J. Parallel Distrib. Comput. 73 (2013) 580–594

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

RISE: A general simulation interoperability middleware container

Khaldoon Al-Zoubi*, Gabriel Wainer

Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

ARTICLE INFO

Article history:
Received 16 July 2012
Received in revised form
22 December 2012
Accepted 26 January 2013
Available online 4 February 2013

Keywords:
Interoperability
Simulation
Web services
REST
Middleware
Experimental framework
Semantic Web

ABSTRACT

In recent years, new services on the Internet have enabled global cooperation; in particular, the Web has enabled new distributed simulation technology. Much research has been devoted to develop middleware interoperability methods on the Web. However, most existing methods have constraints in the structural rules that are placed on the design of middleware interoperability methods. For example, such constraints make it difficult to enhance interoperability via decoupling systems implementations and design, which is essential in open computing networks, as in the case of the Web. In order to achieve such objectives we present the RISE (RESTful Interoperability Simulation Environment) middleware. This all-purpose WS-based distributed simulation middleware decouples design and implementation while allowing composition scalability and dynamicity. Furthermore, it supports experiment-oriented frameworks and has the ability to put Web 2.0 services in the simulation loop. RISE is the first existing middleware to achieve such objectives, and the first to employ RESTful Web-services. We present the foundations for meeting the above objectives, and the distinct characteristics of RISE from existing Web-based approaches.

1. Introduction

Modeling and Simulation (M&S) has evolved to become a discipline that has its own knowledge, formalisms, and methodologies. 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 second concept is *simulation*, which refers to the execution of those models with particular sets of data using a computing device [46]. 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 simulations deal with these issues by executing simulations over multiple processors [17]. Distributed Simulation (DS) is distinguished from parallel simulation by their physical architecture, communication network and latency. A focal point of distributed simulation software has been on how to achieve model reuse via interoperation of different simulation components (without relocating people/equipment to other locations). Other benefits include information hiding such as the protection of intellectual property.

With the expansion of the Internet, the desire toward global cooperation via the Web in the distributed simulation technology has also been on the rise as indicated by number of surveys such as [4,37]. Consequently, much research has been devoted to develop middleware interoperability methods on the Web, particularly using the Simple Object Access Protocol (SOAP) based Web-Services (WS) [30] (e.g. [15,35,42,45]), the High Level Architecture (HLA) standard [23], or a combination of both (e.g. [6,22,48,49]).

The HLA is a distributed simulation architecture mainly used in the defense community since 1996 [23]. HLA-based simulations interact with each other via a common software layer (acting like a bus), called the Run-Time Infrastructure (RTI). On the other hand, SOAP-based WS provide a general interoperability framework on the Web, in which systems (i.e., simulation software) interact through the WS layer, using the Remote Procedure Call (RPC) mechanism. The WS layer transports those RPCs in the form of the Extensible Markup Language (XML) SOAP messages. However, such methods have constraints in the structural rules that are placed on the middleware design methods. In particular, the way they exchange, structure, and use the information is usually tied to internal software design issues, making it difficult to decouple the system implementations and their design. This path usually leads to the need for homogenizing different implementations, which is usually a complex problem to resolve, particularly, in open computing networks, as in the case of the Web. In such open communities, we need interoperating systems that are developed independently without being aware of each other internal software design issues [7]. Any number of systems is expected to be able to join/disjoin the overall distributed environment dynamically at runtime.

Current Web-based distributed simulation systems (particularly using SOAP WS), are mostly under the control of single

* Corresponding author.
E-mail addresses: khaldoon_alzoubi@hotmail.com, kazoubi@connect.carleton.ca (K. Al-Zoubi), gwainer@sce.carleton.ca (G. Wainer).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2013.01.014

team or a closed community, using interfaces that are tied to the implementation (e.g. [35,42]). On the other hand, hiding the implementations (where heterogeneity resides) enhances interoperability, allowing the systems to evolve independently from each other.

Based on the above, our main goal is to develop an all-purpose Web-services based distributed simulation middleware with the following objectives:

(1) The middleware will decouple design and implementation, while allowing *composition scalability* (i.e. any number of systems can join the distributed environment) and *dynamicity* (i.e. systems can be created and destroyed at runtime). The main motivation is hiding these internal details can improve interoperability. This problem has never been investigated before. The assumption has been that, if system A is able to perform distributed simulation on the Web using SOAP WS, there will be a straightforward way to interoperate it with another SOAP-based system. However, in reality, as interoperability is tied to implementation, combining systems developed independently is not trivial (in fact, it usually requires major software design refactoring).
(2) The middleware must serve as a container of any of simulation environment. Therefore, it must be independent of any specific implementation, synchronization algorithms, semantics, and formalisms. This means that the simulation software should run one level above the middleware. Further, *design scalability* must be ensured. This means that the middleware design structure must scale up when adding/removing a supported simulation environment type.
(3) The middleware should support Experiment-oriented frameworks. To do so, we provide *Modeler-oriented Templates*, which are experiment resources created and named by modelers. They can be created for any model, of any settings of any simulation environment. The experiments should be provided with a *Web interface* that enables those simulation experiments to be manipulated externally via the Web, hence enhancing interoperability with other Web-based applications. Finally, all experiment resources must be preserved unless deleted by their owner. This Experimental Framework (EF) approach is one of the characteristics that distinguish the proposed methods here from existing Web-based simulations, which usually build entire specific implementations around specific goals while ignoring other possibilities. For example, a system may focus on visualization access via a Web browser while ignoring distributed simulation and vice versa.
(4) We want to have the ability to put Web 2.0 services in the simulation loop. The ability to create and manipulate experiments on the Web allows putting anything attached to the Web within the simulation loop. This means that one can mix simulations with any service addressed with a URI according to the Web interoperability style. This Web-based information sharing (Web 2.0 [27]) can be used to compose Web-enabled services to produce a new application, known as *mashup* [21].

In order to achieve the goals above, we defined a new simulation architecture based on RESTful WS [32]. The Representational State Transfer (REST) term first appeared in Chapter 5 in [12] to describe the WWW architectural elements. In this client/server model, resources hold representations (states) where these representations are transferable between resources. For example, as shown in Fig. 1, the client call transfers the HyperText Markup Language (HTML) representation from the resource (e.g. www.carleton.ca) to the Web browser.

SOAP-based WS imposes many constraints to interoperability [42]: for instance, one cannot decouple interoperated SOAP-based systems implementations as in most cases the data channels (implemented as procedures) and the exchanged data

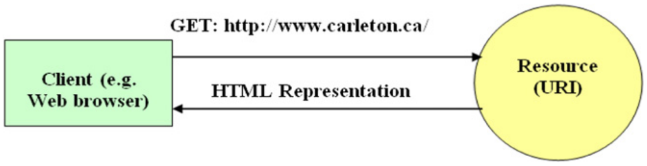


Fig. 1. Representational State Transfer (REST) concept.

(described as programming parameters) are part of implementation itself. Composition scalability is complex as every service (implemented as procedure) at the server side, usually requires a stub on the client side. Instead, as discussed here, RESTful WS interoperability mechanism along with our design and methods, allowed us to meet the listed objectives above.

The RESTful Interoperability Simulation Environment (RISE) middleware (formally known RESTful-CD++) [3,2] is the first existing simulation middleware to achieve the goals stated above, and the first distributed simulation middleware to be based on RESTful WS. RISE serves as a container to hold different simulation environments without being specific to any of them. In RISE, all functionalities are hidden in resources, named with uniform resource identifiers (URIs).[1] Those resources (URIs) are connected to each other via uniform virtual channels in which the simulation synchronization is done using XML messages. Thus, the RESTful interoperability approach allows system designers, as we did in the presented RISE middleware to accomplish the following: (1) decompose the systems in components (i.e. called resources/URIs), (2) hide the implementation within those components, hence separating component interfaces from software implementation. This concept similar is information hiding in modular programming, which is the chief aspect behind the well-known style of object-oriented programming [31]. These fundamentals were adapted by the RESTful WS style, which was adapted by the World Wide Web (WWW), the largest open computing environment. In contrast, existing simulation interoperability approaches do the opposite to these principles by following procedural programming style, hence mixing systems implementation and interface. By going against the Web interoperability principles will always cause serious difficult interoperability issues when interoperating on the Web with other existing systems. These issues became obvious during the current efforts on standardization of Discrete Event System Specifications (DEVS) [46]. This standardization effort is aiming on interoperating various DEVS-based implementations systems via the Web [41].

In the following sections, we present the RISE design, the foundations of meeting the above objectives, and the distinguished characteristics from existing Web-based approaches.

The paper is organized as follows: Section 2 presents background of the concept of simulation interoperability, and we describe how RISE fits within this concept. Section 3 presents the RISE middleware design fundamentals. Section 4 summarizes a number of RISE-based services, applications, and extensions.

2. Background

In recent years, some studies conducted in the form of surveys of experts from different backgrounds (e.g. [4,5,37]) discussed the trends and challenges of distributed simulation. The studies pointed out for the need to enhance cooperation across geographical areas at different levels, as follows: (1) Cooperation at a global level via the Internet, which is driven by economic incentives to form industrial clusters. (2) Hiding/packaging information in components (e.g. to enhance reuse and protect intellectual property

1 Terms Resources and URIs are used here interchangeably.

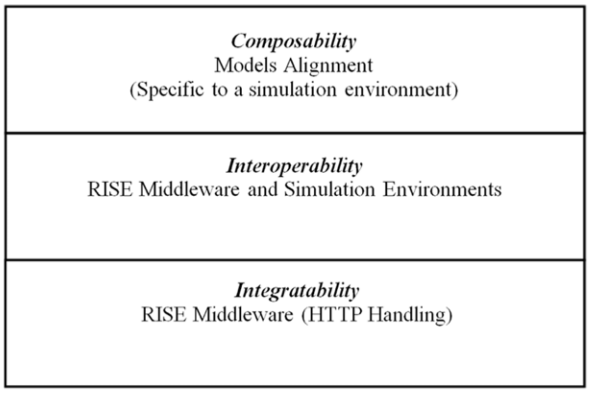

Composability
Models Alignment
(Specific to a simulation environment)
Interoperability
RISE Middleware and Simulation Environments
Integratability
RISE Middleware (HTTP Handling)

Fig. 2. RISE layers within LCIM model layers.

rights). The surveys acknowledged that the most serious challenge to resolve is *interoperability*. Our objectives of enhancing interoperability, particularly by decoupling/hiding implementations and the general Web-based middleware container follow these guidelines.

