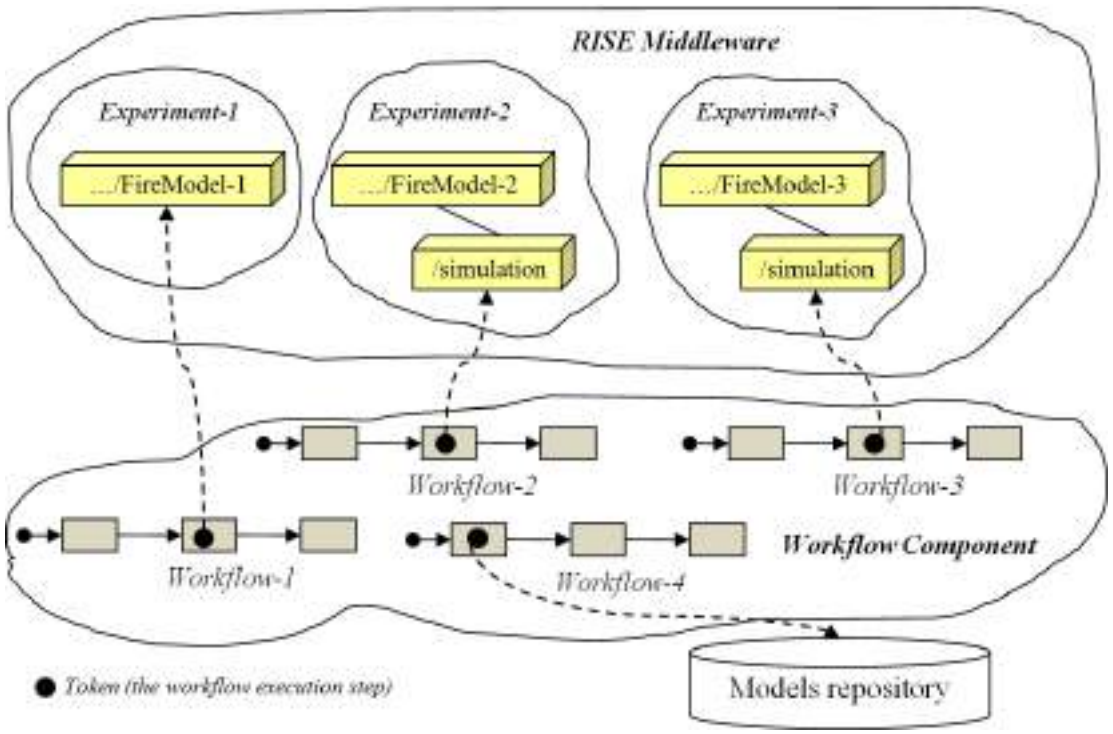

Figure 65: Overview of Simulation Workflows Example

Let us consider the example in Figure 65. This example shows four workflow instances being executed by the workflow component. Assume that the first three workflows have created three RISE experiments to simulate a *Fire* model. In this case, these experiments are handled independently by the workflow component, since each is handled by a workflow instance. For example, Workflow-1 is currently interacting with the Experiment-1 URI *.../FireModel-1* (e.g. submitting model scripts). Workflow-2 and Workflow-3 are currently running simulation on Experiment-2 and Experiment-3 respectively (interacting with URIs *.../FireModel-2/simulation* and *.../FireModel-3/simulation*). However, Workflow-4 is at a computational step not involving an experiment at RISE. This is because a workflow may involve other steps such as

designing a simulation model to be executed later within an experimental framework. Thus, Workflow-4 is interacting with the Models repository to retrieve previously stored models scripts.

This component will be presented as follows. Section 8.2.1 discusses the workflow component architecture (including interaction with different servers and the RISE middleware to manage and execute different workflows). Then, Section 8.2.2 presents a number of experiment workflow patterns using the YAWL graphical notations [136], the interactions with RISE and the workflows XML representation.

8.2.1 Workflow Component Architecture

The workflow component architecture, shown in Figure 66, follows the reference model recommended by the Workflow Management Coalition (WFMC) standard group [148]. In this model, the workflow component is in the center where it interacts (as a client) with other surrounding repositories, servers and perhaps other workflow components. In our case, the workflow component (Figure 66) interacts with the RISE middleware to run simulation experiments. It also interacts with different repositories for storing/retrieving the simulation models and workflows scripts. The Graphical User Interface (GUI) is used to define the workflows graphically according to certain notations (in our case, YAWL that is presented in Section 8.2.2). The workflow modeler's role is to design those workflow blueprints. Once completed, they are compiled by the workflow component into the XML Definition script (discussed in Section 8.2.2), and stored in a repository as XML documents for future use.

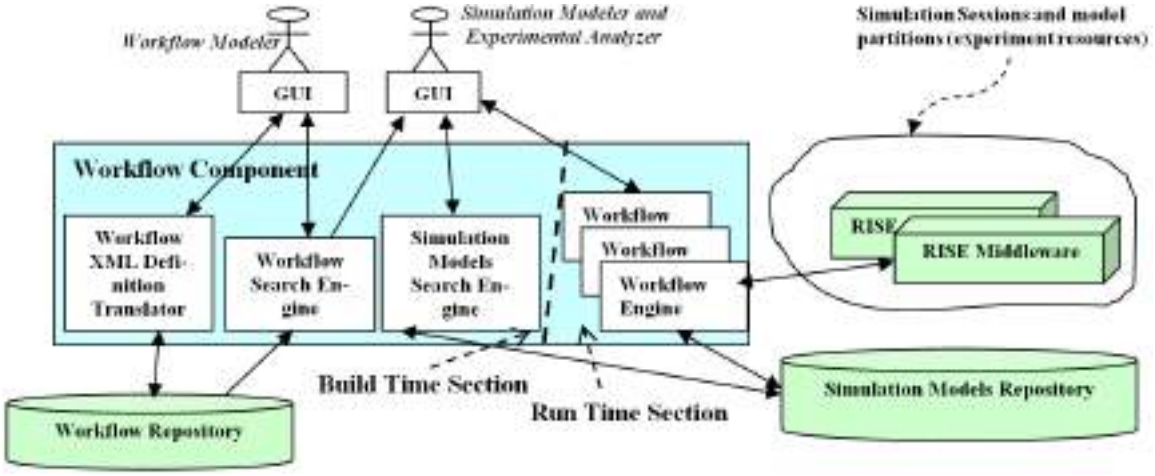

Figure 66: Workflow Component Design Architecture

The workflow components allow both workflow and simulation modelers to submit their search criteria to retrieve certain workflow patterns, as shown in Figure 66. The simulation modelers use workflow patterns in the context of executing experiments and constructing models. On the other hand, the workflow modelers use workflow patterns to construct other needed workflow patterns by reusing certain existing workflows within new constructed workflows.

As also shown in Figure 66, the component operates in two phases: In the *Run Time* phase, workflow engines execute the workflow patterns while all workflows are designed and stored for future reuse in the *Build Time* Phase. In the *Run Time* phase, workflow engines execute the workflow patterns. The workflow engine parses the workflow XML definition document (discussed in Section 8.2.2) and executes each step as applicable (e.g. manipulating simulation experiments at the RISE side). As in the discussed examples later in Section 8.2.2, workflows are designed as computational steps where a token moves through a set of elements such as tasks instances (e.g. start simulation) and conditions (e.g. if simulation is done).

Because the token time is spent inside tasks instances, since they represent the actual work in a workflow. Our focus for the rest of this section is devoted on tasks executions in workflows.

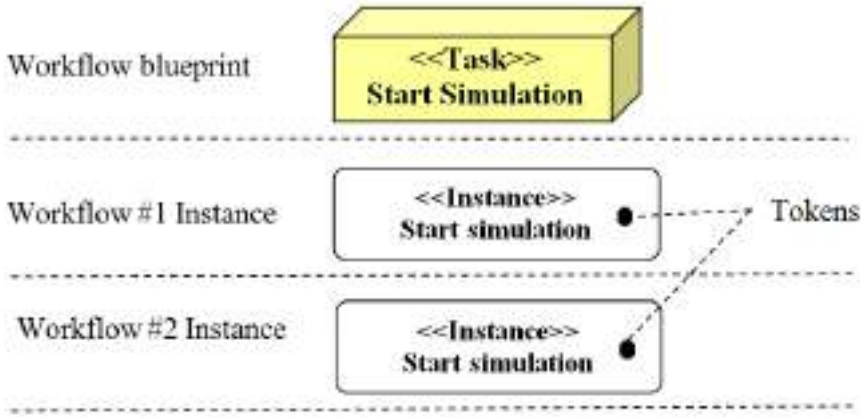

Figure 67: Example of a Workflow Task and Task Instances

It is important to distinguish between a workflow blueprint and a workflow instance. The workflow blueprint is the template that contains tasks (i.e. analogy with the C++ classes) while the workflow instance is a case of that blueprint that contains tasks instances (i.e. analogy with the C++ objects). Thus, workflows instances only exist at runtime. This means that tasks (i.e. computational units) within a workflow blueprint exist in workflows instances as task instances. For example, Figure 67 shows two instances of task "Start Simulation" in Workflow #1 and Workflow#2 instances. Even though these task instances follow the same computational patterns, but their execution is completely two different things. For example, Workflow #1 might be waiting for some reason to start simulation on an experiment instance while Workflow #2 is finishing simulation on another experiment instance. Thus, the state of a task instance is controlled by the state transition diagram (discussed shortly) of that task. This transition diagram becomes active once the workflow token enters the task instance (Figure 67).

In our case, once a token enters a task instance in a workflow, the token follows the state transition diagram to manage its behavior during a task execution. Because a task may have different instances that being executed simultaneously, the workflow component manages those task instances in multiple lists based on their states (i.e. defined in the state transition diagram). The state transition diagram and those lists are discussed next.

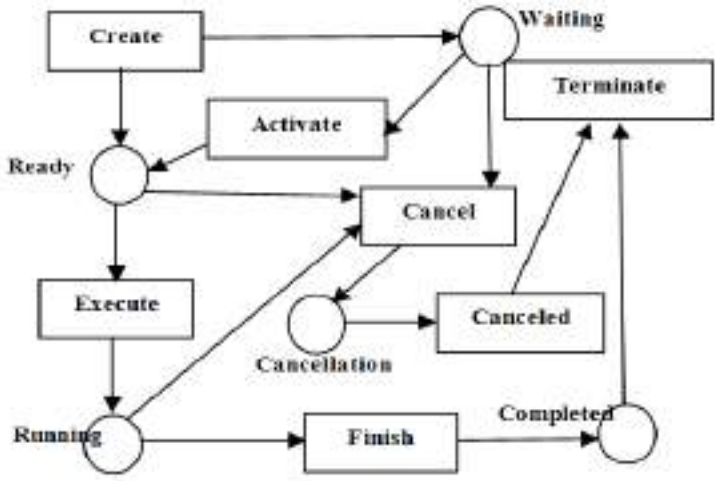

Figure 68: State Transition Diagram for a Workflow Task

The execution semantics of each task in a workflow is expressed in the state transition diagram shown in Figure 68. The diagram consists of conditions (represented by circles) and transitions (represented by rectangles). Conditions are places to hold tokens (a token is an instance of a task), hence multiple instances reside in different condition of their task state transition diagram. Transitions are actions that cause a token to move from a condition to another. Transitions take time to complete. Figure 68 shows five possible conditions: **(1)** "*Waiting*" holds tokens that are not allowed to run. For example, the workflow component might limit the number of running tasks simultaneously. **(2)** *"Ready"* holds tokens that are active and waiting to execute. **(3)**

"Running" holds tokens that are currently being executed. **(4)** *"Completed"* holds tokens that have completed. **(5)** *"Cancellation"* holds tokens that have been canceled before completion.

The workflow component tracks token instances of a task via associating them to lists according to their defined conditions in the state transition diagram. In this case, based on the possible condition, there are four token lists: Waiting, Ready, Running and Completed lists, shown in Figure 69.

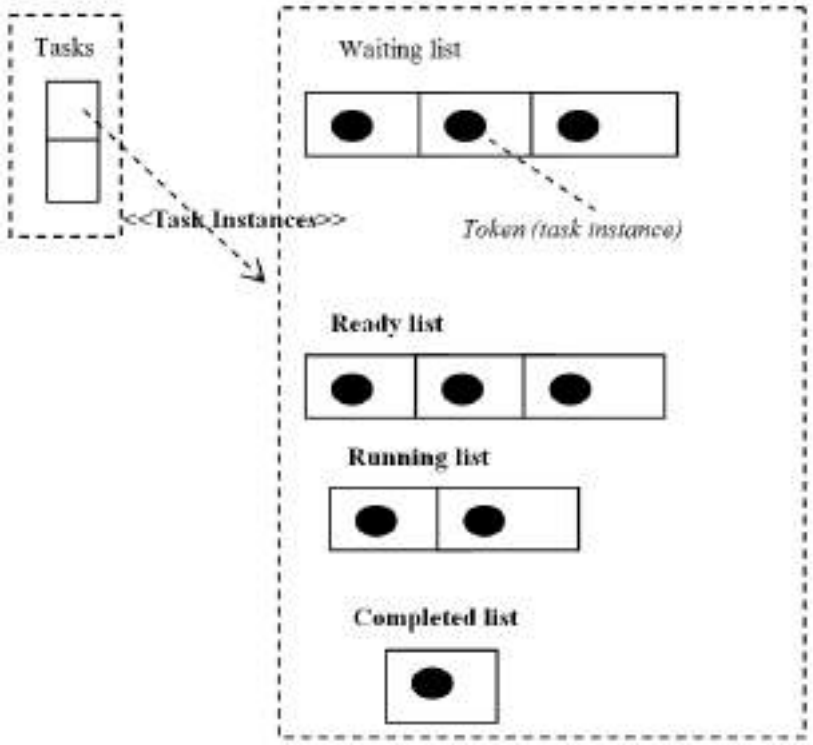

Figure 69: Tracking Token States in Tasks Instances

8.2.2 Experiment Patterns Examples

The workflow patterns examples discussed in this section uses the YAWL [136] graphical notations. This provides modelers with a graphical method to design workflows. However, one of the workflow component objectives is to convert the

graphical notations to XML representation so that they can be stored, retrieved, and executed. This provides machines with processing syntactic for handling workflows. In our discussion next, we briefly describe all of the relevant YAWL notations used in the presented examples in this section, followed by the workflow patterns examples discussion in terms of their graphical notations and XML representation.

Figure 70 shows excerpt of the YAWL notations: conditions are represented by circles and tasks are represented by rectangles. Input and output conditions specify the beginning and the end of a workflow. Atomic tasks are indivisible activities, where composite tasks (denoted by double border) enclose other workflow activities. Tasks are considered automatic, but placing an arrow on a top of a task indicates a user interaction while placing a clock indicates a timer trigger task. The XOR-split element limits the flow output to exclusively one, based on a certain condition. Arcs show the token flow direction.

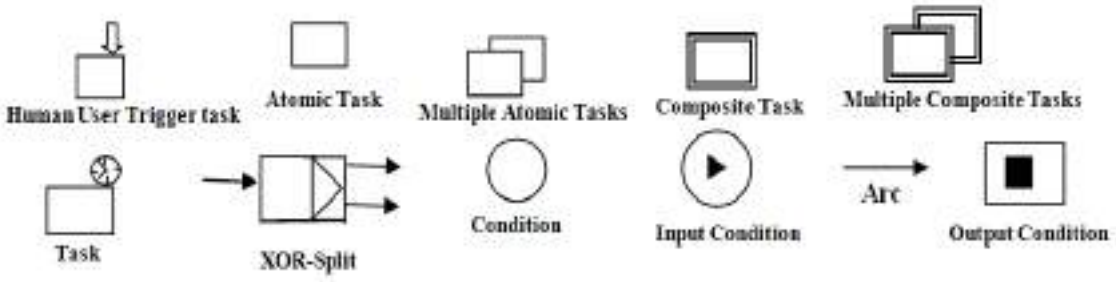

Figure 70: Excerpt of YAWL Notational Elements

The approach followed here is to design workflows in composite tasks so that these composite tasks can be plugged in other workflows, hence allowing their reusability.

The Simulation Workflow Composite Task (SW-CT), shown in Figure 71, is the task that encapsulate all workflows examples presented in this section. The purpose of this task is **(1)** to control the overall pattern of building a simulation model, **(2)** to setup a simulation experiment at the RISE middleware, and **(3)** to execute simulation on that RISE experiment. To do so, this composite task encloses two other composite tasks. The first task is the "Building Simulation Model" Composite Task (BSM-CT), which guides the modeler to construct a simulation model (BSM-CT is discussed shortly). The second task is the "Simulation Experimentation" Composite Task (SE-CT), which sets up multiple experiments at the RISE middleware and executes the simulation on those experiments. The SE-CT is fully automated, hence allowing multiple experiments to run automatically at the same time (SE-CT is discussed shortly).

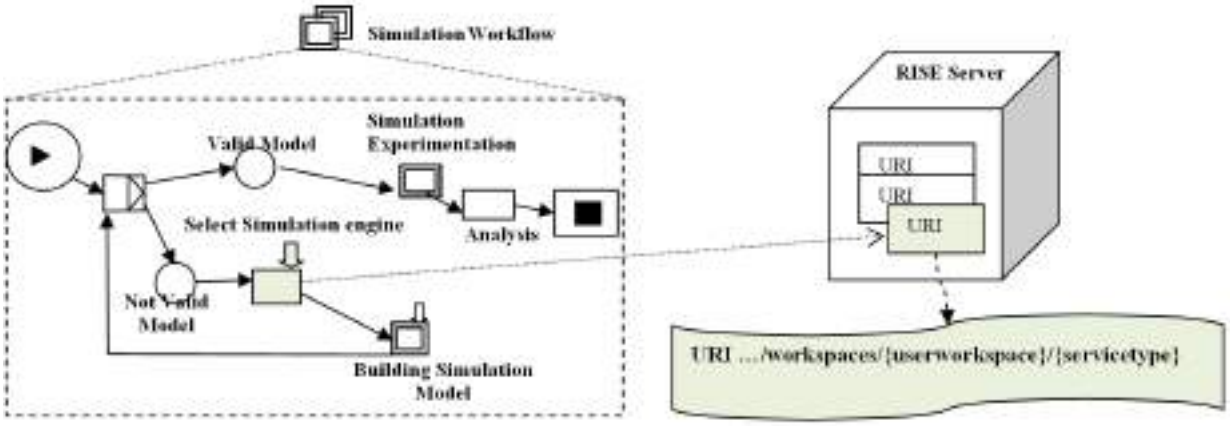

Figure 71: Simulation Workflow Composite Task (SW-CT) and RISE Interactions

Figure 71 shows that if a simulation model is present and valid, each simulation Workflow Composite Task (SW-CT) starts with executing the "Simulation Experimentation" Composite Task (SE-CT). Once the SE-CT is completed, the "Analysis" atomic task handles the output results of all experiment instances. For

example, the analysis may be performed via visualization or by comparing certain variables from experiment instances different simulation results. Note that this task is put outside the SE-CT since it may require extra handling by modelers. However, if a simulation model is not valid, the SW-CT starts up with creating a simulation environment URI at the RISE middleware, using the atomic task “Select Simulation engine”. This atomic task interacts with the RISE middleware (Figure 71) to select a simulation environment to hold all future created experiments. For example, setting up {servicetype} to “DCDpp” would select the DCD++ simulation environment (see API in Appendix-B). Afterward, the SW-CT starts the “Building Simulation Model” Composite Task (BSM-CT) to construct the simulation model that will be executed by the simulation. The flow of the SW-CT task, shown in Figure 71, is summarized as the following:

```
If (valid Model) {
   Start SW-CT // build and executes experiments
   Conduct experiment Analysis
} else if (not valid model) {
   Create a simulation environment on RISE
   Start BSM-CT // to construct simulation model
}
```

Building Simulation Model Composite Task (BSM-CT)

The Building Simulation Model Composite Task (BSM-CT) concerns of constructing the simulation model and its testing and verification. The BSM-CT guides the modeler for searching other existing models in repositories for models reusability. Thus, most of the internal atomic tasks require human intervention (the one with an arrow on the top). Note that BSM-CT uses the Simulation Experimentation Composite Task (SE-CT), to execute the simulation in the experiment framework to test the simulation model being constructed.

