Proceedings of the 2009 Winter Simulation Conference
M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin and R. G. Ingalls, eds.

PERFORMING DISTRIBUTED SIMULATION WITH RESTFUL WEB-SERVICES

Khaldoon Al-Zoubi
Gabriel Wainer

Dept. of Systems and Computer Engineering
Carleton University Centre of Visualization and Simulation (V-Sim),
1125 Colonel By Dr. Ottawa, ON, Canada.

ABSTRACT

Distributed simulations are mainly used to interoperate heterogeneous simulators or geographically distributed models. We present here RESTful-CD++, the first distributed simulation middleware based on REST (Representational State Transfer) Web-services. RESTful-CD++ middleware enables heterogeneous independent-developed simulation components to interoperate with much flexibility and simplicity. REST has the potential to advance distributed simulation state-of-the-art towards plug-and-play or automatic/semi-automatic interoperability. This because of its lightweight approach hides internal software implementation by using universal uniform interface and describing connectivity semantics in form of messages, usually XML. In contrast, other approaches expose functionalities in heterogeneous RPCs that often reflect internal implementation and describe semantics in form of procedure parameters. Further, REST enables simulations to mashup with Web 2.0 applications, which makes simulation in link with any device attached to the Web dynamically at runtime. The CD++ tool is now the first simulation engine to use RESTful middleware to perform distributed simulation in large-scale.

1 INTRODUCTION

Distributed simulation technologies were created to execute simulations on distributed computer systems (i.e., on multiple processors connected via communication networks) (Fujimoto 2000). A focus of distributed simulation software has been on how to achieve model reuse via interoperation of heterogeneous simulation components. Other benefits include reducing execution time, connecting geographically distributed simulation components (without relocating people/equipment), interoperating different vendor simulation toolkits, providing fault tolerance and information hiding – including the protection of intellectual property rights (Boer et al. 2009, Fujimoto 2000).

The defense sector is currently one of the largest users of distributed simulation technology, mainly to provide virtual distributed training environment between remote parties, relying on the High Level Architecture (HLA) (Khul et al. 1999) middleware to provide a general architecture for simulation interoperability and reuse. On the other hand, the current adoption of distributed simulation in the industry is still limited. In recent years, there have been some studies (conducted in the form of surveys) to analyze these type of issues (Boer et al. 2009, Strassburger et al. 2008). Clearly, technology adoption in the industry is based on return-of-investment policies. For example, the authors in (Boer et al. 2009) suggested that, in order to make distributed simulation more attractive to the industrial community, we need a lightweight commercial-off-the-shelf (COTS) based architecture with higher cost/benefit ratio. This means that COTS components must be assembled and interoperated with each other efficiently, effortlessly and quickly. This brings the business environment mentality of "Try-before-buy" (Strassburger et al. 2008) to reduce cost when selecting between different vendors sub-components options to be integrated in the final product. Nevertheless, it has been predicted that in the coming years, the sectors that will drive future advancement in distributed simulation is not only the defense sector, but also gaming industry, the high-tech industry (e.g. auto, manufacturing and working training), emergency and security management (Strassburger et al. 2008). Further, the study illustrated in (Strassburger et al. 2008) found out that the highest rated applications in future distributed simulation efforts include the integration of heterogeneous resources, joining computer resources for complex simulations and training sessions. Therefore, simulation middleware plays a vital role with future advancements, since it is main goal is interoperating different vendor heterogeneous simulations together. Consequently, this study (Strassburger et al. 2008) identifies some research challenges: (1) Plug-and-Play capability: the middleware should be able to support coupling simulation models in such a way that the technical approach and standards gain acceptance in industry. In other words, interoperability should be achieved effortlessly. (2) Automated or semi-automatic semantic interoperability between domains: to achieve the plug-and-play challenge,

interoperability must be achieved at the semantic level. Our presented methodology here is taking distributed simulation in the path toward these goals comparing to other existing approaches. It uses RESTful Web-services to interoperate remote simulation components to perform distributed simulation in large-scale networks.

The non-military distributed simulation community has reached out to other technologies to widen distributed simulation use and to overcome HLA shortcomings such as standards complexity, programming languages dependency and poor scalability. CORBA (Henning et al. 1999) was used during the 1990s and early 2000s to provide interoperability between heterogeneous simulations. CORBA exposes different distributed objects whereas each object exposes various procedures. In this case, those procedures glue different objects operations together, giving the impression of a local procedure call when in fact it is an operation invoked remotely. CORBA operations are described using CORBA IDL, with syntax similar to regular programming languages, which can be compiled into actual programming language code such as Java or C++. The distributed simulation community has also turned to SOAP-based Web-services since its appearance in year 2000 to provide interoperability between different simulation processors. The SOAP-based Web-services (Papazoglou 2007) provides interoperability in a similar way to CORBA Remote Procedure Calls (RPC)-style. It exposes ports (analogous to CORBA objects) where each port is addressed using a URI (analogous to CORBA object references), and it exposes a number of various procedures, usually called services. Those services are described in XML WSDL documents (analogous to CORBA IDL), allowing client programmers to compile WSDL into actual programming language procedures stubs. At runtime, when a client program invokes a service stub: (1) the client converts the RPC into a SOAP message (encoded using XML), (2) wraps the SOAP message within an HTTP message, and (3) sends this HTTP message to the server using the HTTP POST method. Upon the message receipt, the server converts the SOAP message into the appropriate procedure call and responds to the client in the same way. To interoperate two independently developed simulations, the RPCs exposed need to be integrated. For example, when a simulation wants to send an event to another remote simulation, it needs to act as a client and invoke the appropriate procedure of the suitable port of that remote simulation, as shown in Figure 1.