Interoperability enables two or more different software systems to interface and use each service correctly [38]. The complexity of interoperability arises when systems are heterogeneous, as in the case of distributed simulation. This is usually because systems have been developed independently with different semantics (i.e. the meaning of the exchanged information) and/or syntactic (i.e. the rules of structuring and exchanging the information). Since such capabilities are realized in software design and implementation, interoperability needs to be studied from the software perspective, in particular, at the Application Programming Interface (API) level (since this is how systems access and use other systems services).

To hide implementation details and raise the level of abstraction, a layered architecture can be used (similar to the shown model in Fig. 2). In this model, each layer defines its interoperability methods, and provides services to the layer above it. Following this concept, RISE is organized in the following layers: *middleware*, the *simulation*, and *modeling*. The *middleware* layer (discussed in Section 3) provides a number of services to the simulation layer, such as all means of communication and managing all simulation experiments lifecycle and executions. The *simulation* layer deploys different simulation environment types, each of which supports its own time management. The *modeling* layer operates above the simulation layer. This represents the system under study, which is simulated by a specific simulation environment. This RISE model layers match other existing interoperability conceptual layers, particularly the Level of Conceptual Interoperability Model (LCIM) [38], as shown in Fig. 2.

LCIM divides all related interoperability aspects into three general layers (Fig. 2): the LCIM *integrability* layer deals with network and connectivity issues. In RISE environment, the middleware layer uses existing Web protocols to resolve networking and connectivity issues. In this case, the middleware layer conceals the network and any connectivity issues from the upper layers. The LCIM *interoperability* layer deals with the software implementation of interoperation, including simulation and middleware. In RISE, this layer is presented in two layers: *middleware* and *simulation*. As discussed previously, the *simulation* layer contains the simulation engine implementation including simulation algorithms and formalism. This makes the simulation implementation and algorithms independent of the *middleware* layer. Thus, the *simulation* layer can have different types of simulation implementations and algorithms. The *middleware* layer provides management and common interoperability services to the simulation services in the upper layer. For example, the middleware layer manages the distributed aspects in for distributed simulation services, simulation experiments lifecycle, formatting simulation remote messages, interfacing simulation systems with the Web, etc. The LCIM *composability* layer deals with the alignment issues of the models, that is, how to compose various models correctly to meet the overall purpose. In RISE, this is the *modeling layer* discussed previously. RISE composes the models based on the modeler's instructions (e.g. in form of XML) as part of setting up a simulation experiment. In this case, composition here deals with interoperating different partitions. This partition encloses a portion of the distributed model along with the simulation environment. The assumption here is that a simulation environment in a partition is capable of executing the model portion in its partition. However, in the case of the LCIM and other conceptual frameworks (e.g. [11,29]), composability only deals with the conceptual ideas of constructing a model correctly to achieve a solution. Thus, those works only provide modelers with conceptual ideas of building models to represent systems under study correctly. However, composability in RISE goes beyond building a model correctly. For example, in RISE, those models are enabled with a Web interface, and models can be partitioned and simulated over a number of machines in a distributed matter. Of course, models partitions might be heterogeneous, hence are not necessary simulated by the same simulation environment type.

The way current distributed simulation approaches exchange, structure, and use information, is usually tied to programming and implementation, which exposes the heterogeneity of the systems. In order to deal with these issues, we need to homogenize different implementations, which is usually a complex problem to resolve.

In particular, the Common Object Request Broker Architecture (CORBA) based and SOAP-based distributed simulation protocols are complex to interoperate. SOAP-based WS simulations [30] group the 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. SOAP messages in XML (describing RPCs) are not exchanged at the simulation level, but at the Web service technology layer. This is the case for all SOAP-based WS simulations such as [15,35,42,44].

It is worth to note that we tried to achieve our goals based on SOAP-based WS [42], which was one of the first Service Oriented Architecture (SOA)/DEVS simulators, as part of trying to standardize the middleware [1] for DEVS-based systems. Discrete Event System Specification (DEVS) [46] is a modeling and simulation formalism that has been widely used through many different implementations over the last three decades to study complex discrete event systems. A number of DEVS-based implementations are presented in [41]. However, we realized that it was not possible due to the restrictions imposed by SOAP WS structural rules. 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. Composition scalability is complex if every service (implemented as procedure) at the simulations at the server side require stubs on each simulation client. Thus, this interoperability approach is difficult to achieve in open communities as in the case of the Web. This is because in such communities practice, systems need to be designed, implemented, and evolved independently. 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. This approach usually ends in a close community where software developers can discuss with each other to resolve systems API related design issues. The Web Services Description Language (WSDL) role is to describe the RPCs signatures (i.e. names and input/output parameters). However, once the published WSDL document is compiled to programming stubs (usually by a tool), programmers need to code those stubs and compile them with their software [30]. This

Table 1
Comparing RISE to current interoperability approaches.

Approach	Simulation synchronization description	Simulation information channels	Middleware to middleware interoperability	Services addressing
HLA	Procedure parameters (interaction data fields between the RTI and the federates)	Interactions (callback functions between the RTI and the federates)	RTI is implementation specific. RTIs exchange information as regions of attributes	RTI implementation specific
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)
CORBA approaches	Procedure parameters	RPCs (each set is grouped in an object)	Parameters marshaling and unmarshaling	CORBA reference per object
RISE middleware (based on RESTful WS)	XML messages (message-oriented)	Four uniform software channels (HTTP methods)	XML over HTTP	URI template is per resource (service type). Instances (URIs) are created at runtime

fact usually becomes more obvious when examining the interface requirements in the software developer manuals for SOAP-based systems like the well-known Apache AXIS engine [43]. This argument also applies to CORBA-based simulations (e.g. [25]), since they use the same RPC interoperability style, but with different standards.

The HLA standard [23] performs distributed simulation via interfacing a number of federates via the Run Time Infrastructure (RTI). To be part of the Web, the HLA recently allowed interfacing the RTI with a SOAP WS interface (e.g. [6,22,48,49]). However, this WS interface still does not cover RTI-to-RTI interoperability. This is done in the form of programming functions (called interactions); hence, data is exchanged at the federate level via those functions' parameters. The data exchanged between federates is described in programming parameters, while the data channels are realized as programming procedures. Further, the 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.

In contrast, the WWW is the largest existing distributed structure where countless of systems interoperate with each other according the Web standards. Considering this open-community interoperability style, we decided to use RESTful WS [32]. This is mainly because the RESTful interoperability approach allows system designers to decompose systems in resources/URIs while hiding implementation within those components, separating component interface from their software implementation (as we did in the presented RISE middleware). Specifically, the resources exchange messages and they are connected via virtual constant standardized channels. RESTful Web-services are also gaining increased attention with the advent of Web 2.0 [27] and the concept of mashups [21] (a grouping of various services from different providers presented as a bundle in order to provide single integrated service). For example, IBM enterprise mashup solutions [20] aim on integrating Web 2.0 functions as rapid as possible.

The proposed RISE middleware, which is the first RESTful-based simulation middleware, provides a novel approach to decouple interoperability from simulation systems implementations and internal software design issues (Table 1). This is mainly because APIs are moved outside implementations. Information is exchanged using XML messages and transferred via virtual channels (HTTP methods [13] in our case). Table 1 compares RISE with other existing approaches, classifying the way in which synchronization messages are defined and exchanged, and in the way simulation services are accessed, structured, and addressed. These RISE design issues are discussed in Section 3. SOAP-based WS ports are usually connected with specific designed remote procedures, targeting specific systems implementations. On the other hand, the Web-based HLA method is similar to the typical HLA, but federates are able to communicate with their RTI via the Internet using SOAP-based RPC-style.

Other Web-based simulation efforts have mainly focused on visualization and models reusability [8,14,28,33,50] by providing Web access to model repositories and visualization environments. However, these models are only executable on specific simulation environments. Table 2 compares the presented RISE middleware with current Web-based simulation approaches.

Table 2 compares the proposed RISE framework with various Web-based simulation systems. Although there are numerous web-based simulation environments, the references selected for the table reflect the most recent Web-based simulation works. Table 2 summarizes a number of characteristics:

(1) *General middleware* (*Row* #1 *in* Table 2): the ability to support different simulation environments (e.g., the middleware is independent of the implementation).
(2) *Distributed simulation* (*Row* #2 *in* Table 2): several processors perform a single simulation via synchronization through the Web.
(3) *Composition scalability* (*Row* #3 *in* Table 2): the number of partitions in distributed simulation is independent from the way synchronization information is exchanged.
(4) Standardized *information channels* (*Row* #4 *in* Table 2): RISE virtual information channels use the HTTP standards [13]. Existing Web distributed simulation systems often design their specific RPCs. There are few systems based on standardized SOAP-based RPCs, but those implementations are heavily influenced by such standards. For example, the SOAP-based ID-Sim system [15] uses the Globus Toolkit [16] implementation. The Globus Toolkit [16] realizes the SOAP-based OGSI [39] interoperability specifications standards. Because the SOAP-based interoperability rules are tied to programming, the OGSI standards become tied to programming as well. In this case, the IDSim software design architecture (as described in [15]) has heavily been tied to the Globus implementation of the OGSI standards. In general, it is difficult to come up with a software design interface that would fit every system implementation needs.
(5) *Synchronization messages description* (*Row* #5 *in* Table 2): RISE uses XML to describe all synchronization messages. In fact, RISE is the only Web-based simulation system on this list (which is generic) to use XML descriptions. This *message-oriented* approach provides a better method compared to programming parameters. XML messages and the virtual channels put systems APIs outside their implementations, hence, decoupling distributed systems implementations. For example, SOAP-based systems exchange simulation information as programming parameters via RPCs. Even though the WS engines describe those RPCs as XML SOAP messages, information still exchanged as programming parameters at the simulation engines.

Table 2
Comparing current Web-based simulation approaches.

