Distributed Simulation Using RESTful Interoperability Simulation Environment (RISE) Middleware

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. Distributed simulation practice outside the military sector is still limited. Having plug-and-play or automatic middleware interoperability is one of the main challenges is needed to advance distributed simulation, as indicated by several surveys; hence, interoperability must be achieved effortlessly with rational cost. They further indicate the need of having general pluggable container where lightweight commercial-off-the-shelf (COTS) simulation components can be plugged into the container with minimal development time. However, existing middleware solutions have been complex so far to overcome these distributed simulation issues. The RESTful Interoperability Simulation Environment (RISE) is the first existing middleware to be based on RESTful Web-services. RISE uses the Web plug-and-play interoperability style to overcome distributed simulation issues. Our focus here on plugging simulation components into RISE and on interoperating independent-developed simulation engines to perform the same distributed simulation session.

Keywords: Distributed Simulation, REST, Web-service, SOA, Interoperability, DEVS, CD++.

1 Introduction

Distributed simulation technologies were created to execute simulations on distributed computer systems (i.e., on multiple processors connected via communication networks) [15]. Distributed Simulation is a computer program that models real or imagined systems over time. On the other hand, distributed computer systems interconnect various computers (e.g. personal computers) across a communication network. Distributed simulation offers many benefits such as: (1) allowing across-organization simulation collaboration in order to participate in same simulation run without the need of physically being in the same location, hence enabling simulation assets reuse, (2) Allowing complex simulation incremental development. In this case, a complex model can be divided into smaller models so they can be developed and verified individually. Afterward, these smaller models can be integrated together to form the overall complex simulation model. Other benefits also include reducing execution time, interoperating different vendor simulation

toolkits, providing fault tolerance and information hiding – including the protection of intellectual property rights [6][15].

Distributed Simulation middleware is responsible of connecting and synchronizing several simulation components across geographical regions, allowing simulation assets reuse without being physically at the same location. Interoperating scattered simulation assets is the main challenge of a distributed simulation middleware. In practice, making independently developed applications interact with each other is not a trivial task, since this interaction involves not only passing remote messages, but also synchronizing them (i.e. interpreting messages and reacting to them correctly). Particularly, simulation packages can be based on different formalism, implemented independently by different teams, or support different synchronization algorithms. In general, modelers use the simulation tools that they are familiar with, and can be experts within a simulation tool environment, but unable to use others.

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) [21] 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 spite of HLA introduction in 1996. Other technologies such as CORBA and SOAP-based Web-services (WS) were used outside the military sectors to overcome HLA interoperability and scalability issues. However, existing distributed simulation middlewares still lack of plug-and-play interoperability, dynamicity, and composition scalability. Those approaches are described in the background section.

Lack of plug-and-play and dynamic interoperability to interface independent-developed simulation components, and the ability to reuse commercial-off-the-shelf (COTS) simulation components effortlessly are documented needed features in future distributed simulation middleware, as indicated by a number of surveys of experts from different simulation backgrounds such as [6][28]. Those surveys pointed out that having plug-and-play or automatic middleware interoperability is one of the main challenges to advance distributed simulation use in the industry; hence, interoperability must be achieved effortlessly with rational cost. They further indicate the need of having general pluggable container where lightweight commercial-off-the-shelf (COTS) simulation components can be plugged into the container with minimal development time. COTS concept reduces the cost of distributed simulation with the “Try-before-buy” mentality. This concludes that plug-and-play can mean two things. The first one is that any component in the overall system structure can be replaced with another one easily without affecting the entire system. The second one is that independent-developed simulation components can interoperate (synchronize) with each other for the same distributed simulation run. To achieve plug-and-play or Automated/semi-automatic interoperability between independent-developed simulation packages, not only semantics must be standardized but also flexible to adapt to future changes. Further, simulation functionalities should be self-contained components (black boxes) that: (1) Hide their internal software design and implementation, hence interact with other components with self-contained messages (e.g. XML messages) that are not tied to software implementation, uncomplicated to standardize and easy to adapt to future changes. Further, this point becomes more important since already existing simulation packages should be expected to have no

or minimal software implementation changes to comply with any new proposed standards, (2) Connect with other components via universal standardized interface (i.e. uniform connectors). In this case, components can be plugged into a complex structure easily, since they already know how they will be connected to other existing components in the structure, (3) Reached via unique universal standardized addressing scheme from anywhere, and (4) Support dynamic interoperability at runtime. Simulation components should be able to join/disjoin the overall structure without other components pre-knowledge. In other words, no new code or compilation should be required to achieve components interoperability. This point goes beyond simulation components to any device. In this case, real devices may be introduced into the simulation loop without stopping (and perhaps recompiling code) and restarting the current simulation-run in progress. We show here that RESTful Web-services interoperability contains the ingredients to advance distributed simulation on those fronts.

RESTful Web-services [26] imitates the Web interoperability style. The major RESTful Web-services (i.e. Web style) interoperability principles are universal accepted standards, resource-oriented, uniform channels, message-oriented, and implementation hiding. These principles are the Web interoperability characteristics; hence, REST is a reverse engineering of the Web interoperability style. Thus, REST has been in used in many products since the 1990's, but without its official name "REST". On the other hand, the Representational State Transfer (REST) is first used in [13] to describe the Web architecture principles. The name is derived because of the fact that on the Web a resource transfers its representation (state) in a form of a message to another resource. For example, a Web browser transfers a URI representation (e.g. as HTML document) using HTTP GET channel. REST exposes all services as resources with uniform connectors (channels) where messages are transferred between those resources through those uniform channels (i.e. called methods in HTTP standards). Because those characteristics conform to universally accepted standards, REST subsequently contains the recipe of plug-and-play and dynamic interoperability with infinite composition scalability. REST is a style, analogy with object-oriented, therefore, system designers must conform to those principles to obtain those benefits [26].

REST is usually implemented using HTTP, URIs, and usually XML because these are the main pillars of the Web today. In this case, resources (services) are named and addressed by URIs, resources connectors are HTTP channels, and connectivity semantics are usually described in XML messages. RESTful Web Services has been gaining increased attention with the advent of Web 2.0 [25] and the concept of mashup. Mashup applications deliver new functions and services on the Web by combining different information or capabilities from more than one existing source, allowing reusability and rapid development. Nowadays, RESTful Web-service is supported in conjunction with SOAP-based Web-services in tools developed by leading companies such as IBM [19] and Sun Microsystems (e.g. NetBeans IDE [24]).