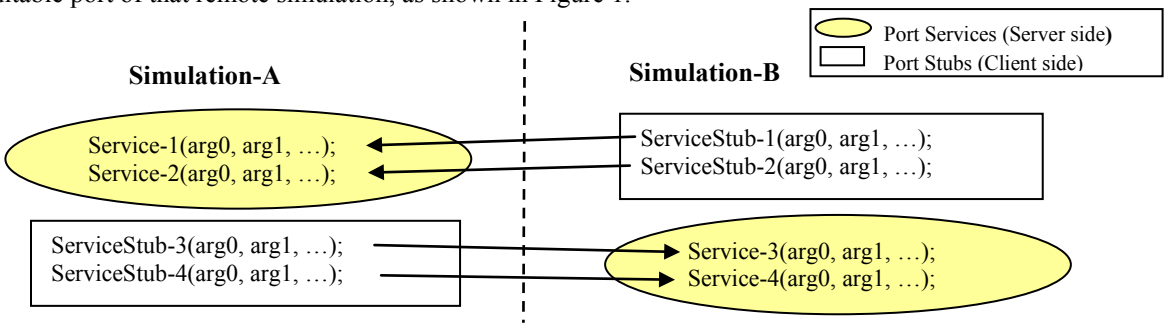


Figure 1: SOAP-based Web-services RPC-based Programming Model

On the other hand, RPCs are heterogeneous (because they were invented by different programmers) and distributed simulation connectivity semantics are described on those RPCs parameters. RPCs directly influence the interoperability integration effort, since they are actually the Application Programming Interface (API) of a simulation component. Further, RPCs often reflect the software internal implementation since they glue distributed software together. WSDL only helps programmers construct service stubs so they can compile them correctly with their software, but programmers still need to integrate the diverse software functionalities. Indeed, interoperating independent-developed simulations where each exposes heterogeneous API (that is tied to internal implementation) and describing semantics in programming procedure parameters is not a trivial task. In reality, coordination among simulations in a distributed environment is much more complex than passing events. They need to implement synchronization algorithms to ensure simulation correctness and efficient execution. They may also need to select a synchronization algorithm over another from different available options. Further, standardizing APIs in form of RPCs can cause inflexibility of possible future improvements like synchronization algorithms, since we need to define new procedures and parameters (that may not be friendly to some simulation components internal implementation). In practice, these interoperability challenges also apply to independent-developed software packages that were based on the same simulation formalism. This is because programmers implement the same functional requirements with different software design and implementation style. We are currently facing these challenges in the DEVS standardization process, which aims on interoperating different DEVS-based simulation packages using SOAP-based Web-services. Our presented proposal described in (Al-Zoubi et al. 2008) attempts to solve some of these issues by minimizing the number of exposed RPCs and transferring simulation messages as XML attachments, allowing simulations to hide (as much as possible) their internal implementation. However, integration is still proven its difficulty because it is uneasy for some simulation packages to hide their internal software implementation when they expose too many ports and RPCs. Clearly, this is not the path to achieve plug-and-play capability or automated/semi-automatic semantic interoperability. A project management in a business has to

question if this task worth the cost, particularly if we need to add several independent-developed simulators and models where each exposes many ports and RPCs.

The Representational State Transfer (REST), instead, provides interoperability by imitating the World Wide Web (WWW) style and principles (Fielding 2000). RESTful Web Services are gaining increased attention with the advent of Web 2.0 (O'Reilly 2005) and the concept of mashups because of its simplicity. Mashups group various services from different providers and present them as a bundle. For example, IBM Mashup Center <http://www-01.ibm.com/software/info/mashup-center/> provides different enterprise mashup solutions, allowing businesses to assemble new applications dynamically from various consumable assets. For instance, WebSphere sMash provides RESTful Web 2.0-based mashup applications, allowing enterprise reuse assets with more flexibility and cost-effectiveness. The main motivation behind mashups is reusing existing assets through the Web (and other benefits are driven from this principle). RESTful Mashups can benefit distributed simulation in a number of ways such as introducing real systems (e.g. sensors, devices, weather data, maps, etc.) in the simulation loop: any device connected to the Internet can be part of the simulation environment, as Web 2.0 extends the concept of devices beyond regular computing machines.

REST separates the software interface from the internal implementation; hence, services can be exposed while the implementation is hidden form consumers. Providers just need to conform to the service agreement, which comes in the form of messages (e.g. XML). This is provided by exposing services as "resources" (instead of RPCs) and manipulated through a uniform interface, usually four HTTP methods: GET (to read a resource), PUT (to create/update a resource), POST (to append to a resource), and DELETE (to remove a resource). These methods are literally the only entrances to those resources (or services). Resources are named (addressed) with unique URIs similar to Web sites. Therefore, a simulation, for example, may search, locate and connect to a Web 2.0 application at runtime, since there is no need to construct services RPC stubs at compile time. REST describes connectivity semantics in form of messages (usually in XML messages, instead of using RPCs parameters). Therefore, REST can separate internal implementation and interface semantics. Obviously, it is extremely flexible to apply possible future changes to XML messages comparing to procedure parameters. Further, it is much easier to integrate two independent-developed simulators when both communicate with XML messages using the same standardized universal uniform interface. However, to achieve full plug-and-play capability, connectivity semantics (messages) must be *standardized*, at least within the same domain such as the DEVS domain. Of course, *standardizing* XML messages is very flexible for future changes or synchronization algorithms improvements (moreover when compared to procedure parameter passing). REST also brings simplicity to distributed simulation, which reduces cost and risk against return benefits in business environment. To show this simplicity affect, for example, Amazon.com services are provided using both REST-based and SOAP-based Web-services. On the other hand, Amazon reports that 95% of the usage is of the REST Web-service in spite of larger retail business partners like ToysRUs that uses SOAP WS (O'Reilly 2005). REST clear-cut interface can enlarge the distributed simulation users instead of only being used by high skilled programmers at this time. REST has been used in many applications such as Yahoo <http://developer.yahoo.com/>, Flicker <http://www.flickr.com/services/ap/>, and Amazon S3 <http://s3.amazonaws.com>. It also used in distributed systems such as NASA SensorWeb (Cappelaere et al. 2009), which uses REST to support interoperability across Sensor Web systems that can be used for disaster management. Another example of using REST to achieve plug-and-play interoperability heterogeneous sensor and actuator Networks is described in (Stirbu 2008). Example of REST usage in Business Process Management (BPM) is described in (Kumaran et al. 2008), which focuses on different methods and tools to automate, manage and optimize business processes. REST has also been used for modeling and managing mobile commerce spaces (McFaddin et al. 2008). A final point that critics sometimes use against REST uniform interface is that what if an RPC operation does not fit within uniform interface (in this case HTTP methods). To summarize the answer is that methods PUT, DELETE, GET and POST are actually the resources gates rather than semantics themselves. Thus, that RPC operation becomes, for example, an XML message that is applied to a resource using a uniform method; hence, the XML message is the service operation. It is suffice to say that all XML SOAP messages (that describe RPCs in SOAP-based Web-services) are sent to a server port (URI) using the POST HTTP method. In this case, the SOAP message itself is the semantic that describes the RPC operation using POST method entry. However, overloading HTTP POST converts HTTP uniform interface into heterogeneous one, hence defeating the uniform interface property of the WWW. See (Richardson et al. 2007) for RESTful Web-services design guidelines and techniques.