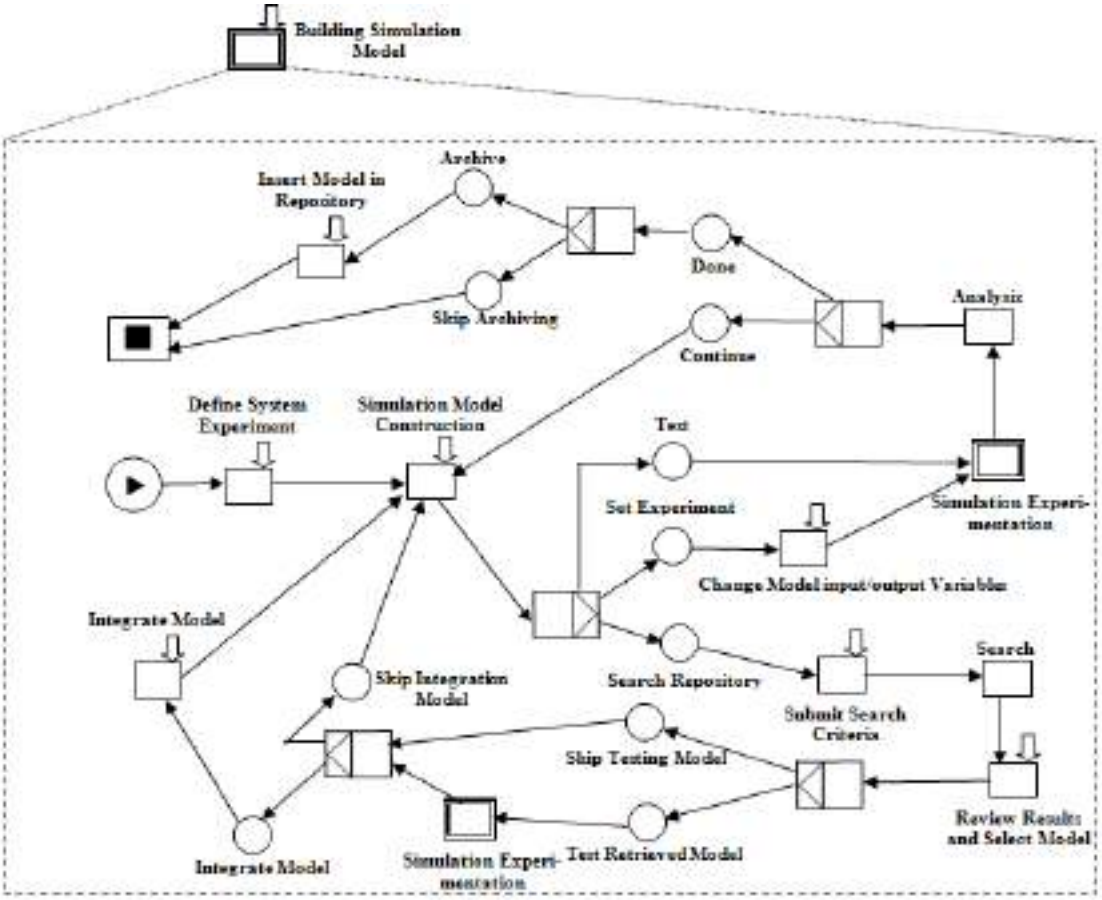

Figure 72: Building-Simulation-Model Composite Task Workflow

The BSM-CT is shown in Figure 72. The task starts with the atomic task "Define System Experiment" to define the purpose of the simulation model with respect to the system understudy. The workflow then moves to the atomic task "Simulation-Model Construction" to build the actual model. The model might be built using graphic notations or by directly writing source code and scripts. The workflow then moves to one of three branches based on conditions "Test", "Set Experiment", or "Search Repository", follows:

- By selecting the "Test" condition, the SE-CT task is used to test the set the experiment and executes the simulation. The "Analysis" task is then used analyze

results. Afterward, there are two options: (1) to continue building the simulation model, or (2) to complete the building process. The second option allows the modeler to archive the model in the repository for possible future reuse.

- By selecting the "Set Experiment" condition, the "Change Model input/output Variables" atomic task enables the modeler to change model parameters. Afterward, it follows the same process of the "Test" condition described above.
- By selecting the "Search Repository" condition, the "Submit Search Criteria" atomic task submits the criteria to the repository. The workflow then allows the modeler to test any found models through executing the simulation via the SE-CT task. The retrieved model can then be integrated with the model being constructed or not integrated. The workflow is then moves back to the "Simulation-Model Construction" atomic task, where one of these three conditions can be selected again.

Simulation Experimentation Composite Task (SE-CT)

The Simulation Experimentation Composite Task (SE-CT) provides an automotive method for conducting multiple experiment instances simultaneously. As shown in Figure 73, the SE-CT contains multiple "Experiment Instances" tasks where each "Experiment" task controls the workflow of a single experiment. Thus, the "Experiment" task interacts heavily with the RISE middleware, since it uses the middleware to create and runs those experiments instances. Note that those experiments might be running as a distributed simulation over various machines or over a single machine. This depends on how such experiments are set and configured on RISE. However, our focus in this section is only on the "Experiment" workflow.

This subsection presents the following topics: **(1)** Describe the "Experiment" workflow YAWL graphical notations, **(2)** Discuss the interactions between an instance of the "Experiment" task and its URIs on the RISE middleware. **(3)** Describe the XML representation of the "Experiment" task, allowing the software to process the workflow.

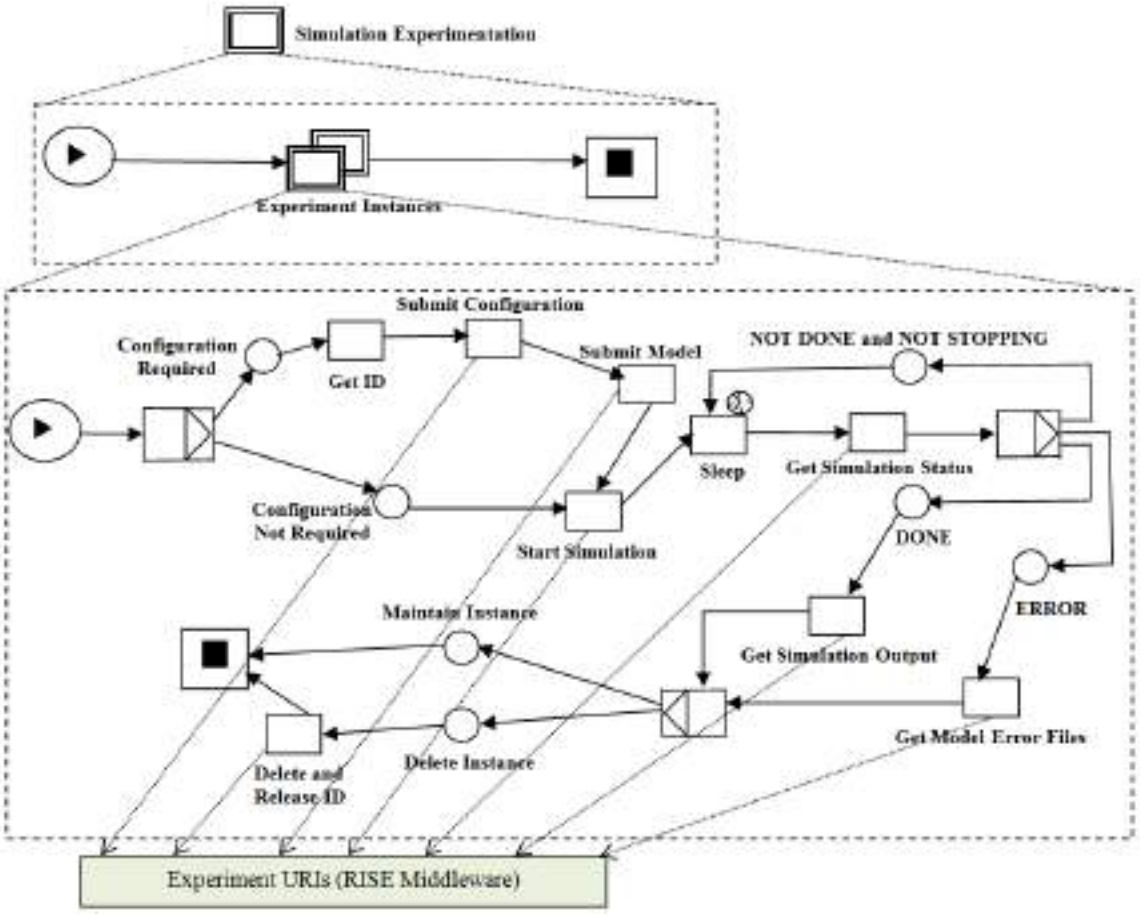

Figure 73: Simulation Experimentation Task Workflow

Figure 73 shows the YAWL graphical notations of the Experiment workflow task. The task starts with making the decision to reload experiment configuration or not to reload. If "Configuration Required" condition is met, the workflow follows the following path: **(1)** the "Get ID" atomic task is executed to assign an ID for the experiment instance. **(2)** The "Submit Configuration" and "Submit Model" atomic tasks are executed

to setup the experiment at RISE side. **(3)** The "Start Simulation" atomic task is executed to start the simulation on that experiment at RISE. However, if "Configuration Not Required" condition is met, the workflow starts the simulation immediately on RISE. Once the simulation started, the workflows keeps checking the simulation status at RISE until it is completed or aborted on error. If simulation completed successfully, the workflow retrieve the simulation results from the experiment at RISE. Otherwise, if exited on error, the error files are retrieved from RISE. Finally, the workflow may choose to delete the experiment instance at RISE, or maintain it for future reuse. The workflow in Figure 73 is summarized in the following:

```
If (configuration Required) {
   Get an ID for this experiment instance // i.e. token id
   Submit updated XML configuration to experiment at RISE
   Submit Model representations to experiment at RISE
} //if
Start simulation in experiment at RISE
While (NOT DONE and NOT STOPPING) {
     Sleep for a number of seconds
     Get Simulation status from the experiment at RISE
     if (status == DONE) {
         GET simulation results from experiment at RISE
     } else if (status == ERROR) {
         GET simulation errors from experiment at RISE
     }
} // while
If (experiment instance to be deleted on RISE) {
   DELETE experiment at RISE and release token ID
}
```

As shown Figure 73, the workflow interacts with RISE in certain atomic tasks. This use is similar to any other client use when setting up experiments and executing simulation on those experiments. Thus, the workflow component, as in the case of any other clients, needs to follow the RISE API (described in Appendix-B) to create and manipulate experiments on RISE. Using the RISE API, the workflow interactions with

RISE are illustrated in Figure 74. In this case, each token (i.e. an experiment instance at the workflow component side) interacts with the URIs associated with that experiment instance at the RISE side. These interactions are discussed next.

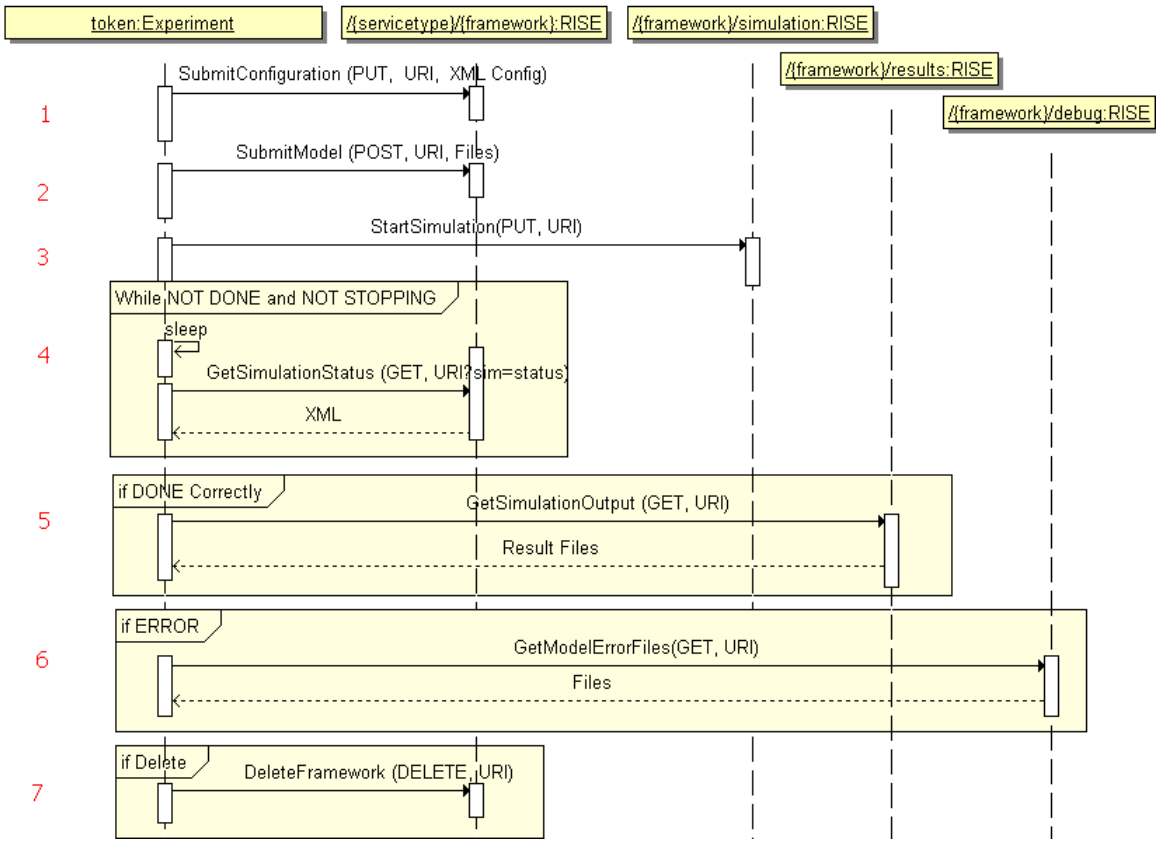

Figure 74: Workflow Token Interactions with Experiment URIs at RISE

Figure 74 shows five components. The first one on the left (i.e. token:Experiment) represents the experiment instance (i.e. token) at the workflow component while the other four components represent the experiment URIs at RISE. Note that those URIs do not usually exist at the same time, as discussed in Chapter 5, but we show all of them on the figure for illustration purposes. These four URIs are: (1) The experiment main URI, usually used for experiment various settings, (2) The active simulation URI that wraps the active simulation, (if simulation is active), (3) The URI that wraps the simulation

results, (if any), (4) The URI that wraps the simulation errors (if any). In this case, each interaction in Figure 74 is uniform with the following information: (1) destination resource URI, (2) resource Channel, (3) and the message to send/receive.

The interactions in Figure 74 are summarized as follows (from top to bottom according to the interaction numbers shown on the figure):

1. Submit updated configuration to experiment at RISE (i.e. PUT channel, main URI, XML message).
2. Submit Model representations to experiment at RISE (i.e. POST channel, main URI, Zipped files).
3. Start simulation in experiment at RISE (i.e. PUT channel, simulation URI, Null message).
4. (The "While" block) Get Simulation status from the experiment at RISE (i.e. GET channel, main URI, XML message).
5. (The If DONE block) GET simulation results from experiment at RISE (i.e. GET channel, results URI, Zipped file).
6. (The If ERROR block) GET simulation errors from experiment at RISE (i.e. GET channel, errors URI, Zipped file).
7. (The If delete block) DELETE experiment at RISE and release token ID (i.e. DELETE channel, simulation URI, Null message).

It is common practice to represent graphical notations in scripts so that it can be stored in repositories and processed by software. Figure 75 is the XML definition for YAWL graphical notations shown in Figure 73.

```
1 <Task>
2   <Name>Simulation-Experimentation</Name><Type>Composite</Type>
3   <Input><Parm1>uri</Parm1><Parm2>ModelPath</Parm2></Input>
4   <Elements>
5       <Splits><XOR><Instances><Instance>XOR-1</Instance>……</Instances></XOR></Split>
6       <Conditions><Condition>COND-Configure</Condition>……</Conditions>
7       <Tasks>
8           ……
9           <Task><Name>Submit-Configuration</Name>
10              <Instances><Instance>
11                  <Name>Submit-Configuration-1</Name>
12                  <Input><Parm1>uri</Parm1><Parm2>ModelPath</Parm2></Input>
13              </Instance></Instances></Task>
14          <Task><Name>Start-Simulation-1</Name>
15              <Instances><Instance>
16              <Input><Parm1>uri+"/simulation"</Parm1></Input>
17          </Instance></Instances></Task>
18          ……
19      </Tasks>
20  </Elements>
21  <Transitions>
22      ……
23      <Transition><Id>xxx</Id><From>XOR-1</From><To>COND-Configure</To>
24      <Transition><Id>xxx</Id><From>COND-Configure</From><To>Get-ID-1</To>
25      <Transition><Id>xxx</Id><From>Get-ID-1</From><To>Submit-Configuration-1</To>
26      <Transition><Id>xxx</Id><From>Submit-Configuration-1</From><To>Submit-Model-1</To>
27      ……
28  </Transitions>
29
30 <!-------------------------------------->
31 <!--Task Submit Configuration -->
32 <!-------------------------------------->
33 <Task>
34  <Name>Submit-Configuration</Name>
35  <Type>Atomic</Type>
36  <Input><Parm1>uri</Parm1><Parm2>ModelPath</Parm2></Input>
37  <Implementation><Type>REST</REST>
38      <Operation>
39          <Channel>PUT</Channel>
40          <Representation><Type>text/xml</Type><File>ModelPath</File></Representation>
41      </Operation>
42  </Implementation>
43 </Task>
44
45 <!-------------------------------------->
46 <!--Task Start Simulation
47 <!-------------------------------------->
48 <Task>
49  <Name>Start-Simulation</Name>
50  <Type>Atomic</Type>
51  <Input><Parm1>uri</Parm1><Input>
52  <Implementation><Type>REST</Type>
53      <Operation>
54          <Channel>PUT</Channel>
55          <Representation>null</Representation>
56      </Operation>
57  </Implementation>
58 </Task>
59 </Task>
```

Figure 75: XML Definition for Simulation-Experimentation Task Workflow

In Figure 75, Line #2 defines the task name and its type. Line #3 defines the input parameters of this task. In this case, the task instance is initialized with the experiment URI at RISE and the directory path of the files that contain the simulation model required

files, typically zipped up in a single file. Lines #4-20 define all internal elements in this composite task. Line #5 defines all split elements in the task (e.g. XOR-1). Line #6 defines the condition elements in the task. Lines 7-19 define internal task instances. In this case, Lines #9-13 creates an instance of atomic task Submit-Configuration with name Submit-Configuration-1. It further initializes it with the RISE experiment URI and the experiment XML configuration document. Line #14-17 creates an instance of task Start-Simulation, and initializes it with the experiment simulation URI. Lines #21-28 define all internal transitions, hence how each internal element instance is connected to other elements instances. The rest of the XML document defines all internal atomic tasks to the SE-CT task. Lines #33-43 presents the Submit-Configuration task: Line #36 defines the input parameters. Lines #37-42 defines the task implementation as of type of REST Web-service. In this case, it uses channel PUT where only XML type of representation is supported. In the same way, the Start-Simulation atomic is defined in Lines #48-58.

8.3 Chapter Summary

This chapter shows how to apply the different methods introduced in previous chapters for other applications (besides DCD++ distributed simulation). To do so, this chapter first presents additional distributed simulation algorithms built with RISE (Section 8.1), and it then shows how RISE could improve simulation experimentation via the use of workflows (Section 8.2).

The algorithms presented in Section 8.1 aim on interoperating independent-developed simulation services (which could be used as a feasible proposal for DEVS standardization [140][141][142][143]). In this case, the algorithms place models in each

partition as black boxes interconnected with other models via input/output ports. The simulation is executed in cycles where all exchanged synchronization messages are described in XML. In this case, the RISE-TM component sends (in parallel) an XML message to all relevant domains, requiring them to execute all internal events at current simulation time. This XML message also forwards all generated external messages to domains. In response, domains execute their internal events, and responds back to RISE-TM with an XML message. This XML message also contains a domain generated external messages to other domains. This synchronization approach is based on the P-DEVS approach, and is has less synchronization overhead. These algorithms are also extended to handle dynamic simulation, where simulation partitions can join/disjoin at runtime.

Because these algorithms require the approval of the DEVS standardization Group [47], these algorithms have not fully implemented in RISE. This is because other DEVS groups may introduce their changes to these algorithms. This type of standardization process is time consuming and difficult to predict its progress. On the other hand, the presented algorithms implementation has fully been planned and partially tested. In case of the DCD++ (presented in Chapter 6), it is expected to be configured by the modeler to use the type of algorithms in the simulation. In this case, all changes are designed in the simulation manager (see Chapter 6) outside the CD++ implementation itself. As result, the simulation manager makes the decisions on how to send/process XML messages, based on the used algorithms. In this case, the CD++ engine “thinks” that it is still using the algorithms presented in Chapter 6. This makes the DCD++ open for other groups’ proposals as soon as those proposals are presented in terms of the required

synchronization rules and the required exchanged XML messages. In other words, they do not dictate the "how to implement" question.

Note that the need to propose different algorithms from the presented schemes in Chapter 6 is that the DCD++ has extended the DEVS-based coordinator into Head/Proxy structure (to reduce synchronization overhead). However, this adds more complexity to other DEVS-based tools that do not support this structure. On the other hand, the presented algorithms here hide this type of internal details. Note further that the presented algorithms have been compared against other proposals in [143]. In this case, other proposals heavily expose their internal implementations to the point that the proposals become like standardizing software implementation. This makes it difficult to bring various independent-developed systems implementations close to each other, as discussed in [143].

This chapter also presented the workflow component design. The workflow component (on the client side) automates the usual manual steps that would have been taken by modelers to create and manipulate experiments on the RISE middleware. They serve as means for automation, repeatability, controlling processes and management. The major objectives of the presented workflow component are **(1)** to execute any number of simulation experiments with different conditions simultaneously, and **(2)** To enhance simulation models construction and reusability. The workflow component operates in two phases: *Run Time* and *Build Time* Phases. In the *Build Time* Phase, workflows are designed (in YAWL graphical notations), converted and stored in repositories (in XML representations) for future reuse. In the *Run Time* phase, the workflow engine parses the workflow XML representation and executes each task in the workflow as applicable.

CHAPTER 9: CONCLUSIONS AND FUTURE WORK

This chapter summarizes the major topics of the thesis and suggests future research directions. Section 9.1 summarizes the thesis major objectives, argument, and results. Section 9.2 suggests different research directions.

9.1 Thesis Summary

This thesis addressed the software issues related to improving distributed simulation interoperability and synchronization algorithms. In particular, our objective was to develop an all-purpose Web-services based distributed simulation middleware to enhance interoperability methods on the Web between different simulation systems. To do so, we defined certain features similar to the following that we wanted this middleware to support:

- The middleware was designed as a general container to keep the door open for supporting additional simulation environments (beside DCD++ discussed in Chapter 6). Further, this container approach is not limited to simulation systems, but also can be applied to different applications as described in section 9.2.
- The middleware methods had to hide systems heterogeneity in the way information is exchanged, accessed and described.
- The middleware had to be independent of any specific implementation and should allow flexibility of interoperated simulation systems such as in the ability of those systems to interoperate with different synchronization protocols.