	Characteristic	RISE middleware	Web API access to simulation services (e.g. [24,45,47,50])	SOAP-based WS and/or HLA with SOAP interface distributed simulation (e.g. [35,33,42,44])	Programming procedure API to simulation services (e.g. [9,26,36])
1	General middleware (ability to support different simulation environments)	Yes	No	No	No
2	Distributed simulation (partition simulation among different machines)	Yes	No	Yes	No
3	Composition scalability	Yes	No	No	No
4	Distributed simulation standardized information channels	Yes	No	No (except [15], it uses OGSI [39])	No
5	Distributed simulation synchronization messages description	XML	No	Programming parameters	No
6	Experiments direct access on the Web (experiments seen as URIs)	Yes	Yes	No (accessed on Web via programming procedures)	No (accessed on Web via programming procedures)
7	Experiments named with modelers choice of URIs	Yes	No	No	No
8	Dynamic experiments	Yes	No	No	No
9	API XML description	Yes (WADL; can be done in WSDL 2.0)	No	Yes (WSDL 1.1) (except [25], it uses CORBA IDL)	No
10	Interoperable with Web 2.0	Yes	Yes	No	No

(6) *Experiments have direct access on the Web* (*Row* #6 *in* Table 2): RISE exposes all experiments as URIs on the Web. This means that experiments are directly accessed and manipulated as any other URI on the Web. This is important because it eases interoperability with other Web-based applications.
(7) *Resource names of their choice* (*Row* #7 *in* Table 2) for any supported simulation environment. This follows a general template pattern as discussed in Section 3.2.2.
(8) *Dynamic experiments* (*Row* #8 *in* Table 2): in RISE, experiments URIs can be created, deleted, and manipulated at runtime, as discussed in Section 3.2.2.
(9) In RISE, the API XML *publication description* (*Row* #9 *in* Table 2) is based on the Sun Microsystems Web Application Description Language (WADL) standards [19] and can be done in WSDL 2.0. This XML standard allows clients to generate RISE API automatically (WSDL 2.0 is the recommended standard since year 2007 [10]).
(10) *Interoperable with Web* 2.0 (*Row* #10 *in* Table 2): it is simple to due to the use of RESTful WS.

3. Rise middleware principles

In this section, we will discuss the three RISE middleware fundamentals: its *uniform interface* (Section 3.1), the *resource-orientation* design (Section 3.2), and the *message-orientation* mechanisms (Section 3.3). We first summarize how our objectives are met using these principles.

In general, RISE spreads services over a number of resources/URIs, and these resources exchange synchronization information in form of XML messages via uniform channels, as shown in Fig. 3.

To decouple systems software related design issues (our first objective in Section 1), is to place the APIs (i.e. how data is exchanged and described) outside systems interoperation. In RISE, this is accomplished via message-orientation and uniform interfaces. Further, composition scalability is achieved because all the interoperating resources (URIs) are always connected with the same channels. These channels are virtual (i.e. they are part of the exchanged messages), and they automatically exist upon a resource creation, providing dynamicity. The virtual channels conform to the universal HTTP methods standards [13], hence they can be used to create/destroy resources at runtime. Thus, RISE resources are designed externally as URIs that can be manipulated according to HTTP standards. The second objective is that the middleware must serve as container to hold different simulation environments. To meet this objective, RISE organizes the resources hierarchically, allowing design scalability. Resources are expressed as URI templates (created at runtime using HTTP standards), which led to a layered interoperability where simulations reside on top of the middleware. This resource-oriented design also allowed us to meet our third objective: defining experiment blueprints directly attached to the Web. Experiments (and not only simulators and models) are seen externally as URIs. That is, the middleware interoperates at the Web layer using the Web standards and style (i.e. based on RESTful WS). This also meets our fourth objective: we have the ability to interoperate and to put other Web 2.0 RESTful services in the simulation loop.

3.1. Uniform interface

RISE uses a uniform interface for each resource, i.e., they 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 the ideal choice to realize those channels. RISE uses those HTTP methods and treats them as software virtual channels as in Fig. 4.

The GET channel is used to read information from resources (such as simulation status and results). The PUT channel creates a resource or updates existing data in a resource (i.e., experiment settings). POST is used to append new information to an existing resource (such as XML synchronization messages 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 WADL [19] description of all the RISE API.

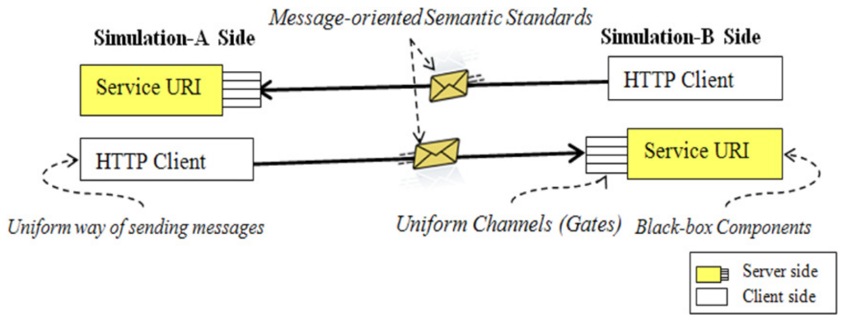


Fig. 3. Overview of RISE main concepts.

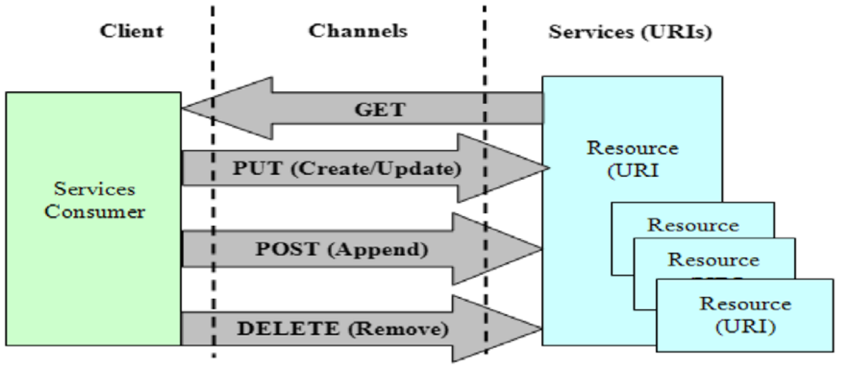


Fig. 4. Uniform channels for RISE resources.

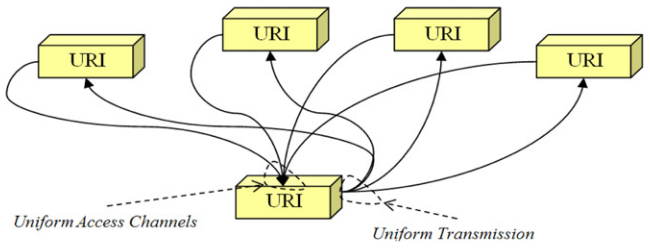


Fig. 5. RISE information channels structure.

Having these standardized channels for each resource, we can achieve composition scalability and improve dynamicity in distributed simulations. Since the channels of each resource exist automatically upon that resource creation, the foundations for dynamic interoperability already exist. Further, because each resource is connected with the same number of virtual channels (regardless of the number of remote resources); composition is scalable, as shown Fig. 5.

Based on the above, RISE uses a uniform way to send messages and access system resources. Four parameters are needed for a message: the *destination* URI, the *channel name*, the *message syntax* (e.g., defined in XML), and the *actual message data*. This means that the message transmission mechanism can be implemented in a single procedure, where each message is transmitted within its own thread [3]. Once the message arrives at its destination, RISE needs to forward it to the appropriate local resource (i.e. the service URI). This is done in three steps (Fig. 6):

(1) The Router (i.e. a thread in RISE) checks if the URI matches one of the templates in the server. If so, it starts a thread (from the threads pool) and initializes it with the HTTP envelope information along with an instance of the Java class associated with the subject URI template.
(2) The proper operation (of the Java object) is invoked based on the message channel. At this point, the message in the HTTP envelope is processed (e.g. converting the received XML message, and sending it to a simulation engine). A resource is implemented as a Java class, and channels as operations in that class. These operations take HTTP requests as inputs, and produce HTTP responses.
(3) The HTTP response is generated, and the message thread is terminated.

As seen on the second step in Fig. 6, a message always enters a resource through uniform channels. 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 resource states, while the others do (i.e. write data). RISE realizes the access mechanism by protecting every resource with a filter, as shown in Fig. 7. The filter performs the following steps: (1) Authentication, which verifies the username and password in the request; 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 error (HTTP code 401), if either fails. Otherwise, the received request is processed.

3.2. General resource-oriented design

In this section, we describe the resource-oriented hierarchical design, the Experimental Framework (EF) blueprint, and the resources database.

3.2.1. Resources hierarchical design

A resource on the Web is conceptually intended to capture a target of a hypertext Ref. [12]. 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 (analog to the concept of a *class*, which can have many *instances*). To do this, RISE applies the concept of URI *templates* [18] to deploy resources types. A URI Template is a URI with variables (placed between

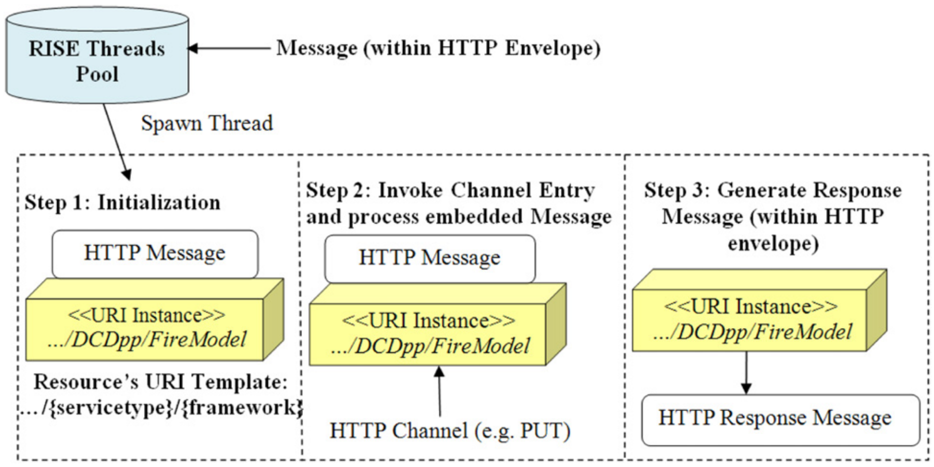

RISE Threads Pool
Message (within HTTP Envelope)
Spawn Thread
Step 1: Initialization
HTTP Message
<<URI Instance>> .../DCDpp/FireModel
Resource's URI Template: .../{servicetype}/{framework}
Step 2: Invoke Channel Entry and process embedded Message
HTTP Message
<<URI Instance>> .../DCDpp/FireModel
HTTP Channel (e.g. PUT)
Step 3: Generate Response Message (within HTTP envelope)
<<URI Instance>> .../DCDpp/FireModel
HTTP Response Message

Fig. 6. Processing received messages in RISE.

