2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation

Using REST Web-Services Architecture for Distributed Simulation

Khaldoon Al-Zoubi Gabriel Wainer
Department of Systems and Computer Engineering
Carleton University Centre for Visualization and Simulation (V-Sim)
Ottawa, ON K1S-5B6 Canada
kazoubi@connect.carleton.ca , gwainer@sce.carleton.ca

Abstract

In recent years, Web Services technologies have been successfully used for simplifying interoperability while providing scalability and flexibility in multiple applications, including distributed simulation software. The RESTful-CD++ simulation Server provides Web Services according to the REST principles by exposing services as URIs and consumed via HTTP messages. Therefore, the server becomes a service part of the Web that can be easily mashed-up with other applications and simulation software. In contrast, RPC-style SOAP-based Web Services use the Web as a transmission medium by exposing few URIs and many RPCs. RESTful-CD++ is (to our best knowledge) the only existing RESTful system in this area. Further, this distributed simulation package provides pioneering distributed simulation services using the Web architectural style. We present an overview of the principles, design and implementation of the RESTful-CD++ HTTP server and DCD++ simulation. We show that REST fulfills WS objectives with a much better and easier style than the SOAP-based systems.

1. Introduction

In recent years, Web Services (WS) technologies have been successfully used for simplifying interoperability while providing scalability and flexibility in multiple applications, including distributed simulation software. WS is to provide interoperability (i.e. interfacing remote heterogeneous applications), thus, these services can be combined and exposed in a bundle (called *mash-ups*). SOAP-based WS are provided as Remote Procedure calls (RPC) on top of the Web (HTTP). Instead, the Representational State Transfer (REST) style provides is an architectural style whose principles are easy to understand and design. REST was coined in chapter five of [3] to reveal the principles behind the Web architecture. REST WS are promising because of their lightweight and simplicity comparing to SOAP-based WS.

The Discrete Event System Specification (DEVS) [11] formalism has been extensively used to study discrete event systems. The CD++ [9] is modeling and simulation toolkit executes DEVS and Cell-DEVS models following the definition of the DEVS abstract simulator where it separates modeling from simulation. We have created a distributed simulator based on WS HTTP servers (called *RESTful-CD++)* strictly following the REST principles and style [3] (inheriting all of the Web architecture benefits).

RESTful-CD++ is a URI-oriented HTTP server that spreads exposed services over a number of resources. We designed the server URIs similar to regular Web sites, making services easy to understand and to use by clients. Each resource is manipulated via a few standardized HTTP methods. This resource-oriented approach takes the object-oriented style to the extreme since every object is only allowed to expose services via a few set of methods (making the server design and implementation extensible, scalable and modifiable).

The RESTful-CD++ server is literally a part of the Web great mash-up, hence simplifying interoperability with other Web applications. The server can consume/provide services from/to any application on the Web by interfacing with any application that can understand HTTP messages. Further, the server can still consume services from SOAP-based WS (in this case, the server acts like a SOAP client). Further, the RESTful-CD++ server can be deployed as standalone or as a Servlet within an HTTP container.

RESTful-CD++ acts as a container which can be extended to provide different services easily and quickly (hence it is straightforward to extend uniform resources structure). For example, the server currently provides the services of Distributed CD++. However, more services can be added without affecting other supported services (or even clients that already own resources on the server). This is similar to when a Web site changes some of its resources. Further, programming at the HTTP level revealed many performance issues (that are hidden when many RPCs spread over the code) such as transmitting simulation messages simultaneously in the distributed environment.

2. Background and Motivation

1087-4097/09 $25.00 © 2009 IEEE
DOI 10.1109/PADS.2009.16

IEEE computer society

Discrete Event System Specification (DEVS) [11] is M&S specification aims to study discrete event systems. The formalism expresses a model as a number of connected behavioral (atomic) and structural (coupled) components. These components are connected together through external ports, and events are exchanged among models via those ports. The models change their state only upon the occurrence of an event. The basic building component of DEVS models is the *atomic* DEVS model, formally defined as:

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

At any given time, the model is in some state $s \in S$, and it stays in this state for the period specified by the time advance function $ta(s)$. When the lifetime expires, the model activates the output function λ and can generate an output value $y \in Y$. It then changes its state as indicated by the internal transition function δ_{int}. The model changes its state as defined by the external transition function δ_{ext} if it receives one or more external events $x \in X$ before the expiration of $ta(s)$. The confluent transition function δ_{con} is used to resolve collisions of external events with internal transitions. A DEVS coupled model is formally defined as:

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

The model is composed by a number of components M_d interconnected. The external input coupling *EIC* specifies the connections between external and component inputs, while the external output coupling *EOC* describes the connections between component and external outputs. The connections between the components themselves are defined by the internal coupling *IC*.