Based on these ideas, we designed RESTful Interoperability Simulation Environment (RISE) middleware (formally called RESTful-CD++ [1][2][35]). RISE strictly follows the Web standards and interoperability style, hence, to avoid losing the main provided benefits such as plug-and-play and dynamic capabilities. Our main

motivations behind proposing such plug-and-play middleware with dynamic capabilities is to provide practical solutions for distributed simulation identified needed capabilities to overcome its limited use in the industry while maintaining rational cost. Having dynamic plug-and-play/automatic interoperability is recognized needed capabilities in a distributed simulation middleware [6][28][35].

Thanks to RESTful Web-services principles, RISE, which is the first existing RESTful WS middleware, is designed as a multipurpose online plug-and-play simulation interoperability middleware. First, the middleware provides a pluggable container to support different simulation components (e.g. CD++ [34]); hence, components become online Web services with minimal development time. Plugging commercial-off-the-shelf (COTS) simulation components quickly reduces cost and increases reusability. We plugged the distributed CD++ (DCD++) into RISE, allowing conservative-based distributed simulation between different CD++ instances. RISE-based DCD++ is described here along with its synchronization algorithms. Second, RISE forms the foundation for developing distributed simulation standards [3][30][31][32][33]. From our DEVS standardization [30][31][32][33] experience and the rationale behind CORBA declining [17], having practical standards indicates certain features that standards must be have such as simple to support, avoid software changes to legacy systems, allow legacy systems to use their existing resources (e.g. modeling methods), and allow different teams to evolve independently. The RISE-based standard is described here, aiming in interoperating independent-developed simulation packages.

In addition, RISE provides different functionalities that are not covered here such as making simulation assets part of workflows, Web 2.0 mashup, and Data fusion (DF). Workflows enable simulation experiments automation, repeatability and reusability, as described in [4]. Mashup concept groups various services from different providers and presents them as a bundle in order to provide single integrated service. IBM enterprise mashup solutions [19] [20] argue that integrating different RESTful plugging functions (called widgets) enable self-designed service Aggregation and information, rapid application development, unlock legacy systems via Web 2.0 [25] without major software upgrade. Thus, one of RISE objectives is to mashup applications/devices into simulation loop, allowing better-obtained results and analysis. DF is defined as collecting information from different sources to achieve inferences, which potentially leads to better accuracy from relaying on a single source of information. DF is applied by the military to build integrated images from various information sources in battlefields [27]. DF is similar to mashup in a sense of putting information into simulation loop. DF is highly dynamic, which makes it easier to achieve using RESTful WS plug-and-play interoperability.

2 Background on Distributed Simulation

At present, most works in distributed simulation are invested in optimizing simulation algorithms and in achieving efficient interoperability between different independent-developed simulation entities. These two areas define the current

challenges of distributed simulation and future trends [28]. For further thorough details, we discuss distributed simulation current state-of-the-art in [35].

Parallel/distributed simulations are typically composed of a number of sequential simulations where each is responsible of part of the entire model. Each of these subparts is a sequential simulation, which is usually referred to as a logical process (LP). The main purpose of synchronization algorithms is to produce the same results as if the simulation were preformed sequentially in a single processor. The second purpose is to optimize the simulation speed by executing the simulation as fast as possible. They fall in two categories: Conservative and optimistic. Conservative algorithms were introduced in late 1970s by Chandy-Misra [9] and Bryant [8]. This approach always satisfies local causality constraint via ensuring safe timestamp-ordered processing of simulation events within each LP. In current systems, the common implementation of conservative-based distributed simulation cycle to advance simulation time (e.g. [9][10][39]) is summarized as follows: (1) Time-coordinator requests minimum time from all LPs. (2) Time-coordinator calculates global minimum time, broadcasts it to all LPs, and waits for their replies. (3) Time-coordinator instructs all LPs to execute events with the minimum global time, waits for all LPs replies, and starts again with step #1. In optimistic algorithms, each LP maintains its Local Virtual time (LVT) and advances "optimistically" without explicit synchronization with other processors. On the other hand, a causality error is detected if a LP receives a message from another processor with a timestamp in the past (i.e. with a time-stamp less than the LVT); such messages are called straggler messages. To fix the detected error, the LP must rollback to the event before the straggler message timestamp; hence undo all performed computation. Time Warp algorithms focus on providing efficient rollback by reducing memory and communication overhead such as the mechanisms presented in [15].

Distributed simulation Middleware main objective is interfacing different simulation environments, allowing synchronization for the same simulation run across a distributed network. Those simulation entities are usually heterogeneous. For example, each simulation environment may differ from other entities in its simulation engine, algorithms, model representation, and formalism. This comes as no surprise that a number of surveys placed the middleware of distributed simulation as the most area of interest to overcome current distributed simulation challenges and to meet future expectation, as indicated by a number of surveys of experts of different simulation background [6][28].

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) [21] 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. Further, HLA could not make a breakthrough in the industry since its adoption in 1996 due to a number of issues such as its complexity, tied to programming languages and lack of interoperability in interfacing different Run-Time Infrastructure (RTI) vendors, since RTI-to-RTI interface is not standardized. RTI is the software layer that connects and synchronizes different HLA simulation entities (called federates) together where federates are interfaced with local RTIs via callback function interface (Figure 1). The HLA interoperability and scalability issues have caused the

consideration of using existing Service-oriented architectures (SOA) technologies in distributed simulation middleware, mainly CORBA [18], SOAP-based WS [12], and RESTful WS [26].