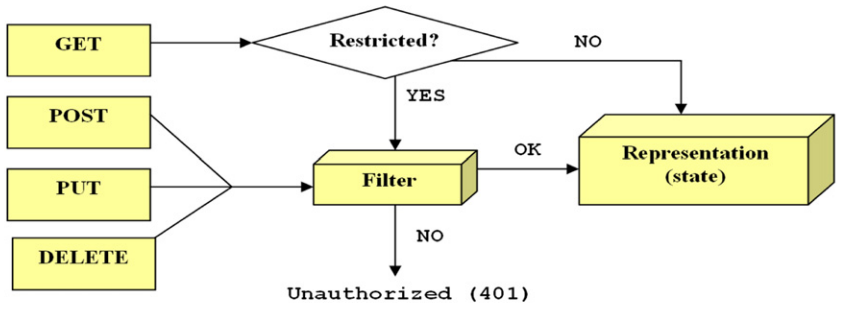


Fig. 7. Resources authorization access process.

braces '{}') which can be substituted with the appropriate values to obtain the actual URI instances. For example, "*username*" in the template "*users*/{*username*}" can be substituted with any string to obtain the actual URI instance such as "*users/Bob*". The use of URI templates allowed us to achieve our goals in terms of middleware organization, since URIs can be created and named at runtime to wrap concrete services.

Fig. 8 shows how resources are organized hierarchically, with multiple instances of each template created simultaneously by different users (the RISE API is detailed in [3]). For example, the template "{*userworkspace*}" allows any number of modeler workspaces to be created, separating the modelers' experiments from each other. The "{*servicetype*}" template allows each modeler to select a simulation service. This allows modelers to create experiments based on different environments. For instance, setting it to "*DCDpp*" will activate the Distributed CD++ environment [3]. It also allows RISE middleware developers to support additional simulation environments types without affecting other existing types. The "{*framework*}" template indicates that the modelers may create any number of Experimental Frameworks with any supported simulation environment type (see "{*servicetype*}" discussion above). As shown in Fig. 8, these URIs are linked to each other, allowing a URI to be discovered via its parent URIs. For example, the "{*servicetype*}" template does not only indicate a simulation environment type, but also serves as a structural resource for its children. For instance typing "…/*Bob/DCDpp/*" in a Web browser will return all of Bob's URIs experiments that use the DCD++.

The concept of providing services as general resources led to the layered interoperability concept discussed in Section 2, which can be seen in Fig. 9. The *middleware* layer provides services to the *simulation* layer (i.e., all means to exchange all information, URI encapsulation, experimental frameworks, authentication, database, and data distributed simulation). The middleware allows

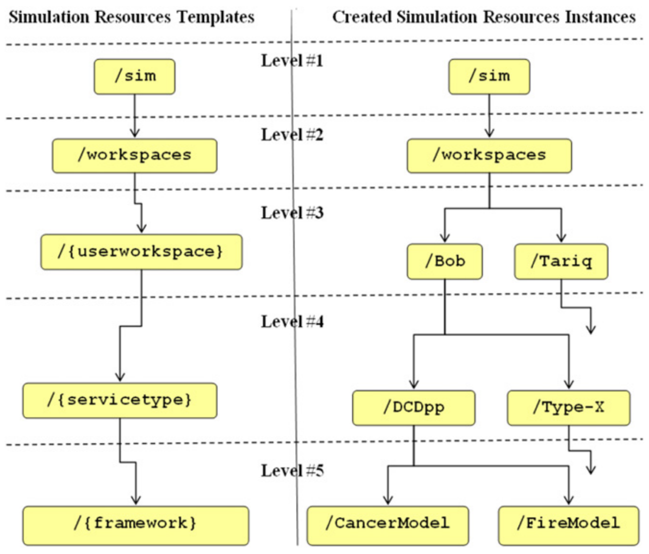


Fig. 8. Excerpt of resources templates with instances examples.

adapting the data distribution scheme to different simulation methods (as described by the RISE implementation in [3]). The *simulation* layer deploys different simulation environment types (e.g. conservative/optimistic, distributed/parallel, etc.), each of which can support their own time management based on their internal algorithms. The actual simulation engines (which are responsible of executing the model) are created only when the simulation is started within a simulation experiment. The modeling layer operates on top of a simulation environment (its model representation is compatible only within its associated simulator). In RISE, the *modeling* layer schemes define the level of

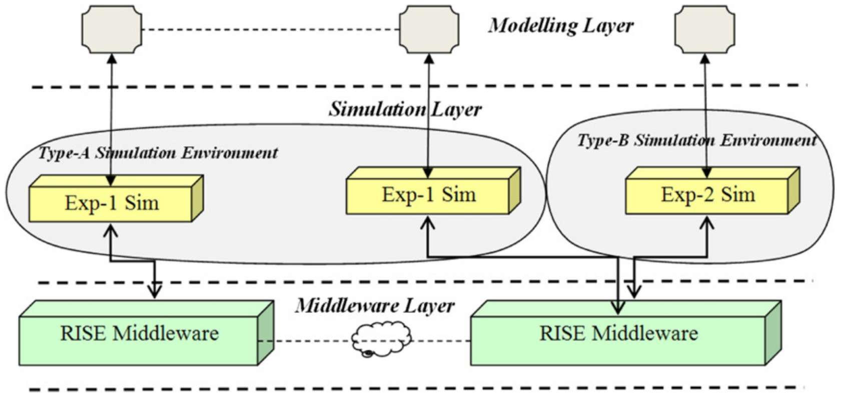


Fig. 9. RISE middleware general simulation container on a machine.

partitioning granularity in the distributed model. For example, the DCD++ simulator [3] supports partitioning as low as DEVS [46] atomic models. Note that DEVS expresses a system as a number of behavioral models (called atomic) and structural components (called coupled). DEVS coupled model may consist of a number of atomic and other coupled models; DEVS atomic models are indivisible blocks. Thus, DCD++ partitions themselves might be fragmented or exist as single blocks. However, the scheme described in [2] places an entire model as a black box in each partition.

Using this architecture, any number of experiments of any type may be conducted at the simulation layer. For example, Fig. 9 shows two experiments with two simulation environments: *Type*-A (e.g. distributed conservative) and *Type*-B (e.g. single engine sequential). In this example, <.../*Type*-A/*Exp*-1>corresponds to the experiment "*Exp*-1" of simulation system "*Type*-A". Likewise, <.../*Type*-B/*Exp*-1>corresponds to the experiment "*Exp*-1" of simulation system "*Type*-B". In this case, *Exp*-1 is distributed over two machines; hence, *Type*-A algorithms must synchronize their activities, and the middleware supports these services. On the other hand, *Exp-2* is running on a single machine, hence, the middleware is only providing a Web access to such experiment. Of course, those experiments are executing separately of each other, and they may belong to the same user or different users.

3.2.2. *Experimentation blueprint*

As discussed in the previous section, the simulation resources (i.e., the second layer in Fig. 9) are typically part of experiments instances. Because the resources are defined as templates, each template is an experiment blueprint. This means that the creation and manipulation of experiments follows a generic pattern discussed in this section.

The Experimental Framework (EF) template is shown in Fig. 10. Here, the "{*servicetype*}" template is used to select a given simulation environment. The "{*framework*}" template is used to create an experiment and its main URI. This URI provides a web interface to the encapsulated settings and data of an experimental framework. The "{*framework*}/*simulation*" template is used to wrap an active simulation for an experiment. This URI provides a web interface to the running simulation for a given experiment. The "{*framework*}/*results*" template is used to store simulation results once a simulation is completed. The "{*framework*}/*debug*" template is used to store any errors related with the model under simulation.

We will illustrate the experiment template in Fig. 10 using an example summarizing the modeler's activities. The first step is to create an experiment (typically via client software). For instance, the modeler requests to create an experiment via the PUT channel to URI <.../*Bob*/*DCDpp*/*MyModel*>. RISE will then create an experiment with the requested URI, and it will update

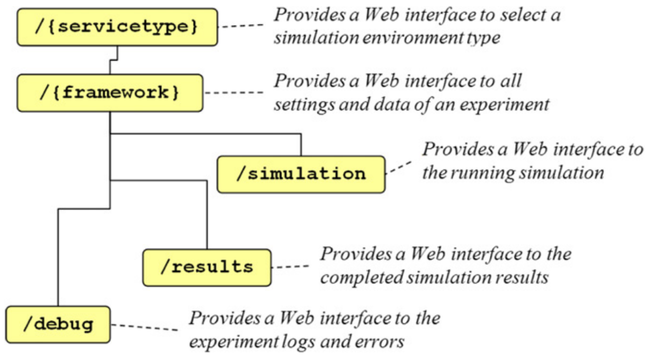

/{servicetype}
Provides a Web interface to select a simulation environment type
/{framework}
Provides a Web interface to all settings and data of an experiment
/simulation
Provides a Web interface to the running simulation
/results
Provides a Web interface to the completed simulation results
/debug
Provides a Web interface to the experiment logs and errors

Fig. 10. Simulation experimental framework template.

it with any information received with the request. In this example, this URI indicates that the experiment name is "*MyModel*", using the "*DCDpp*" simulation environment, and belongs to user "*Bob*". The following steps in this experiment will interact with this URI, submitting the necessary settings and data to run the simulation. The actual data submitted can be specific to the simulation environment. For example, the RISE-based DCD++ [3] requires the modeler to submit (via the PUT channel) an XML message describing the CD++ model-partitioning scheme over a number of machines. For instance, the following XML document shows a CD++ model partitioned over two machines; the "*Producer*" and the "*Consumer*".

```
<Partitions>
    <Partition IP=''10.0.40.175'' PORT=''8080''>
        <MODEL>Producer</MODEL>
    </Partition>
    <Partition IP=''10.0.40.162'' PORT=''8080''>
        <MODEL>Consumer</MODEL>
    </Partition>
</Partitions>
```

The above scheme only assigns DEVS atomic models to partitions, since they are the indivisible blocks in the DEVS formalism. However, these atomic models and their relations are described using the CD++ modeling language. Thus, the modeler also needs to submit the CD++ model to the experiment URI before simulation can take place. This is usually done in a form of zipped file (via the POST channel) to URI <.../*Bob*/*DCDpp*/*MyModel*>. At this point, the simulation can be started (or restarted). This is done via the PUT channel to URI <.../*Bob*/*DCDpp*/ *MyModel*/*simulation*>. This newly created URI becomes the active simulation Web address. Further, once the simulation started, the middleware creates all other partitions in the distributed environment. At this point, the algorithms