CD++ [9] is a modeling and simulation toolkit that offers a varied simulation engines to execute DEVS and Cell-DEVS models on different platforms (e.g. parallel, distributed and real-time). For each DEVS atomic model, users need to implement the various DEVS functions in C++. On the other hand, for DEVS coupled models and Cell-DEVS models, users can specify the model in a model configuration file using a built-in specification language. CD++ has been successfully providing distributed simulation using the SOAP-based WS [9], which paved the way toward interfacing CD++ with other applications for providing a mashup approach (by combining services from multiple sources). This can lead to higher degree of reuse and reduced time to set up and run experiments, and making sharing among remote users more effective. For example, the DEVS community is in the progress of interfacing various DEVS-based simulation tools (e.g. CD++) using Web-service technology towards a DEVS standard interoperability protocol [1]. We believe (as showed here) that WS can achieve its objectives using the resource-oriented REST principles with more simplicity, flexibility and scalability than the RPC-style SOAP-based WS (as proven by the millions who use the World-Wide-Web – WWW - everyday).

The current generation of WS has been successfully used to perform distributed simulation [7] [8] [9]. It exchanges data with SOAP messages wrapped within HTTP messages (this is done by the SOAP engine [10] and the HTTP server layers). WSDL [2] is used to describe exposed services; enabling clients to build services as stubs to be compiled with their programs. On the other hand, SOAP applies RPC on top of HTTP. Once an RPC is invoked, the SOAP engine converts it into a SOAP message and wraps it as a HTTP request sent via the HTTP server. Exposed RPCs (usually called *services*) have heterogeneous interfaces, and programmers need to learn each procedure purpose along with each single parameter. WSDL only converts those RPCs into programming stubs, but does not teach programmers how to use them, making difficult composing and mashing-up services, and constraining reuse. The RPC-style is suitable for closed communities that can coordinate changes among each other to avoid breaking each other's software.

Instead, the WWW has been extremely successful in providing an outsized, scalable and interoperable mash-up system that is simple and easy to understand. What make the Web successful in doing so? (1) It is message-oriented and uses open standards. All changes are done to message contents (e.g. HTML, HTTP messages). Thus, if you follow message-format standards, you will be able to communicate with anyone on the Web. (2) Open Standard Protocols. For instance, HTTP is the Web protocol; hence, if you need to interface with anyone in the Web, follow the HTTP standards. (3) The Web employs a standardized global addressing scheme. For example, every "resource" in the Web is named with a unique URI. Resources are read (e.g. via a Web-browser) by sending HTTP request messages to that resource's URI (i.e. using the GET method). In response, the server returns an HTTP response message with the resource data to the client (e.g. as HTML, XML, etc.). Updating HTTP methods (e.g. POST) work the same way, but by transferring data from the client side to a resource on the server. REST [3] derives its principles from the Web. To name few of the REST principles: (1) it is stateless (message-oriented). Every request should have all of the necessary information to be processed. (2) It uses a uniform interface (usually HTTP methods [4]). (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.

REST principles are suitable for open communities that allowed RESTful applications to decouple clients from servers where each side can progress indepen-

dently from the other. The Web principles have been proven in simplifying interoperability while still providing scalability, flexibility and client simplicity. These principles have inspired us to provide WS at the HTTP layer (to be part of the Web mash-up) rather than using the Web as a transmission medium as in the case of the RPC-style SOAP-based WS. REST simplifies the client side, which is one of the strongest advantages of the Web.

REST has been successfully used by numerous vendors, including Yahoo (http://developer.yahoo.com/), Flicker (http://www.flickr.com/services/ap/), and Amazon S3 (http://s3.amazonaws.com). To the best of our knowledge, there are no REST-based simulators (or even distributed programming software in general). Most distributed simulation systems have been provided using SOAP-based WS and other approaches (see [9] for the complete list). This motivated us to build a REST-based simulator for CD++, providing distributed simulation at the Web layer, avoiding the heavyweight RPC-style mechanism.