To summarize, RESTful Web-services is a lightweight approach that hides internal software implementation by using universal standardized uniform interface and describing connectivity semantics in form of XML messages. The uniform-interface property serves as "resources" gates, hence simpler to design, reuse and control. Resources are viewed as uniform blocks that are viewed as black boxes with unique universal addresses (URIs) and can be integrated into the distributed environment seamlessly even at runtime. Describing semantics in XML messages leads to simpler connectivity standards and flexibility to future changes such as synchronization algorithms enhancement, standards updates and interoperability provision between different standards. Further, REST employs the "hypertext" concept, which means resources (services) can be creatively linked to each other (e.g. similar to browsing a Web site). This has the potential to empower simulation with more

intelligence when searching and locating new services, real-systems or data at runtime. Linking information together can benefit simulation. For example, LPs may be structured in a hierarchy tree (parent-child relationship); in this case, an LP may look up another LP address (URI) from the latter LP parent. Furthermore, RESTful distributed simulation is literally accessed by any device attached to the Internet, as advocated by the Web 2.0 mashup principles. This type of design is a recipe for a plug-and-play (or at least automatic/semi-automatic) distributed simulation interoperability. In contrast, exposing simulation functionality in RPCs (called services), which describe semantics in terms of programming procedure parameters and often reflect internal implementation, is clearly not the path toward plug-and-play interoperability. Further, services stubs are constructed at compile time, which deprive them of the ability to use new services at runtime (static-thinking type of design approach). Furthermore, services are not linked with each other. For example, a SOAP-based port does not return links to other service ports, making them isolated from each other.

Based on these ideas, we designed RESTful-CD++ (Figure 2) the first existing distributed simulation middleware based on REST. The RESTful-CD++ middleware design is discussed in *Section 3*. The middleware main purpose is to expose services as URIs. Therefore, RESTful-CD++ routes a received request to its appropriate destination resource and apply the required HTTP method on that resource. This makes the RESTful-CD++ independent of a simulation formalism or a simulation engine. CD++ (Wainer 2009), which is based on Discrete Event System Specification (DEVS) formalism (Zeigler et al. 2000), is selected to be the first simulation engine to be supported by the RESTful-CD++ middleware. The simulation manager, shown in Figure 2, manages the distributed CD++ (DCD++) simulation such as the geographic existence of model partitions, XML simulation messages and synchronization algorithms. DCD++ simulation manager design is discussed in *Section 4*. The simulation manager is seen externally as a URI (e.g. similar to web site URIs). Thus, it can be accessed similar to any other URI from any device attached to the Internet (e.g. Web-browsers, laptops, cell phones, etc.). On the other hand, is a component that manages a distributed simulation logical processor (LP) instance, in our case an LP is a CD++ simulation engine (Figure 2). Therefore, LPs exchange XML simulation messages among each other according to their wrapped URIs (in this case using the HTTP POST method as discussed in *Section 3*).

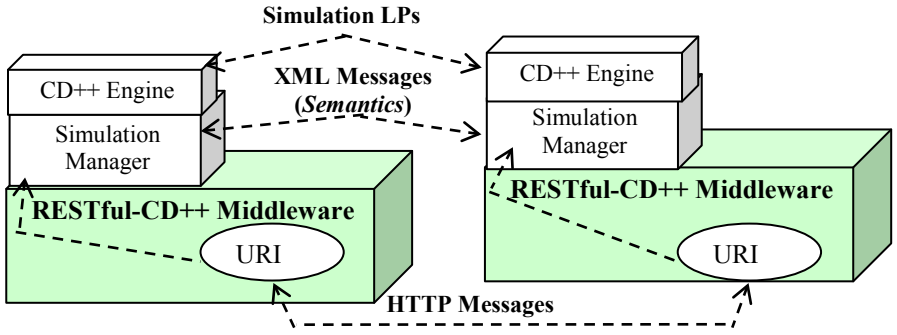


Figure 2: RESTful-CD++ Middleware Distributed Simulation Session

LPs cooperate among each other to simulate the same DEVS model structure in the same simulation run where each LP is responsible for simulating a segment of the overall model hierarchy. The model is partitioned among different LPs using an XML configuration document, hence allowing seamless modeling reconfiguration and repartitioning in the network. Therefore, the RESTful-CD++ exposes its APIs as a regular Web-site URIs that can be mashed up with other Web 2.0 applications to introduce, for example, real systems or data in the simulation loop. In addition, it is still capable of consuming services from SOAP-based Web Services. In this case, it needs to build the required RPC stubs for that service at compile time and then communicates with it using SOAP messages.

2 BACKGROUND

Discrete Event System Specification (DEVS) (Zeigler et al. 2000) is M&S specification aims to study discrete event systems. The model consists of components connected together through external port(s), where events are exchanged among models via those ports. The models being simulated changes state only at discrete points in time (upon the occurrence of an event). The P-DEVS formalism (Chow et al. 1994) expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. The basic building component of DEVS models is the atomic DEVS model. Coupled models contain one or more coupled/atomic models.

CD++ (Wainer 2009) is modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models. CD++ uses “simulators” to simulate atomic models and “coordinators” to simulate coupled models in a parent-child relationship. For each DEVS atomic model, users need to implement the various functions in C++ as required by the DEVS formalism. On

the other hand, for DEVS coupled models and Cell-DEVS models are defined using CD++ scripts. In addition, CD++ has been successfully providing distributed simulation using the SOAP-based WS (Wainer et al. 2008). Using Web-services have paved the way toward interfacing CD++ with other applications for providing a mash-up approach. For example, the DEVS community is in the progress of interfacing various DEVS-based simulation tools (e.g. CD++) using Web-service technology toward DEVS standardization interoperability protocol. On the other hand, we believe (as we show here) that REST resource-oriented Web-services can achieve mash-up and interoperability at the Web level with much simplicity, flexibility and scalability than the RPC-style approaches.

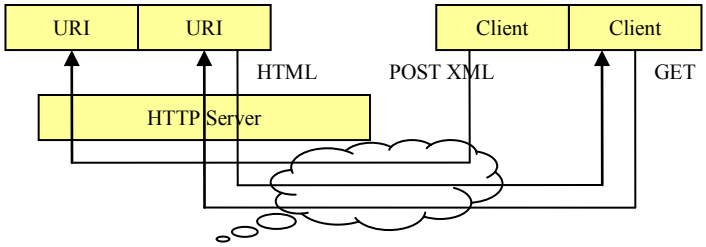


Figure 3: REST-based WS Programming Model

The Representational State Transfer (REST) provides interoperability (mash-up) by imitating the World Wide Web (WWW) style and principles (it was coined in chapter five of (Fielding R. T. 2000)). It exposes services as "resources" (which are named with unique URIs similar to Web sites) and manipulated with uniform interface (usually HTTP methods). For example (as shown in Figure 3), a client apply HTTP GET method to a resource's URI to retrieve that resource representation (e.g. as HTML). This is what happens when you browse a Web site. Further, a client can transfer data (e.g. as XML) by applying HTTP methods PUT or POST to a URI. Therefore, URI templates (Gregorio 2008) are the RESTful WS APIs whereas RPCs are the SOAP-based WS APIs. The Web Mash-up approach is appropriate for open communities where each party can evolve independently as soon as they follow the Web standards. Therefore, RESTful WS provide promising mash-up systems with simplicity and scalability. To name few of the REST principles: (1) Stateless (message-oriented): every request should have all of the necessary information to be processed. (2) Uniform interface (are usually HTTP methods). (3) Every "thing" is exposed as a "resource" (and named with URIs). (4) Representation (resources state): captures a resource data, which is transferable to other resources.