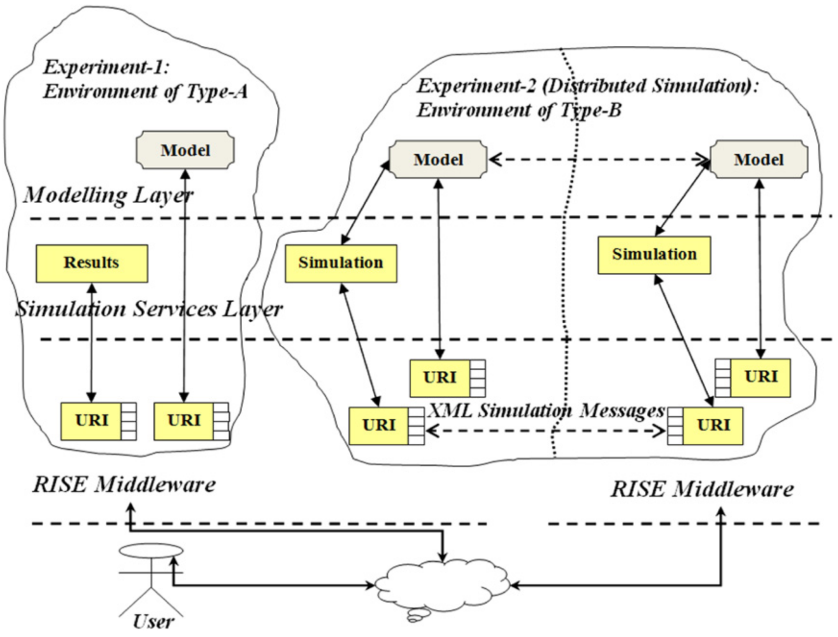


Fig. 11. Example of experiment instances.

in each partition synchronize their execution among each other via exchanging XML messages to each other active simulation URIs.

As discussed above, each of the experiment resources instances are attached to the Web, (since they are URIs) and all the information flows to/from those resources through a set of uniform channels, as discussed in Section 3.1. In fact, the URIs act as wrappers to concrete simulation services and data. To show how the simulation experiments are interfaced and wrapped in URIs, consider the example shown in Fig. 11. We can two experiment instances where *experiment*-1 is of *Type*-A (e.g. a sequential simulation) while *experiment*-2 is of *Type*-B (e.g. a conservative distributed simulation). Here, *experiment*-1 holds the results from a previous simulation (completed state) while *experiment*-2 is executing a simulation distributed over two computers (active simulation state). In distributed simulations like *experiment*-2, the algorithms in each partition synchronize their execution among each other via exchanging XML messages with the URI "{*framework*}/*simulation*". This URI wraps all the simulation components in each partition, including the simulation engines. Each resource in the RISE API is described in terms of its URI, supported channels, messages exchanged via those channels (to execute a function), and type of error responses that may be generated.

Each of the experiments instances follows a workflow shown in Fig. 12. As we can see, the modeler first needs to create an EF instance on RISE, and submit all of the necessary files and configuration settings to it. The EF instance creation is performed via the PUT channel where the experiment settings (e.g. model partitioning) may be optionally submitted as an XML message. If this message is received by 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*).

After the EF instance is created, the modeler can update the existing data via this URI. For instance, models scripts can be submitted as a zipped file via the POST channel. Furthermore, the experiment settings may be updated sending new XML messages via the PUT channel. 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*". 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 is complete).

Once the experiment is set up (Fig. 12), the simulation can be started by creating the Active Simulation URI (e.g. .../*FireModelWithRain*/*simulation*). However, before this, the middleware verifies that the experiment has been set up correctly. This, for example, includes all of the model script files and configurations. The type of the information submitted to an experiment depends on the selected simulation environment for that experiment. Recall that the "{*servicetype*}" template can used to select different simulation environments for a given experiment, as shown in Fig. 10. This Active Simulation URI is to manipulate simulation in an experiment during execution such as: (1) sending distributed simulation synchronization messages (via the POST channel), (2) inserting external events (via the POST channel), and (3) reading simulation results (via the GET channel to URI .../*simulation*?*sim* = *results*). In this case, a copy of the collected simulation results up to the time of receiving the request is sent back to modeler.

Once the Active Simulation URI is correctly created (Fig. 12), the necessary components are deployed on each partition to manage and execute the simulation. At this point, the simulation can exist on one of the following states: ERROR (e.g., a problem in the scripts), ABORTED (the modeler sent a DELETE request), or DONE (the simulation is complete). When the simulation is successful, a results resource (e.g. .../*FireModelWithRain*/*results*) is created to hold all simulation outputs. These results can be downloaded at anytime (for instance, to replay a simulation's visualization without running it again). In order to retrieve results while the simulation is in progress, the modeler needs to issue a read request (via the GET channel) to URI .../{*framework*}/*simulation*? *sim* = *results*. In this case, the client does not need to wait until the simulation completed to obtain results. If the simulation aborts, the errors are stored in the debugging resource (e.g. .../*FireModelWithRain*/*debug*).

3.2.3. *Resources database*

One of the objectives of RISE is to preserve all the experiments' instances unless deleted by an authorized user. Thus, we need a database to maintain all resources instances and settings.

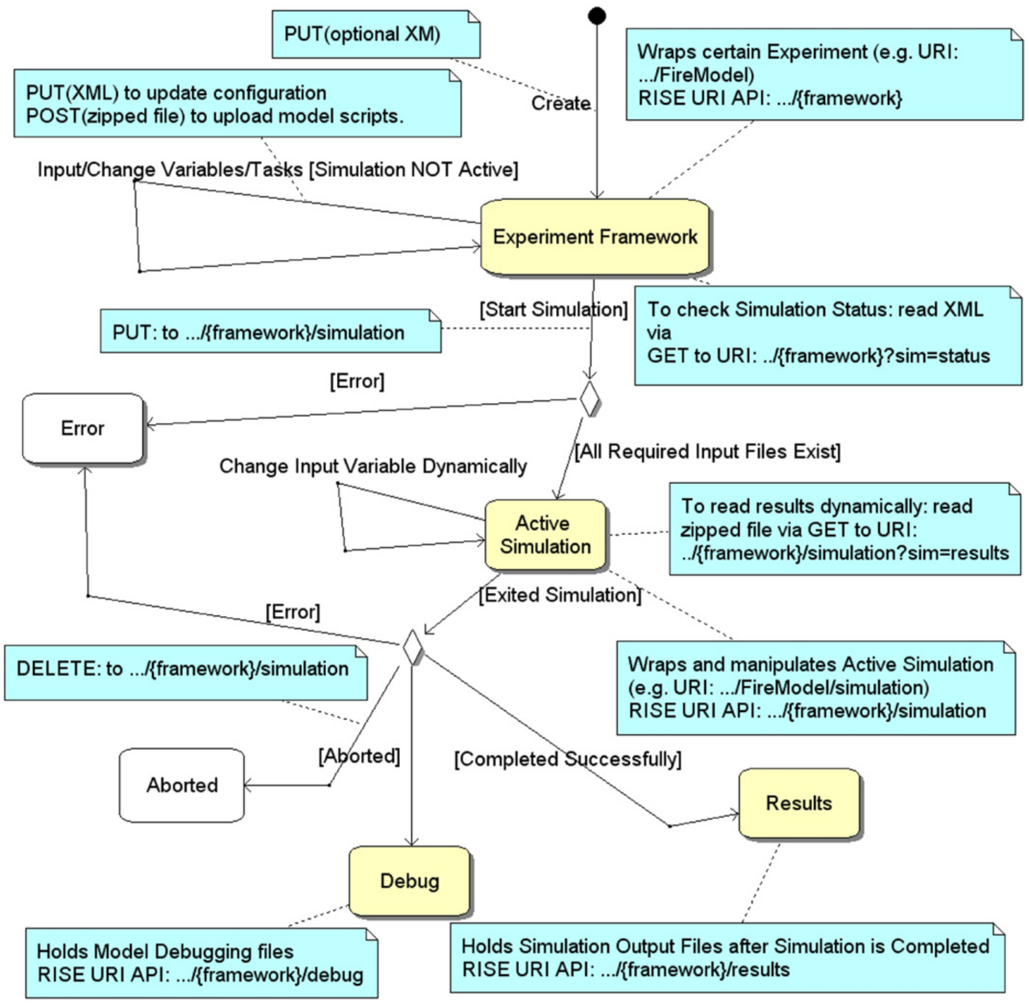

PUT(optional XM)
Create
Wraps certain Experiment (e.g. URI: .../FireModel)
RISE URI API: .../{framework}
PUT(XML) to update configuration
POST(zipped file) to upload model scripts.
Input/Change Variables/Tasks [Simulation NOT Active]
Experiment Framework
[Start Simulation]
To check Simulation Status: read XML via
GET to URI: ../{framework}?sim=status
PUT: to .../{framework}/simulation
[Error]
Error
[All Required Input Files Exist]
Change Input Variable Dynamically
Active Simulation
To read results dynamically: read zipped file via GET to URI:
../{framework}/simulation?sim=results
[Exited Simulation]
[Error]
DELETE: to .../{framework}/simulation
Wraps and manipulates Active Simulation
(e.g. URI: .../FireModel/simulation)
RISE URI API: .../{framework}/simulation
[Aborted]
[Completed Successfully]
Aborted
Results
Debug
Holds Model Debugging files
RISE URI API: .../{framework}/debug
Holds Simulation Output Files after Simulation is Completed
RISE URI API: .../{framework}/results

Fig. 12. Simulation experimental framework pattern context.

The database is divided into sections, and 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, as each incoming message is processed in its own thread (a single thread can only modify one object at a time, but different threads can manipulate different objects simultaneously). The database is transactional, i.e., a transaction can only insert an object in the database when the previous transaction to the same object is completed.

As seen on Fig. 13, 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 types (e.g. DCD++) that are currently used in this user's experiments. Each service object contains the list of the experiments objects created by the subject user for a specific simulation environment type. The database main issues are summarized as follows: (1) RISE divides the database into sections, each section belonging to one username account. This minimizes the number of threads needed to manipulate the same data objects simultaneously. (2) The database (stored in the file system) and the objects in memory have to be synchronized at all times (without degrading performance) as server reboots or failures may happen. The database and cache synchronization do not affect distributed simulation performance, because in RISE, a simulation in an experiment always needs to be restarted if the server fails (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.