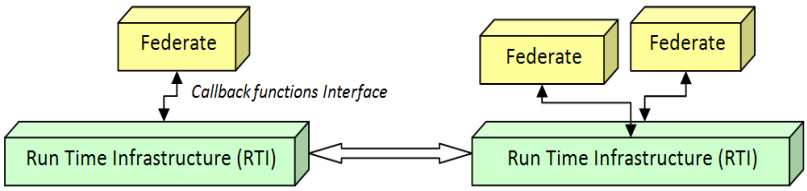

Figure 1: HLA Interaction Overview Model

WSDL and SOAP are the main elements enable SOAP-based Web-services (WS) interoperability. SOAP-based Web-services provides interoperability in a similar way as CORBA: WSDL corresponds to IDL role whereas SOAP corresponds to ORB data marshalling/serialization function. Further, Web-service ports addressed by URIs whereas CORBA objects addressed by references. Both ports and objects contain a collection of procedures (i.e. called services by WS) similar to a Java/C++ classes. Those procedures glue software components across the network, hence providing and RPC-style type of software interoperability, as shown in Figure 2. The server exposes a group of services via ports (Figure 2). Each service is actually an RPC where semantic are described via that procedure parameters. Client programmers need to construct service stubs with their software at compile time. Clients, at run time, consume a service by invoking its stub, which is in turn converted into XML SOAP message (to describe the RPC call), wrapped within HTTP message and sent to the server port, using the appropriate port URI. Once the message is received at the server side, the HTTP server passes the message into the SOAP layer (usually called SOAP engine like Apache AXIS [36]). SOAP engines are usually running inside HTTP servers as Java programs, called Servlets. The SOAP layer parses the SOAP message and converts it into an RPC call, applied to the appropriate procedure of the proper port. The server returns results into clients in the same way. Thus, the SOAP message role is to provide a common representation among all parties to the invoked procedure at runtime. In a distributed simulation environment, different components act as peers to each other. This means that each acts as client when it needs to send information while acts as a server via exposing different RPCs (i.e. services), as shown in Figure 2. Service providers need to publish their services, as XML WSDL documents. Clients programming stubs (Figure 2) are generated via compiling the WSDL document into programming stubs. Programmers then need to write the body of those stubs and compiling them with their software. [23][29] are examples of SOAP-based WS distributed simulation.

In reality, RPCs are heterogeneous interface, since they were invented by different programmers, and need to be written and compiled before being used. RPCs also expose internal implementation, leading to impractical and complex interoperability standards. It is almost impossible to interoperate independent-developed simulation systems via RPC-style without requesting major software implementation changes. This makes it impractical to support. Further, existing solutions lack of composition scalability, for example, programming stub is needed for every remote service.

However, in case of HLA the scalability is even worst, since the RTI is treated like a bus where all simulation entities use it for synchronizations. Furthermore, API complexity makes it difficult for distributed simulation to break outside expert programmers circle.

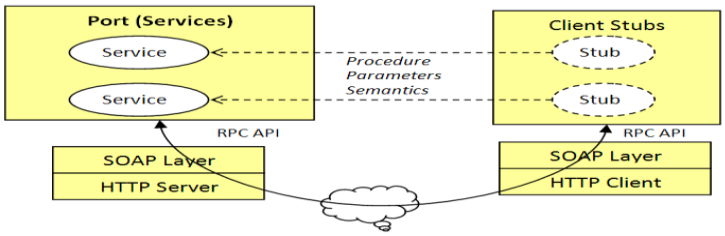

Figure 2: SOAP-based WS RPC-based Architecture Model

RESTful WS exposes all services as resources with uniform connectors (channels) where messages are transferred between those resources through those uniform channels. REST is usually implemented using HTTP, URIs, and usually XML because these are the main pillars of the Web today. In this case, resources (services) are named and addressed by URIs, resources connectors are HTTP channels (usually called methods), and connectivity semantics are usually described in XML messages (Figure 3). RESTful applications APIs are expressed as URI templates [16] that can be created at runtime. Variables in URI templates (written within braces {}) are assigned at runtime by clients before a request is sent to the server, enabling clients to name their services URIs at the server side. For example, username in template <…/users/ {username}> can be substituted with any string to get the actual URI instance (such as <…/users/user1> or <…/users/user2>). Further, URIs may include query variables to define the request scope by appending them to a URI after the question mark "?". For instance, request via GET channel to URI <http://www. google.com/search?q=DEVS> would instruct Google search engine to return information only about keyword "DEVS". RESTful services can be described formally using XML either using Application Description Language (WADL) [37] or WSDL 2.0 [22][38].

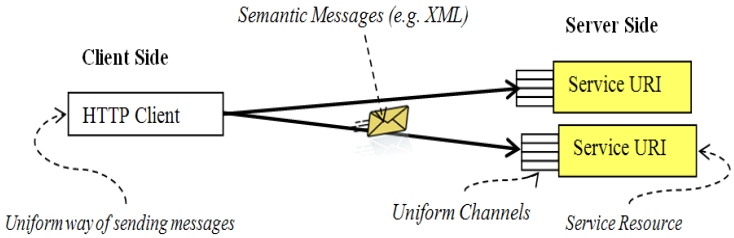

Figure 3: RESTful WS Architecture Model

From distributed simulation viewpoint, there are some differences between SOAP-based WS and RESTful WS as follows: (1) SOAP groups all services as procedures and expose them via a port (i.e. addressed by single URI) whereas REST exposes each service as a resource (i.e. addressed by single URI). (2) SOAP-based WS

communicates simulation information (i.e. semantics) in form of procedure parameters whereas REST defines them as XML messages. (3) SOAP-based WS transmits all SOAP messages (i.e. RPC description) using HTTP POST channel whereas REST uses all HTTP channels to transfer simulation semantics. (4) SOAP-based WS clients need to have a stub for each corresponding service while REST clients communicate in the same uniform way. (5) SOAP-based WS client stubs skeleton usually built via tools, but they still need to be written, integrated with existing software and compiled by programmers whereas REST does not usually require this process, hence follows a dynamic approach.

REST critics usually raise few issues such as REST only uses the four HTTP channels to transfer all messages so that those methods might not be enough for some applications: mainly, GET (to read), POST (to append new data), PUT (to create/update), and DELETE (to remove). This misleading comes from naming those virtual channels as "methods" in HTTP standards (RFC 2616 [14]), hence being confused with regular programming methods. Perhaps, it is ample to mention that SOAP-based WS transfers all SOAP messages using only the HTTP POST channel, thus, single method is enough in this case. Another issue is that REST heavily depends on HTTP, on other hand; SOAP-based WS can send SOAP messages using different protocol from HTTP like TCP/IP. This is because SOAP is a message describes an RPC via a network so that it can be sent using TCP socket. This is a misleading issue because: (1) HTTP is the Web protocol, thus sending SOAP messages using different protocol from HTTP makes it not Web service any more, hence complicates interoperability with other heterogeneous even further. (2) REST is message-oriented, thus, those messages are portable to different protocols like TCP/IP. For example, all simulation synchronization messages presented here portable to different protocol, similar to SOAP. However, a universal standard is part of REST principles and makes no sense to use different protocol from HTTP.