3 RESTFUL-CD++ MIDDLEWARE

The RESTful-CD++ is general middleware (shown in Figure 2) to expose services as URIs to the external world. This makes it independent of simulation formalism or a specific simulation engine. The RESTful-CD++ strictly follow HTTP universal standards.

The RESTful-CD++ is a URI-oriented architecture where service URIs are structured in hierarchal tree (parent-child relationship) similar to a regular Web site, as shown in Figure 4. Figure 4 shows the server URI template (APIs) that is used to construct every possible URI. URI Templates (Gregorio 2008) are URIs with variables (placed between braces '{}'). The variables are substituted with appropriate values to get the actual URI instances at runtime. URI Templates simplify both clients and server sides. Clients can easily know what part of the URI is under their control, hence enabling them to name their resources. This comes handy when, for example, clients want to name different simulation frameworks for different scenarios. Further, makes URIs (resources names) easier to remember by clients. URI Templates also benefit the server side, since it becomes easier to verify all possible paths that clients can use to manipulate exposed resources (since clients can only get to those resources through those URIs).

The root URI (/cdpp) is split into three subordinate resources (Figure 4): (1) URI (…/admin) branch (Line #1 of Figure 4) is used for administrative services such as create/delete/update accounts, general server configuration and retrieving server logs. (2) URI (…/util) branch (Line #1 of Figure 4) is used for utilities purposes that might be helpful for client programs. (3) URI (…/sim) branch (Lines 1-9 of Figure 4) is used to structure simulation resources. Simulation contains a number of workspaces where a specific workspace (i.e. resource {user workspace} to hold a workspace name) may contain a number of supported services (i.e. resource {servicetype} to define a service type such as DCD++). A simulation service (e.g. DCD++) may contain a number of simulation frameworks (i.e. resource {framework}). For example, simulation framework /cdpp/sim/workspaces/Bob/DCDpp/FireModel belongs to workspace "Bob", service "DCDpp" and "Fire-Model" is the name of the framework. The server APIs (Figure 4) can easily be extended to hold more service types such as other simulation engines in addition to DCD++.

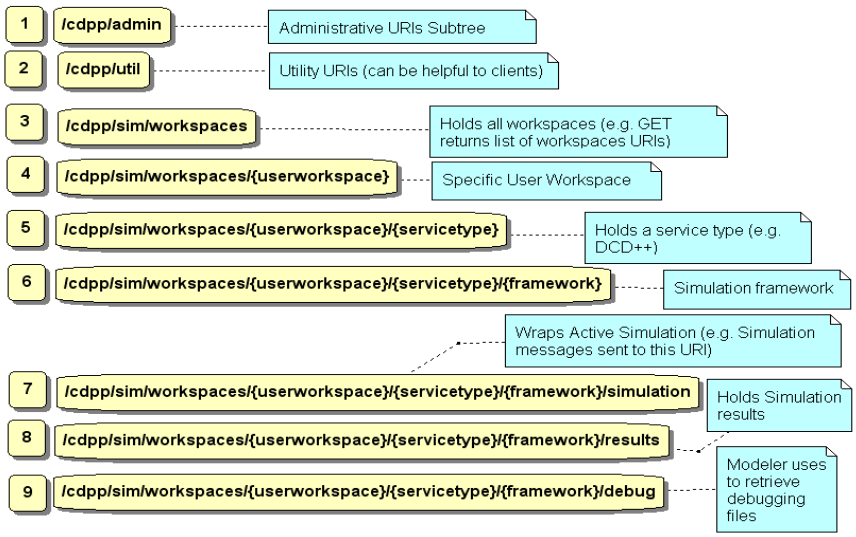


Figure 4: RESTful-CD++ Middleware URIs Template (APIs)

The server stores users' resources in database, which is divided into user sections where each section belongs to a user. This allows multiple requests from different users to modify the database without blocking each other, since each request lives in its own thread. Each user's section (as shown in Figure 5) contains an account object (i.e. username, password, etc.) and a workspace object. The workspace contains a list of simulation services (e.g. DCD++). Each service contains a list of simulation framework. The framework keeps track of the necessary information to find simulation results or active simulation managers (discussed in next section). Therefore, looking up a user data becomes straightforward from the URI variables, shown in Figure 4.

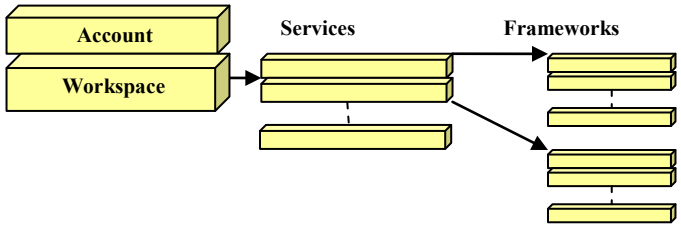


Figure 5: User Section in the Database

The RESTful resource-oriented approach has not only simplified URI APIs (from both client and server side) and database, but also routing and implementation. Each URI template in Figure 4 has a java class to handle its requests and to generate an HTTP response. There are three steps carried out upon receiving a request (as shown in Figure 6). *Step 1*: the Router checks if the URI matches one of the URI templates shown in Figure 4. If so, it creates a java thread and initializes it with the HTTP request along with an instance of the Java class resource that is associated with the subject URI. Note that the request thread owns the HTTP message and the Java class resource. However, the database is shared among all threads. *Step 2*: The proper operation (of the Java class resource) is invoked based on the HTTP method in the request. *Step 3*: The HTTP response is generated and the request thread is terminated. The server spawns a java thread for each request to reduce response time; hence, requests are handled simultaneously. This prevents short-time requests (e.g. checking simulation status) of being blocked by long-time requests (e.g. downloading large files). This design choice comes naturally since the server divides services among number of resources so it is better and more efficient to spawn a thread per request.