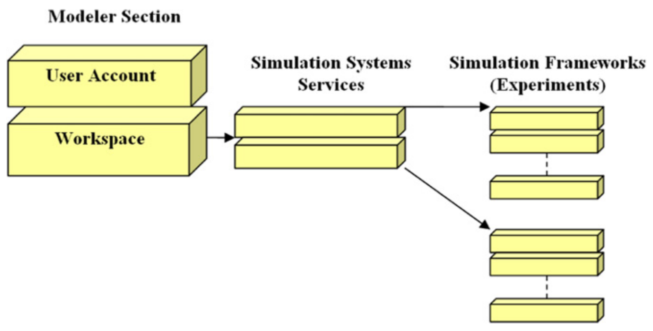

Modeler Section
User Account
Workspace
Simulation Systems Services
Simulation Frameworks (Experiments)

Fig. 13. User section in the database.

3.3. *Message-orientation*

In this section, we discuss the information flow between resources through the virtual uniform channels discussed earlier. Since the 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 the resources. 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, the semantics of all synchronization messages is described in XML [3]. One of reasons is to enhance the decoupling

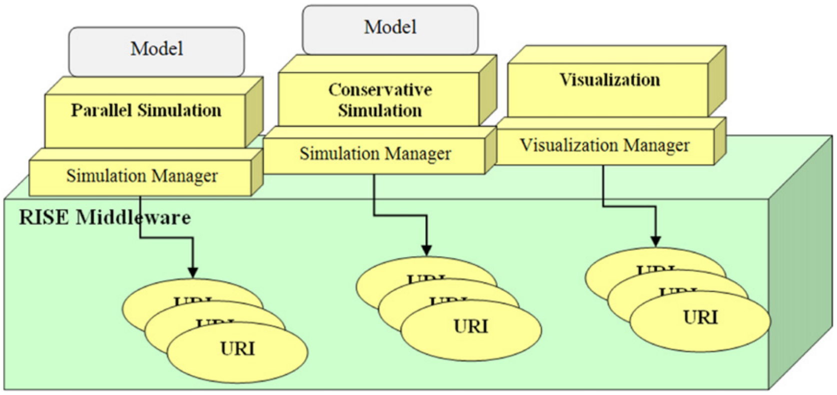


Fig. 14. Overview of RISE typical simulation environments.

of different systems implementations in a number of ways: (1) the design decisions for handling simulation messages become an internal issue that does not need to conform to other systems implementations. (2) The synchronization algorithms can be designed at a level higher than programming, independent of programming details, languages and implementations. This is because the focus of this message-oriented protocol is on what the XML messages can send and the messages expected in return. On the other hand, how these messages are handled during implementation is out of the protocol scope (i.e., one does not need to be a programmer to design the synchronization algorithms).

The flexibility of XML message description also allowed us to enhance performance by aggregating multiple remote messages in distributed simulations. Multiple simulation messages were grouped in a single XML message to reduce the number of remote transmissions. The flexible message description can also be used to support multiple synchronization protocols, accommodating different interoperability domains. In order to transmit a message uniformly, we use four pieces of information: the destination URI, the channel name, the message syntax, and the actual message data. To accommodate different protocols, the software needs to pack the XML messages according to the synchronization protocol semantics. From the sending routine viewpoint, sending different types of XML messages is the same. At the receiver, the XML messages are processed similarly to the transmission. The information packed in an XML message still needs to be mapped to the local software implementation, but these issues are irrelevant to other systems implementations (and synchronization protocols), which is a fundamental factor for decoupling systems implementations and improving interoperability.

4. Examples: services and applications

In this section, we show different examples of the use of RISE. To do so, we will provide an overview of a number of RISE-based services. Some of them are placed within the RISE general container. Other applications are placed on the client side, where they can use RISE services.

4.1. Simulation and visualization environments

RISE is designed as a container for different simulation environments, which are interfaced with the RISE component on the same machine via the IPC queues. However, from the API viewpoint, adding new services to the URI template structure is similar to regular Web site URIs.

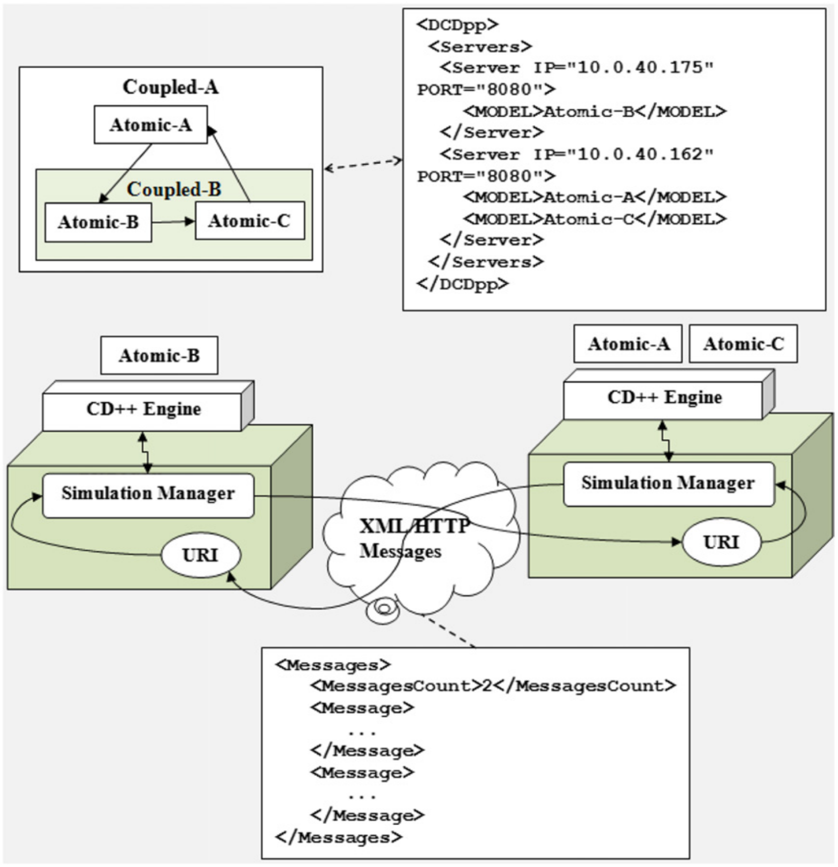
Fig. 14 shows an example of three types of services: visualization, conservative, and parallel simulation. The simulation/visualization managers shown in Fig. 14 are actually a part of the RISE middleware. The Manager component usually extends the RISE generic component to handle environment specifics such *data distribution* across the distributed environment. For example, the visualization environment can interoperate with the open Second Life visualization environment [34]. The visualization manager manages the visualization representations and their distributions to registered clients who view them locally. The other simulation environments in the figure need *time management* according to their specific mechanisms. However, if a simulation environment needs to synchronize with other remote systems, the manager also handles the data distribution mechanisms.

We next show an example based on the distributed CD++ (DCD++) simulator [3]. This is a modeling and simulation toolkit capable of executing DEVS [46] and Cell-DEVS models [40]. Fig. 15 shows (on the top-left of the figure) a DEVS model, called *Coupled*-A. DEVS coupled models are the structural models that contain other internal coupled/atomic models. In this example, *Coupled*-A consists of two models: *Atomic*-A and *Coupled*-B, each including input and output ports. The input port of *Atomic*-A is connected to the *Coupled*-B output port (and vice versa). *Coupled*-B also consists of two atomic models: *Atomic*-B and *Atomic*-C. The XML partitioning scheme (on the top-right of Fig. 15): it assigns *Atomic*-B to a partition and *Atomic*-A and *Atomic*-C to the other partition. A partition is identified by the machine's IP address and the middleware TCP port. As the two partitions in Fig. 15 have different IP addresses, each of them is executed on a different machine. The modeler can change this XML partitioning scheme at anytime.

The modeler runs an experiment by creating the URI …/*DCDpp*/{*framework*}/*simulation* on the main RISE server (i.e. the server used to create the DCD++ experiment). As a result, the main RISE server creates the *simulation manager* component within the local EF partition (Fig. 15). This *simulation manager* creates all necessary local components; including the CD++ engine in its partition. Furthermore, it contacts the remote RISE servers to create other partitions of the experiment. The CD++ simulation environments only exist during active simulation. At this point, each CD++ engine recognizes the parts of the model it is supposed to execute (based on the XML partitioning scheme). In this example, the CD++ on the left executes *Atomic*-B while the CD++ on the right executes *Atomic*-A and *Atomic*-C. The CD++ engines recognize the ports interconnections based on the coupled model definition file. Once all components are created, the algorithms synchronize their local activities via XML synchronization messages between each other URIs [3]. The DCD++ experiment follows the pattern described in Section 3.2.2.

4.2. Distributed simulation standards foundations

As a proof of concept of a semantic synchronization protocol with specific requirements, the scheme in this section can be used



```
<DCDpp>
 <Servers>
  <Server IP="10.0.40.175"
PORT="8080">
    <MODEL>Atomic-B</MODEL>
  </Server>
  <Server IP="10.0.40.162"
PORT="8080">
    <MODEL>Atomic-A</MODEL>
    <MODEL>Atomic-C</MODEL>
  </Server>
 </Servers>
</DCDpp>
```

```
<Messages>
   <MessagesCount>2</MessagesCount>
   <Message>
      ...
   </Message>
   <Message>
      ...
   </Message>
</Messages>
```

Fig. 15. Overview of RISE-based DCD++ environment.

to standardize synchronization interfaces between different DEVS implementations [41]. To do so, the scheme uses a centralized time manager, as adopted by the P-DEVS synchronization protocol [46] (which, implemented by most existing DEVS tools, is the basis for the current efforts on DEVS standardization [41]).