3 RISE Middleware API

Each experiment is wrapped up and manipulated via a set of URIs (i.e. an experiment API), hence allowing their online access from anywhere. Simulation experiment is various resources (URIs) hold all necessary information for simulation setup such as model scripts and model partitions where they are simulated in a single simulation run. These URIs are created and manipulated according to the middleware URI template (API), shown in Figure 4. The full RISE design and API described in [1][2]. The URI API template can be created at runtime. Variables (written within braces {}) in URI templates are assigned at runtime by clients before a request is sent to the server. The resource that best matches the request's URI will receive the request and it will become its responsibility to respond to the client.

Line #1 (Figure 4) shows a specific user workspace. This allows multiple users to use the middleware where each owns a single workspace (e.g. …/workspaces/Bob). Line #2 holds a specific service supported by RISE such as DCD++ (e.g. …/workspaces/Bob/DCDpp). In this case, for instance, other simulation components may be supported by RISE similar to adding new links to a Web site. Modelers

(clients) usually interact with a number of resources during the course of a simulation experiment, as shown in Lines 3-6: (1) the framework resource (Line #3) holds an experiment input data (such as the model source code, simulation input variables and sub-models interconnections). The POST channel is used to submit files to a framework. PUT is used to create a framework or update simulation configuration settings. DELETE is used to remove a framework. The GET channel is used to retrieve a framework state. (2) A simulation resource (Line #4) wraps an active simulation engine (e.g. CD++), which interacts with other remote simulation, if any. It is worth to note that in case of DCD++, this URI is the modeler single entry to a simulation experiment. However, it needs to communicates with other URIs (e.g. on different machines) to perform distributed simulation. This resource exchanges synchronized messages with other simulation entities (in case of distributed simulation) via the POST channel, and POST can be used by modelers to input variables in order to manipulate simulation at runtime dynamically. The PUT channel is used to create this resource, hence to start simulation. The DELETE channel is used to abort simulation and remove this resource. (3) The results resource (Line #5) holds the simulation output files (if the simulation was completed successfully). The GET channel is used to retrieve results where the DELETE channel is used to remove those results. The PUT and POST channels are disabled for this resource. (4) The debug resource (Line #6) holds model-debugging files. For example, a modeler can print debugging information inside his model source code to be retrieved later via this resource. The GET channel is used to retrieve model-debugging files where the DELETE channel is used to remove those files. PUT and POST channels are disabled for this resource.

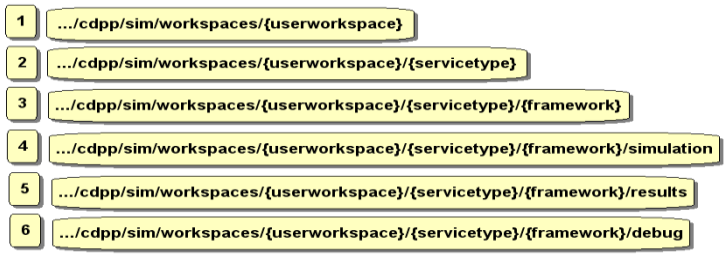

Figure 4: Simulation Experiment API in RISE

4 RISE-based Distributed CD++ Simulation Algorithms

This section discuss the distributed CD++ (DCD++) simulation session between different CD++ instances. At this point, a modeler should have already created an experiment URI (i.e. …/{framework}) where {framework} is the experiment name created by the modeler. Note that this URI is the parent for all other URIs that are created or deleted during the simulation process. This section is divided into two parts: the first part discusses the distributed simulation architecture while the second part discusses the simulation synchronization algorithms. The CD++ is plugged into RISE as shown in Figure 5 where each CD++ instance is reached via a URI and accessed via HTTP channels.

The purpose of the simulation manager (Figure 5) component is to manage a distributed CD++ (DCD++) simulation engine instance in the distributed simulation environment where various DCD++ instances participate to execute single simulation experiment. A simulation engine instance is usually called Logical processor (LP) in the distributed simulation environment, CD++ in our case. The simulation manager is able to synchronize a DCD++ instance with another remote DCD++, using the presented algorithms and semantics here. It is also capable to synchronize a DCD++ instance with none-CD++ simulation engine using standard protocols semantics, hence the ability of multiple semantics support. In DCD++, single DEVS or Cell-DEVS model is partitioned among those DCD++ engines where each instance simulates its partition.

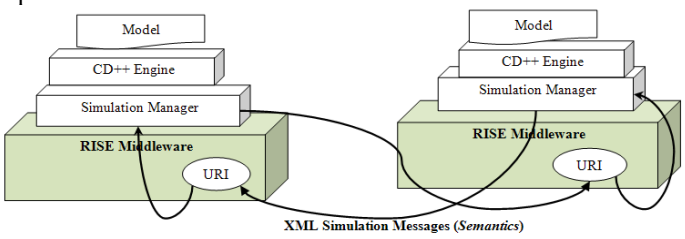

Figure 5: Distributed Simulation between two CD++ instances

DCD++ follows the conservative synchronization approach in which the casualty is strictly prohibited. On the other hand, it provides a number of improvement techniques comparing to other existing conservative-based simulation summarized as follows: (1) it avoids the required steps to loop all simulation entities to calculate the simulation global minimum time and then broadcasting it to all entities before an entity being able to proceed. This allows Root coordinator (which manages time) to start a new simulation phase without asking each logical processor (LP) its minimum time. (2) It aggregates remote simultaneous events together in single XML message, hence reducing the cost of several network messages to the cost of one message. (3) Provides modelers with experimental framework template where they can freely create as many as they like of different simulation scenarios. (4) It avoids unnecessary remote message transmission when it can be performed locally. (5) It avoids involving irrelevant models within current simulation phase (i.e. models that do not have events to execute or to send/receive at current time). This method can ignore huge part of the model partitions at certain simulation phases. (6) It uses simultaneous message transmissions to avoid blocking messages when a number of messages need to be sent to multiple remote LPs. (7) Exploiting thread-pool concepts to avoid creating a thread every time a message is sent. (8) Reusing TCP connections to transmit multiple HTTP messages to avoid establishing a connection with every message, which is very expensive.

4.1 Distributed CD++ (DCD++) Architecture

In the RESTful DCD++ grid various machines need to coordinate and exchange simulation messages (as HTTP messages) to carry out the simulation. Each physical machine in the grid needs to have at least one instance of the RISE middleware installed on it, since the DCD++ is plugged into it, as shown in Figure 6. DCD++ instances act as peers to each other. This means that when a simulation message is sent to an URI, the sender is an HTTP client, delivering an HTTP request using an HTTP channel where the receiver URI is a server, processing HTTP requests and responding with HTTP responses according to the HTTP standards.