Resources (URIs) are manipulated via HTTP uniform interface methods (according to RFC 2616): GET, PUT, DELETE, and POST. GET is used to retrieve resources representations (states). The server returns the resource representation in the body of the HTTP response message. Therefore, users who can perform GET operation on a resource have read privileges from that resource. PUT creates a resource (URI) when it does not exist or updates the resource if it already exists. This enables clients to name their URIs with meaningful names. POST appends data to a resource (e.g. XML simulation message). DELETE removes a resource. These methods serve as the gates of a resource (as shown in Figure 7), providing uniform design pattern. Therefore, resources can be viewed as blocks, which construct the whole system structure (analogy with object-oriented design). On the other hand, semantics are described in form of messages that transfer resources states (representa-

tion) fully or partially among each other; hence, REST stands for Representational State Transfer. For example, XML simulation messages, discussed in next section, are example of such semantics. Having standardized uniform interface that everyone knows not only makes interoperability simpler on the client side, but easier and safer implementation on the server side. For example, we know GET does not change a resource, but PUT method does change a resource state. However, it can be tedious to identify read-only methods when having many different methods spread over the programming code.

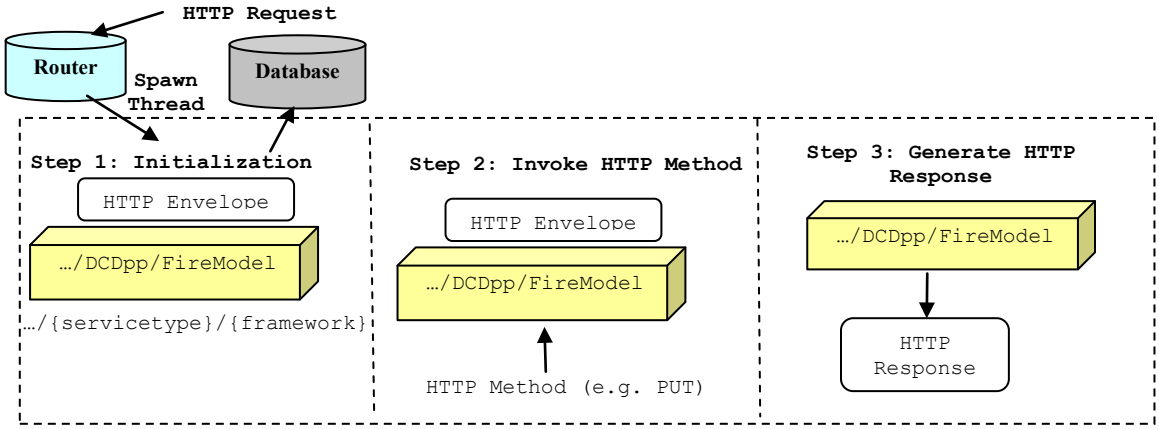


Figure 6: Processing Incoming Request Example

The request is *authenticated* (at Step #1 in Figure 6) according to the HTTP Basic authentication (Franks et al. 1999) to verify a user identity (i.e. username and password). Basic authentication is widely supported by Web browsers and Web programming languages (e.g. Java and JavaScript). Further, it does not influence performance because only one HTTP message is transmitted from client to server, instead of communicating back and forth to authenticate a message as in the case of Digest Access Authentication method (Franks et al. 1999). Once a message is validated, the server checks if the user is allowed to perform a certain task (*authorization* process). Thanks to REST uniform interface, this process is straightforward, as shown in Figure 7. All modifier methods must be performed by the owner of the resource (i.e. the one who created them). On the other hand, GET is performed by anyone by default, unless a resource is restricted by its owner.

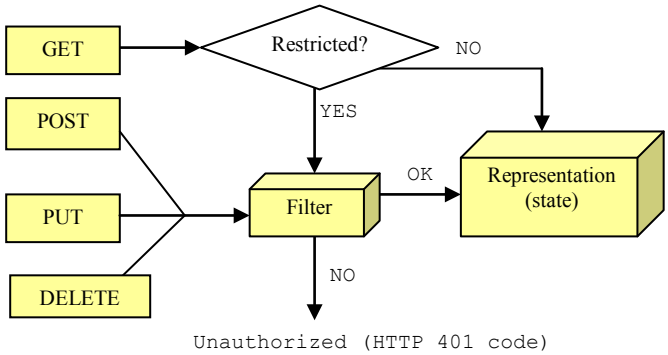


Figure 7: Resources Authorization Process

Each resource (Figure 4) has specification that defines the supported HTTP methods (and their responses), possible HTTP errors (e.g. code 401), incoming/outgoing representations media type. For example, the framework resource (Line #6 of Figure 4) supports the following four operations: (1) GET method. It returns XML or HTML document, describing the simulation framework configuration. (2) DELETE method to delete a framework. (3) PUT method. It creates/updates a simulation framework configuration. (4) POST method. It submits zipped or text model files used for CD++ simulation. Resource simulation (Line #7 of Figure 4) manipulates active simulation LP according to the following operations: (1) PUT method: It creates (starts) the simulation. (2) DELETE method: It aborts the simulation. (3) POST method: It sends messages to active simulation LP. (4) GET method: It is used to read values from simulation in progress. The active simulation resource is automatically deleted upon simulation normal completion. In addition, the {framework}/results resource (Line #8 of Figure 4) is created, enabling modelers to retrieve simulation results.

The RESTful-CD++ middleware server always responds to clients (even if a request causes a software failure in the server). In case of software failure, the RESTful-CD++ returns to the client (who sent the request) HTTP status 500 (Internal

Server Error) and keeps running. This increases system robustness since a user request cannot bring the server down preventing all users of using deployed services.

4 DISTRIBUTED CD++ (DCD++) SIMULATION

We plugged CD++ into the RESTful-CD++ middleware, enabling it to perform distributed simulation using RESTful WS (we call this extension DCD++). To do so, we needed to add the simulation manager component, shown in Figure 2, to manage distributed simulation semantics at runtime such as such as the geographic existence of model partitions, XML simulation messages and synchronization algorithms. In our design, each CD++ engine instance is a separate logical processor (LP) with its private simulation queue events. The overall model is partitioned among different LPs using an XML configuration document, hence allowing seamless modeling reconfiguration and repartitioning in the network. LPs synchronize simulation session between each other using XML simulation messages. These messages are wrapped in HTTP envelopes and sent to an LP's URI (see Line #7 of Figure 4) using HTTP POST method, as shown in Figure 8.