3. RESTful-CD++ Server Design

RESTful-CD++ is a URI-oriented architecture where all services are divided into URIs and manipulated via HTTP uniform interface methods. Figure 1 shows the server URI template [6] used to construct every possible URI. URI Templates [6]are URIs with variables (placed between braces '{}'). Variables are substituted with appropriate values to get the actual URI instances at runtime, which simplifies both clients and servers. Clients can easily know what part of the URI is under their control. Servers can easily verify all the possible paths that clients can use to manipulate exposed resources. Each resource (Figure 1) includes a specification that defines the supported HTTP methods (and their responses), possible HTTP errors (e.g. code 401 for not-found resource), incoming/outgoing representations and media type. The root URI is split into three subordinate resources: (1) `/admin` is used for administrative services such as create/ delete/update accounts, general server configuration and retrieving server logs. For example, a PUT request to an absent URI *(/admin/accounts/Bob)* creates an account with name *Bob*. (2) `/util` is used for utilities that might be helpful for client programs. (3) `/sim` is used to structure simulation resources. It contains a number of *workspaces*, each of which may contain a number of supported services (i.e., the resource `{userwork-space}` holds a workspace name that can contain a `{servicetype}` to define a service type). A simulation service (e.g. DCD++) may contain a number of *frameworks*. Clients use frameworks to setup their simulation (i.e., sending configuration files). For example, */cdpp/sim/workspaces/Bob/DCDpp/FireModel* is a framework that belongs to workspace *Bob*, service *DCDpp* and uses the framework *FireModel*.

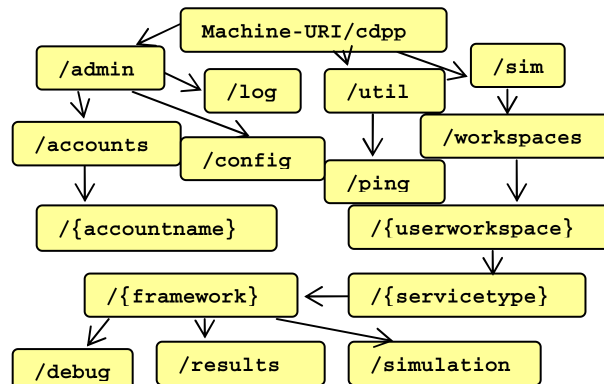

Figure 1: RESTful-CD++ URIs Template

The *framework* resource supports four methods: (1) GET, which returns an XML or HTML document describing the framework configuration. (2) DELETE, to delete a framework. (3) PUT, to create/update a framework. (4) POS, to submit files used for CD++ simulation. The resource `{framework}/simulation` manages active simulations according to the following operations: (1) PUT creates (starts) the simulation. (2) DELETE aborts the simulation. (3) POST sends messages to an active simulation (e.g., DCD++ servers use it to exchange XML simulation messages). (4) GET reads values from a simulation in progress. The active simulation resource is automatically deleted upon normal completion. In addition, `{framework}/results` is created, enabling to retrieve simulation results. The server spawns a Java thread for each HTTP request, which is terminated upon generating HTTP response to the client. This allows many requests to be handled simultaneously, hence improving response time. The server also requires all incoming requests to be validated according to the HTTP Basic authentication [5]. However, authentication is not required (by default) for GET requests.

4. RESTful Distributed CD++ Design

Modeler clients creates their simulation resources on the main server (which is a RESTful-CD++ HTTP server that the modeler owns a user account on). The modeler then needs (for the first time) to create a simulation framework on the main server and submit all necessary files to it. After that, the modeler can start a simulation via creating a *{framework}/simulation* resource. As a result, the main server (acting as client)

creates the necessary resources on support servers and starts the simulation everywhere.

Figure 2 shows example of a Distributed CD++ simulation session among three servers. The three DCD++ simulation engines (written in C++) are plugged into the RESTful-CD++ HTTP server (written in Java), which they coordinate among each other to simulate a CD++ model by exchanging XML messages within HTTP envelopes.