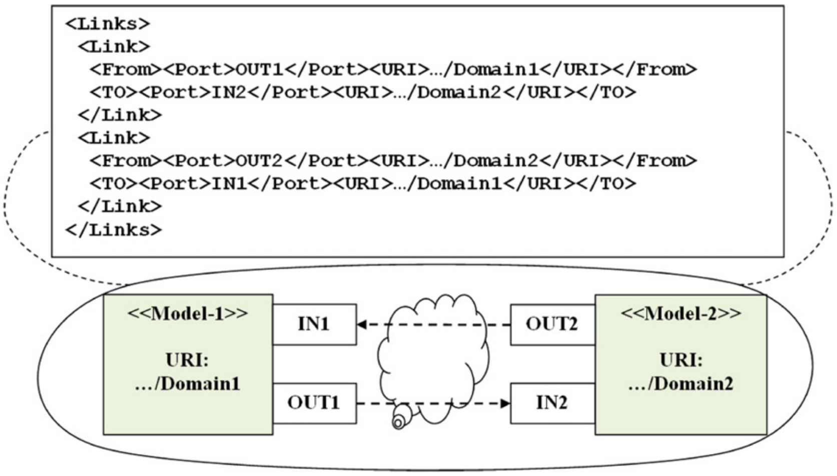
As discussed earlier, RISE philosophy is that systems implementations can interoperate easily if they are decoupled (that is, the proposed algorithms and standardized protocols should not enforce the software design, while respecting legacy systems designs). To do so, the RISE-based algorithms place models in each partition as black boxes interconnected with other models via input/output ports [2]. The simulation is executed in cycles where the synchronization messages exchanged are in XML. These algorithms can handle dynamic simulations, whose partitions can join/disjoin a runtime. The main idea is that systems algorithms synchronize the simulation by exchanging XML messages wrapped within HTTP envelopes, leaving the implementation details for the developers.

As seen in Fig. 16, this scheme [2] puts each model as a black box on each partition. This makes it easier to interoperate heterogeneous models that are executable by a specific simulation environment. The only information that each system needs to identify is how the model influences other remote models. This information is provided to each system (domain) in XML, as in the example shown in Fig. 17.

The synchronization scheme in Fig. 17 adapts the DEVS synchronization style, which can be accepted and supported by the numerous distributed DEVS simulators (and, on the other hand, additional synchronization schemes can be proposed). In our scheme, the RISE Time Manager (RISE-TM) component advances the simulation in cycles comprised of two steps:

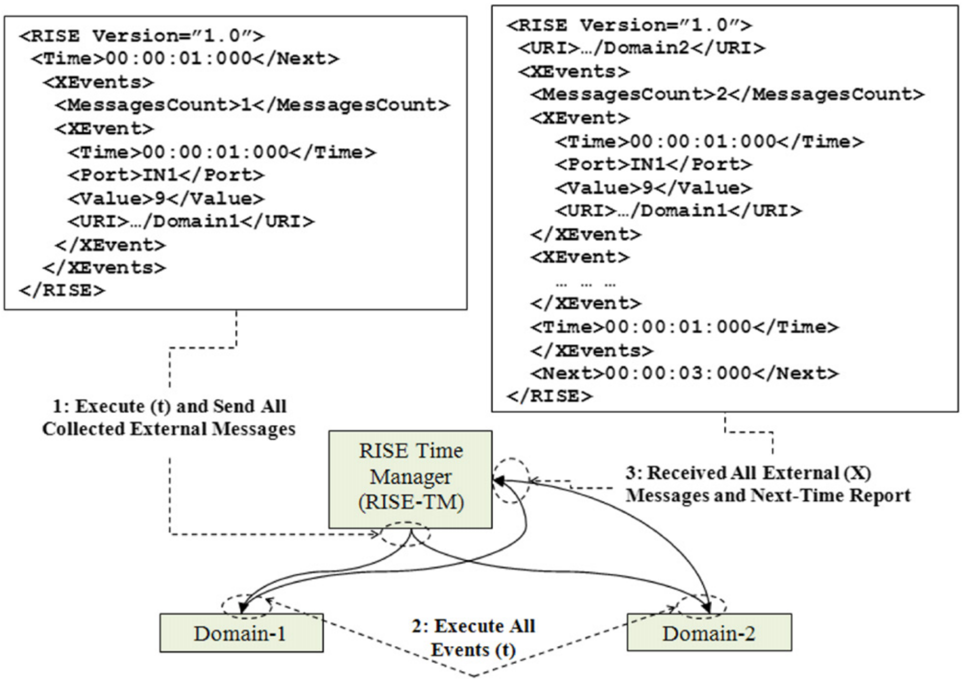
(1) RISE-TM requires all domains to execute all of their events at the current (or newly calculated) RISE time (i.e. the time that simulation partitions are allowed to execute events at). This is done via sending (in parallel) one XML message to each relevant domain, including the current RISE time and all the 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 message containing all external messages generated for other domains (if any). All generated external messages must be stamped with the current RISE time (or larger). This message also contains the next event time in the sender partition, which is the time of the next event in a partition larger than RISE time.
(2) Once RISE-TM receives replies from all the relevant domains, it calculates the next RISE time. RISE-TM merges all the external messages generated, and passes them to all the relevant domains at the beginning of the next simulation cycle. If RISE-TM finds the new RISE time to be infinity (or receives a stop request from the modeler), the simulation ends.

Thus, synchronization via exchanging messages describing information in XML enhances protocol support flexibility (for example, comparing to RPCs describing information in programming parameters). This is because, in practice, systems design and implementations are less sensitive to XML messages comparing to programming procedures interface. For example, DCD++ can interoperate with another DCD++ via the scheme in Fig. 15. However, DCD++ can also interoperate with other DEVS tools by using the scheme presented in this section too. In this case, a DCD++ can be set up (as part of an experiment) to use a specific protocol when synchronizing the simulation with other remote DEVS-based system. In this case, the DCD++ sends/handles XML messages according to the required protocol rules. This concept can be extended for



```
<Links>
 <Link>
  <From><Port>OUT1</Port><URI>.../Domain1</URI></From>
  <TO><Port>IN2</Port><URI>.../Domain2</URI></TO>
 </Link>
 <Link>
  <From><Port>OUT2</Port><URI>.../Domain2</URI></From>
  <TO><Port>IN1</Port><URI>.../Domain1</URI></TO>
 </Link>
</Links>
```

Fig. 16. Models interconnection across domains.



```
<RISE Version="1.0">
 <Time>00:00:01:000</Next>
  <XEvents>
   <MessagesCount>1</MessagesCount>
   <XEvent>
    <Time>00:00:01:000</Time>
    <Port>IN1</Port>
    <Value>9</Value>
    <URI>.../Domain1</URI>
   </XEvent>
  </XEvents>
</RISE>
```

```
<RISE Version="1.0">
 <URI>.../Domain2</URI>
 <XEvents>
  <MessagesCount>2</MessagesCount>
  <XEvent>
    <Time>00:00:01:000</Time>
    <Port>IN1</Port>
    <Value>9</Value>
    <URI>.../Domain1</URI>
  </XEvent>
  <XEvent>
    ... ... ...
  </XEvent>
  <Time>00:00:01:000</Time>
  </XEvents>
  <Next>00:00:03:000</Next>
</RISE>
```

1: Execute (t) and Send All Collected External Messages
RISE Time Manager (RISE-TM)
3: Received All External (X) Messages and Next-Time Report
Domain-1
2: Execute All Events (t)
Domain-2

Fig. 17. Models interconnection across domains.

non-DEVS simulators. For example, what do we need to do to interoperate, let us say, a DEVS-based simulator on one end, with a simulator based on Process Algebra models on the other end? In practice, we can use RISE to solve this issue in two steps (carried out by users familiar with both systems simulation engines, though they do not need to be programmers). The first step is to determine if a synchronization algorithm can be developed to interoperate both systems (in this case, both systems are interoperable). The second step is to describe the information exchanged in XML and the rules for exchanging such messages, similarly to the example in Fig. 17. In this case, such protocol can be supported by systems using the RISE middleware to interoperate.

Conclusion

We presented a new concept and ideas for distributed simulation, implemented as the RISE middleware. We introduced the software design of this middleware along with a number of RISE-based services and applications examples. We showed that the three design principles of RISE (i.e. general resource-orientation, uniform-interface, and message-orientation) can achieve various objectives with respect to interoperability. Particularly, hiding interoperating systems heterogeneities (by decoupling systems APIs from internal implementations), composition scalability (by advocating uniform-interface), and dynamicity (since information channels automatically exist). We also showed how the middleware could be designed to serve as a general container to hold simulation environments. This is done via the concept of layered interoperability: The *middleware* layer (the interoperability methods), the *simulation* layer (i.e. the simulation environments; in our examples above we used DCD++ as a proof of concept), and the *modeling* layer (i.e. partitioning granularity). Thus, the middleware design is not specific to any simulation package, keeping the door open for additional extensions. Interoperating in the Web style at the Web layer level (i.e. RESTful WS) allows the middleware to take advantage of any new Web-based applications and ideas such as Web 2.0. Doing this with SOAP-based WS is difficult, mainly because SOAP creates an RPC layer above the Web layer that exposes internal systems implementations to each other. On the other hand, RISE provides better interoperability applying RESTful WS principles. 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. (2) Projects that are expecting to interoperate with systems outside their control. In this case, as we did in RISE design, systems APIs can be moved outside the implementation, decoupling the system components. (3) Projects that are expecting to scale up well when composing large number of systems (components). The REST style has proved to work well on the WWW, which RISE imitates. (4) Projects that expect having different components joining/disjoining 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 to join/disjoin all times.

References

[1] K. Al-Zoubi, G. Wainer, Interfacing and coordination for a DEVS simulation protocol standard, in: Proc. 12th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications, DS-RT2008, 2008, pp. 300–307.

[2] K. Al-Zoubi, Wainer, Performing distributed simulation with RESTful Web-services, in: Proc. 2009 Winter Simulation Conference, WSC 2009, 2009, pp. 1323–1334.

[3] K. Al-Zoubi, G. Wainer, Using REST Web services architecture for distributed simulation, in: Proc. 23rd ACM/IEEE/SCS Proceedings of Principles of Advanced and Distributed Simulation, PADS2009, 2009, pp. 114–121.

[4] C. Boer, A. Bruin, A. Verbraeck, Distributed simulation in industry—a survey, part 3—the HLA standard in industry, in: Proc. 2008 Winter Simulation Conference, WSC2008, 2008, pp. 1094–1102.

[5] C. Boer, A. Bruin, A. Verbraeck, A survey on distributed simulation in industry, Journal of Simulation 3 (1) (2009) 3–16.

[6] A. Boukerche, F. Iwasaki, R. Araujo, E. Pizzolato, Web-based distributed simulations visualization and control with HLA and Web services, in: Proc. 12th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications, DS-RT'08, 2008, pp. 17–23.

[7] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.