Suppose, for example, a modeler created simulation framework and submitted all necessary simulation files to its URI (see Line #6 of Figure 4). After that, the modeler can start a DCD++ simulation via creating resource "…/{framework}/simulation" (see Line #7 of Figure 4) on the main server (which is the RESTful-CD++ server that is the modeler interfacing with). As a result, the main server (acting as client) creates the necessary resources on supportive servers and starts the simulation everywhere. Figure 8 shows a DCD++ simulation example between two LPs. As shown in the figure, the modeler manipulates entire active simulation via main LP's URI that he created to start the simulation on the main server. The figure further shows the message path when an LP desires to send an event to a remote LP, as the following. (1) It passes the simulation event to its dedicated simulation manager through the operating system Inter Process Communication (IPC) queues. This is because LPs (CD++ engines) run as separate processes. (2) The simulation manager then figures out the subject remote LP URI based on the overall model partitions, packs the event as XML message, and sends it to the remote URI. At destination, the RESTful-CD++ server passes the message to the proper simulation manager, which in turn passes it to the LP via IPC queues. Note that all HTTP messages are authenticated via HTTP Basic scheme (Franks et al. 1999), hence each RESTful-CD++ server should have a user account on all other servers (similar to any other client).

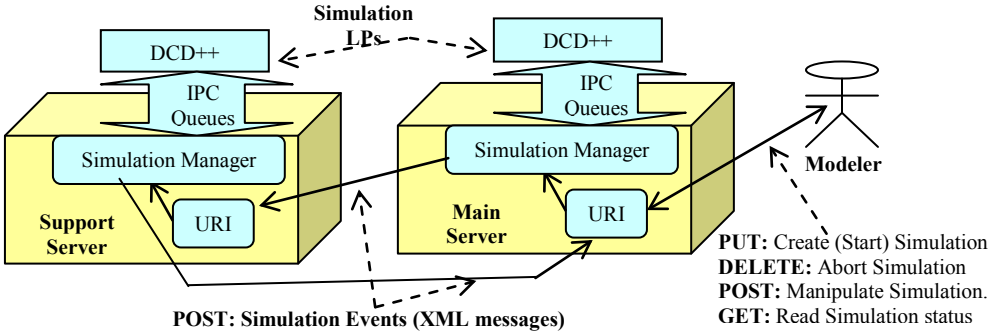


Figure 8: DCD++ Session between Two LPs Example

To improve performance, simulation managers spawn a thread for each transmitted message, allowing concurrent message transmission. This is because HTTP calls are synchronous; hence, the process is blocked until a response is received back. Figure 9 shows an example of two simulation managers. The top manager is sending two concurrent messages (each message is actually an HTTP client thread) where the bottom manager is receiving two messages concurrently (via the same URI). Therefore, receiving messages by a simulation manager must be thread-safe to avoid message contention.

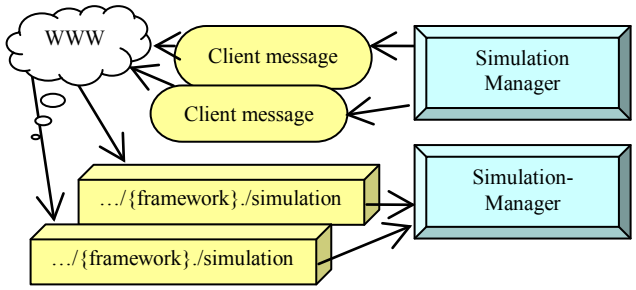


Figure 9: Sending/Receiving DCD++ Simulation Messages

To put what discussed so far in an example: A modeler can perform DCD++ simulation in four steps as shown in Figure 10. The modeler can create a simulation framework and submits all of the CD++ model files in Step #1 and #2 (see Line #6 of Figure 4). These two steps are only needed once. The modeler can then start the simulation in step #3 (see Line #7 of Figure 4) and retrieve simulation results in Step #4 (see Line #8 of Figure 4).

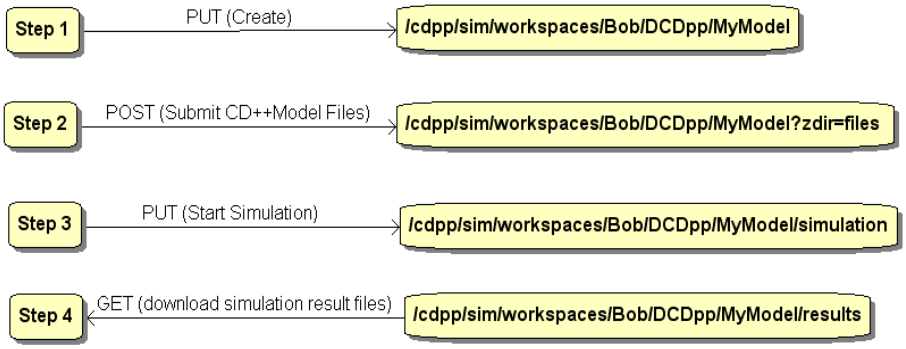
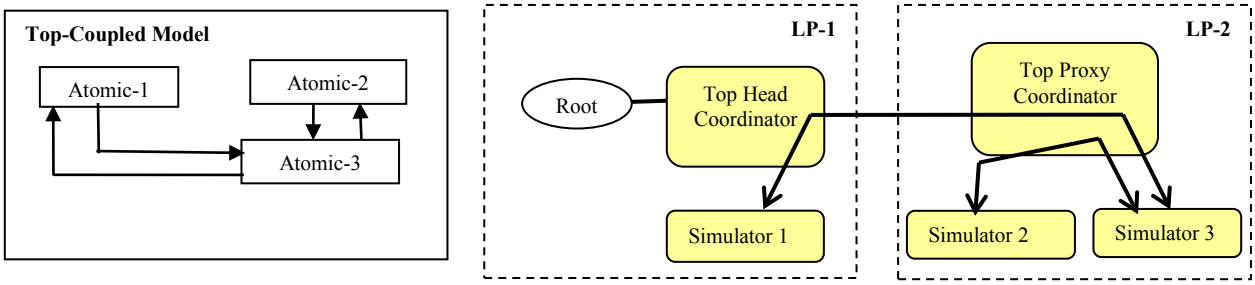


Figure 10: DCD++ Steps by a Client Example

Suppose, as shown in Figure 11-A, a DEVS coupled model that consists of three atomic models. An atomic model forms an indivisible block. A coupled model is a model that consists of one or more coupled/atomic models. The atomic Atomic-3 has two input and two output ports. Each of the input ports is connected to an output port of Atomic-1 and Atomic-2. Additionally, each of Atomic-3 output ports is connected to an input port of Atomic-1 and Atomic-2. Suppose that this model hierarchy is partitioned between two LPs where LP-1 contains Atomic-1 and LP-2 contains Atomic-2 and Atomic-3. DCD++ simulates this model hierarchy as shown in Figure 11-B. Simulator processor simulates an atomic model and Coordinator processor simulates a coupled model. DCD++ uses head/proxy structure to reduce the number of exchanged remote messages when coordinating a distributed coupled model simulation. In this case, for example, messages exchanged between Simulator #2 and Simulator #3 (in Figure 11-B) can be handled locally by the proxy coordinator (on behalf of head coordinator), hence converting a remote message into a local one. The head/proxy approach also group remote exchanged messages between models at different LPs to reduce the number of costly remote messages. For example, all messages from Simulator #1 heading Simulator #2 (within a simulation phase) are grouped in one XML messages (see Figure 12 example). This solution not only cuts the number of remote messages between two coordinators (in a simulation phase) to only one message, but also ensures the correct message order arrival to a coordinator, since simulation events (within the same phase) are transmitted simultaneously, as shown in Figure 9.