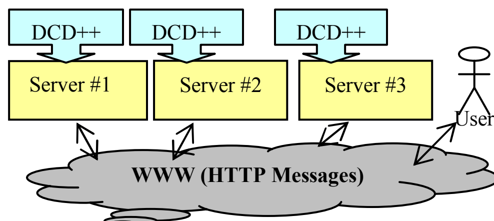

Figure 2: RESTful Distributed Simulation Example

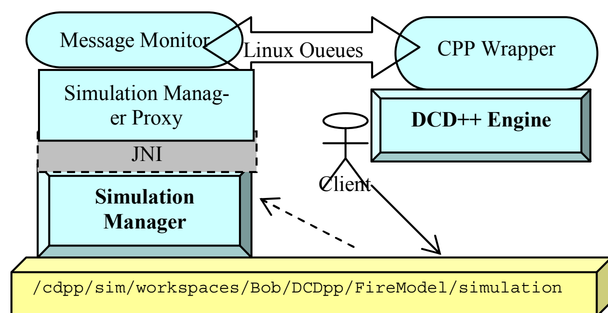

Figure 3: DCD++ Simulation Resource Example

Figure 3 shows a conceptual look of the main server during a single simulation session. The URI example in Figure 3 is the address to manipulate the active simulation. The *Simulation Manager* component manages a session for a DCD++ simulation engine. Upon receiving a DCD++ simulation message by the server Router: (1) a Java thread is spawned to handle the received message, which represents the target URI. (2) The XML message is parsed and validated. (3) The message is forwarded to the proper simulation manager. (4) An HTTP response is generated and the request thread is terminated. Simulation managers also act as clients when they transmit messages to remote simulation URIs in the DCD++ grid. To improve performance, they 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. The *simulation manager proxy* (Figure 3) interfaces with the C++ models in CD++. All messages between the DCD++ simulation engine and its simulation manager are exchanged using Linux queues.

The simulation manager on the main server creates the necessary resources on all support servers. All participants in the simulation conference must have a username and password on all other servers (similar to any other clients). Everybody in the conference is allowed to POST a message while the main server (the resources owner) is the only one allowed to perform other HTTP methods, taking the advantage of the uniform interface.

DCD++ uses *Coordinator* objects to simulate coupled models and *Simulator* objects to simulate atomic models. The simulation is divided into three phases: *Initialization*, *Collection* and *Transition*. Initialization starts when the topmost coupled model receives an Initialization (I) message. Collection starts when the *Root coordinator* sends a collect message (@) to the top model in the hierarchy. In this phase, all output (Y) and external (X) messages are collected. Transition starts when the Root coordinator sends an internal (*) message to the top model. In this phase, all collected external and scheduled internal messages are executed. Each phase ends with a Done (D) message, which contains the calculated next imminent event time.

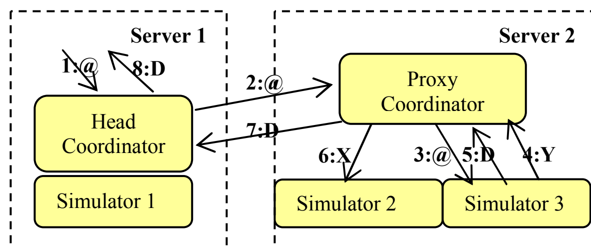

Figure 4: Head/Proxy Local message routing

A *Head/Proxy* algorithm (also used in our SOAP-based DCD++ [9]) places *proxy coordinators* on remote machines to handle children's messages on behalf of the *head coordinator*. For example, the Proxy Coordinator in Figure 4 routes the output message (4:Y) from Simulator 3 to Simulator 2 locally (6:X). Thus, the proxy makes local decisions (on behalf of the Head Coordinator) to avoid unnecessary remote message transmission, improving performance.

On the other hand, the algorithm still transmits all remote messages. For example, suppose that Simulator 3 in Figure 5 is transmitting a message for Simulator 1. The proxy sends it to the Head Coordinator as (5:X) followed by the *Done* message (8:D) to mark the end of the proxy collection phase. What if the Head receives the external message (5:X) after the *Done* message (8:D)? It will treat external (5:X) as part of the next collection phase, which leads to incorrect simulation. Further, assume the message (5:X) is sent (in reverse direction) from the Head to the proxy during the collection phase. In this case, the simulation will deadlock if the internal message (*) from the Head (which starts

the next transition phase) is received by the proxy before that external message. The proxy coordinator will be waiting forever for an imaginary *Done* message (as acknowledgment to the internal message) from its children.