Figure 6 shows an example of three DCD++ engines in distributed simulation conference where each DCD++ instance is plugged into RISE middleware. This conference represents an experiment during active simulation. The modeler manipulates and interacts with the simulation via the main DCD++ instance URIs, which resides on the main RISE middleware. The main RISE is the server that the modeler has on it a user account, selects it to setup experiments, and executes them. CD++ simulation engines are actually online simulation services that can be reached via URIs and accessed via HTTP channels. Thus, a main RISE in an experiment is not necessary the main middleware in another experiment. Further, the main server (e.g. machine #3 in Figure 6) sets up experiment resources on supportive servers on behalf of the modeler. In this case, the main RISE owns those resources; hence, it instructs supportive servers to hide all of its resources from external users. After all, those resources are URIs on the Web.

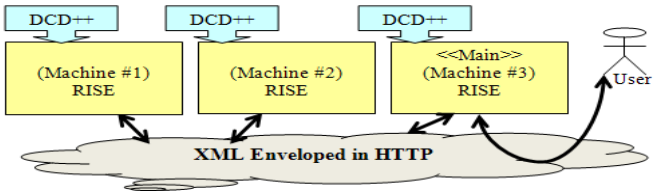

Figure 6: Conceptual View of a Distributed Simulation Session

Plugging components (e.g. DCD++) into the middleware provides a separation between provided services and the middleware. This clearly provides a number of advantages such as simulation components become independent of underlying technology, hence moving easily to another technology that might appear in the future, and applying the concept of pluggable container middleware where lightweight commercial-off-the-shelf (COTS) simulation components can be plugged into the middleware with minimal development time. COTS concept reduces the cost of distributed simulation with the business mentality of "Try-before-buy" attitude [6][28].

Each active DCD++ simulation component instance is wrapped by URI (…/{framework}/simulation), as shown in Figure 7. The modeler creates this URI via PUT channel on the main RISE server to start the simulation, which in turn starts the simulation on other supportive RISE servers. The request to start a simulation on RISE creates all necessary Inter-Process Communication (IPC) queues, simulation managers and the DCD++ simulation engines. During active simulation, as shown in Figure 7, simulation managers send messages to remote active-simulation URIs

(where it is then passed to the corresponding simulation manager). Simulation managers communicate with the actual CD++ simulation engines via operating system IPC queues, since CD++ runs as a separate process outside RISE middleware. It is worth to note that the modeler may use URI (…/{framework}/simulation) to manipulate simulation during runtime such as inserting an external event (i.e. simulation input variable) to change the course of the simulation. This is helpful during simulation training session when instructors like to change conditions during an exercise.

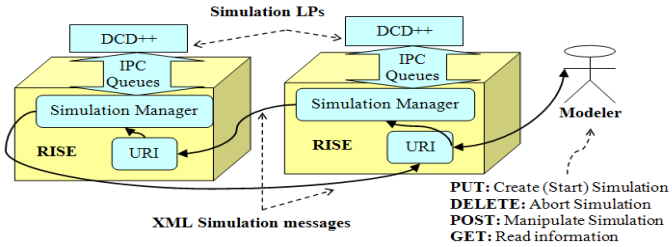

Figure 7: DCD++ Simulation Session between Two Machines

The DCD++ virtual network (Figure 7) is constructed and destructed based on the way a modeler partitions the model under simulation between different machines. Figure 8 shows example of DCD++ XML model partitioning information for both standard DEVS models (the top figure) and the Cell-DEVS model (the bottom figure). Model partitioning is a section of a larger XML configuration document for customizing the entire experiment options. The model-partitioning document describes each atomic model or cells zone location. Thus, the DCD++ virtual network shown in Figure 7 is reconstructed, if modeler redistributed models across different machines. Note that the DCD++ simulation session is actually performed among different URIs (within one or more RISE instances) coordinating among each other. However, those URIs are usually located on different physical machines. Note further that a RESTful-CD++ is identified via its port and IP address (Figure 8), relieving modelers of figuring out full URIs path every time a model is moved to another machine.

```
<DCDpp>
    <Servers>
      <Server IP="10.0.40.162" PORT="8080">
        <MODEL>reception</MODEL>
      </Server>
      <Server IP="10.0.40.175" PORT="8080">
        <MODEL>cuthair</MODEL>
        <MODEL>checkhair</MODEL>
      </Server>
    </Servers>
</DCDpp>
```

(A) Distributing Three Atomic DEVS Models Between Two Machines

```
<DCDpp>
    <Servers>
      <Server IP="10.0.40.175" PORT="8080">
      <ZONE>fire(0,0)..(14,29)</ZONE>
    </Server>
      <Server IP="10.0.40.162" PORT="8080">
      <ZONE>fire(15,0)..(29,29)</ZONE>
    </Server>
    </Servers>
</DCDpp>
```

(B) Spliting a Cell-DEVS Model Between Two Machines

Figure 8: XML Model Partitioning Example

The modeler (i.e. client GUI software) is expected to check on the active simulation status periodically. This is usually done via GET channel to URI

(…/{framework}?sim=status). In this case, RISE responds with an XML document similar to the following: <*Simulation*><*Status*>*RUNNING*</*Status*></*Simulation*>. The simulation goes into different states (from the modeler viewpoint), as shown in Figure 9: IDLE, INIT, RUNNING, ABORTED, ERROR, DONE and STOPPING. When a framework is created, the status is initialized with the IDLE state, which indicates that the simulation was never run on this framework. It moves into the INIT state upon receiving the request to start the simulation. The simulation goes into RUNNING state, if initialization was successful. The RUNNING state indicates that all simulation engines everywhere are up and running. In this state, the CD++ simulation engines can exchange simulation messages. Further, the modeler can manipulate simulation like inserting external events. Furthermore, dynamic online simulation results can be retrieved during this state. The simulation goes from RUNNING state to ERROR because of various possible errors such as failing to transmit a simulation message or a server failure in the grid. Further, the simulation goes into ABORTED state, if the modeler chooses to stop the simulation during the RUNNING state (via applying DELETE method to resource {framework}/simulation). In the normal completion, the simulation goes into STOPPING state. In this state, the main server collects simulation results from all supportive servers. The simulation goes into ERROR if it fails to stop properly such as failing to collect results from supportive servers or failing to stopping supportive simulations. Upon normal completion, the simulation status goes into DONE state, which means simulation results can now be retrieved from URI …/{framework}/results. Note that releasing system resources such as Linux queues, threads and processes occur in all exiting states: ABORTED, ERROR and STOPPING.