A: Coupled model consists of three atomic models

B: Model hierarchy during simulation split between two LPs

Figure 11: Head/Proxy Modeling Structure

DCD++ LPs follows the conservative algorithm approach, which always satisfies local causality constraint via ensuring safe timestamp-ordered processing of simulation events within each LP. The simulation is performed in phases where LPs are synchronized at beginning of each phase. Each LP (which is a CD++ engine) has its own unprocessed event queue and the simulation is cycling between phases. In this case, the Root Coordinator starts a phase by passing a simulation message to the topmost Coordinator in the hierarchy, as shown in Figure 11-B. This message is propagated downward in the hierarchy. In return, a DONE message is propagated upward in the hierarchy until it reaches the Root Coordinator. Each model processor uses this DONE message to insert the time of its next change (i.e. an output message to another model, or an internal event

message) before passing it to its parent coordinator. A coordinator always passes to its parent the least time change received from its children. Once the Root coordinator receives a DONE message, it advances the clock and starts a new phase safely without worrying about any lingering transit messages in the network. Further, each coordinator in the hierarchy knows which child will participate in the next simulation phase. Furthermore, each LP can safely process any event exchanged within a phase since an event is generated at the time it suppose to be executed by the receiver model. In this approach, the Root coordinator does not need to contact any of the LPs because they are already synchronized.

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

Figure 12: Grouping Simulation Messages Example

Finally, the main simulation manager starts a watchdog thread that periodically (2 minutes in our case) sends a message to check on the supportive simulation health (hence abort simulation, if a supportive fails). Supportive simulations also watch the main simulation manager health, enabling them to release system resources quickly such as processes, threads and any other operating system resources, if the main server fails.

5 PERFORMANCE ANALYSIS

We discuss here two test cases to study the RESTful-CD++ server (middleware) and DCD++ behavior when pressure increases. The idea of both cases is to increase the number of clients handled simultaneously. All results are the average of 50 different runs. The clients were run by one computer (i.e. each client is a separate thread) and are asked to send requests to the server at the same time. The "Near" clients sent their requests to the server from the same room through a local wireless network where the "Remote" clients use the Internet. The response time is measured (by clients) from the time a request is sent until the response is received back. First case studies the response time of the server when load increases, since the server handles each client request in a separate thread. Figure 13 shows the first test case response-time for concurrent remote and near clients. The results clearly show that the server holds its ground under pressure, since response time increases slowly. The results show that the more clients, the less of the jump of the response times.

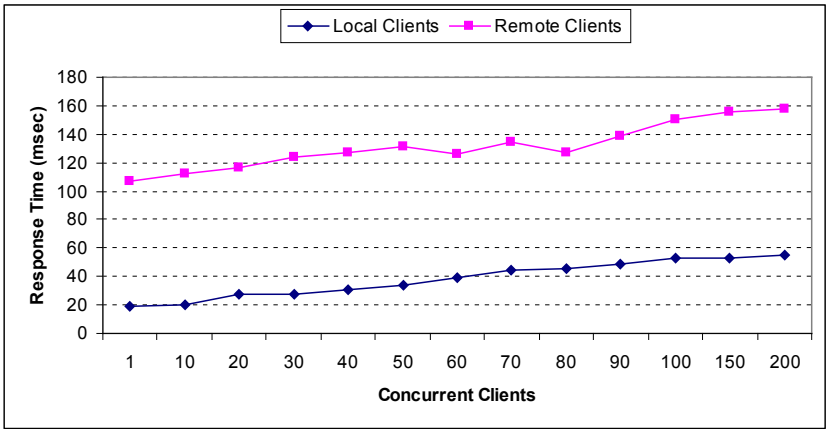


Figure 13: Server Concurrent Clients Response Time

The second case aims on studying the DCD++ simulation engines response time to clients under pressure. Messages are exchanged between simulation engines and RESTful server via IPC queues (i.e. they can be potential bottleneck when load increases), as shown in Figure 8. Note that simulation managers in the DCD++ grid do not actually wait for HTTP responses,

since those messages are sent in completely separate threads. Simulation managers only know the bad news when failure occurs during message transmission. We repeated the local-clients requests, but with requests (that across via IPC queues) to the DCD++ engines, as shown in Figure 14. The results show that the queues load start to show at 70 concurrent clients. On the other hand, the queues showed non-bottleneck and held their ground (at least up to 200 simultaneous clients), as shown in Figure 14.