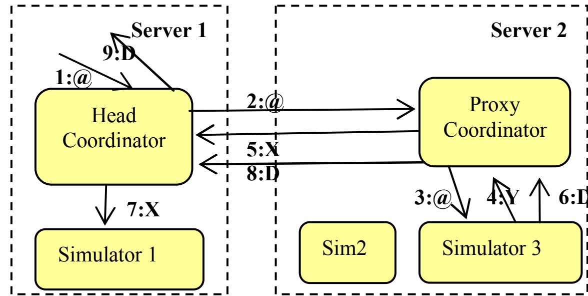

Figure 5: Head/Proxy Remote message routing

These scenarios are possible in RESTful-CD++ because all remote messages are transmitted concurrently, (i.e. each lives in a separate thread). We cannot guarantee that several messages transmitted in few milliseconds apart would reach their destination in their transmission order. The original Head/Proxy algorithm works in SOAP-based DCD++ [9], because messages are transmitted using RPC-style via the SOAP-engine. In this case, the SOAP engine converts the RPC into a SOAP sent in an HTTP envelope. Since HTTP requests are blocking, the simulation messages arrive in their transmission order. This is an example of how RPCs can deceive even experienced programmers by appearing innocent while they are still expensive blocking remote calls. Thus, a new Head/Proxy version was created.

Simulation messages can be categorized into *synchronization* messages (i.e. D, *, I and @), and *content* messages (Y, X). Synchronization messages are used to synchronize the start/end of simulation phases. Thus, they are exchanged at the boundary of simulation phases. For example, external message (5:X) must always arrive at the head coordinator before the *Done* message (8:D) (in Figure 5) to ensure causality and avoid deadlocks. The new head/proxy algorithm achieves this by sending all content messages with the first synchronized message heading to their destined processor. This solution not only ensures the correct message order arrival to a coordinator, but also reduces the number of remote messages between two coordinators (in a simulation phase) to only one message. Figure 6 shows an example of two simulation messages bundled in one XML document, showing the XML flexibility to overcome issues when compared to RPCs in the SOAP-based WS.

```
<Messages>
 <MessagesCount>2</MessagesCount>
 <Message>
   <MessageType>X</MessageType>
   <Time>08:50:00:00</Time>
   <SrcProcId>2</SrcProcId>
   <PortId>5</PortId>
   <Value>9.0</Value>
   <SenderModelId>3</SenderModelId>
   <DestProcId>1</DestProcId>
 </Message>
 <Message>
   <MessageType>D</MessageType>
   <Time>08:50:00:00</Time>
   <SrcProcId>2</SrcProcId>
   <NextChange>00:00:00:00</NextChange>
   <SenderModelId>3</SenderModelId>
   <Proxy>True</Proxy>
   <DestProcId>1</DestProcId>
 </Message>
</Messages>
```

Figure 6: Simulation Messages Example

The simulation message contains (at least) the following information: Message type, simulation time, source processor Id, destination port Id, content value, next change time, sender model Id, and destination Processor Id. DCD++ keeps unique IDs for each model, port and processor (i.e. coupled model coordinator or atomic model simulator) in the DCD++ grid. This is ensured by the way model files are parsed by each participant DCD++ simulation engine. Therefore, it is necessary to resubmit model files to each server, if modeler changes any of them.

5. RESTful-CD++: Implementation

The RESTful-CD++ HTTP server consists of five Java packages (shown in Figure 7): *Main*, *Data*, *Resources*, *Utility* and *Simulation Admin* (i.e. note that the DCD++ simulation engine shown in Figure 2 and Figure 3 implementation is not discussed here). The *Main* subsystem starts the server and initializes major components such as URIs Router, Database, logging, simulation managers' administration and communication. *Utility* provides helper classes for all other subsystems such as XML parsing utilities, server logging, file-system services and HTML builder documents. *Data* holds and organizes the server database. The *Database* is divided into sections for each user, hence, thread requests from different users do not need to block each other. Each user section is divided into user account and workspace. A workspace may contain any number of supported services (e.g. DCD++) where a service may contain any number of simulation frameworks.