- The middleware had to scale in the ability of interfacing more services into the middleware, and in the ability of composing any number of partitions in the distributed simulation session.
- The middleware had to realize simulation within experiments frameworks. In this case, the middleware had to provide modelers the mechanism to create and manipulate those experiments via the Web. Of course, the experiments were expected to maintain all related data and settings, unless changed by their authorized owners.

On the other hand, in order to meet the above objectives, we first had to solve the problem of hiding systems heterogeneity (that resides in implementation) in components while allowing composition scalability and dynamicity. As discussed in Chapter 2 and 3, none of the existing approaches up to date was suitable to solve this problem. This is mainly due to the way in which current distributed simulation approaches exchange, structure, and use information, which is tied to programming and implementations, exposing systems *heterogeneity*. This path usually leads to the need for homogenizing different implementations, which is usually a complex problem to resolve.

We initially focused on meeting our objectives using a SOAP-based WS framework. However, the SOAP WS structural rules proved difficult for completely solving the above research questions. For example, the WS ports that contain software implementations cannot be created at runtime. Further, data channels must be implemented as procedures where a stub for each procedure is required in each user system. This causes composition scalability difficulties and dynamicity problems (since stubs need to be compiled with each system implementations). Although we limited our procedures to a few (implemented in a single WS port), those channels were still

embedded (and compiled) with the internal implementation. To reduce exposing the internal implementation, we exchanged and described the simulation synchronization messages as XML messages. In this case, the entire XML message is sent as a single SOAP attachment. The lesson learned here was that the syntactic and structural interoperability rules characterize the level of freedom of a software designer when defining the methods for middleware interoperability. This experience showed us that decoupling systems implementations (where heterogeneity resides) is difficult task. This is because interoperating via programming procedures plant the actual links between systems inside implementations. This makes system implementations and internal software design issues easily being influenced by each other designs. Therefore, we concluded that this interoperability approach is difficult to be achieved in open communities as in the case of the Web. In this type of communities, systems need to be designed, implemented, and evolve independently from external systems. Further, in open comminutes collaboration, any number of participant systems should be able to join/disjoin the overall distributed structure at runtime without necessary a pre-knowledge of other systems. On the other hand, the SOAP-based WS ports (along with their procedures) had to be created and compiled before even starting up the system. Thus, this approach is a close community oriented where software developers can discuss with each other to resolve systems API related design issues.

In contrast, the WWW is the largest existing distributed structure where countless of systems interoperate with each other according the Web standards. This open-community interoperability style is the main advantage of using the RESTful WS, which adopts the Web interoperability style. Because of the characteristics of this style, we

showed that the proposed RISE middleware had first solved the research problem of hiding systems heterogeneity (implementation) in components while allowing composition scalability and dynamicity. Hiding implementations indicate that software related design and implementations issues become system internal issues, hence irrelevant to other external systems. This is one of the major contributions of the RISE middleware (comparing to current approaches) of being able to decouple systems implementations. Further, the solution of this problem consequently allowed RISE to meet the desired objectives such as decoupling systems implementations in the way information is exchanged, accessed and described. In this case, RISE presented all services as types in resources (addressed by URI templates). Thus, the resources (i.e. services types) can be created and named with URIs at runtime. Regardless of the number of URIs (resources) in the distributed environment, they are always automatically connected with the same HTTP methods (channels). Furthermore, as channels are realized outside implementations and resources synchronized their activities in XML messages, resources APIs are decoupled from implementations. Thus, systems heterogeneity is hidden. Solving this problem allows RISE to be a general middleware and to provide experimental frameworks that are literally attached to the Web (because they are externally seen as URIs similar to any other URIs on the Web).

To prove the concept of general middleware that is by being able to hold different simulation environments. We interfaced the CD++ engine to RISE so that distributed simulation between different CD++ engines (each placed in a partition and in charge of simulating a portion of the entire distributed CD++ model)s. In this case, the algorithms synchronize the activities between partitions URIs by exchanging XML messages. To

reduce remote messages transmission through the Internet, we developed an aggregation scheme to group multiple simulation messages in XML. These algorithms performed substantially better when compared to the SOAP-based DCD++. Furthermore, we showed that using RISE methods was able decouple distributed synchronization from software design specifics. In this case, the algorithms presented in Chapter 8 were designed as the synchronization rules and the messages that they require to communicate without dictating the way systems have to implement them. The RISE experiment blueprint was also used to automate the simulation experimentation process via using the concept of workflows. In this case, workflows (at the client side) can dynamically create and manipulate various experiments simultaneously on RISE (the server side).

The outcomes of the research showed that the interoperability syntactic and structural rules played a major role for enhancing the interoperability methods at the software level. This is mainly because of applying the RESTful WS principles, which adopts the WWW interoperability principles. However, as in the case of any technology, the decision to use such technology can be different from a project to another. Therefore, we recommend the use of RESTful WS principles in Web-service based projects that contain some or all of the following characteristics:

1. Projects that desire to interoperate with applications that use the Web interoperability style such as Web 2.0 and mashup solutions. In fact, REST eases interoperability with any application attached to the Web (e.g. Web browser) because they use the same methods of interoperability. It is worth to note that the SOAP-based WS creates another RPC layer for applications to interoperate above the Web layer.

2. Projects that are expecting to interoperate with systems outside their control. In this case, as we did in this thesis, systems APIs can be moved outside systems implementations, allowing systems implementations decoupling. This allows systems development to evolve independently from each other.
3. Projects that are expecting to scale well when composing large number of systems. In this case, the REST style has been proving to work well on the WWW, which REST imitates.
4. Projects that are expecting to have different components join/disjoin the distributed structure dynamically at run time. The REST (Web) style has been proven to work on the WWW by having countless number of systems join/disjoin all the time.

However, the above points may not work well for every project. For example, it may be a better choice to migrate an existing CORBA-based system to a SOAP-based WS. This is because both styles are very similar to each other.

9.2 Future Work

The following list includes a number of topics for future research in the context of simulation interoperability and access via the Web:

- **RISE-Enabling Additional Simulation and Visualization Environments**

The RISE middleware is designed as general container to hold more simulation services beside the DCD++ presented in Chapter 6. From a design viewpoint, those new services (e.g. Parallel CD++ [93]) can be interfaced with RISE via the IPC queues as we did in Chapter 6. However, from the API viewpoint (Appendix-B), adding new services

to the URI template structure is similar to regular Web site URIs. For example, Figure 76 shows example of three types of additional services: Visualization, Conservative simulation, and Parallel simulation systems.

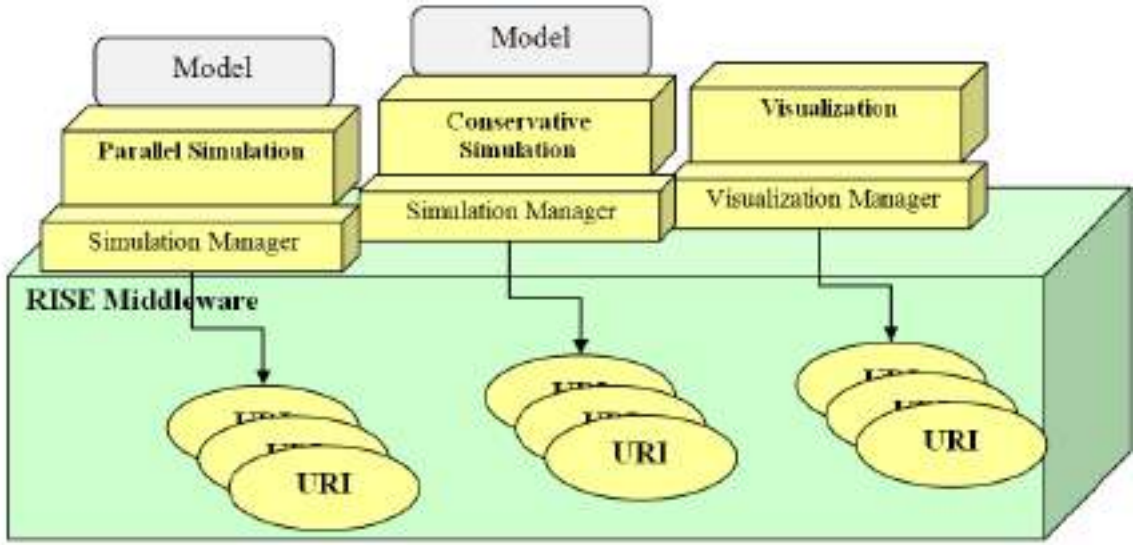

Figure 76: An Overview of Additional RISE-based Services

The Simulation/Visualization managers shown in Figure 76 are actually part of the RISE middleware layer. The Manager component usually extends the RISE generic component to handle an environment specifics such *data distribution* across the distributed environment (see the DCD++ implementation extension in Appendix-A). For example, the Visualization environment (e.g. RUBE [53]) can interoperate with the open Second Life visualization environment [120]. In this case, the visualization manager, for instance, manages the visualization representations and their distributions to registered clients who view them locally. The other shown simulation environments need to perform simulation *time management* according to their specific mechanisms. For example, the parallel simulation might be similar to PCD++ [93], which executes optimistic-based simulation on parallel machines. However, if a simulation environment

needs to synchronize simulation with other remote systems to perform distributed simulation, the simulation manager also needs to handle *data distribution* mechanisms. This separation of functionalities is similar to the DCD++ environment presented in Chapter 6.

- **Mashups: Putting Information in Simulation Loop**

The *mashup* concept groups various services from different providers and presents them as a bundle in order to provide single integrated service. Different RESTful WS based mashup tools already exist to allow enterprise mashup such as the IBM Mashup Center [70]. Thus, this concept can be used to put information in simulation loop.

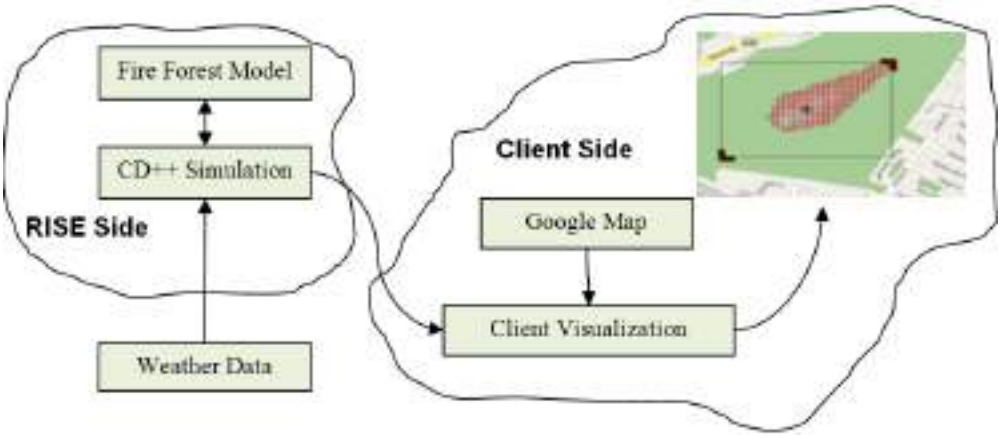

Figure 77: Mashup Example

One of the interesting directions is to integrate RISE services (e.g. DCD++) with the IBM mashup tools. Those tools allow users to click and drag on RESTful services (called widgets) like Google maps to compose them together. The other direction is to develop mashup application to integrate different RESTful services, including RISE services. For example, Figure 77 shows an example of mashing up three sources: (1) CD++ simulation of the *Fire Forest* model (on RISE side), (2) real Weather Web-service

input, and (3) Google map (on Client side). In this case, the weather data is input in real-time to the fire spreading simulation and displayed on top of Google map.

- **Data Fusion (DF)**

Data fusion (DF) is defined as collecting information from different sources to achieve inferences, which potentially leads to better accuracy from relaying on a single source of information [123]. DF is applied by the military to build integrated images from various information sources in battlefields [123]. In this case, information can be combined and forwarded to different simulators at runtime. DF applications go beyond defense sector. These applications include manufacturing, health and environment [26]. DF is highly dynamic and requires infinite composition scalability, which RISE can help in this regard. In this case, DF URIs can be created by various software systems at run time so that messages can be exchanged by those URIs. Further, semantics standards are being developed (i.e. called Battle Management Language (BML) [68][119]), which brings DF a huge step toward being fully interoperated in a simulation loop. BML is an explicit standardized language to express commands for real troops, simulated troops, and robotic forces.

REFERENCES

[1] Al-Zoubi K.; Wainer, G. “Interfacing and Coordination for a DEVS Simulation Protocol Standard”. Proceedings of the IEEE/ACM Distributed Simulation and Real-Time Applications (DS-RT 2008). Vancouver, BC, Canada. 2008.

[2] Al-Zoubi K.; Wainer, G. “Performing Distributed Simulation with RESTful Web-Services Approach”. Proceedings of the Winter Simulation Conference (WSC 2009). Austin, TX, USA. 2009.

[3] Al-Zoubi K.; Wainer, G. “Using REST Web Services Architecture for Distributed Simulation”. Proceedings of Principles of Advanced and Distributed Simulation (PADS 2009), Lake Placid, New York, USA. 2009.

[4] Al-Zoubi K.; Wainer, G.; “Managing Simulation Workflow Patterns using Dynamic Service-Oriented”. Proceedings of the Winter Simulation Conference (WSC 2010). Baltimore, Maryland, USA. 2010.

[5] Al-Zoubi K.; Wainer, G. “RISE: REST-ing Heterogeneous Simulation Interoperability”. Proceedings of the Winter Simulation Conference (WSC 2010). Baltimore, Maryland, USA. 2010.

[6] Anita A., Gordon M., David S. "Aggregate Level Simulation Protocol (ALSP) 1993 Confederation Annual Report", the MITRE Corporation. 1993. <http://ms.ie.org/alsp/biblio/93_annual_report/93_annual_report_pr.html>. Accessed March 2009.

[7] Apache Tomcat. <http://tomcat.apache.org/>. Accessed October 2008.

[8] Atom. <http://tools.ietf.org/html/rfc4287>. Accessed November 2010.

[9] Babineau W., Barry P., Furness Z., "Automated Testing within the Joint Training confederation (JTC)", Proceedings of the Fall 1998 Simulation Interoperability Workshop, Orlando, FL, USA. September 1998.

[10] Banks J.; Carson J.; Nelson B.; Nicol D. “Discrete-Event System Simulation”. Pearson Prentice Hall. Upper Saddle River, NJ. 2005.

[11] Banks C. "Introduction to Modeling and Simulation". Chapter 1 in book “Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains”. Catherine Banks, John Sokolowski Editors. Wiley. New Jersey, 2010.

[12] Boer C., Bruin A., Verbraeck A. “Distributed simulation in industry -- a survey, part 3 -- the HLA standard in industry”. Proceedings of Winter Simulation Conference (WSC 2008). Miami, FL, USA. 2008.

[13] Boer C., Bruin A. and Verbraeck A. “A survey on distributed simulation in industry”. Journal of Simulation. Vol. 3, No. 1, pp. 3–16. March 2009.

[14] Bonan H.; Yiping Y.; Bing W.; "Mapping from BOM conceptual model definition to PDES models for enhancing interoperability". Proceedings of IEEE 7th International Conference on System Simulation and Scientific Computing (ICSC2008). Beijing, China. October 2008.

[15] Boon G.; Lendermann, P.; Yoke, M.; Low, H.; Turner, S.J.; Xiaoguang W.; Taylor, S.J.E.; "Interoperating AutoSched AP using the High Level Architecture". Proceedings of the Winter Simulation Conference (WSC2005). Orlando, FL, USA. December 2005.

[16] Boukerche, A.; Gu, Y.; "An Efficient Adaptive Transmission Control Scheme for Large-Scale Distributed Simulation Systems". IEEE Transactions on Parallel and Distributed Systems (TPDS), Vol. 20, No. 2, pp. 246-260. February 2009.

[17] Boukerche, A.; Zhang M.; Shadid A.; "DEVS Approach to Real-time RTI Design for Large-scale Distributed Simulation Systems". Simulation. Vol. 84, No. 5, pp. 231-238. May 2008.

[18] Boukerche, A.; Iwasaki, F.M.; Araujo, R.; Pizzolato ,E.B. "Web-Based Distributed Simulations Visualization and Control with HLA and Web Services". Proceedings of the IEEE/ACM Distributed Simulation and Real-Time Applications (DS-RT 2008). Vancouver, BC, Canada. 2008.

[19] Boukerche A., Zhang M., Xie H. "An Efficient Time Management Scheme for Large-Scale Distributed Simulation Based on JXTA Peer-to-Peer Network". Proceedings of the IEEE/ACM Distributed Simulation and Real-Time Applications (DS-RT 2008). Vancouver, BC, Canada. 2008.

[20] Boukerche A., Shadid A., Zhang M. "Efficient Load Balancing Schemes for Large-Scale Real-Time HLA/RTI Based Distributed Simulations". Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2007). Chania, Crete Island, Greece. 2007.

[21] Boukerche A., Lu K. "Design and performance evaluation of a real-time RTI infrastructure for large-scale distributed simulations". Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2005). Montreal, Quebec, Canada. October 2005.

[22] Boukerche A., McGraw N.J., Dzermajko C., Lu K., "Grid-Filtered Region-Based Data Distribution Management in Large-Scale Distributed Simulation Systems". Proceedings of 38th Ann. Simulation Symp. (ANSS '05). 2005.

[23] Boukerche A., Roy A., "Dynamic Grid-Based Approach to Data Distribution Management," Journal of Parallel and Distributed Computing, Vol. 62, No. 3, pp. 366-392, 2002.

[24] Booth D., Haas H., McCabe F., Newcomer E., Champion M., Ferris C., Orchard D. "Web Services Architecture". 2004. <http://www.w3.org/TR/ws-arch/>. Accessed November 2010.

[25] Box D., Ehnebuske D., Kakivaya G., Layman A., Mendelsohn N., Nielsen H., Thatte S., Winer D. “Simple Object Access Protocol (SOAP) 1.1”. May 2000. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>. Accessed March 2009.

[26] Brooks R., Iyengar S., Multi-sensor Fusion: Fundamentals and Applications with Software, New Jersey; Prentice Hall,1998.

[27] Brown P. “Information Architecture with XML: A Management Strategy”. Wiley. England, 2003.

[28] Bryant, R. E. “Simulation of packet communication architecture computer systems”. Massachusetts Institute of Technology. Cambridge, MA, USA. 1977.

[29] Byrne J., Heavey C., Byrne P. “A review of Web-based simulation and supporting tools”. Simulation Modelling Practice and Theory. Vol. 18, No. 3, pp. 253-276. March 2010.

[30] CAE RTI. <http://www.cae.com>. Accessed October 2010.

[31] Calvin, J.; Dickens, A.; Gaines, B.; Metzger, P.; Miller, D.; Owen, D.; “The SIMNET virtual world architecture”. Proceedings of Virtual Reality Annual International Symposium (IEEE VRAIS 1993). Los Alamitos, CA, USA. 1993.

[32] Cappelaere, P.; Frye, S.; Mandl, D. “Flow-enablement of the NASA SensorWeb using RESTful (and secure) workflows”. 2009 IEEE Aerospace conference. Big Sky, Montana, USA. March 2009.

[33] CD++ toolkit. <http://cell-devs.sce.carleton.ca>. Accessed March 2010.

[34] Chandy, K. M. and J. Misra. “Distributed Simulation: A Case Study in Design and Verification of Distributed. Programs”. IEEE Transactions on Software Engineering. Vol. SE-5, No. 5, pp. 440-452. 1979.

[35] Chandrasekaran, S.; Silver, G.; Miller, J.A.; Cardoso, J.; Sheth, A.P.; “Web service technologies and their synergy with simulation”. Proceedings of Winter Simulation Conference (WSC 2002). San Diego, California, USA, USA. 2002.

[36] Cheon, S.; Seo, C.; Park, S.; Zeigler, B.P. “Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System”. Proceedings of the Advanced Simulation Technologies Conference, Arlington Virginia. April, 2004.

[37] Cho, Y.K.; Zeigler, B.P.; Sarjoughian, H.S.; “Design and implementation of distributed real-time DEVS/CORBA”. Proceedings of IEEE International Conference on Systems, Man and Cybernetics (ICSMC2001). Tucson, AZ, USA. 2001.

[38] Chow, A.; Zeigler, B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism”. Proceedings of Winter Simulation Conference (WSC 1994). Lake Buena Vista, FL, USA. 1994.

[39] Christensen E., Curbera F., Meredith G., Weerawarana S. “Web Services Description Language (WSDL) 1.0”. <http://xml.coverpages.org/wsdl20000929.html>. Accessed March 2009.

[40] Christensen, E; Curbera, F.; Meredith, G.; Weerawarana, S “Web Service Desctiption Language (WSDL) 1.1”. March, 2001. <http://www.w3.org/TR/wsdl>. Accessed March 2009.

[41] Chinnici R., Moreau J., Ryman A., Weerawarana S. “Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language”. June 2007. <http://www.w3.org/TR/wsdl20/>. Accessed October 2010.

[42] Chinthaka E. “Enable REST with Web services, Part 1: REST and Web services in WSDL 2.0”. <http://www.ibm.com/developerworks/webservices/library/ws-rest1/>. Accessed November 2010.

[43] Chung M., Kyung C. “Improving Lookahead in Parallel Multiprocessor Simulation Using Dynamic Execution Path Prediction”. Proceedings of Principles of Advanced and Distributed Simulation (PADS 2006). Singapore. May 2006.

[44] Cubert R., Fishwick R., “A framework for distributed object-oriented multimodeling and simulation”. Proceedings of Winter Simulation Conference (WSC 1997). Atlanta, Georgia, USA. December 1997.

[45] Davis P., Anderson R. “Improving the Composability of Department of Defense Models and Simulation”. Santa Monica, CA, Rand Corporation. 2003. <http://www.rand.org/pubs/monographs/MG101.html>. Accessed May 2011.

[46] DEVSJAVA. "Extensible Modeling and Simulation Framework (XMSF) C4I Testbed". < http://www.acims.arizona.edu/SOFTWARE/software.shtml#DEVSJAVA>.