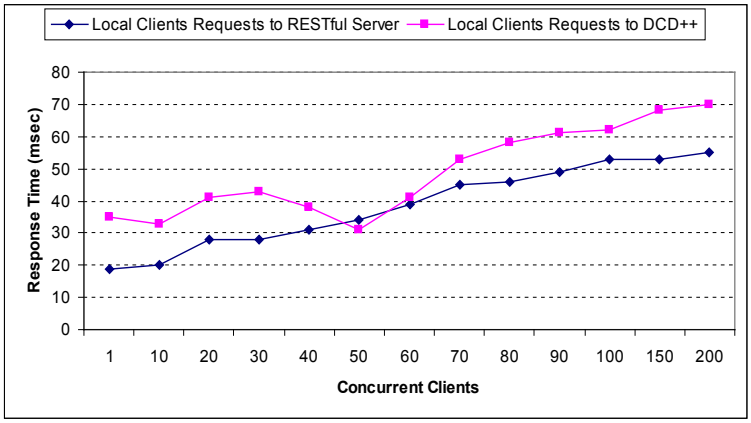


Figure 14: DCD++ Concurrent Clients Response Time

6 CONCLUSIONS

We presented here RESTful-CD++ middleware which is the first existing distributed simulation middleware based on REST WS, enabling heterogeneous independent developed simulation components to interoperate with much flexibility and simplicity. RESTful-CD++ exposes services as well-connected resources (URIs) and manipulated via HTTP methods (uniform interface). RESTful-CD++ is a service container, therefore it is independent of any simulation formalism or simulation engines. We plugged CD++ simulation engine into the RESTful-CD++ middleware to provide distributed simulation in a large scale using the Internet. In this case, simulation semantics is defined in form of XML messages. We showed that RESTful WS approach has the potential to advance distributed simulation state-of-the-art toward plug-and-play or automatic/semi-automatic interoperability because it hides internal implementation (via its uniform interface and describing connectivity semantics in form of messages). In contrast, other approaches expose functionalities using heterogeneous RPCs that often reflect internal implementation and describe semantics in form of programming parameters. This makes integrating independent-developed simulation components difficult and costly to do, since integration may spread into internal implementation. It is not easy to convince involved parties to do any internal implementation changes that could jeopardize their simulation integrity. Therefore, standardizing simulation semantics in form of messages like XML is the commonsense path to pursue. Further, those standardized XML messages can reach remote simulation using universal standardized entrances (REST uniform interface). Finally, REST approach is not a new hypothesis that we need to experience, but it has already been proven on the WWW. Further, it makes simulation accessible from any device attached to the Internet as advocated by Web 2.0 RESTful mashup concept. It is hard to imagine nowadays that any advanced device does not have access to the Internet. REST can bring these devices effortlessly into the simulation loop at runtime. On the other hand, other approaches need to compile RPC service stubs before being able to use them.

REFERENCES

Al-Zoubi, K., and G. Wainer. 2008. Interfacing and Coordination for a DEVS Simulation Protocol Standard. In *Proceedings of the 2008 ACM/IEEE Distributed Simulation and Real-Time Applications (DS-RT)*, 300-307. Vancouver, BC, Canada.

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

Cappelaere, P., S. Frye, D. Mandl. 2009. Flow-enablement of the NASA SensorWeb using RESTful (and secure) workflows. In *Proceedings of the 2009 IEEE Aerospace conference*. Big Sky, Montana, USA.

Chow A., and B. Zeigler. 1994. Parallel DEVS: a parallel, hierarchical, modular modeling formalism. In *Proceedings of the 1994 Winter Simulation Conference*, ed. J.D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, 716-722. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Fielding, R. T. 2000. Architectural Styles and the Design of Net-work-based Software Architectures. Ph.D. thesis, University of California, Irvine. Available via <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm> [Accessed October 2008].

Fujimoto, R. M. 2000. Parallel and distribution simulation systems. New York: John Wiley & Sons.

Franks, J., P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, P. Luotonen, and L. Stewart. 1999. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617. Available via <http://www.ietf.org/rfc/rfc2617.txt> [Accessed March 2009].

Gregorio, J. 2008. URI Templates. Available via <http://bitworking.org/projects/URI-Templates/> [Accessed March 2008].

Henning, M., and S. Vinoski. 1999. Advanced CORBA programming with C++. Reading, MA: Addison–Wesley.

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

Kumaran, S., L. Rong, P. Dhoolia, T. Heath, P. Nandi, and F. Pinel. 2008. A RESTful Architecture for Service-Oriented Business Process Execution. In *Proceedings of the IEEE 2008 International Conference on e-Business Engineering (ICEBE)*, 197-204. Xi'an, China.

McFaddin, S., D. Coffman, J.H. Han, H.K. Jang, J.H. Kim, J.K. Lee, M.C. Lee, Y.S. Moon, C. Narayanaswami, Y.S. Paik, J.W. Park, and D. Soroker. 2008. Modeling and Managing Mobile Commerce Spaces Using RESTful Data Services. In *Proceedings of the IEEE 2008 International Conference on Mobile Data Management (MDM)*, 81-89. Beijing, China.

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

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

Richardson, L., and S. Ruby. 2007. RESTful Web Services: 1st edition. Sebastopol, California: O'Reilly Media, Inc..

Stirbu, V. 2008. Towards a RESTful Plug and Play Experience in the Web of Things. In *Proceedings of the IEEE 2008 International Conference on Semantic Computing (ICSC)*, 512-517. Santa Clara, CA, USA.

Strassburger, S., T. Schulze, and R. Fujimoto. 2008. Future trends in distributed simulation and distributed virtual environments: results of a peer study, In *Proceedings of the 2008 Winter Simulation Conference*, eds. S. J. Mason, R. R. Hill, L. Mönch, O. Rose, T. Jefferson, J. W. Fowler, 777-785. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Wainer, G., R. Madhoun, and K. Al-Zoubi. 2008. Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web Services: Simulation Modelling Practice and Theory. 16(9):1266-1292.

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

Zeigler, B., T. Kim, and H. Praehofer. 2000. Theory of Modeling and Simulation: 2nd Edition. Academic Press.

AUTHOR BIOGRAPHIES

KHALDOON AL-ZOUBI is a PhD Candidate in Electrical Engineering within the Department of Systems and Computer Engineering in Carleton University, Ottawa, Canada. He is also a senior software analyst and programmer with over 12 years of industry experience occupying a number of seniority and leadership positions. His industry experience spreads over wide range of areas such as embedded software and mobility, air-traffic software management and telecommunications, and security software for explosive and narcotics detections. His email is <kazoubi@connect.carleton.ca>.

GABRIEL WAINER, SMSCS, SMIEEE, received the M.Sc. (1993) and Ph.D. degrees (1998, with highest honors) of the University of Buenos Aires, Argentina, and Université d'Aix-Marseille III, France. In July 2000, he joined the Department of Systems and Computer Engineering, Carleton University (Ottawa, ON, Canada), where he is now an Associate Professor. He has held positions at the Computer Science Department of the University of Buenos Aires, and visiting positions in numerous places, including the University of Arizona, LSIS (CNRS), University of Nice and INRIA Sophia-Antipolis (France). He is author of three books and over 190 research articles, edited four other books, and helped organizing over 90 conferences. He was PI of different research projects and recipient of various awards (NSERC, Precarn, Usenix, CFI, CONICET, ANPCYT, CANARIE, IBM Eclipse Innovation, SCS and others). He is the Special Issues Editor of the Transactions of the SCS, and Associate Editor of the International Journal of Simulation and Process Modeling. He was a member of the Board of Directors of the SCS, and a chair of the DEVS standardization study group (SISO). He is Director of the Ottawa Center of The McLeod Institute of Simulation Sciences and chair of the Ottawa M&SNet, and one of the investigators in Carleton University Centre for advanced Simulation and Visualization (V-Sim). His current research interests are related with modelling 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>.