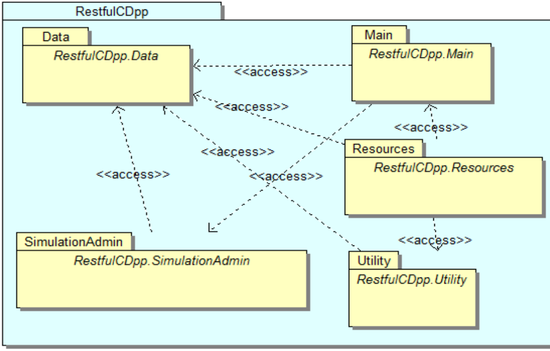

Figure 7: RESTful-CD++ Architecture Overview

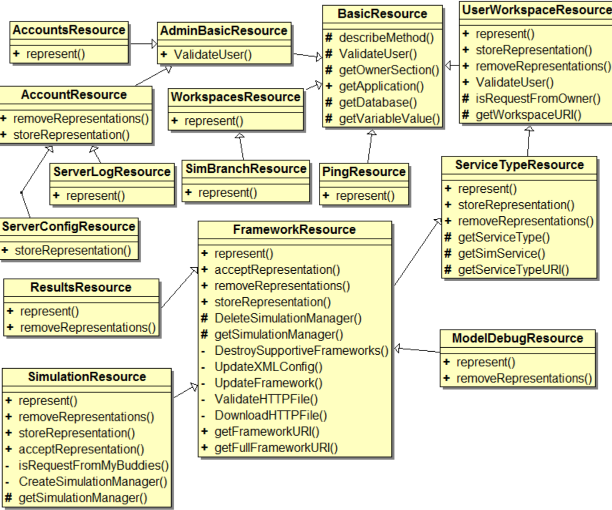

Figure 8: Resources-Subsystem Overview

Figure 8 shows the *Resources Subsystem*, which handles client HTTP requests. There is a Java class to process each URI in the template of Figure 2. For example, requests to *.../{framework}/simulation* are handled by *SimulationResource*. Resource classes parse the variable parts of a URI to figure out the appropriate stored Java objects in the database. For example, a request to *.../sim/workspaces/ Bob/DCDpp/FireModel* is handled by *FrameworkResource*, which can determine that the request belongs to the framework *FireModel* of service *DCDpp,* owned by workspace *Bob*. After that, the appropriate operation (based on the HTTP method in the request) is invoked. Each Java class in Figure 8 (at most) contains the following operations: *represent, storeRepresentation*, *accept Representation* and *removeRepresentations* (to handle HTTP methods GET, PUT, POST and DELETE respectively).

The *SimulationAdmin* classes (shown in Figure 9) manage active simulation sessions. The *SimulationManager* class performs all necessary operations to start, stop, control and monitor a simulation engine.

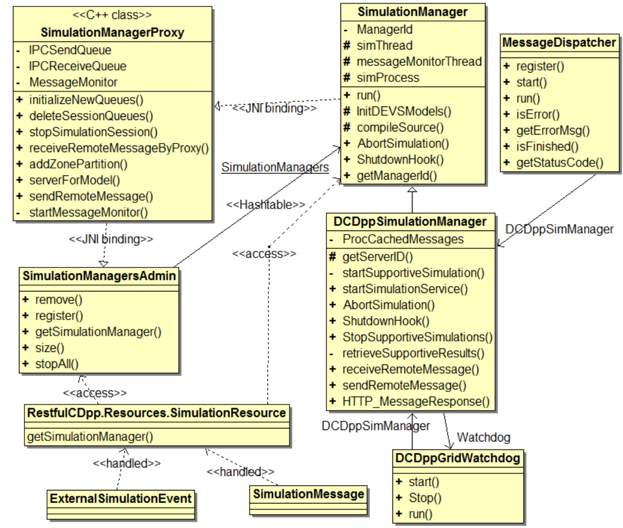

Figure 9: SimulationAdmin -Subsystem Overview

DCDppSimulationManager is extended from *SimulationManager* class to handle DCD++ simulation management in geographically distributed locations. *SimulationManagersAdmin* administrates all simulation managers in the server. The *SimulationManagerProxy* class operations are used by Simulation managers to manipulate their correspondent CD++ simulation engines (written in C++). *DCDppGridWatchdog* implements the watchdog thread, which periodically checks on the health of simulation resources in the DCD++ grid. *MessageDispatcher* is used (by DCD++ simulation managers) to send a simulation message (as a separate thread) to remote simulation resources (URIs) in the DCD++ grid. *MessageDispatcher* is actually an HTTP client with the purpose to send single message and reports the transmission status to the subject simulation manager.