[47] DEVS Standardization Group. <http://cell-devs.sce.carleton.ca/devsgroup/>. Accessed March 2011.

[48] DuBois P. “MySQL”. 4th edition. Addison-Wesley. 2009.

[49] Erl T., Karmarkar A., Walmsley P., Haas H., Yalcinalp, L.U., Liu K. Orchard D., Tost A., and Pasley J. “Web Service Contract Design and Versioning for SOA”. Prentice Hall. 2008.

[50] Fielding, R. T. “Architectural Styles and the Design of Network-based Software Architectures”, Doctoral dissertation, University of California, Irvine, 2000. Available at: <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>. Accessed October 2008.

[51] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T. “Hypertext Transfer Protocol -- HTTP/1.1”. RFC 2616. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>. Accessed October 2008.

[52] Foster, I., Kesselman C. “Globus: A Metacomputing Infrastructure Toolkit”. International Journal on Supercomputer Applications, Vol. 11, No. 2, pp: 115-128. 1997.

[53] Fishwick, P., Jinho L., Minho P. “RUBE: a customized 2D and 3D modeling framework for simulation” Proceedings of Winter Simulation Conference (WSC 2003). Gainesville, FL, USA. December 2003.

[54] Franks J.; Hallam-Baker P.; Hostetler J.; Lawrence S.; Leach P.; Luotonen P.; Stewart L. “HTTP Authentication: Basic and Digest Access Authentication” RFC 2617. <http://www.ietf.org/rfc/rfc2617.txt>. Accessed October 2008.

[55] Frécon E.; Stenius M. “DIVE: A scalable network architecture for distributed virtual environments”, Distributed Systems Engineering Journal. Vol. 5, No. 3, pp. 91-100. September 1998.

[56] Fischer M. "Aggregate Level Simulation Protocol (ALSP) - Future Training with Distributed Interactive Simulations", U. S. Army Simulation, Training and Instrumentation Command. International Training Equipment Conference. The Hague, Netherlands. 1995.

[57] Fitzgibbons J., Fujimoto R., Fellig D., Kleban D., Scholand A. “IDSim: An extensible framework for interoperable distributed simulation” Proceedings of the IEEE International Conference on Web Services (ICWS2004). San Diego, California, USA. 2004.

[58] Fujimoto, R. M. “Parallel and distribution simulation systems”. New York: John Wiley & Sons. 2000.

[59] Fujimoto, R.; Hunter, M.; Sirichoke, J.; Palekar, M.; Kim, H.; Suh Wonho. “Ad Hoc Distributed Simulations”. Proceedings of Principles of Advanced and Distributed Simulation (PADS 2007). San Diego, California, USA. June 2007.

[60] Gan, B. P.; Liu, L.; Jain, S.; Turner, S. J.; Cai, W. T. and Hsu, W.J. “Distributed Supply Chain Simulation Across the Enterprise Boundaries”. Proceedings of Winter Simulation Conference (WSC 2000). Orlando, FL, USA. December 2000.

[61] Goetz B.; Peierls T.; Bloch J.; Bowbeer J.; Holmes D.; Lea D. “Java Concurrency in Practice”. Addison-Wesley Professional. 2006.

[62] Gregorio J. URI Templates. <http://bitworking.org/projects/URI-Templates/>. Accessed October 2008.

[63] Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.; Nielsen, H. “SOAP Version 1.2 Part 1: Messaging Framework”. 2003. <http://www.w3.org/TR/soap12-part1/>. Accessed October 2008.

[64] Gong Li; Liu Gao-Feng; Liu Zhong; An Ru-Kui; “An opened model with Web Service in discrete event simulation”. Proceedings of Future Computer and Communication (ICFCC 2010). Wuhan, China. 2010.

[65] Halpin, D.W.; Jen, H.; Kim, J.; “A construction process simulation Web service”. Proceedings of Winter Simulation Conference (WSC 2003). New Orleans, Louisiana,USA. 2003.

[66] Henning M., “The Rise and Fall of CORBA”. Communications of the ACM. Vol. 51, No. 8, August 2008. Also available at <http://queue.acm.org/detail.cfm?id=1142044>. Accessed March 2010.

[67] Henning, M., and S. Vinoski. “Advanced CORBA programming with C++”. Addison–Wesley. 1999.

[68] Hieb, M.R. and Schade, U., “Formalizing Command Intent Through Development of a Command and Control Grammar.” in 12th ICCRTS. Newport, Rhode Island, June 2007.

[69] Hong, L; Wan, H; Wang, Y; Chen, X.; Zou, D. “Extending HLA/RTI to WAN Based on Grid Service”. Proceedings of IEEE Asia-Pacific Services Computing Conference (APSCC2008). Yilan, China, 2008.

[70] IBM Mashup Center. <http://www-01.ibm.com/software/info/mashup-center/>. Accessed June 2009.

[71] IBM. “Why Mashups Matter”. <ftp://ftp.software.ibm.com/software/lotus/lotusweb/portal/why_mashups_matter.pdf>. Accessed June 2009.

[72] IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA) - frameworks and rules. Technical Report 1516, IEEE (2000).

[73] IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA) - federate interface specification. Technical Report 1516.1, IEEE (2000).

[74] IEEE: Standard for modeling and simulation (M&S) High Level Architecture (HLA) - object model template (OMT) specification. Technical Report 1516.2, IEEE (2000).

[75] IEEE 1278.1-1995 - Standard for Distributed Interactive Simulation - Application protocols.

[76] IEEE-1278.2-1995 - Standard for Distributed Interactive Simulation - Communication Services and Profiles.

[77] IEEE 1278.3-1996 - Recommended Practice for Distributed Interactive Simulation - Exercise Management and Feedback.

[78] IEEE 1278.4-1997 - Recommended Practice for Distributed Interactive - Verification Validation & Accreditation.

[79] Jacobs I., Walsh N. “Architecture of the World Wide Web, Volume One”. <http://www.w3.org/2001/tag/webarch/>. 2004. Accessed October 2008.

[80] Jefferson, D. R. “Virtual time”. ACM Transactions on Programming Languages and systems. Vol. 7. No. 3. pp. 405-425. 1985.

[81] JXTA. <https://jxta.dev.java.net/>. Accessed March 2009.

[82] Khul F., Weatherly R., Dahmann J.: “Creating Computer Simulation Systems: An Introduction to High Level Architecture”. Prentice Hall (1999).

[83] Kakivaya G., Layman A., S. Thatte S., Winer D. “SOAP: Simple Object Access Protocol”. Version 1.0. 1999. <http://www.scripting.com/misc/soap1.txt>. Accessed March 2009.

[84] Ke P.; Turner, S..; Wentong C.; Zengxiang L.. “A Service Oriented HLA RTI on the Grid”. Proceedings of IEEE International Conference on Web Services (ICWS 2007). Salt Lake City, Utah, USA. July 2007.

[85] Kepler. <https://kepler-project.org/>. Accessed March 2010.

[86] Khargharia, B.; Hariri, S.; Parashar, M.; Ntaimo, L.; Kim, B. “vGrid: A Framework for Building Autonomic Applications”. Proceedings of the 1st International Workshop on Challenges for Large Applications in Distributed Environments (CLADE 2003), pp. 19-26. 2003.

[87] Khul F., Weatherly R., Dahmann J.: “Creating Computer Simulation Systems: An Introduction to High Level Architecture”. Prentice Hall (1999).

[88] Kim, K.; Kang, W. “CORBA -Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One”. Proceedings of the International Conference on Computational Science and its Applications (ICCSA2004). Assisi, Italy. 2004.

[89] Kim, K. and Kang, W., "A Web Service Based Distributed Simulation Architecture for Hierarchical DEVS Models". Proceedings of the 13th International Conference on AI, Simulation, Planning in High Autonomy Systems, (AIS 2004). Jeju Island, Korea, October 2004.

[90] Kumaran, S.; Rong Liu; Dhoolia, P.; Heath, T.; Nandi, P.; Pinel, F. “A RESTful Architecture for Service-Oriented Business Process Execution”. Proceedings of IEEE International Conference on e-Business Engineering (ICEBE '08). Xi’an, China. October 2008.

[91] Lenoir, T. and Lowood, H. “Theaters of wars: the military – entertainment complex”. <http://www.stanford.edu/class/sts145/Library/Lenoir-Lowood_TheatersOfWar.pdf>. Accessed March 2009.

[92] Liang, S. “Java Native Interface (JNI), Programmer’s Guide and Specification”. Addison-Wesley. 1999.

[93] Liu, Q., and Wainer G., “Parallel Environment for DEVS and Cell-DEVS Models”, SIMULATION, Vol. 83, No. 6, pp. 449-471, 2007.

[94] MÄK High Performance RTI. <http://www.mak.com/products/rti.php>. Accessed March 2009.

[95] Mandel L. “Describe REST Web services with WSDL 2.0”. <http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/>. Accessed May 2009.

[96] Mathure, M.A.; Jonnalagadda, V.; Zalewski, J.; “Heterogeneous architecture and testbed for simulation of large-scale real-time systems”. Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2003). Delft, Netherlands. October 2003.

[97] Mattern, F. “Efficient algorithms for distributed snapshots and global virtual time approximation”. Journal of parallel and distributed computing. Vol. 18, No. 4. pp. 423-434. August 1993.

[98] McFaddin, S.; Coffman, D.; Han, J.H.; Jang, H.K.; Kim, J.H.; Lee, J.K.; Lee, M.C.; Moon, Y.S.; Narayanaswami, C.; Paik, Y.S.; Park, J.W.; Soroker, D. “Modeling and Managing Mobile Commerce Spaces Using RESTful Data Services”. 9th IEEE International Conference on Mobile Data Management (MDM'08). Beijing, China. April 2008.

[99] McLean T., Fujimoto R., Fitzgibbons B.” Middleware for real-time distributed simulations”. Journal of Concurrency and Computation: Practice and Experience. Vol. 16 No. 15, pp. 1483-1501. November 2004.

[100] McGrath, D.; Hunt, A.; Bates, M.; “A simple distributed simulation architecture for emergency response exercises”. Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2005). Montreal, Quebec, Canada. October 2005.

[101] Mittal S., Risco-Martín J.L., and Zeigler B.P., “DEVS-based simulation web services for net-centric T\&E,” in Proceedings of the 2007 summer computer simulation conference (SCSC2007). San Diego, California, USA. 2007.

[102] Möller, B. and Dahlin, C. “A First Look at the HLA Evolved Web Service API”. Proceedings of 2006 Euro Simulation Interoperability Workshop, Simulation Interoperability Standards Organization. 06E-SIW-061. 2006.

[103] NetBeans IDE <http://www.netbeans.org/)>. Accessed June 2009.

[104] Noelios Restlet Engine (NRE). <http://www.noelios.com/products/restlet-engine>. Accessed October 2008.

[105] O'Reilly T. “What Is Web 2.0”. <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>. Accessed May 2009.

[106] Page E., Briggs R, Tufarolo J. “Toward a family of maturity models for the simulation interconnection problem”. Proceedings of the Simulation Interoperability Workshop. Arlington, VA. April 2004.

[107] Page E., “Beyond speedup: PADS, the HLA and web-based simulation”. Proceedings of Winter Simulation Conference (WSC 1999). Atlanta, Georgia. December 1999.

[108] Papazoglou, M. “Web Services: Principles and Technology”. Prentice Hall. 2007.

[109] Papelis Y., Madhavan P. "Modeling human behavior". Chapter 9 in book “Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains”. Catherine Banks, John Sokolowski Editors. Wiley. New Jersey, 2010.

[110] poRTIco. <http://www.porticoproject.org/index.php?title=Main_Page>. Accessed March 2009.

[111] pRTI™. <http://www.pitch.se/products/prti>. Accessed March 2009.

[112] Padmanabhuni S., Chaudhari P., Bharti S., Kumar S. “WSDL 2.0: A Pragmatic Analysis and an Interoperation Framework”. June 2007. <http://soa.sys-con.com/node/219029>. Accessed October 2010.

[113] Petty M., Weisel E. “A composability lexicon”. Proceedings of the Spring Simulation Interoperability Workshop. Orlando, FL. March 2003.

[114] Restlet API. <http://www.restlet.org/>. Accessed October 2008.

[115] Richardson L., Ruby S. “RESTful Web Services”, 1st edition. O’Reilly Media, Inc., Sebastopol, California. 2007.

[116] RSS. <http://www.rssboard.org/rss-specification>. Accessed November 2010.

[117] Samadi, B. “Distributed simulation, algorithms and performance analysis”. Computer science department, PhD Thesis, University of California, Los Angeles. 1985.

[118] Sawhney, M.; Verona, G.; Prandelli, E. “Collaborating to create: The Internet as a platform for customer engagement in product innovation”. Journal of Interactive Marketing, Vol. 19, No. 4, pp. 4-17. Fall 2005.

[119] Schade, U. and Hieb, M.R., "Formalizing Battle Management Language: A Grammar for Specifying Orders," 06S-SIW-068, in Spring Simulation Interoperability Workshop, April 2006.

[120] Second Life. <http://secondlife.com/>. Accessed May 2011.

[121] Seo, C.; Park, S.; Kim, B.; Cheon, S.; Zeigler, B. “Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment”. Proceedings of Advanced Simulation Technologies conference (ASTC2004). Arlington, VA. USA. 2004.

[122] Simple Framework. <http://www.simpleframework.org/>. Accessed March 2009.

[123] Shahbazian E., “Introduction to DF: Models and Processes, Architectures, Techniques and Applications,” in Multisensor Fusion, Kluwer Academic Pulishers, 2000, pp. 71-97.

[124] Strassburger, S.; “The Road to COTS-Interoperability: From Generic HLA-Interfaces Towards Plug-and-Play Capabilities”. Proceedings of Winter Simulation Conference (WSC 2006). Monterey, CA, USA. December 2006.

[125] Strassburger S., Schulze T., Fujimoto R. “Future trends in distributed simulation and distributed virtual environments: results of a peer study”. Proceedings of Winter Simulation Conference (WSC 2008). Miami, FL, USA. 2008.

[126] Stirbu, V. “Towards a RESTful Plug and Play Experience in the Web of Things”. IEEE International Conference on Semantic Computing (ICSC 2008). Santa Clara, CA, USA. August 2008.

[127] Tan G., Xu L., Moradi F., Zhang YS, “An Agent-based DDM Filtering Mechanism”. Proceedings of MASCOTS, San Francisco, USA, Aug 2000.

[128] Taha, H.A. “Simulation with SIMNET II”. Proceedings of Winter Simulation Conference (WSC 1991). Phoenix, Arizona, USA. December 1991.

[129] Taha, H.A. “Introduction to SIMNET v2.0”. Proceedings of Winter Simulation Conference (WSC 1988). San Diego, California, USA. December 1988.

[130] Tolk A. "Interoperability and Composability". Chapter 12 in “Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains”. Catherine Banks, John Sokolowski Editors. Wiley. New Jersey, 2010.

[131] Trident. <http://connect.microsoft.com/Trident>. Accessed March 2010.

[132] Taylor S., Strassburger S., Turner, S., Low, M.., Xiaoguang W., Ladbrook, J.; “Developing Interoperability Standards for Distributed Simulation and COTS Simulation Packages with the CSPI PDG”. Proceedings of Winter Simulation Conference (WSC 2006). Monterey, California, USA. December 2006.

[133] Taylor S., Xiaoguang W., Turner S., Low M., “Integrating heterogeneous distributed COTS discrete-event simulation packages: an emerging standards-based approach”. IEEE Systems, Man, and Cybernetics Society. Vol. 36, No. 1, pp. 109-122. January 2006.

[134] Tuecke, S., Foster I., et al. “Open Grid Services Infrastructure (OGSI) Version 1.0. Open Grid Services Infrastructure Working Group (OGSI-WG)”. June 2003. <http://xml.coverpages.org/OGSI-SpecificationV110.pdf>. Accessed May 2011.

[135] Ulriksson, J.; Ayani, R.; “Consistency Overhead using HLA for Collaborative Work”. Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2005). Montreal, Quebec, Canada. October 2005.

[136] Van der Aalst W., ter Hofstede A. “YAWL: Yet Another Workflow Language”. Information Systems. Vol. 30, No. 4, pp. 245-275. June 2005.

[137] Wainer, G.; Madhoun, R.; Al-Zoubi, K. “Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web Services”. Simulation Modelling Practice and Theory. Vol. 16, No. 9, pp. 1266-1292. October 2008.

[138] Wainer, G. and Al-Zoubi, K. "An Introduction to Distributed Simulation". Chapter 11 in book “Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains”. Catherine Banks, John Sokolowski Editors. Wiley. New Jersey, 2010.

[139] Wainer, G. “Discrete-Event Modeling and Simulation: A Practitioner's Approach”. CRC press, Taylor & Francis Group. Boca Raton, Florida. 2009.

[140] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “DEVS Standardization: Foundations and Trends”. Chapter 15, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

[141] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “An Introduction to DEVS Standardization”. Chapter 16, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

[142] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “Standardizing DEVS Model Representation”. Chapter 17, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

[143] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. “Standardizing DEVS Simulation Middleware”. Chapter 18, “Discrete-Event Modeling and Simulation: Theory and Applications.” G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. December 2010.

[144] Web Application Description Language (WADL). <https://wadl.dev.java.net/>. Accessed October 2008.

[145] Web Services-Axis. <http://ws.apache.org/axis/>. Accessed October 2008.

[146] Weske M. “Business Process Management: Concepts, Languages, Architectures”. Springer-Verlag Berlin Heidelberg. New York. 2007.

[147] Wilson B.J. “JXTA”. New Riders Publishing. 2002.

[148] Workflow Management Coalition. < http://www.wfmc.org/ >. Accessed October 2009.

[149] W3C. "W3C Semantic Web Frequently Asked Questions". <http://www.w3.org/2001/sw/SW-FAQ>. Accessed March 2008.

[150] XML. “Extensible Markup Language (XML) 1.0 (Fifth Edition)”. < http://www.w3.org/TR/REC-xml/>. 2008. Accessed November 2010.

[151] Zelle J. “Python Programming: An Introduction to Computer Science”. 2nd Edition. Franklin, Beedle & associates. 2004.

[152] Zeigler, B.P.; Doohwan K. “Distributed supply chain simulation in a DEVS/CORBA execution environment”. Proceedings of Winter Simulation Conference (WSC 1999). Phoenix, Arizona, USA. December 1999.

[153] Zeigler, B.; Kim, T.; Praehofer, H. “Theory of Modeling and Simulation”. 2nd Edition. Academic Press. 2000.

[154] Zeigler, B.; Hammods, P. “Modeling & Simulation-Based Data Engineering: Pragmatics into Ontologies for Net-Centric Information Exchange”. Academic Press. 2007.

[155] Zhang, M.; Zeigler, B.; Hammonds, P. "DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies". < http://acims.eas.asu.edu/PUBLICATIONS/PDF/devsRmi.pdf >. Accessed July 2009.

[156] Zhang H.; Wang H.; Chen, D. “Integrating web services technology to HLA-based multidisciplinary collaborative simulation system for complex product development”. Proceedings of IEEE 12th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2008). Xi'an, China. April 2008.

[157] Zhao H., Georganas N.D.”HLA Real-time Extension”. Proceedings of the IEEE Distributed Simulation and Real-Time Applications (DS-RT 2001). Cincinnati, Ohio, USA. 2008.

[158] Zhu H.; Li G.; Zheng L. “Introducing Web Services in HLA-based simulation application”. Proceedings of IEEE 7th World Congress on Intelligent Control and Automation (WCICA 2008). Chongqing, China. June 2008.

[159] Zhu S. Du Z. Chai X. “GDSA: A Grid-Based Distributed Simulation Architecture”. Proceedings of the sixth IEEE International Symposium on Cluster Computing and the Grid Workshops (CCGRID 2006). Singapore. May 2006.

APPENDIX-A: RISE MIDDLEWARE IMPLEMENTATION

RISE middleware is implemented in Java and can be deployed as a Servlet running within any HTTP container (e.g. Tomcat [7]) or as a standalone HTTP server directly on top of TCP/IP. From the implementation viewpoint, there is no difference between the standalone or Servlet version except in the way it is packaged and deployed. This appendix describes the RISE five subsystems (i.e. java packages) and the RISE types of deployment.

A.1 RISE Subsystems

Figure 78 shows the implementation overview of the RISE Middleware. It consists of five java packages shown in Figure 78. RISE subsystems (Figure 78) summarized as follows:

1. The Main subsystem (Section A.1.1): It initializes the server application and major components such as communication, server logging, database, RISE URIs internal router, and the shutdown and cleaning routines.
2. Data subsystem (Section A.1.2): It holds and organizes the server database. The Database holds two object types: the server general configuration and the user specific data sections.
3. Resources subsystem (Section A.1.3): It handles received HTTP requests. Each URI request is processed by a java class in this package, which also generates the HTTP response.

4. Utility subsystem (Section A.1.4): This package provides the XML parsing utilities, server logging (based on java logging), file services and the constructing of HTML documents.
5. SimulationAdmin subsystem (Section A.1.5): It manages/wraps active simulation services. For example, Java classes in this subsystem synchronize simulation with remote simulation entities to ensure correct and efficient distributed simulation.