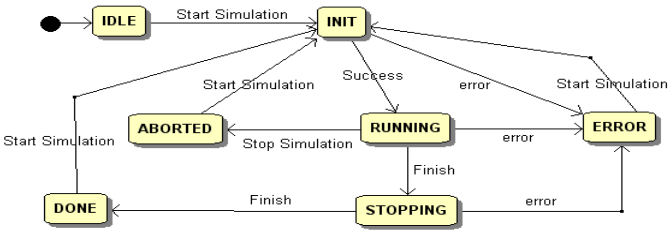

Figure 9: Simulation State Diagram

Simulation is automatically aborted (to ensure simulation accuracy) by a simulation manager, if, for any reason, it fails to transmit a simulation message to a remote simulation URI during a session. In this case, if the simulation manager is supportive, it aborts simulation and silently removes itself from the distributed simulation conference. On the other hand, if it is the main simulation manager, it also aborts simulation on all other supportive servers, since it is the actual owner of all simulation resources in the session. To make the matter worse, suppose a supportive server fails while the main server is waiting for a DONE simulation message from a process on that failed supportive server (simulation phases are discussed in next section). In this case, the Root coordinator, which drives the whole simulation, cannot advance the simulation to another phase because it is waiting for a DONE message from a dead simulation participant. This is a deadlock situation. To overcome this

possibility of deadlock, the main simulation manager starts a watchdog thread at the beginning of the simulation (and stops it at the end of the simulation) to keep watching all supportive simulation resources, as shown in Figure 10. The watchdog sends periodic (e.g. every two minutes) messages to every simulation URI checking if it is alive or dead. The main simulation manger only hears from the watchdog the bad news, which leads to aborting the simulation everywhere. Therefore, the session stays in deadlock at most for the watchdog period before the simulation is aborted. Supportive servers also need to watch the main server (Figure 10). This allows them to release system resources such as processes, threads, connections, and IPC queues.

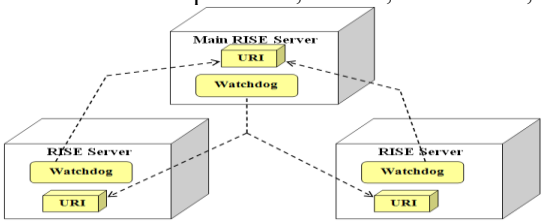

Figure 10: Watchdog Periodic Checking in a Simulation Session

HTTP messages are synchronous. This means that when an HTTP message is sent via TCP connection, the sender is blocked until a response is received. This argument also applies to the RPC-style SOAP-based web-services because SOAP messages (that describe RPCs) are enveloped in HTTP messages; hence, it is still an HTTP synchronies transmission. This fact often goes unnoticed by SOAP-based WS programmers. This is because SOAP engines handle SOAP messages at a different layer of the software stack. Further, SOAP engines are often used from a third-party provider through available open source like Apache AXIS [36]. HTTP synchronies transmission is obviously a performance concern, particularly when multiple messages need to be sent at the same time to different destinations. For this reason, simulation messages are transmitted concurrently where each message lives on its own thread. Figure 11 shows 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 (assuming via the same URI). Therefore, receiving messages by a simulation manager must be thread-safe to avoid message contention, since each request is handled by a separate thread. Further, in this case only the sender-message thread is blocked until the HTTP response is received back without blocking the entire application or other messages transmission. Note that all RISE threads are started from a thread pool, avoiding a new thread creation every time a thread is started.

Security is always a concern when communicating in cloud computing environment as in the case of RESTful-Web services. Other based RESTful Web-services such as Amazon Web-service (AWS) which requires developers to apply for an “Access Key ID” and a “Secret Access Key” [5]: The “Access Key ID” identifies the developer who is accessing AWS while the “Secret Access Key” is used to generate a keyed-Hash Message Authentication Code (HMAC), enabling AWS to authenticate the user. HMAC is calculated over service (i.e. URI), operation (e.g. user authorized to use POST channel or not to use), and timestamp (i.e. to prevent replay attacks). To prevent in-flight tampering, AWS recommends all requests should be

sent over HTTPS [5]. This scenario is portable for RISE. On the other hand, we chose to encode user name and password into a single string with base 64 encoding according to HTTP Basic Authentication method, defined in RFC 2617. This method does not add extra overhead, and it is supported by Web browsers and Web programming languages. Therefore, all simulation messages need to be authenticated according to this method. Note that the main server authorizes all simulation participants to use POST channel on all URIs, allowing them to pass simulation messages within the simulation conference.

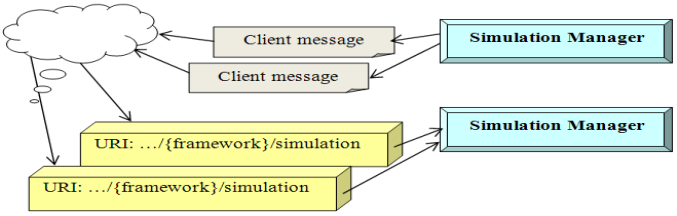

Figure 11: Concurrent Message Passing to/from Simulation Managers

4.2 DCD++ Simulation Synchronization Algorithms

DCD++ executes the model by passing messages among the different processors in the model hierarchy. Coordinators are the processors responsible for executing coupled models while Simulators are associated with atomic models and they are responsible for executing each of the DEVS functions defined by the model depending on the time and type of the received message. A Root coordinator is in charge of driving the simulation as a whole and interacting with the environment, since DCD++ is a conservative-based engine. Because DCD++ is a conservative-based engine, there is a special coordinator called Root coordinator which is responsible for the following: (1) Starting and stopping the simulation, (2) Connecting the simulator with the environment in terms of passing external events/output from/to the environment, and (3) Advancing the simulation clock. As shown in Figure 12, "coordinator" processors coordinate the simulation of one or more coupled/atomic models where "simulator" processors simulate atomic models. The processors are created and initialized at the beginning of the simulation in a hierarchy that matches the model hierarchy in terms of the parent-child relationship.