[8] J. Byrne, C. Heavey, P. Byrne, A review of Web-based simulation and supporting tools, Simulation Modelling Practice and Theory 18 (3) (2010) 253–276.

[9] E. Chia, M. Shamsir, Z. Hussein, S. Hashim, GridMACS portal: a grid web portal for molecular dynamics simulation using GROMACS, in: Proc. 4th IEEE Asia International Conference On Mathematical/Analytical Modelling and Computer Simulation, AMS2010, 2010, pp. 507–512.

[10] R. Chinnici, J. Moreau, A. Ryman, S. Weerawarana, Web services description language (WSDL) version 2.0 part 1: core language, 2007. http://www.w3.org/TR/wsdl20/ (accessed March 2012).

[11] P. Davis, R. Anderson, Improving the composability of department of defense models and simulation, Rand Corporation, Santa Monica, California, 2003. http://www.rand.org/pubs/monographs/MG101.html (accessed March 2012).

[12] R. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Dissertation, Dept. of Computer Science, Univ. of California, Irvine, CA, USA, 2000.

[13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext transfer protocol–HTTP/1.1. RFC 2616, 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html (accessed March 2012).

[14] P. Fishwick, L. Jinho, P. Minho, RUBE: a customized 2D and 3D modeling framework for simulation, in: Proc. 2003 Winter Simulation Conference, WSC2003, 2003, pp. 755–762.

[15] J. Fitzgibbons, R. Fujimoto, D. Fellig, D. Kleban, A. Scholand, IDSim: an extensible framework for interoperable distributed simulation, in: Proc. IEEE International Conference on Web Services, ICWS2004, 2004, pp. 532–539.

[16] I. Foster, C. Kesselman, Globus: a metacomputing infrastructure toolkit, International Journal on Supercomputer Applications 11 (2) (1997) 115–128.

[17] R. Fujimoto, Parallel and Distribution Simulation Systems, John Wiley & Sons, New York, 2000.

[18] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, URI Templates, 2010. http://tools.ietf.org/html/draft-gregorio-uritemplate-04 (accessed March 2012).

[19] M. Hadley, Web application description language, WADL, 2009. http://www.w3.org/Submission/wadl/ (accessed October 2008).

[20] IBM Mashup Center. http://www-01.ibm.com/software/info/mashup-center/ (accessed March 2012).

[21] IBM Software Group. Why mashups matter, 2008. ftp://ftp.software.ibm.com/software/lotus/lotusweb/portal/why_mashups_matter.pdf (accessed March 2012).

[22] P. Ke, S. Turner, C. Wentong, L. Zengxiang, A service oriented HLA RTI on the grid, in: Proc. 2007 IEEE International Conference on Web Services, ICWS 2007, 2007, pp. 984–992.

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

[24] T. Kim, S. Hoon Hong, Y. Chung, I. Park, Web-based CAD framework for low cost SoC design prototyping, in: Proc. 2010 IEEE International SoC Design Conference, ISOCC2010, 2010.

[25] K. Kim, W. Kang, CORBA-based, multi-threaded distributed simulation of hierarchical DEVS models: transforming model structure into a non-hierarchical one, in: Proc. International Conference on Computational Science and its Applications, ICCSA2004, 2004.

[26] H. Leong, D. Brutzman, D. McGregor, C. Blais, Web services integration on the fly for service-oriented computing and simulation, in: Proc. of the 2009 Spring Simulation Multiconference, SpringSim2009, 2009.

[27] T. O'Reilly, What Is Web 2.0, 2005. http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html (accessed March 2012).

[28] E. Page, Beyond speedup: PADS, the HLA and web-based simulation, in: Proc. 1999 Winter Simulation Conference, WSC1999, 1999, pp. 2–9.

[29] E. Page, R. Briggs, J. Tufarolo, Toward a family of maturity models for the simulation interconnection problem, in: Proc. 2004 Simulation Interoperability Workshop, SIW2004, 2004.

[30] M. Papazoglou, Web Services: Principles and Technology, Prentice Hall, 2007.

[31] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM 15 (12) (1972) 1053–1058.

[32] L. Richardson, S. Ruby, RESTful Web Services, O'Reilly Media, Inc., Sebastopol, California, 2007.

[33] R. Rocha, R. Araujo, M. Campos, A. Boukerche, Understanding and building interoperable, integrable and composable distributed training simulations, in: Proc. 14th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications, DS-RT2010, 2010, pp. 121–128.

[34] Second Life. http://secondlife.com/ (accessed March 2012).

[35] C. Seo, B. Zeigler, Automating the DEVS modeling and simulation interface to web services, in: Proc. of the 2009 Spring Simulation Multiconference, SpringSim2009, 2009.

[36] W. She, I. Yen, B. Thuraisingham, WS-Sim: a web service simulation toolset with realistic data support, in: Proc. 34th IEEE Computer Software and Applications Conference Workshops, COMPSACW2010. 2010.

[37] S. Strassburger, T. Schulze, R. Fujimoto, Future trends in distributed simulation and distributed virtual environments: results of a peer study, in: Proc. 2008 Winter Simulation Conference, WSC2008, 2008, pp. 777–785.

[38] A. Tolk, Interoperability and composability, in: C. Banks, J. Sokolowski (Eds.), Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains, Wiley, New Jersey, 2010, pp. 373–402.

[39] S. Tuecke, I. Foster, et al. Open grid services infrastructure (OGSI) version 1.0. open grid services infrastructure working group (OGSI-WG) 2003. http://xml.coverpages.org/OGSI-SpecificationV110.pdf (accessed March 2012).

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

[41] G. Wainer, K. Al-Zoubi, S. Mittal, J. Risco Martín, H. Sarjoughian, B. Zeigler, in: G. Wainer, P. Mosterman (Eds.), Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press. Taylor and Francis, 2010, pp. 389–494 (Chapters 15–18).

[42] G. Wainer, R. Madhoun, K. Al-Zoubi, Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web services, Simulation Modelling Practice and Theory 16 (9) (2008) 1266–1292.

[43] Web-services AXIS. http://ws.apache.org/axis/java/user-guide.html (accessed March 2012).

[44] Q. Xiang, G. Chen, Y. Wang, Distributed simulation based on web enabling HLA, in: Proc. 2nd IEEE International Conference on Artificial Intelligence, Management Science and Electronic Commerce, AIMSEC2011, 2011, pp. 2062–2064.

[45] T. Yoo, K. Kim, S. Song, H. Cho, E. Yucesan, Applying web services technology to implement distributed simulation for supply chain modeling and analysis, in: Proc. 23rd ACM/IEEE/SCS Proceedings of Principles of Advanced and Distributed Simulation, PADS2009, 2009, pp. 863–873.

[46] B. Zeigler, H. Praehofer, T. Kim, Theory of Modeling and Simulation, Academic Press, San Diego, CA, 2000.

[47] S. Zhang, P. Coddington, A. Wendelborn, A national grid submission gateway for eScience, in: Proc. 7th IEEE International Conference on E-Science, e-Science2011, 2011, pp. 23–30.

[48] H. Zhang, H. Wang, D. Chen, Integrating web services technology to HLA-based multidisciplinary collaborative simulation system for complex product development, in: Proc. 12th IEEE International Conference on Computer Supported Cooperative Work in Design, CSCWD2008, 2008, pp. 420–426.

[49] S. Zhu, Z. Du, X. Chai, GDSA: a Grid-based distributed simulation architecture, in: Proc. 6th IEEE Int'l Symp. on Cluster Computing and the Grid Workshops, CCGRID2006, 2006, pp. 66–71.

[50] K. Zhu, H. Song, J. Gao, Web-based atmospheric nucleation data management and visualization, in: Proc. 2nd IEEE Int'l Conference on Networking and Distributed Computing, ICNDC2011, 2011, pp. 127–131.

Khaldoon Al-Zoubi received both Ph.D. (2011) in Electrical and Computer Engineering and M.C.S. (2006) from Carleton University (Ottawa, Ontario, Canada). He received a B.Sc. in Electrical and Computer Engineering (1995) from the University of Louisiana at Lafayette (Lafayette, Louisiana, USA). His current research interests are related with modeling methodologies, parallel/distributed simulation, Grid Computing and Real-Time systems. He is also a senior Software Engineer and Programmer with over 14 years of industry experience occupying a number of seniority and leadership positions throughout the USA and Canada. His industry experience spreads over wide range of areas such as embedded software and mobility, air-traffic software management and telecommunications, security software for explosives and narcotics detections. His email is kazoubi@connect.carleton.ca.

Gabriel Wainer, SMSCS, SMIEEE, received the M.Sc. (1993) at the University of Buenos Aires, Argentina, and the Ph.D. (1998, with highest honors) at the University of Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. After being Assistant Professor at the Computer Science Department of UBA, in July 2000 he joined the Department of Systems and Computer Engineering at Carleton University (Ottawa, ON, Canada), where he is now Full Professor. He has held visiting positions at the University of Arizona, LSIS (CNRS), University Paul Cezanne, University of Nice, INRIA Sophia-Antipolis (France), UCM (Spain) and others. He is the author of three books and over 260 research articles; he edited four other books, and helped organizing over 120 conferences, including being one of the founders of SIMUTools and SimAUD. He was PI of different research projects (funded by NSERC, CFI, GRAND, MITACS, Autodesk Research, IBM, Intel, INRIA, CANARIE, Precarn, Usenix, CONICET, ANPCYT). Prof. Wainer is the Vice-President Conferences, and was a Vice-President Publications and a member of the Board of Directors of SCS. He is Special Issues Editor of SIMULATION, member of the Editorial Board of IEEE Computing in Science and Engineering, Wireless Networks (Elsevier), Journal of Defense Modeling and Simulation (SCS), and International Journal of Simulation and Process Modeling (Inderscience). He is the head of the Advanced Real-Time Simulation lab, located at Carleton University's Centre for advanced Simulation and Visualization (V-Sim). He is also the Director of the Ottawa Center of The McLeod Institute of Simulation Sciences and chair of the Ottawa M&SNet. He has been the recipient of various awards, including the IBM Eclipse Innovation Award, SCS Leadership Award, and various Best Paper awards. He has been awarded Carleton University's Research Achievement Award (2005–2006), the First Bernard P. Zeigler DEVS Modeling and Simulation Award, and the SCS Outstanding Professional Award (2011). His current research interests are related with modeling methodologies and tools, parallel/distributed simulation and real-time systems. His e-mail and web addresses are gwainer@sce.carleton.ca and www.sce.carleton.ca/faculty/wainer.