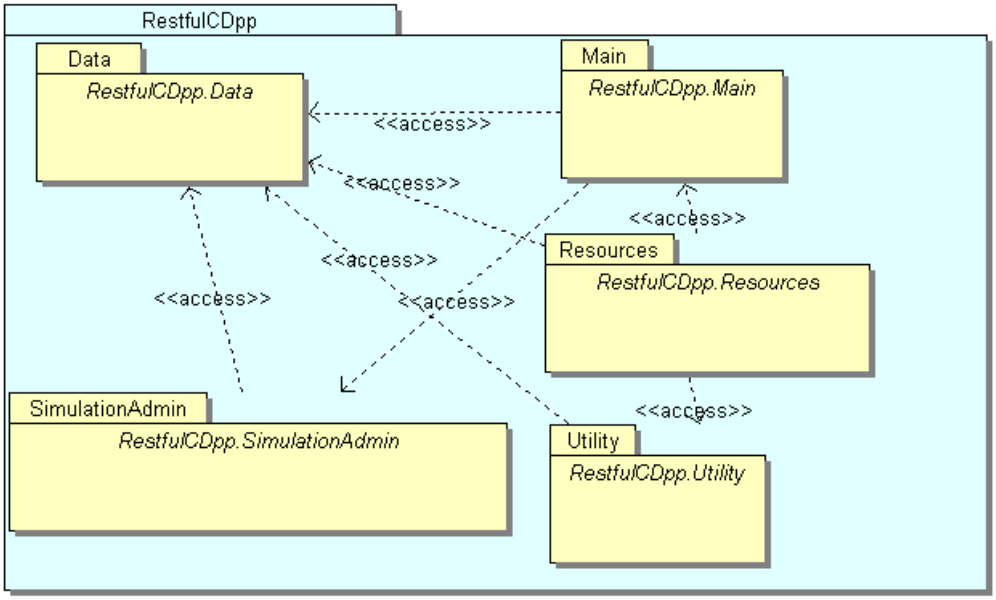

Figure 78: RISE Server Architecture Overview

A.1.1 Main Subsystem

Figure 79 shows the Main subsystem Java classes and their interaction with other subsystem classes. This subsystem contains four major classes: *Application*, *ThreadPool*, *ServletListener* and *ShutdownHook*.

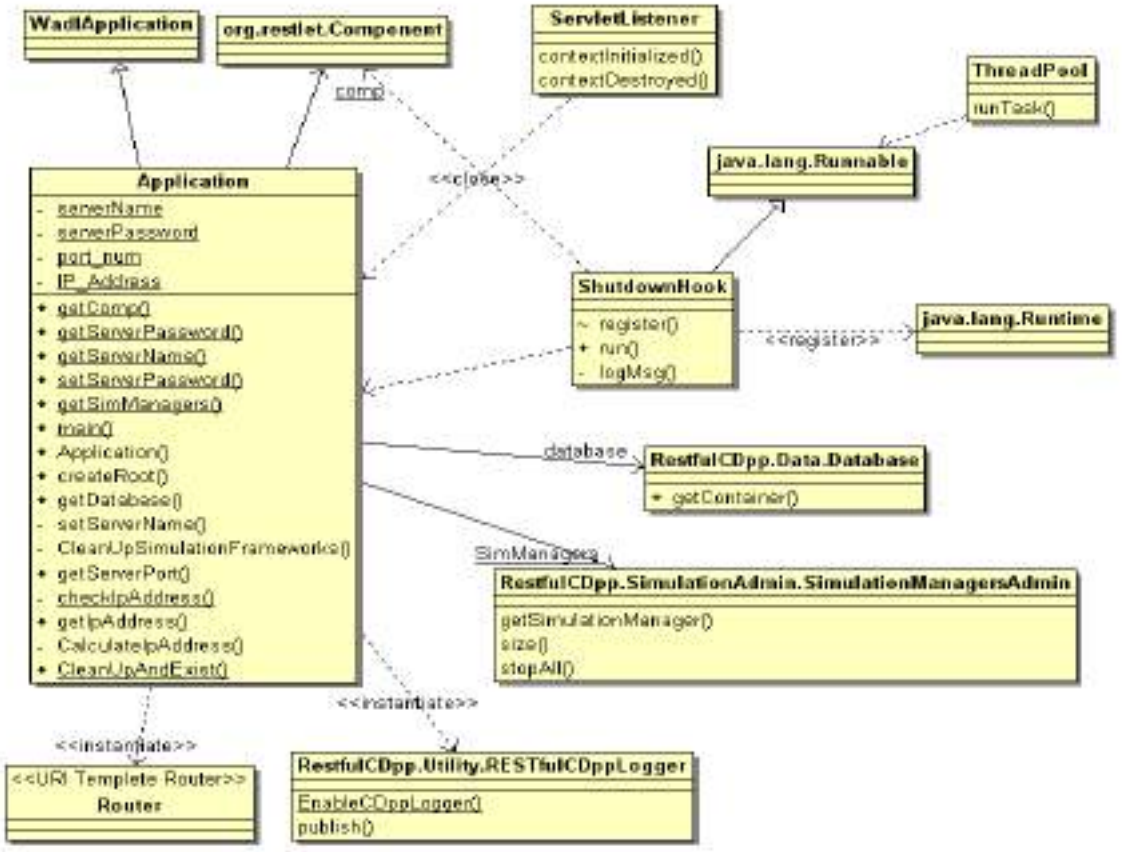

Figure 79: Main-Subsystem Architecture Overview

ThreadPool class manages all system threads. It is based on Java thread pools [61], which consist of *worker* threads. Using worker threads improves performance since it minimizes thread creation overhead. Allocations and de-allocations of thread objects can heavily degrade performance in large-scale applications due to the significant memory management overhead. In this case, a thread is taken out of the pool every time a thread needs to be created to perform a task such as send a simulation message. Consequently, the thread is returned to the pool once the task is completed, allowing future tasks to reuse it.

ServletListener and *ShutdownHook* classes are used to shutdown the RISE middleware gracefully for Servlet and the Standalone deployments respectively. The *ServletListener* is registered with the Servlet container (e.g. Tomcat) where the *ShutdownHook* is registered with Java Virtual Machine (JVM). These classes act as blocked threads waiting for a shutdown event from the Servlet container or JVM. In this case, the RISE middleware always performs the same cleaning routine even if the RISE is terminated via directly killing its operating system process. The cleaning routine (implemented in operation *CleanUpAndExist* in *Application* class) performs the following: **(1)** It aborts all active simulations (if any) along with any remote active simulations, if necessary. This is easily done via invoking operation *stopAll* of the *SimulationManagersAdmin* class. **(2)** It closes communication connectors, if middleware is deployed as standalone. The Servlet container handles this part if middleware is deployed as Servlet. **(3)** It marks the shutdown as normal, hence avoiding any cleaning with the next start up, and **(4)** it closes the database. This completes any pending transactions and unlocks the database, making it reusable with the next start up or by another middleware.

The *Application* class is the main class of the RISE middleware. It creates and makes major structures accessible by all other subsystems. The *Application* class performs a number of activities. The following summarizes the major tasks: **(1)** It creates and initializes the Thread Pool to handle all *worker* threads. **(2)** It starts the middleware logging system. This allows the necessary information of all incoming requests to be logged along with their responses like a typical HTTP server. This feature can be disabled by the administrator. **(3)** It creates and initializes the Router class, which will be

in charge of routing all incoming requests based on their URIs to the proper Java class. **(4)** It creates and initializes the *SimulationManagersAdmin* class (from the SimulationAdmin subsystem) to manage all active simulation managers that will be created in the server. **(5)** It creates and initializes the Database class (from the Data subsystem) to starts the server database. It creates a new database with one administrative account, if it does not exist. **(6)** It registers classes *ServletListener* and *ShutdownHook* to perform proper cleaning, as discussed earlier, and **(7)** it starts communication.

A.1.2 Data Subsystem

Data subsystem (Figure 80) holds and organizes the server database, as shown in Figure 80. The database file stores two types of java objects: the *UserState* class (i.e. contains a user data section) and *ServerConfig* class (i.e. contains the server general configuration). There is always one instance of class *ServerConfig* where one or more instances of the *UserState* class may exist in the database.

The *Database* class holds the database objects and provides methods to external classes to manipulate the database such as deleting, retrieving and storing objects. Database objects are cached in memory after the first access, preventing more accesses to the database file. The database is transactional which means a transaction blocks other transactions until it is completed. All transactions are committed (to the file) to ensure their synchronization with the cache. Note that the entire server is multi-threaded so that writing changes to the database file is performed in a separate thread.

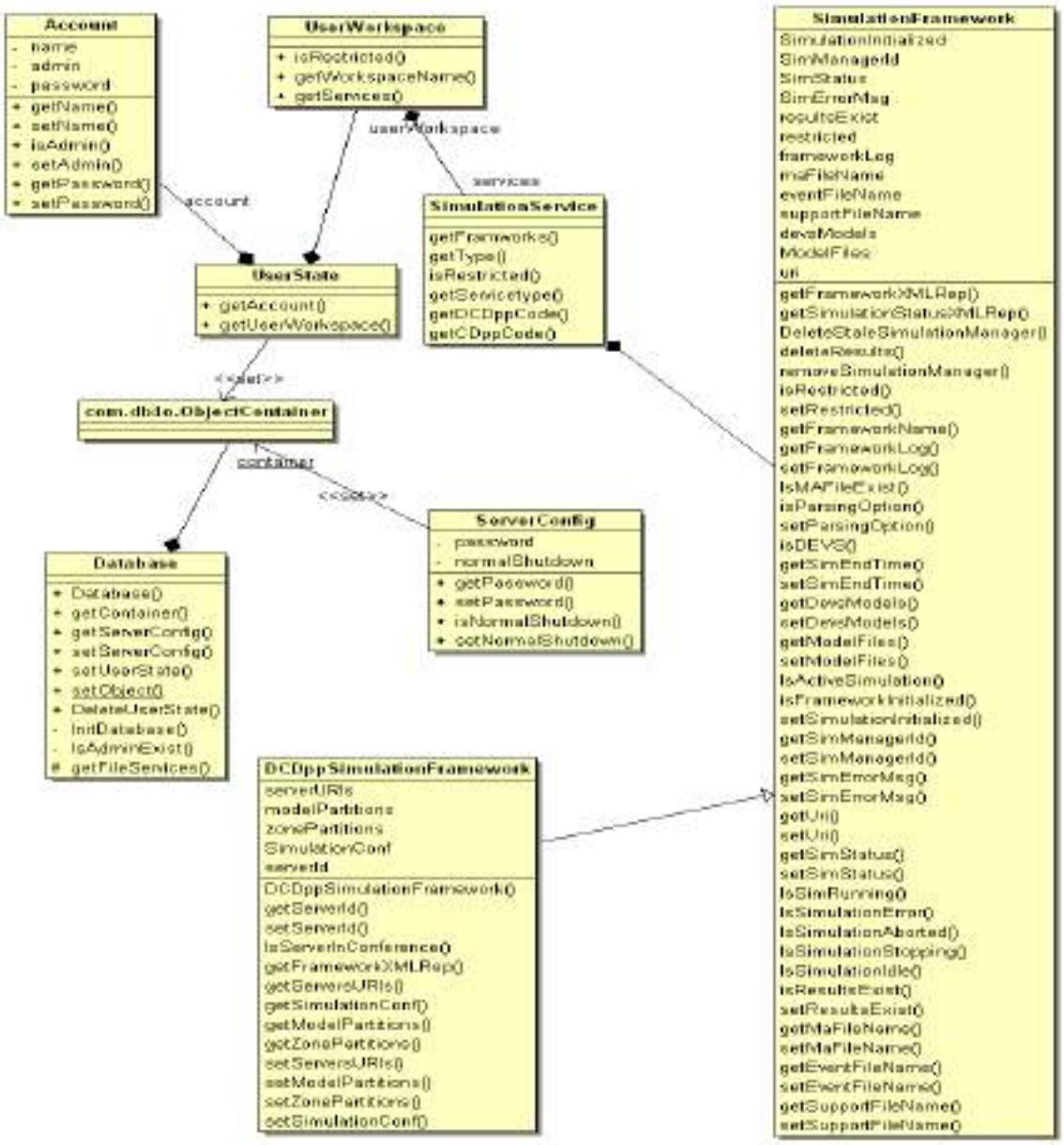

Figure 80: Data-Subsystem Architecture Overview

The *ServerConfig* class holds the server general configuration such as username and password to be used in the DCD++ grid when using remote servers.

The *UserState* class represents a user section in the database. Therefore, there is no need to lock the entire database when receiving several simultaneous requests from different clients. In this case, several threads can update the database safely, since each

incoming runs in a separate thread. However, a user section is protected if more than request is received from that user to change the same section. The *UserState* class (see Figure 80) consists of the *Account* class and the *UserWorkspace* class. The *Account* class holds the user account related information such as username, password and account privileges. The *UserWorkspace* class holds a list of simulation services. Therefore, the server, for example, may support none-simulation services by creating a class for that new service within the *UserWorkspace* class.

The *SimulationService* class holds a simulation service type such as the DCD++ simulation service. Thus, more simulation environments like DCD++ can be added to use the RISE middleware. This class contains a list of simulation frameworks (i.e. experiments) specific for a service type.

SimulationFramework class holds the necessary information related to any simulation framework (as shown in Figure 80). The DCD++ extends the *SimulationFramework* class to *DCDppSimulationFramework* class to hold data related to the geographically distributed simulations such as URIs, model partitions and zone partitions. DCD++ is more complex than any other CD++ extension from the RISE server viewpoint. This is because the RISE manages distributed simulation across the grid among several servers.

A.1.3 Resources Subsystem

This section discusses the implementation of the resources (i.e. URI templates) that handle clients messages enclosed in HTTP envelopes (e.g. XML messages from remote distributed simulation partitions). Each URI request is processed by a java class in the resources subsystem package and an HTTP response is accordingly generated.

Therefore, each URI template in RISE API is handled by a Java class in Figure 81. Table 13 summarizes this URIs mapping to Java classes. The server does not store resources owned by clients according to URIs because it does not scale and is difficult to manage. On the other hand, it only stores relevant data to resources as discussed in the Data subsystem. The RISE Router starts a thread from a pool to handle each incoming request. Afterward, the following steps are taken: **(1)** an instance of a resource class (shown in Figure 81) is created, based on the destined URI template (Table 13), and **(2)** The appropriate operation of the subject resource class is invoked depending on the HTTP channel in the request. Note that the Java classes in the Resources subsystem are thread-safe, since they are private data for each request thread.

Table 13: Resources URI Templates Mapping to Java Classes in Figure 81

Resource URI Template	Java Class
/cdpp/admin/log	ServerLogResource
/cdpp/admin/config	ServerConfigResource
/cdpp/admin/accounts	AccountsResource
/cdpp/admin/accounts/{accountname}	AccountResource
/cdpp/util/ping	PingResource
/cdpp/sim	SimBranchResource
/cdpp/sim/workspaces	WorkspacesResource
/cdpp/sim/workspaces/{userworkspace}	UserWorkspaceResource
/cdpp/sim/workspaces/{userworkspace}/{servicetype}	ServiceTypeResource
/cdpp/sim/workspaces/{userworkspace}/{servicetype}/{framework}	FrameworkResource
/cdpp/sim/workspaces/{userworkspace}/{servicetype}/{framework} /simulation	SimulationResource
/cdpp/sim/workspaces/{userworkspace}/{servicetype}/{framework} /results	ResultsResource
/cdpp/sim/workspaces/{userworkspace}/{servicetype}/{framework} /debug	ModelDebugResource

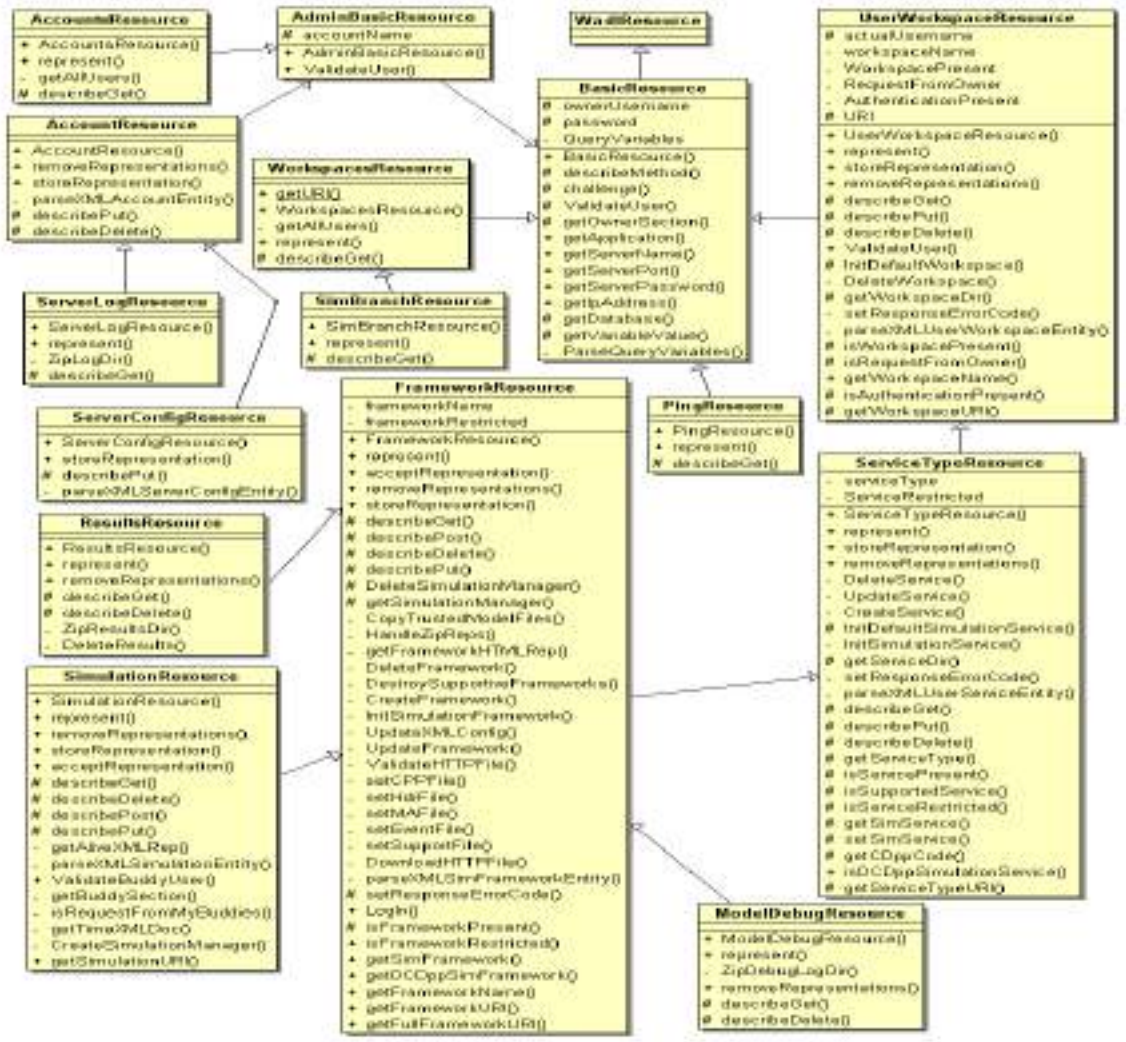

Figure 81: Resources-Subsystem Architecture Overview

The HTTP methods (channels) are implemented by the following operations, if supported by the resource: **(1)** GET channel is handled by the *represent* operation. **(2)** PUT channel is handled by the *storeRepresentation* operation. **(3)** POST channel is handled by the *acceptRepresentation* operation. **(4)** DELETE channel is handled by the *removeRepresentations* operation. Operations *allowGet*, *allowPost*, *allowPut* and *allowDelete* used by resources to disable HTTP GET, POST, PUT and DELETE channels respectively. Operations *describeGet*, *describePost*, *describePut* and

describeDelete used by resources to generate XML WADL [144] description for HTTP GET, POST, PUT and DELETE channels respectively.

The *BasicResource* class contains the general and default functionalities for all resources, since it is the parent of the resource-class hierarchy, as follows: **(1)** it performs HTTP Basic authentication for the incoming request, if needed. For example, it may generate Basic authentication response challenge, allowing clients to reenter their credentials, if they are missing. **(2)** Preparing common structures for processing requests such as: **(2.1)** Processing URIs query variables, and **(2.2)** Retrieving necessary Java objects from the database (or cache). **(3)** It provides access channels to manipulate the database, communication component and utilities.

The *BasicResource* class is extended into four classes *AdminBasicResource*, *WorkspacesResource*, *PingResource* and *UserWorkspaceResource*, which are discussed next. The *PingResource* class validates a user and responds with the proper HTTP status. This class serves as a utility class allowing client programs to verify a user credential.

The *AdminBasicResource* class handles the general functionalities of classes in charge of handling administrative URIs. For example, it validates a user based on allowed administrative privileges. It is extended into the following classes: **(1)** *AccountsResource* class: It allows administrative users to GET an XML document, listing all user accounts. **(2)** *AccountResource* class: It allows administrative accounts to create/delete an account. Further, it allows users to change their passwords. It extends the following classes: **(2.1)** The *ServerLogResource* class enables users to GET the server logs in a zipped directory. **(2.2)** The *ServerConfigResource* class allows administrative

users to change server general configuration such as its password (in order to be used when manipulating remote resources on servers in the DCD++ grid).

The *WorkspacesResource* class returns a list of existing public workspaces as HTML or XML document. A client workspace is always returned (regardless if it is restricted or not restricted), if the request is generated by the owner with the proper authentication. The *SimBranchResource* class is extended from class *WorkspacesResource* to provide general HTML description about CD++ simulation.

The *UserWorkspaceResource* class is used to manipulate a user workspace, allowing owners to create, update or a delete their workspaces. This class allows reading requests by everyone unless the owner has restricted the workspace. For example, main servers always restrict their workspaces on support servers to hide them from other users.

The *ServiceTypeResource* class (extended from the *UserWorkspaceResource* class) is used to manipulate a user service (e.g. DCD++), allowing clients to manipulate their service resources. This class can be extended to support other service types beside DCD++ using its method *isSupportedService*.

The *FrameworkResource* class (extended from the *ServiceTypeResource* class) handles requests to frameworks of any simulation service type including DCD++. Users create, update, read and delete frameworks via this class. This class downloads modeler zip/text files from HTTP requests and sets frameworks configuration.