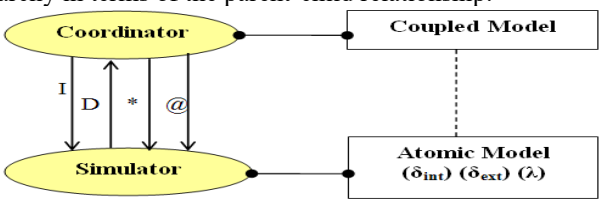

Figure 12: Message exchange during a simulation cycle

A number of simulation messages are used to synchronize simulation among processors hierarchy, shown in Figure 12. Simulation messages can be categorized as follows: (1) Content messages represent events generated by a model. Content

simulation messages include External messages (X) and Output messages (Y). Output messages (Y) are usually converted to external messages for their destinations. (2) Synchronization messages cause the simulation to move into another simulation phase (those phases discussed next). Synchronization messages include Initialize message (I), Internal message (*), Collect message (@), and Done message (D). Initialize message (I) starts the initialization phase. Internal message (*) starts the transition simulation phase. The top model Coordinator propagates it downward in the hierarchy. Collect message (@) starts the collection phase. Done message (D) marks a simulation phase end. It is also used by Coordinators to identify which children needs to be simulated at the next phase. It further used to calculate the global minimum simulation time.

The simulation phases for the entire simulation are driven by the Root coordinator (which is the parent of the highest model's coordinator). They are divided into three phases (shown in Figure 13): (1) Initialization phase initializes all models in the hierarchy; hence, it eventually executes every initialization method of every atomic model. In response, a DONE message propagates upward in the model hierarchy where each Coordinator calculates the minimum next change of its children and passes it in a DONE message to its parent. Eventually, the Root receives DONE message with smallest time, which updates the simulation clock and starts the Collection phase. (2) In the Collection phase, some of the output messages are collected to ensure their execution at the same time with internal events. (3) In the Transition phase, all the collected external messages are executed along with simultaneous internal events. The Root coordinator handling of a DONE message arrival is described in Figure 14.

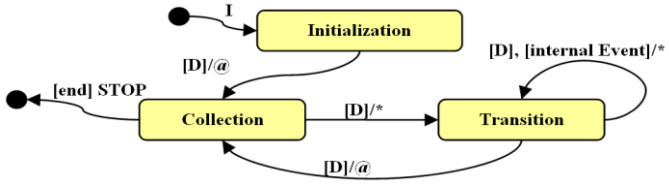

Figure 13: Root Coordinator Simulation Phases State Diagram

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

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

Figure 14: Root Coordinator Handling DONE Message Algorithm

The head/proxy is originally intended to solve redundant number of messages from remote CD++ processors back to their parent coordinator. The main motivation

behind Head/Proxy algorithm is that network messages in distributed environment are expensive and have direct affect on performance. For example, assume the coordinator in Figure 15 is coordinating three simulators where two of its children (simulator #2 and #3) are residing on a remote machine. Figure 15 shows a fragment of the collection phase messages when the coordinator receives a collect (@) message from its parent. As shown in Figure 15, Simulator 3 sends an output message to the parent coordinator to translate it to external message for Simulator 2. Obviously, the transmission of those two messages (in Figure 15) could have been avoided if another coordinator (we call proxy) was placed in server 2 so that converting the output message (from simulator 3) to an external message (to simulator 2) is done locally, as shown in Figure 16.

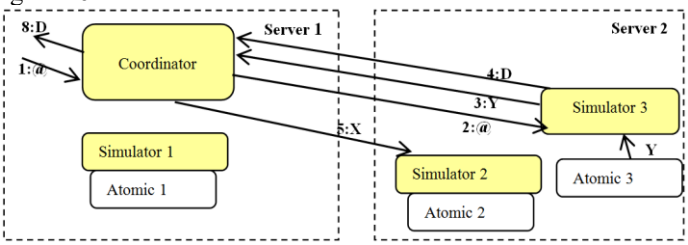

Figure 15: Unnecessary remote messages in distributed simulation

The idea of the head/proxy depends on using two kinds of coordinators for each coupled DEVS/Cell-DEVS model: (1) Head Coordinator: is responsible for synchronizing the model execution, interacting with upper level coordinators and message routing among the local and remote model components. (2) Proxy Coordinator: is responsible for message routing among the local model components dispensing with the need to send remote messages if the head coordinator is residing on a different machine than that used to run the sending and receiving processors. The advantage of using proxy coordinators (as shown Figure 16) is that converting all remote messages between local processors to local messages. The proxy coordinator forwards one DONE message to the head coordinator once it receives all DONE messages from its children. Note that in this collection phase (Figure 16) simulator #2 does not forward the external message to the Atomic #2 model. In this phase, simulator #2 inserts the external message in its bag, waiting for the next internal (*), which starts the next transition phase. This allows simulator #2 to execute any scheduled internal events along with the already collected external message simultaneously.

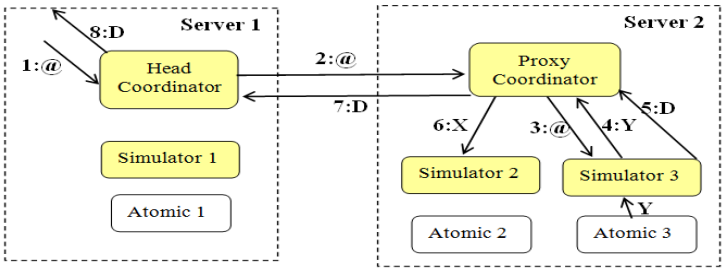

Figure 16: Proxy Advantage of Preventing Unnecessary remote messages

Proxy coordinators avoid remote message transmission when it is possible to route them locally, but still forward all none-local messages to the head coordinator. For example, suppose the output message from simulator #3 is transmitted to simulator #1 (instead of simulator #2), as shown in Figure 17. In this case, the proxy coordinator has no choice but to transmit the external message to the head coordinator remotely. The external message ends up queued at simulator #1, waiting to be executed in the next transition phase upon receiving internal (*) message from head coordinator. In fact, simulation events that are exchanged in the same simulation phase are simultaneous events; hence, they need to be executed in the same virtual time. To clarify this point, consider how Root coordinator advances simulation Time and phases, as shown Figure 18, which is a depiction of the model hierarchy partitions shown in Figure 17. Assume that Simulators #2 and #3 outputs a job to simulator #1 every two seconds where Simulator #1 takes one second to process each regardless of the number of jobs are being process. In this simple example, shown in Figure 18-A: (1) as part of simulation initialization, I message is sent to the Head coordinator, which passes it to Simulator #1 and Proxy coordinator. Consequently, the Proxy reroutes message I to Simulator #2 and #3. Simulator #2 and #3 reply with D messages with a scheduled change in two seconds from now. (2) Root advances time to (t2) and starts Collection phase by sending @ message to Head coordinator, which only sends it to the proxy. This message is not send Simulator #1 because it did not schedule a change in previous phase, hence becomes irrelevant in this phase. The Proxy passes @ message to Simulator #2 and #3, which cause them to send two jobs (i.e. Y message) to Simulator #1 (via Head and Proxy coordinators). Simulator #1 receives these Y messages as external messages (X) where it holds them to be executed in the next phase. (3) Root starts transition phase causing Simulator #1 to schedule a change at one second from now (when it will execute the two received jobs). In addition, Simulators #2 and #3 schedule a change at two seconds from now (when they will produce their next jobs).