6. Client Example

The DCD++ simulation can be performed with the following four steps (Step #1 and #2 are only needed once):

Step #1: Creating a DCD++ framework via applying the PUT method to the URI (for instance, `…/cdpp/sim/ workspaces/Bob/DCDpp/MyModel`). The server then creates the framework *MyModel*. Workspace *Bob* and service *DCDpp* are created, if they do not already exist. Further, the modeler needs to configure the DCD++ grid (upon creation or in a separate update request). For example, the following XML configuration document places two atomic models on two DCD++ servers:

```
<DCDpp>
 <Servers>
   <Server IP="10.0.40.8" PORT="8080">
     <MODEL>Producer</MODEL></Server>
```

```
    <Server IP="10.0.40.9" PORT="8282">
      <MODEL>Consumer</MODEL></Server>
  </Servers>
</DCDpp>
```

Step #2: Submitting the required CD++ model files via applying POST method to the framework URI. The easiest way is to submit all of the CD++ files in a zipped directory to URI …/`DCDpp/MyModel?zdir=files` (where *files* is the name of the directory when extracted).

Step #3: Start the simulation by applying PUT method to create URI …/`DCDpp/MyModel/simulation`. The client program can check the simulation status (via GET) to URI …/`MyModel?sim=status`.

Step #4: Once the simulation is completed, one can retrieve the results by applying the GET method to URI …/`MyModel/results`.

7. Performance

This section provides two sets of experiments. The first set aims on studying the REST system under pressure since many users are expected to use the system services simultaneously. The second set presents comparison results against the SOAP-based DCD++ [9] system.

Clients in the first set of experiments sent their requests simultaneously to the server from the same room as of the server. They also resend their requests to the server using the Internet across the city. The response time is measured, by clients, from the time a request is sent until the response is received back. Figure 10 shows the response time (averaged over 50 different runs) for both LAN and Internet clients. It also shows the difference between LAN and Internet response times, which indicates the same behavior regardless of the number of clients. This is because messages round-trip delay across the Internet is the major contribution in the difference, which is independent of our executed tests. The results also show that the server reacts to the number of requests at the beginning, but it holds its ground when the pressure increases (i.e. the jumps in response time get smaller when clients more than 100).

The second set of experiments compares distributed simulation using the presented REST-based WS DCD++ here against the SOAP-based WS DCD++ [9]. The next set of experiments used three CD++ models in the simulation: a Fire model (30x30 Cell-DEVS) used to simulate fire behavior in forests, a ship evacuation model (49x27 Cell-DEVS) used to simulate human behavior in case of ship evacuation and a Barbershop model (standard DEVS model) simulates barbershop customer service. The models were split evenly between two machines connected to the same Ethernet; hence remote simulation messages round trip time delay is measured to be around 4 ms.

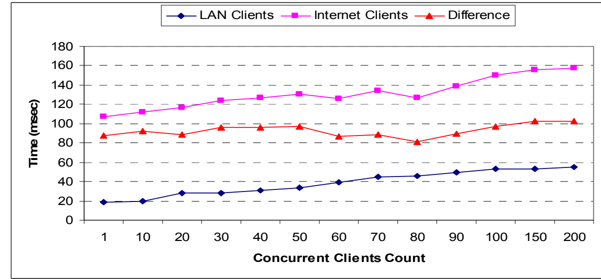

Figure 10: Concurrent Clients Response Time

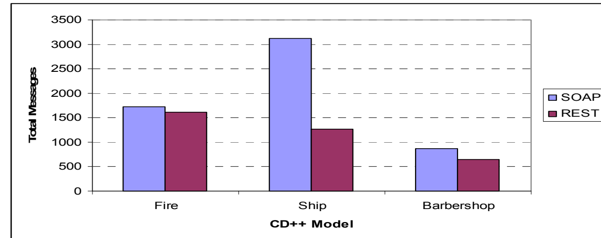

Figure 11: Total Exchanged Remote Simulation Messages

Figure 11 shows the total exchanged messages between the two machines to synchronize the distributed simulation. The shown difference is because the REST-system bundles remote messages in one XML message to reduce number of remote messages as previously discussed here. This improvement offered itself because of exchanging simulation messages in XML documents rather than parameters in a procedure. In this case, the degree of improvement depends on the studied model. On the other hand, models execution times (i.e. averaged over 25 different runs) still very close to each other on the Ethernet as shown in Figure 12. This is because using the Ethernet almost cancels the gain that is obtained from reducing the number of exchanged messages because of the trivial delay time of messages.