The *ResultsResource* and *ModelDebugResource* classes (extended from the *FrameworkResource* class) enable users to download simulation results and debugging files respectively as zipped file.

The *SimulationResource* class (extended from the *FrameworkResource* class) wraps simulation managers and interfaces users with active simulations. This class routes all requests to a simulation manager such as passing simulation messages. HTTP PUT channel (operation *storeRepresentation*) starts simulation and creates a simulation manager. HTTP DELETE channel (operation *removeRepresentations*) aborts simulation. HTTP GET channel (operation *represent*) retrieves information from active simulation. HTTP POST (operation *acceptRepresentation*) allows modelers to manipulate active simulation. Further, it is used to exchange messages among servers in the DCD++ grid. This class only (via channel *ValidateBuddyUser*) allows participant servers in a simulation conference to POST a simulation message.

A.1.4 Utility Subsystem

Utility subsystem classes (Figure 82) provides helper classes for all other subsystems, providing four main functionalities: XML parsing utilities, server logging, file services and HTML builder documents. Resources subsystem classes (shown Figure 81) construct on their own transmitted XML documents to clients. However, they rely on the Utility subsystem classes to parse incoming XML documents from clients. Utility classes store all received data in XML documents in java structures, allowing resources classes to manipulate them conveniently.

The *RESTfulCDppLogger* class is responsible of logging incoming requests along with their HTTP status codes. The RISE logging is based on Java logging. The *HTMLHandler* class is used by any resource class to construct an HTML document to be sent to clients. The RISE server usually responds (to HTTP GET channel) by HTML documents to be friendlier when the request is made from a Web-browser. Of course,

XML documents are also supported (if requested by clients) to be friendlier when request is made from a client program. The *FileSystemServices* class provides general file services such as zipping directories, copying directories, unzipping files etc.

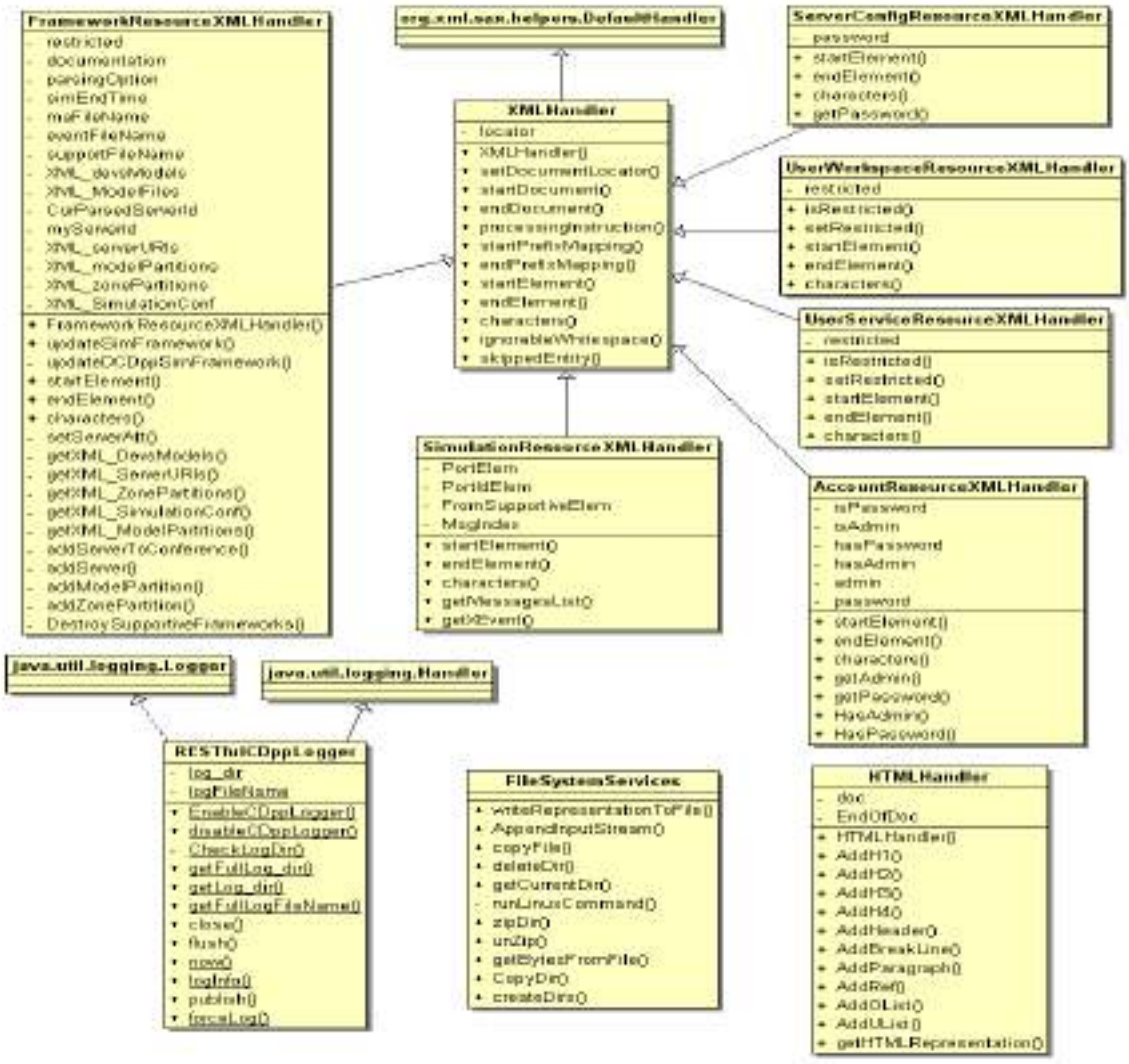

Figure 82: Utility-Subsystem Architecture Overview

Utility XML parsing is based on the Simple API for XML (SAX) parser. As shown in Figure 82, there are six XML parsing classes extended from class *XMLHandler* to process XML documents: **(1)** The *SimulationResourceXMLHandler* class: **(1.1)** It parses modeler's messages to manipulate active simulation such as inserting external

events into active simulation. **(1.2)** it parses XML simulation messages. Multiple messages usually received in a single XML document (see algorithms in Chapter 6 and Chapter 7). Consequently, *SimulationResource* class (shown Figure 81) can retrieve all received messages (using operation *getMessagesList*) and pass them to the proper simulation manager. **(2)** The *FrameworkResourceXMLHandler* class parses simulation framework XML configuration (see Figure 89 example) and store them in local java structures. The class *FrameworkResource* (shown Figure 81) can then update a framework configuration using channel *updateSimFramework*. **(3)** The *AccountResourceXMLHandler* class is used by class *AccountResource* to process XML accounts information. **(4)** The *ServerConfigResourceXMLHandler* class is used by class *ServerConfigResource* to update RISE general configuration. **(5)** The *UserServiceResourceXMLHandler* class is used by class *UserServiceResource* to manipulate a simulation service configuration. **(6)** The *UserWorkspaceResourceXMLHandler* class is used by class *UserWorkspaceResource* to manipulate a workspace configuration.

A.1.5 SimulationAdmin Subsystem

The *SimulationAdmin* subsystem classes (shown in Figure 83) manage the active simulation of an experiment (or a simulation partition, if distributed simulation). This is what we have been calling simulation manager throughout this thesis (see Chapter 5 and Chapter 6). First, received simulation messages are captured by URI <.../ *{framework}/simulation*>, which is handled by the *SimulationResource* class (Figure 83). During simulation initialization, the *SimulationResource* class (Figure 83) creates a

Simulation manager (i.e. a class in this discussed *SimulationAdmin* subsystem) to handle all received simulation messages during simulation.

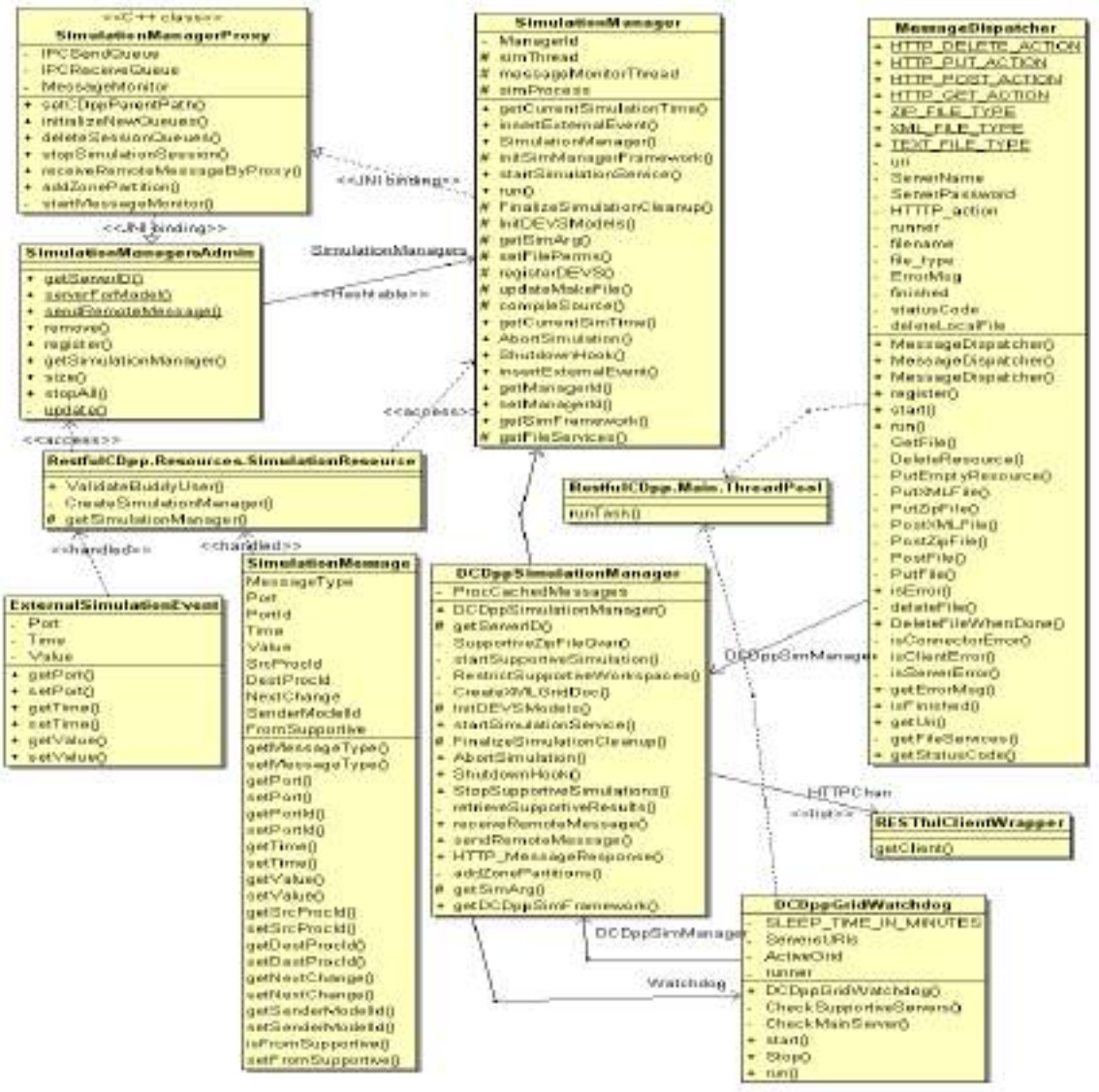

Figure 83: SimulationAdmin-Subsystem Architecture Overview

The SimulationAdmin subsystem (shown in Figure 83) contains the following major classes: *DCDppGridWatchdog*, *ExternalSimulationEvent*, *RESTfulClientWrapper*, *MessageDispatcher*, *SimulationMessage*, *SimulationManagersAdmin*,

SimulationManagerProxy (written in C++), *SimulationManager*, and *DCDppSimulationManager*. These classes are discussed in the following.

The *DCDppGridWatchdog* class implements the watchdog thread to check on the health of remote simulations resources in the distributed simulation environment. The watchdog (of the DCD++ main simulation manager) keeps sending periodic messages (one minute in our case) to check if support simulations are alive. This avoids deadlocks in the distributed simulation and ensures releasing operating system resources as soon as possible. On the other hand, support simulation managers uses watchdog to watch the main simulation to ensure releasing system resources, if simulation fails. Each instance of the *DCDppSimulationManager* class owns an instance of class *DCDppGridWatchdog*.

The *ExternalSimulationEvent* class holds an external event data received from modeler to manipulate active simulation dynamically. In this case, this event is passed (by the simulation manager) to the CD++ simulation engine so it can be executed as the proper simulation time.

The *RESTfulClientWrapper* class wraps HTTP client implementation. HTTP client sends an HTTP message to remote HTTP server and waits for its response. It performs further tasks such as encrypting credentials according to HTTP Basic authentications and managing connections with TCP layer. This class manages TCP connections in a pool, allowing connections reuse for multiple messages, since allocating a TCP connection for each single message is expensive. This wrapper allows HTTP client communication part to be reimplemented or upgraded from a third party easily without affecting the entire system.

The *MessageDispatcher* class is used to dispatch messages where each message transmission lives in a separate thread. In this case, a simulation manager is able to transmit many messages simultaneously. The *MessageDispatcher* class relies on the *RESTfulClientWrapper* class to handle HTTP client communication details. Therefore, messages are transmitted using both thread and TCP connection pools.

The *SimulationMessage* class holds the data of a received simulation messages. In this case, when multiple simultaneous messages are received in a single XML document, they are passed as a list of instances of the *SimulationMessage* class. This is part of preparing messages to the simulation engine.

The *SimulationManagersAdmin* class keeps track of all Simulation manager instances currently handling active simulations. The *SimulationManagersAdmin* class provides a number of tasks that are summarized as follows: **(1)** it is used as router by the CD++ simulation engines running on local machine. The CD++ simulation engine sends remote events via operating system IPC to the *SimulationManagersAdmin* class, which then routes to the proper simulation manager associated with that CD++ engine. This routing is performed (via method *sendRemoteMessage*) based on the manager ID sent along with the event. **(2)** It provides CD++ engines needed information on-demand such as the ID of their associated simulation manager (via method *getSimulationManager*), and the server ID (via method *getServerID*), enabling them to realize their model partitions of the entire model hierarchy. **(3)** It is used by the RISE middleware to create a new simulation manager (via method *register*) or to delete a simulation manager (via method *remove*). These operations are thread-safe (via private method *update*), since they may be performed by multiple threads simultaneously. **(4)** It is used to abort all

simulations (via method *stopAll*). This is only used when the middleware is prematurely shutting down while several active simulation still running. This is important to release operating system resources, particularly CD++ engines that run as separate processes.

The *SimulationManagerProxy* class (written in C++) is used by Simulation manager to manipulate its correspondent CD++ simulation engines. The *SimulationManagerProxy* class can be viewed as the C++ side of the Java simulation manager. The *SimulationManagerProxy* class handles the IPC communication between a CD++ engine and its associated simulation manager. Thus, there is an instance of the *SimulationManagerProxy* class for each simulation manager instance. Upon a message receipts (via IPC) from the CD++ engine, the *SimulationManagerProxy* invokes the proper method of the simulation manager. This is done through the *SimulationManagersAdmin* class as earlier discussed above. On the other hand, a simulation invokes the proper method of the *SimulationManagerProxy* to communicate with its correspondent CD++ engine. For example, the simulation manager invokes method *receiveRemoteMessageByProxy* to pass a message to the *SimulationManagerProxy*, which then forwards it via IPC. Java and C++ crossing implementation is based on the Java Native Interface (JNI) [92].

The *SimulationManager* and *DCDppSimulationManager* classes implement the RISE-based Distributed simulation semantics and algorithms, which are already discussed in previous chapters.

The *SimulationManager* class implements the default part of managing a simulation session. Some of these tasks: **(1)** it creates the *SimulationManagerProxy*, (which is its C++ side) along with the IPC message monitor thread to supervise incoming

IPC messages from the CD++ engine. **(2)** It starts the CD++ engine as an operating system process. This process lives a separate Java thread to avoid blocking the entire simulation manager. This thread is blocked until the CD++ engine process is completed or aborted.

The *DCDppSimulationManager* class is extended from the *SimulationManager* class to handle DCD++ simulation management in geographically distributed environment. The *DCDppSimulationManager* class keeps track with all information related to other remote simulations and their model partitions. Some of these tasks are summarized as follows: **(1)** it passes received messages from remote simulation to its correspondent CD++ engine. This is done with the help of the *SimulationManagerProxy* class. **(2)** It creates the communication channels with all remote simulation entities. This is done with the help of class *RESTfulClientWrapper*. It further transmits all messages to remote simulations on behalf of its correspondent CD++ engine. It follows the algorithms discussed in previous chapters to aggregate (and transmit) simultaneous events in single. This is done with the help of class *MessageDispatcher* (to dispatch the message) and class *RESTfulClientWrapper* (to handle communication details). **(3)** It initializes and starts the distributed simulation (via method *startSimulationService*) at all simulation partitions. **(4)** It starts/stops watchdog thread to watch the participant in the distributed simulation environment. This is done with the help of the *DCDppGridWatchdog* class. **(5)** If it is the main simulation manager, it starts and stops simulation on all support nodes (via operation *startSupportiveSimulation*). It further collects results and debugging data from support entities.

A.2 Middleware Deployment

The RISE middleware can be deployed in two ways, as shown in Figure 84: Standalone HTTP server or as a Servlet engine within an HTTP container.

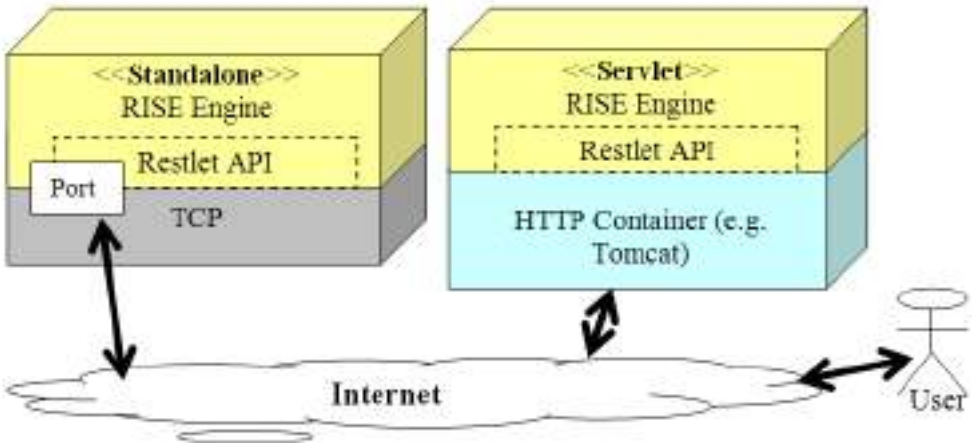

Figure 84: RISE Type of Deployments

Standalone HTTP server: In this case, the RISE engine relies on Simple framework [122] to carryout HTTP connections and communications. The standalone version is an actual HTTP application layer. Therefore, it should have its own dedicated HTTP port to be able to run it simultaneously with other HTTP servers on the same machine. This type of deployment provides convenience particularly when testing distributed simulation on the same physical machine, since multiple middleware instances may be run simultaneously on the same machine.

Servlet engine running inside any HTTP container such as Tomcat [7]: In this the HTTP container forwards all URIs starting with "/cdpp" to the RISE Servlet. This type of deployment takes advantage of Web-services available open source software for performance issues, since performance often vary between a third-party software to another (even from a release to another from the same vendor).

Both types of deployment are interchangeable. For example, one may deploy the server as standalone or redeploy it as a Servlet inside an HTTP container without affecting clients' services. This is because clients always consume services from the same URIs; hence, they still receive the same services regardless of the server type of deployment.

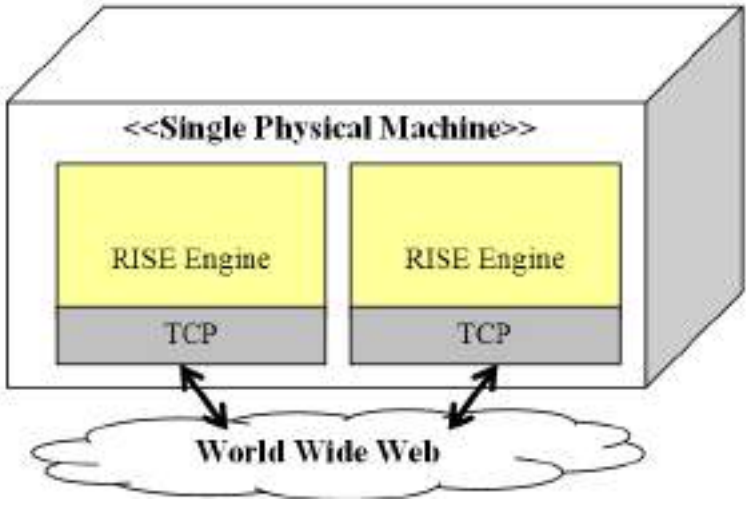

Figure 85: Multiple Instances of RISE Running on a Single Machine

The RISE middleware Servlet (Figure 84) is started by starting the HTTP container (e.g. Tomcat). In this case, the HTTP container routes to the RISE middleware all received HTTP message with the URIs that start with </cdpp>.

The RISE standalone is simply an application layer on top of TCP as shown Figure 84. This means that several instances of the RISE may easily run on the same physical machine, as shown in Figure 85. Simply, copy the RISE files into different folders and starts each one of them on a different TCP port. This fact is a pulse because we now only need one physical machine to be able to test the server behavior on any number of machines, particularly when testing distributed simulation. It is beyond doubt

that for example running one hundred instances (to test distributed simulation) on the same physical machine is the same as utilizing one hundred distributed physical machines (from testing viewpoint). This is because each instance sends/receives messages through the TCP (which then passes it to the IP layer to be routed to its destination); hence at this point it doesn't matter if the message needs to be routed (by the IP) to the other side of the world or to the same machine. Note that the RISE uses the Restlet API [114] (which is realized by the Noelios Restlet Engine (NRE) implementation [104]) to provide set of APIs, mainly allowing the RISE to access HTTP message contents and to hide the communication details.

A.3 Interfacing CD++ With RISE

As discussed in Section 6.2, the DCD++ partition (in a simulation experiment) is realized between the simulation manager and the CD++ engine. In this case, the simulation manager and the CD++ exchanges simulation messages via the IPC queues. This simulation manager is implemented as part of the RISE middleware. This part has previously discussed in the *SimulationAdmin* subsystem classes (Section A.1.5). This subsystem contains the *SimulationManagerProxy* (written in C++), and *DCDppSimulationManager* classes. The *DCDppSimulationManager* class implements the DCD++ simulation manager functionalities, while the *SimulationManagerProxy* class is the C++ side of the simulation manager. In this case, Java and C++ crossing implementation is based on the Java Native Interface (JNI) [92]. In this section, we cross to the CD++ simulation engine. Figure 86 shows the CD++ Processors hierarchy and the *CPPManager* class.

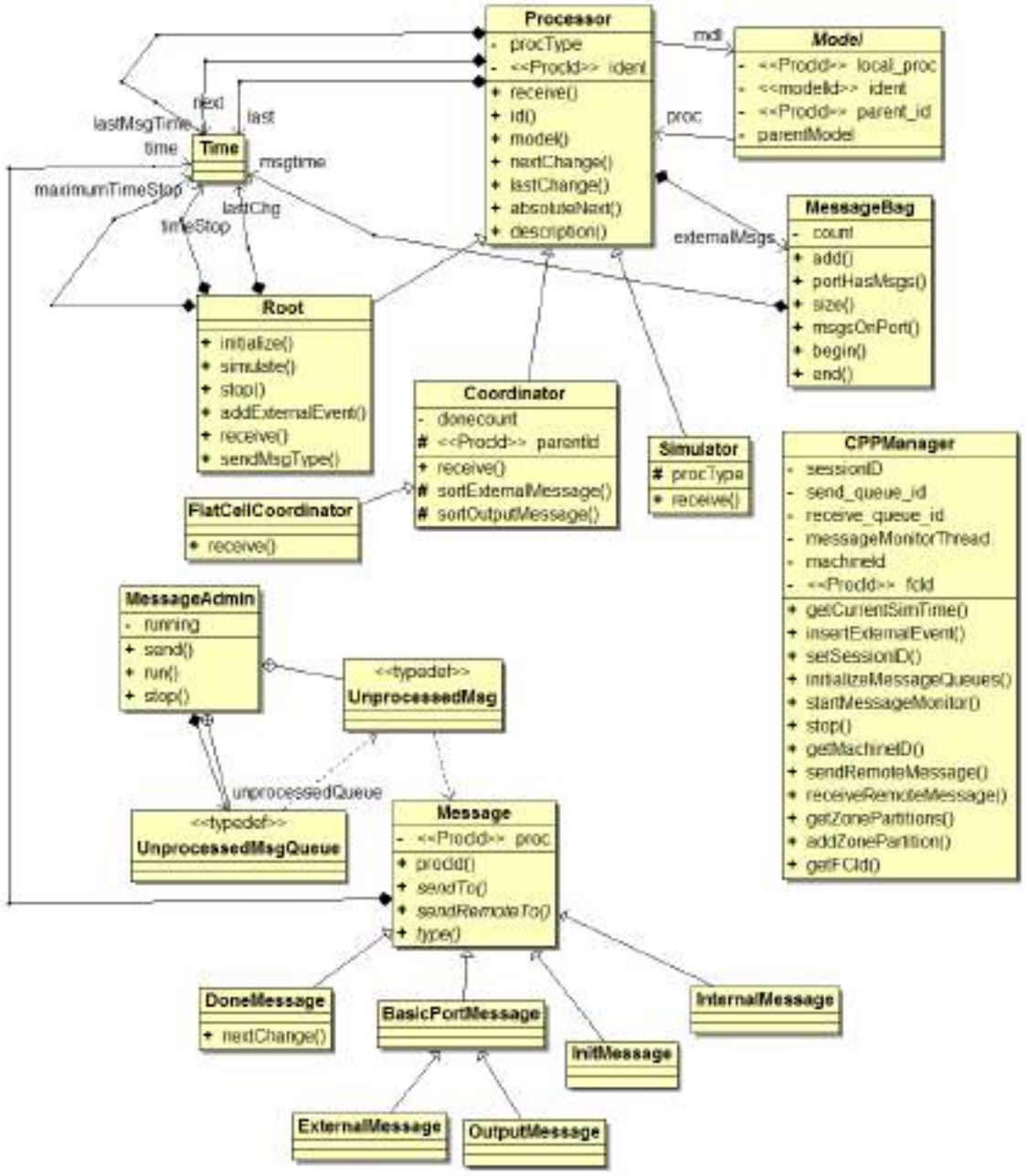

Figure 86: CD++ Processors Hierarchy

The *CPPManager* class (Figure 86) is responsible for interfacing the CD++ simulation engine with the *SimulationManagerProxy* class via IPC queues. The *CPPManager* class creates two message queues: send_queue_id (for sending messages to the simulation

manager) while receive_queue_id (for receiving messages from the simulation manager). The functionality of the *CPPManager* includes:

- Initializing the message queues used for communication with the simulation manager (*initializeMessageQueues*).
- Querying and retrieving the model partitions from the simulation manager in RISE side (*machineForModel*, *addZonePartition*).
- Querying the current execution time and inserting external events while the simulation is running (*getCurrentSimulationTime*, *insertExternalEvent*).
- Sending remote messages while running distributed simulations (*sendRemoteMessage*). This method takes a C++ message and sends it to the simulation manager to be sent to the remote machine.
- Receiving remote messages while running distributed simulations (*receiveRemoteMessage*). This method receives a message from the simulation manager and constructs a C++ message to be processed by the simulator.
- Stopping the simulation when receiving a stop message from the simulation manager (*stop*).

Each processor (in the CD++ hierarchy shown in Figure 86) keeps track of the model that is responsible for executing the model hierarchy described shortly. The processor class is the parent of all the classes in charge of executing the model. Those include the *Simulator*, *Coordinator*, *FlatCellCoordinator*, and *Root* classes. The *Processor* class implements the basic functionality required by all simulation classes: (1) Receiving and processing the different simulation messages, (2) Sending messages and scheduling simulation events via class *MessageAdmin*. The *Simulator* class extends the

Processor class and executes the functions of the atomic DEVS model corresponding to the type of the received message. For example, when a *Simulator* receives a collect message from its parent coordinator, it executes the output function associated with its atomic model. The *Coordinator* class is responsible for forwarding messages among the *Simulators* and for synchronizing the events taking place during the simulation. The *FlatCellCoordinator* class is in charge of executing flat Cell-DEVS models, which differ from Cell-DEVS models in that they are executed by one processor instead of using a processor for each cell in the cell space. The *Root* coordinator is in charge of starting and stopping the simulation, interacting with the environment, and clock advancement. Messages are implemented as separate classes, each representing a message type with all the classes inheriting the *Message* class. Different messages have different attributes; for example, the Done Message class has an extra field (nextChange) to indicate the time of the next state change.

The model loading mechanism in the original CD++ is based on parsing the model definition files and creating the corresponding simulator/coordinator for each of the model components. Those components can be atomic DEVS models, coupled DEVS models, atomic Cell-DEVS models, coupled Cell-DEVS models, and flat-coupled Cell-DEVS models. However, in DCD++, the model loading mechanism includes loading the partitioning information as part of the model loading process; the partitioning information is retrieved from the simulation manager components through the *CPPManager* class (was shown in Figure 86). Atomic models are assigned to run on a specific machine and a coupled model can span different machines with each of its components running on an individual machine.

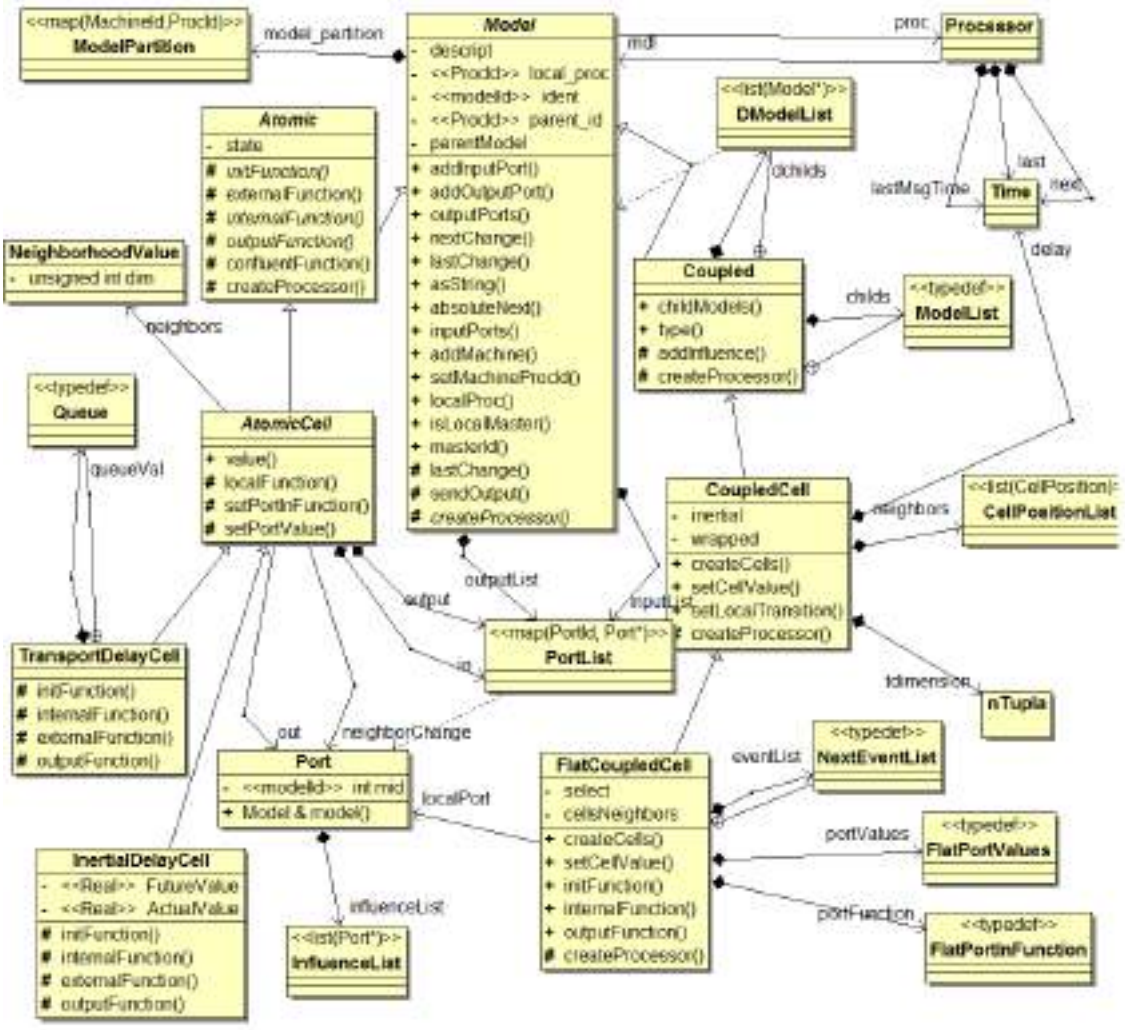

Figure 87: CD++ Models Hierarchy

Figure 87 shows the relationship between the different classes representing the model hierarchy in CD++. The figure also shows that each model keeps track of the processor that is responsible of executing it (the processor hierarchy shown in Figure 86). During the simulation initialization, the simulation loads all required models, and create processors to executed the loaded models (via createProcessor method) along with the organization of child-parent relationship among the created processors, and it then assigns those processors to the appropriate machines according to the models that they responsible to execute (via invoking addMachine method of the appropriate model).

APPENDIX-B: RISE APPLICATION PROGRAMMING INTERFACE (API)

This appendix summarizes the RISE Application Programming Interface (API) with more focus on the simulation blueprint experiment related API, since they form the URI templates for all simulation experiments. Note that the full RISE API (i.e. URI templates) is detailed in the middleware user and developer manuals. This appendix provides the following information:

- It provides an overview description of the entire RISE API.
- It discusses the experiment API (i.e. URI templates) specifications. This discussion is presented in terms of supported functions via the supported HTTP channels. It further provides the possible generated faults.
- It discusses the API XML description based on WADL standards [144]. This allows client machines to retrieve standardized XML description of the RISE API at all time. It is worth to note that RISE always constructs this WADL document upon the client request receipt to ensure the latest version of the API.

B.1 RISE API Overview

RISE is a URI-oriented (i.e. resource-oriented) middleware where all deployed services are wrapped in URIs and manipulated via uniform channels. In this case, the API is expressed as URI template [62]. URI Templates are URIs with variables (placed between braces '{}'). Variables are substituted with appropriate values to get the actual URI instances at runtime, which simplifies both clients and servers. Clients can easily

know what part of the URI is under their control. Further, servers can easily verify all the possible paths that clients can use to manipulate exposed resources.

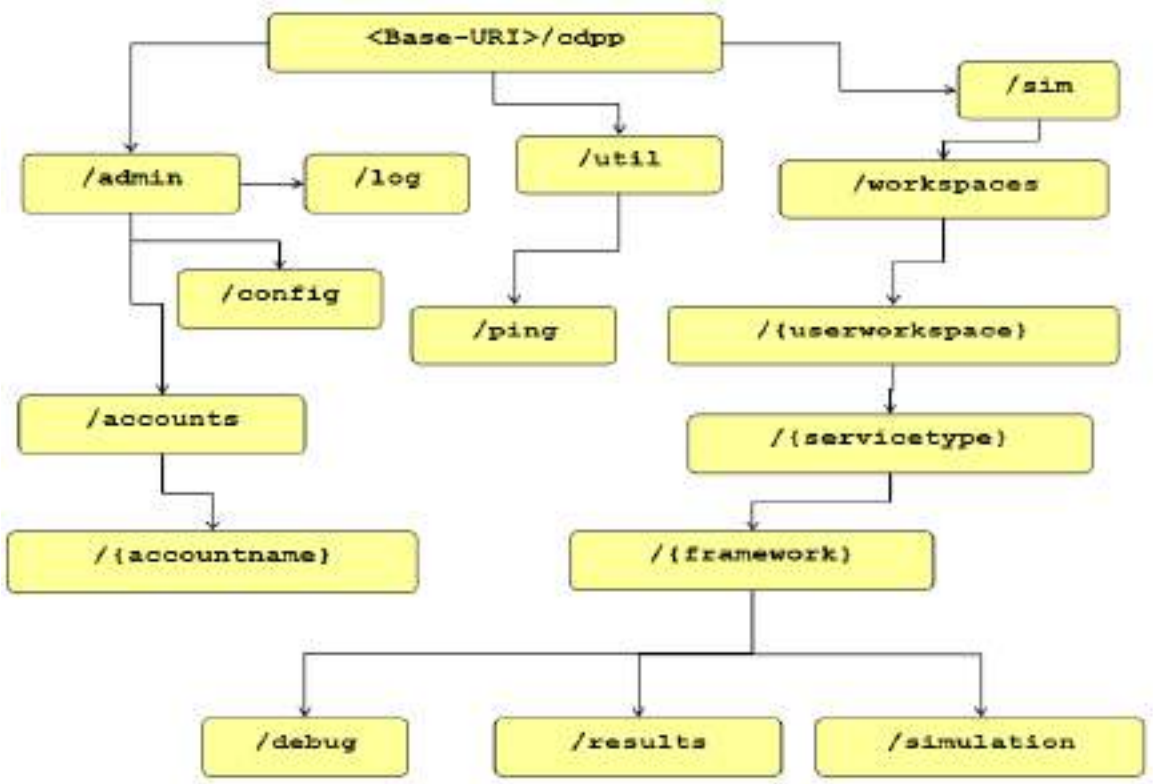

Figure 88: RISE Resources URI Template Overview

Figure 88 shows RISE API overview. Each resource (Figure 88) includes a specification that defines the supported access channels (and their responses); possible generated errors (e.g. HTTP code 401 for not-found resource), incoming/outgoing representations (messages) and media type (e.g. XML). The root URI is split into three subordinate resources (Figure 88): **(1)** *"/admin"* resource is used for administrative services such as creating user accounts, general middleware configuration and retrieving middleware logs. For example, sending an XML message via PUT channel to an absent URI *".../admin/accounts/Bob"* creates an account with name *Bob* according the privileges set in that XML message. **(2)** *"/util"* resource is used for utilities that might be

helpful for modeler client applications. **(3)** *"/sim"* resource is the parent of all simulation resources regardless of their type, as discussed next.

Resource *"/sim/workspaces"* contains all modelers' simulation resources, organized in different sections. Resource *"/sim/workspaces /{userworkspace}"* denotes to a specific modeler workspace that will contain all of his/her experiments resources of any type. Thus, resource *"/sim/workspaces"* may contain a number of different workspaces such as *"/sim/workspaces/Bob" and "/sim/workspaces/Tariq"*, etc. For example, sending a read request via GET channel to resource *"/sim/workspaces"* will return an HTML or XML documents with all user workspaces URIs, as in the case of a regular Web site. Each of user workspace may contain a number of different simulation environments (e.g. DCD++) supported by RISE. A simulation environment type is wrapped in resource *"/sim/workspaces/{userworkspace}/{servicetype}"*. For instance, *"{servicetype}"* needs to be set to *"DCDpp"* to use the DCD++ simulation environment. In this case, more simulation service types may easily be added without affecting the entire API or any existing modelers' resources. Typically, each simulation environment (e.g. DCD++) can contain any number of experiments of any name wrapped in resource *"/sim/workspaces/{userworkspace}/{servicetype}/{framework}"* where *"{framework}"* represents the experiment name (e.g. *fireModel*, *battleModel*, etc.). For example, URI *"/cdpp/sim/workspaces/Bob/DCDpp/FireModel"* is an experiment (framework) with *FireModel* name, DCD++ simulation environment service which belongs to workspace *Bob*. The *"{framework}"* resource is the parent URI of all of the simulation experiments. They are discussed next in Section B.2.

B.2 Experiment URIs Specifications

Resource *"{framework}"* is the parents of all resources belong to the same experiment. Resource *"{framework}"* along with its subordinate resources form the experiment blueprint because the can wrap any experiment, at any state, in any simulation environment, in any modeler workspace, as discussed in Chapter 5. We discuss them in this section in terms of their use by client software.

Resource *"{framework}"* (discussed in Section B.2.1) is used to manipulate an experiment setup such as distributing model partitions and submodels interconnections, etc. In Resource *"{framework}"* URI, POST channel is used to submit/update simulation model necessary information to a framework. PUT is used to create a framework or to update simulation configuration settings. DELETE is used to remove a framework. GET channel is used to retrieve fully/partially a framework state either as an XML or HTML documents.

Resource *"{framework}/simulation"* (discussed in Section B.2.2) is used to wrap active simulation. This URI is used to interoperate with a simulation in progress of an experiment. This resource must only be active during simulation, and it is automatically removed by a domain upon completion. DELETE is used to abort simulation. PUT is used to start simulation. POST is used to send simulation synchronization messages and to manipulate simulation dynamically. GET is used read dynamic results during simulation or active simulation status. This resource only exists during active simulation, and it is automatically removed upon completion.

Resource *"{framework}/results"* (discussed in Section B.2.3) is used to store simulation results once simulation is completed. Resource *"{framework}/debug"*

(discussed in Section B.2.4) is used to store any debugging related information regard the subject model under simulation.

B.2.1 Experiment Resource: {framework}

URI Template: **<…>/{servicetype}/{framework}**

Example: **<…>/DCDpp/FireModel**

This resource holds the experiment settings such as models scripts, distributed simulation partitions configuration, etc. Thus, all model files submitted by clients are made to this resource. Note that the simulation-framework is considered restricted (i.e. read-only by owner) if itself has been restricted or belongs to a restricted URI parent.

Table 14: Specifications Summary for Resource {framework}

Operation	HTTP Channel	HTTP Success Response Code	Request Representation Format	Response Representation Format
Delete Experiment	DELETE	200 (OK)	None	None
Submit File	POST	200 (OK)	• Text file (text/plain) • Zip file (directory) (application/zip)	None
Create Experiment	PUT	201(Created)	[*Optional*] XML(text/xml)	None
Update Experiment	PUT	200 (OK)	XML (text/xml)	None
Get Experiment state	GET	200 (OK)	None	• XML (text/xml) • HTML (text/html)

Table 14 summarizes the experiment framework resource supported operations. Table 14 specifies five columns (left to right): The first column indicates the name of the operations. Those operations classify the purpose of messages requests via a specific HTTP channel. The second column indicates the operation HTTP channel. The third column specifies the HTTP response code, if operation succeeded. The forth column

specifies the allowed data formats (i.e. messages) from clients to RISE via a channel. This is only applicable for the PUT and POST channels. The fifth column specifies the data format returned from RISE to clients. This is only applicable for the GET channel.

The first row (Table 14) defines operation *Delete Experiment*. It uses DELETE channel to delete a simulation experiment instance along with all related information. Note that deletion of distributed simulation experiments also leads to the deletion of all experiments partitions on all machines.

The second row (Table 14) defines operation *Submit File*. It uses the POST channel to submit files to the experiment framework. In this case, there are two types of supported files: text files and zip files. For example, a modeler can submit models scripts as individual text files or as a single zipped file.