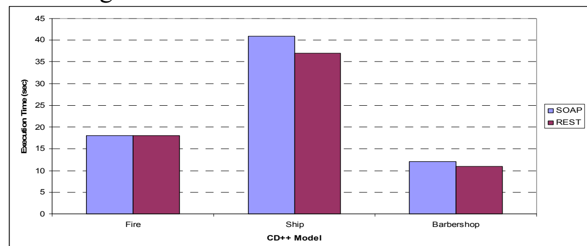

Figure 12: Models Execution Times over the Ethernet

The results in Figure 13 aim on studying concurrency on both systems, since these systems are usually used by different modelers at the same time. It starts by simulating one evacuation model, then two of them simultaneously, etc. Results (i.e. execution time aver-

aged over 25 different runs) are almost the same for both systems. However, the REST systems start showing slightly better performance after four concurrent models.

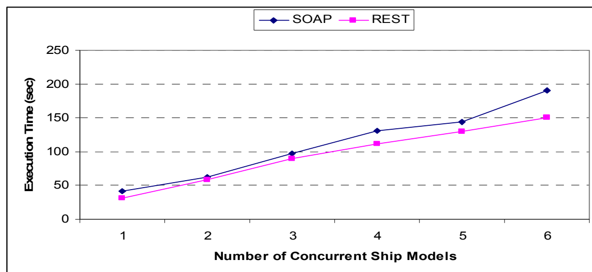

Figure 13: Execution Time of Concurrent Ship Models

The results presented here show that REST-style can be used to provide distributed simulation as in the case of SOAP-based WS. Further, the REST programming model is much easier to introduce techniques to improve performance. The comparison results here only use both systems to study software implementations, which can be influenced by many factors such as using third-party software and general software design. On the other hand, interoperability is the major benefit of WS and is usually put ahead of performance. We believe that interoperability is provided with much flexibility and simplicity by the REST-style comparing to the SOAP-based style as it has previously been shown here. Particularly the underlying communication in the both systems is the same. The SOAP-WS invokes RPCs by sending XML SOAP messages in HTTP messages whereas REST-WS directly works with HTTP messages themselves.

8. Conclusions

The RESTful-CD++ HTTP server provides WS using the Web principles (as defined by the REST-style). The server strictly follows the REST style and the Web principles in order to capitalize on the Web benefits such as scalability, interoperability and simplicity. In contrast, the SOAP-based WS use RPC-style by exposing few ports (i.e. single URI per port) and many operations (i.e. services). RPCs glues server and client programs together which prevent them from evolving independently.

The RESTful-CD++ server supports distributed CD++ (DCD++) simulation. DCD++ servers coordinate among each other via XML messages according to the head/proxy algorithm new version (which bundles remote messages together).

Being part of the Web mash-up, designing with the Web principles (e.g. resources with uniform interface and message-oriented open standards), the heavy-weight of the SOAP-WS, and having the Web global addressing scheme have inspired us to develop the RESTful-CD++ server. Furthermore, the server is still able to consume services from the SOAP-based WS.

9. References

[1] Al-Zoubi K.; Wainer, G. “Interfacing and Coordination for a DEVS Simulation Protocol Standard”. In *Proceedings of DS-RT 2008*. Vancouver, BC, Canada. October 2008.

[2] Christensen, E; Curbera, F.; Meredith, G.; Weerawarana, S.” Web Service Desctiption Language (WSDL) 1.1”. March, 2001. http://www.w3.org/TR/wsdl. [Accessed October 2008].

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

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

[5] Franks J.; Hallam-Baker P.; Hostetler J.; Lawrence S.; Leach P.; Luotonen P.; Stewart L. “HTTP Authentication: Basic and Digest Access Authentication” RFC 2617. Available at: http://www.ietf.org/rfc/rfc2617.txt. [Accessed: Feb. 2009]

[6] Gregorio J. URI Templates. http://bitworking.org/projects/ URI-Templates/ [Accessed: February 2009]

[7] Mittal S., Risco J. and Zeigler B. "DEVS-Based Simulation Web Services for Net-Centric T&E". Proceedings of SCSC. San Diego, CA. 2007.

[8] Möller B., Löf S.: “A Management Overview of the HLA Evolved Web Service API”. http://www.pitch.se/images/06f-siw-024.pdf . Accessed February 2009.

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

[10] Web Services-Axis. Available via http://ws.apache.org/axis/ [Accessed October 2008].

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