The third row (Table 14) defines operation *Create Experiment* while the fourth row defines operation *Update Experiment*. Both operations use the PUT channel to write the XML configuration document to the experiment framework. In this case, if the request is received to absent URI, it is treated as the Create operation; otherwise, it is treated as the Update operation. Figure 89 shows an example of such XML configuration document for the DCD++ experiment types. In this example, The XML configuration document is divided into four main sections (Figure 89): General (line #2-3), files (lines #4-13), CD++ options (lines #14-17) and the DCD++ model partitioning (lines #18-28) sections. The first three sections are optional where the fourth section is mandatory for DCD++ experiments frameworks. Lines #2 indicates if the experiment is restricted (i.e. read-only by owner) or not restricted. Line #3 shows the framework documentation. This usually states the experiment purpose. Lines #4-13 describes the model files section. Files

section is mandatory if files are submitted as zipped folder instead of submitting them individually (see operation *Submit File*). For example, Lines 5-12 describe the CD++ *Barbershop* model files. For instance, Line #10 lists the C++ *Checkhair.h* header file, which contains the C++ *Checkhair* class. Lines #14-17 show the CD++ options. Line #15 indicates the simulation stoppage time. Lines #18-28 show the DCD++ model partitions section. In this example, the *Barbershop* model atomic models are partitioned between two RISE servers, as previously discussed in Chapter 6.

```
1 <ConfigFramework>
2   <Restricted>false</Restricted>
3   <Doc> This DEVS model simulates Barber shop. </Doc>
4   <Files>
5      <File ftype="ev">barber.ev</File>
6      <File ftype="ma">barber.ma</File>
7      <File ftype="src">Reception.cpp</File>
8      <File ftype="hdr" class="Reception">Reception.h</File>
9      <File ftype="src">Checkhair.cpp</File>
10     <File ftype="hdr" class="Checkhair">Checkhair.h</File>
11     <File ftype="src">Cuthair.cpp</File>
12     <File ftype="hdr" class="Cuthair">Cuthair.h</File>
13  </Files>
14  <Options>
15     <TimeOp>null</TimeOp>
16      …
17  </Options>
18  <DCDpp>
19     <Servers>
20        <Server IP="10.0.40.162" PORT="8080">
21           <MODEL>reception</MODEL>
22        </Server>
23        <Server IP="10.0.40.175" PORT="8282">
24           <MODEL>cuthair</MODEL>
25           <MODEL>checkhair</MODEL>
26        </Server>
27     </Servers>
28  </DCDpp>
29 </ConfigFramework>
```

Figure 89: DCD++ Experiment XML Configuration Document Example

The second row (Table 14) defines operation *Get Experiment state*. It uses the GET channel to read the experiment state. The state is the entire experiment configuration and any active simulation status within the experiment. The operation

returns an HTML document by default (making it easier to Web-browser users) or as XML document (making easier for machines processing). Both of these documents described next.

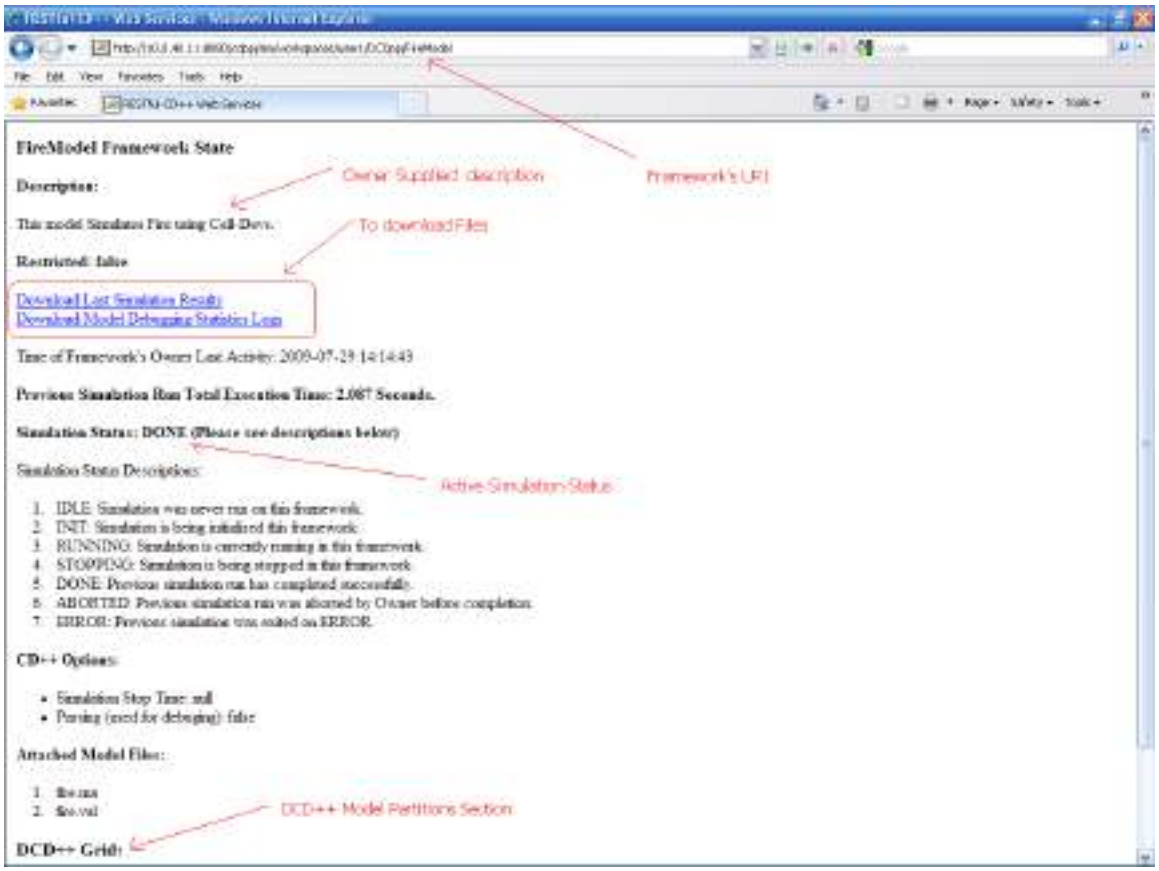

Figure 90: Excerpt of Displayed Framework State Using a Web-browser

Figure 90 shows an example of the returned HTML document displayed by a Web-browser. This HTML document is sent to a Web-browser by simply typing the experiment framework URI in the Web-browser address bar. This is because Web-browsers always send their requests via the HTTP channel. The shown document in Figure 90 displays the experiment information such as the models partitioning, the simulation status, and the links to download results or debugging files. These links are the URIs of subordinates resources, discussed in the next subsections.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <ConfigFramework>
3   <Name>BarberModel</Name>
4   <Restricted>false</Restricted>
5   <Simulation>
6     <Status>DONE</Status>
7   </Simulation>
8   <Doc>This DEVS model simulates Barber shop.</Doc>
9   <Files>
10    <File ftype="ma">barber.ma</File>
11    <File ftype="ev">barber.ev</File>
12    <File ftype="simple">Reception.h</File>
13    <File ftype="simple">Reception.cpp</File>
14    <File ftype="simple">Cuthair.h</File>
15    <File ftype="simple">Checkhair.h</File>
16    <File ftype="simple">Checkhair.cpp</File>
17    <File ftype="simple">Cuthair.cpp</File>
18  </Files>
19  <Options>
20    <ParsingOp>false</ParsingOp>
21  </Options>
22  <DCDpp>
23    <Servers>
24      <Server NAME="10_0_40_66_8282">
25        <Model>cuthair</Model>
26        <Model>reception</Model>
27        <Model>checkhair</Model>
28      </Server>
29    </Servers>
30  </DCDpp>
31 </ConfigFramework>
```

Figure 91: Example of Experiment State in XML Representation

Figure 91 shows an example of the returned XML document of an experiment. Client software reads the XML document (instead of the default HTML document), by simply setting query variable *"fmt"* to *"xml"* such as *<.../BarberModel?fmt=xml>*. The shown XML document in Figure 91 contains the same information in the XML configuration document, discussed in Figure 89. However, it also includes the current simulation status in lines 5-7 (simulation status are previously discussed in Chapter 6). Note that the client software can use query variables to return excerpt of the experiment framework. For example, the simulation status (lines 5-7) can read by clients by setting

query variable *"sim"* to *"status"* such as <*.../BarberModel?fmt=xml*>. In this example, Lines 5-7 will only be returned to the client.

Table 15 summarizes the possible generated errors by the experiment framework. In this case, errors are classified based on the received message channel.

Table 15: Faults Summary for Resource {framework}

HTTP Channel	HTTP Response Error Code	Error Description
DELETE	404 (Not Found)	o Framework does not exist. o Failed Authentication (where Framework is restricted).
	403 (Forbidden)	Simulation is Currently Active (i.e. resource {framework}/simulation exists)
	401 (Unauthorized)	Failed Authentication (where Framework is not restricted).
PUT	400 (Bad Request)	o Request contains bad XML configuration document. o Request is trying to attach a new framework to unsupported simulation service.
	415 (Unsupported Media type)	Received unsupported media type. XML is the only allowed format from clients. DCD++ servers are allowed to PUT zip files among each other.
	404 (Not Found)	o Failed Authentication (where Framework is restricted). o Framework does not exist (if the request did not come from the owner).
	401 (Unauthorized)	Failed Authentication (where Framework is not restricted).
	403 (Forbidden)	Simulation is Currently Active (i.e. resource {framework}/simulation exists)
GET	404 (Not Found)	o Framework does not exist. o Failed Authentication (where Framework is restricted).
	401 (Unauthorized)	Missing Authentication (where Framework is restricted). In this case, a challenge is sent to give the user the chance to provide a username and password.
POST	400 (Bad Request)	o Server received empty file. o Server could not unzip the received file. o Server could not read extracted files. o Errors in URI query variables.

HTTP Channel	HTTP Response Error Code	Error Description
	415 (Unsupported Media type)	Request contains unsupported media type. Text and Zip are only supported files.
	404 (Not Found)	o Framework does not exist. o Failed Authentication (where Framework is restricted).
	401 (Unauthorized)	Failed Authentication (where Framework is not restricted).
	403 (Forbidden)	Simulation is Currently Active (i.e. resource {framework}/simulation exists)
General	501 (Not Implemented)	Request contains Unsupported HTTP channel.
	500 (Internal Server Error)	The server encountered an unexpected condition, which prevented it from fulfilling the request.

B.2.2 Active-Simulation Resource: {framework}/simulation

URI Template: **<…>/{servicetype}/{framework}/simulation**

Example: **<…>/DCDpp/FireModel/simulation**

This resource represents an active simulation with an experiment framework. It is created when simulation is started and is deleted when the simulation is completed (or aborted). Therefore, clients use this resource to manipulate active simulation such as inserting an external event, passing simulation messages among distributed servers or retrieving results during simulation. Table 16 summarizes the active-simulation resource supported operations. Table 16 specifies five columns (left to right): The first column indicates the name of the operations. Those operations classify the purpose of messages requests via a specific HTTP channel. The second column indicates the operation HTTP channel. The third column specifies the HTTP response code, if operation succeeded. The forth column specifies the allowed data formats (i.e. messages) from clients to RISE via a channel. This is only applicable for the PUT and POST channels. The fifth column

specifies the data format returned from RISE to clients. This is only applicable for the GET channel.

Table 16: Specifications Summary for Resource {framework}/simulation

Operation	HTTP Channel	HTTP Success Response Code	Request Representation Format	Response Representation Format
Start Simulation	PUT	202 (Accepted)	None	None
Stop Simulation	DELETE	202 (Accepted)	None	None
Submit Message	POST	202 (Accepted)	XML	None
Get Simulation State	GET	200 (OK)	None	XML or Zipped file

The first row (Table 16) defines operation *Start Simulation*. It uses the PUT channel to start the simulation within an experiment instance. This means that the resource (*{framework}/simulation*) is created. As a result, the simulation engine is created on the local machine. Further, the simulation is started on all other machines, if the experiment is distributed, as discussed in Chapter 6 and Chapter 8.

The second row (Table 16) defines operation *Stop Simulation*. It uses the DELETE channel to delete the resource and abort simulation. Further, if experiment is distributed, simulation on all machines is also aborted.

The second row (Table 16) defines operation *Submit Message*. It uses the POST channel to submit XML messages. These messages could be XML synchronization messages exchanged as part of distributed simulation algorithms (see algorithms in Chapter 6 and 8), or as modelers XML messages to manipulate simulation. For example, Figure 92 shows an example of a simulation event that can be sent by modelers to manipulate active simulation. In this case, this external event is inserted in the simulation

event list to be executed as any other event. The XML external event message consists of the following (Figure 92): Line #2 shows the simulation *Time* (e.g. 09:53:10:000), Line #3 shows the *Port* Name, and Line #5 shows the event *Value*.

```
1   <XEvent>
2     <Time>09:53:10:000</Time>
3     <Port>finished</Port>
4     <Value>9</Value>
5   </XEvent>
```

Figure 92: Example of Simulation External Even Message

The second row (Table 16) defines operation *Get Simulation State*. It uses the GET channel to read information from the active simulation. The returned information is either an XML document to indicate the simulation health or as a zipped file to download simulation results that are being computed.

To get simulation health, query variable *"sim"* needs to be set to *"status"* such as *<.../fireModel/simulation?sim=status>*. In this case, the middleware returns a message similar to the following: *<Simulation>ALIVE</Simulation>*. This message is mainly used to check if a simulation partition failed during distributed simulation (see DCD++ watchdog in Chapter 6). Further, to get simulation results, query variable *"sim"* needs to be set to *"results"* such as *<.../fireModel/simulation?sim=results>*. In this case, all results are returned in a single zipped file. This prevents, in some cases, clients of waiting long time until simulation completed to be able to get the entire results.

Table 17 summarizes the possible generated errors by the active simulation within an experiment framework. In this case, errors are classified based on the received message channel.

Table 17: Faults Summary for Resource {framework}/simulation

HTTP Channel	HTTP Response Error Code	Error Description
DELETE	404 (Not Found)	o Framework does not exist. o Failed Authentication for restricted Framework o Simulation is not active.
	401 (Unauthorized)	Failed Authentication (where framework is not restricted).
PUT	400 (Bad Request)	o Simulation Resource has already been created. o Server could not Start Simulation Manager.
	404 (Not Found)	Failed Authentication (where framework is restricted).
	401 (Unauthorized)	Failed Authentication (where framework is not restricted).
GET	404 (Not Found)	o Framework does not exist. o Failed Authentication (where framework is restricted). o Simulation is not active.
	401 (Unauthorized)	Missing Authentication (where framework is restricted). In this case, a challenge is sent to give the user the chance to provide a username and password.
	400 (Bad Request)	o Errors exist in query variables. o Simulation is not running.
POST	400 (Bad Request)	o Request contains bad XML. o Simulation is not running. o Not Allowed to POST received XML Contents. o Errors in URI query variables.
	415 (Unsupported Media type)	Received unsupported media type.
	404 (Not Found)	o Framework does not exist. o Failed Authentication (where framework is restricted).
	401 (Unauthorized)	Failed Authentication (where framework is not restricted).
General	501 (Not Implemented)	Request contains Unsupported HTTP channel.
	500 (Internal Server Error)	The server encountered an unexpected condition, which prevented it from fulfilling the request.

B.2.3 Simulation-Results Resource: {framework}/results

URI Template: **<…>/{servicetype}/{framework}/results**

Example: **<…>/DCDpp/FireModel/results**

This resource holds the last simulation-run result files. It is automatically created upon normal simulation completion. Note that results still retrieval during simulation via resource *"{framework}/simulation"*, as discussed in section B.2.2. Table 18 shows the two operations supported by this resource as follows:

- *Download Results* operation. It uses the GET channel to retrieve all simulation results files in a single zipped file. Note that this operation allows results to be downloaded via Web browsers similar to regular Web sites.
- *Delete Results* operation. It uses DELETE channel to allow users to force results deletion URI.

Table 19 summarizes the possible generated errors by the active simulation within an experiment framework. In this case, errors are classified based on the received message channel.

Table 18: Specifications Summary for Resource {framework}/results

Operation	HTTP Channel	HTTP Success Response Code	Request Representation Format	Response Representation Format
Download Results	GET	200 (OK)	None	Zipped file
Delete Results	DELETE	200 (OK)	None	None

Table 19: Faults Summary for Resource /{framework}/results

HTTP Channel	HTTP Response Error Code	Error Description
DELETE	404 (Not Found)	o Framework does not exist. o Failed Authentication for restricted Framework o Simulation is currently active. o Results do not exist.
	401 (Unauthorized)	Failed Authentication (where Framework is not restricted).
GET	404 (Not Found)	o Framework does not exist. o Failed Authentication (where Framework is restricted). o Results do not exist.
	401 (Unauthorized)	Missing Authentication (where Framework is restricted). In this case, a challenge is sent to give the user the chance to provide a username and password.
	507 (Insufficient Storage)	Server could not Zip results.
General	501 (Not Implemented)	Request contains Unsupported HTTP channel.
	500 (Internal Server Error)	The server encountered an unexpected condition, which prevented it from fulfilling the request.

B.2.4 Simulation-Debug Resource: {framework}/debug

URI Template: **<…>/{servicetype}/{framework}/debug**

Example: **<…>/DCDpp/FireModel/debug**

This resource holds all debug log files that can be helpful for modelers to debug their models when there are problems, particularly during simulation. For example, a modeler may decide to place debugging print lines within his CD++ model code to trace his model behavior. In this case, those debugging print lines are dumped into a file in this resource, allowing modeler to retrieve this information. Further, for example, compilation

results are written to a file in this resource, allowing a modeler to figure out any compiling problems if, for instance, a DEVS model C++ source code fails compilation. Table 20 shows the two operations supported by this resource as follows:

- *Download Debug Files* operation: It uses the GET channel to retrieve all simulation debug files in a single zipped file. Note that this operation allows debug files to be downloaded via Web browsers similar to regular Web sites.
- *Delete Debug files* operation: It uses DELETE channel to allow users to delete existing debug files, allowing fresh start, particularly when framework has been used for a long time.

Table 21 summarizes the possible generated errors by the active simulation within an experiment framework. In this case, errors are classified based on the received message channel.

Table 20: Specifications Summary for Resource {framework}/debug

Operation	HTTP Channel	HTTP Success Response Code	Request Representation Format	Response Representation Format
Download Debug Files	GET	200 (OK)	None	Zipped file
Delete Debug Files	DELETE	200 (OK)	None	None

Table 21: Faults Summary for Resource /{framework}/debug

Channel in Request	HTTP Response Status	Error Description
DELETE	404 (Not Found)	o Framework does not exist. o Failed Authentication for restricted Framework o Simulation is still in progress.
	401 (Unauthorized)	Failed Authentication (where Framework is not restricted).

Channel in Request	HTTP Response Status	Error Description
GET	404 (Not Found)	o Framework does not exist. o Failed Authentication (where Framework is restricted).
	401 (Unauthorized)	Missing Authentication (where Framework is restricted). In this case, a challenge is sent to give the user the chance to provide a username and password.
	507 (Insufficient Storage)	Server could not Zip results.
General	501 (Not Implemented)	Request contains Unsupported HTTP channel.
	500 (Internal Server Error)	The server encountered an unexpected condition, which prevented it from fulfilling the request.

B.3 API XML Description

Describing RISE URI template in a standardized XML description is important. This allows this information to be consumed and processed automatically by machines. In our case, the RISE middleware provides XML description for the entire API based on the Web Application Description Language (WADL) [144]. This WADL document is dynamically constructed by the server and sent back to the client in response to a request via the HTTP OPTIONS channel to the root URI (*"/cdpp"* in our case). The returned WADL document is always up-to-date since it is constructed dynamically by the software. Sun Microsystems NetBeans IDE [103] and WADL2Java [144] are examples of tools that support WADL standards.

Figure 93 shows an example of the overall WADL document generated by the RISE. Lines 1-3 show the document header. Line #4 indicates the RISE base URI address (*http://localhost:8282/cdpp/*), which means that each resource's URI starts with this base

address followed by its path as defined in the WADL document. For example, Line #5 starts the description of the first resource with path “accounts”. Therefore, the URI of this resource becomes (*http://localhost:8282/cdpp/accounts*). Line #9 starts the description of a template resource where the value of {accountname} must be supplied at runtime (e.g. *http://localhost:8282/cdpp/accounts/bob*). Lines 5-15 show the description for each resource supported in the API.

```
1 <?xml version="1.0" standalone="yes"?>
2 <?xml-stylesheet type="text/xsl" href="wadl_documentation.xsl"?>
3 <application xmlns="http://research.sun.com/wadl/2006/10">
4   <resources base="http://localhost:8282/cdpp/">
5      <resource path="admin/accounts">
6           …
7           …
8      </resource>
9      <resource path="admin/accounts/{accountname}">
10          …
11          …
12      </resource>
13          …
14          …
15  </resources>
16 </application>
```

Figure 93: RISE WADL Document Structure Example

WADL describes the specification of each resource such as the supported uniform interface channels (HTTP channels in our case), the request/response requirements and list all possible generated faults. It also allows documentation to be added to the document. For example, Figure 94 shows the full description of the resource that was originally shown within Lines 5-8 in Figure 93. The resource shown in Figure 94 supports only HTTP GET channel. Thus, this read-only resource, hence all other channels are disabled. In this example, Lines #9 shows that the server respond is returned as XML document while lines 10-18 list all possible errors.

```
<resource path="admin/accounts">
   <method id="GetAccountsList" name="GET">
      <request>
       <param style="header" name="Authorization" required="true">
           <doc>Authenticated Call with HTTP Basic Method</doc>
       </param>
      </request>
      <response>
         <representation mediaType="text/xml"/>
         <fault mediaType="text/html" status="401">
            <doc>UNAUTHORIZED: Authentication not provided</doc>
         </fault>
         <fault mediaType="text/html" status="401">
            <doc>UNAUTHORIZED: Wrong Authentication Provided</doc>
         </fault>
         <fault mediaType="text/html" status="406">
            <doc>NOT_ACCEPTABLE: User needs Admin Privileges</doc>
         </fault>
      </response>
   </method>
</resource>
```

Figure 94: RISE WADL Resource Description Example