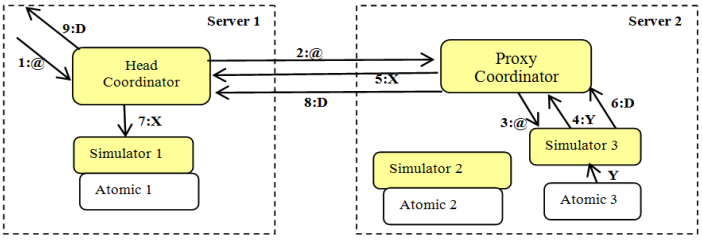

Figure 17: Head/Proxy Remote messages Transmission

Figure 18 shows two types of messages: Remote and local. All exchanged messages between the Head and Proxy coordinator are remote messages (shown in red); hence, they are usually measured in range of milliseconds to seconds. On the other hand, all other messages are local (shown in blue); hence, they are measured in few microseconds in DCD++, since a processor simply sends a message by inserting it in the unprocessed events queue. To reduce the communication high cost, the DCD++ groups simultaneous events heading to the same destination in one message. For example, as shown in Figure 18-B, the proxy sends two Y messages and D

messages for the cost of one message. This shows huge improvement in performance, particularly, for models with intensive communication overhead. Grouping remote simultaneous events make sense for obvious performance reasons, but also avoid inaccurate simulation results or deadlock in the simulation. This is because P-DEVS messages, as previously mentioned, belong into two categories: (1) Content messages (i.e. Y and X) represent DEVS models communication. These messages must be exchanged within a simulation phase. (2) Synchronization Messages (I, @, * and D) synchronize the start or an end of simulation phase; hence, they mark simulation phases boundaries. Therefore, Content messages must arrive at destination within the correct simulation phase to be executed at the correct virtual time. Further, synchronization messages must arrive at the start or end of the correct simulation phase to ensure correct simulation and to avoid deadlock, since a Coordinator may hang forever waiting for a synchronization message to be able to start a new phase or end the current phase. Of course, we can never guarantee message arrival at destination in the same order of their transmission order. On the other hand, DCD++ guarantees the correct message-order arrival by grouping messages in one XML document, as shown Figure 19.

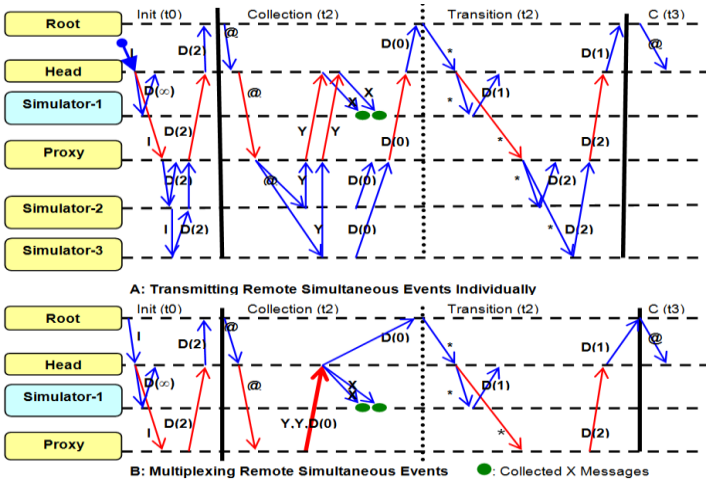

Figure 18: DCD++ Simulation Phases and Time Advancement

```
<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 19: Multiplexing Simultaneous Simulation Messages Together

The simulation message contains (at least) the following information (see Figure 19): 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. In this case, simulation managers always organize messages in the order they received from the DCD++ engine, allowing them to be handled in the correct order upon arrival at destination. Simultaneous messages aggregation is accomplished by having message bags in simulation managers to hold content messages to remote processors where those messages are sent with the first synchronization message (i.e. indicates the start/end of a phase) heading to the same processor, according to the shown algorithm in Figure 20. Message aggregation shows clearly that XML message-oriented semantics is much flexible to handle than procedure parameters semantics as in the RPC-style approaches.

```
SendRemoteMessage () {
   If (Remote Processor does not have a message bag) {
      Start Msg bag for Remote Processor;
   }
   Insert Msg in Message bag;
   If (this is a Synchronized Msg) {
      Start XML Document Builder;
      Pack Msgs count in XML Document;
      For (all messages in the Processor's bag) {
         Pack Msg in XML Document;
      }
      Close XML Document;
      Release Processor's Message bag;
      Send XML Document to remote URI;
   }
}
```

Figure 20: Dispatching Simulation Messages in Single XML Document

5 Distributed Simulation Interoperability Standards

The need for a widely accepted standardized framework is growing necessity nowadays, allowing sharing and reusability across organizations, laboratories and research teams. On the other hand, the specialization of knowledge and fragmentation in the distributed simulation field has also grown than ever. This caused the DEVS simulation community to start the interoperability standardization effort to interoperate various DEVS-based simulation packages together (e.g. CD++ [34]). DEVS standard proposals [30][31][32][33] categorized the standards into two parts: (1) Standardizing DEVS model representation allows a platform-independent DEVS model representation so that it can be executed by a DEVS-based simulator. In this case, a model may be retrieved and executed locally without the need to perform distributed simulation for obvious performance reasons. (2) Since, it is not always possible to run simulation locally on single or multiprocessor machine, the second part deals with Standardizing Interoperability Middleware protocol for interfacing different simulation environments allowing synchronization for the same simulation run across a distributed network regardless of their model representation, as shown in Figure 21. The second part is handled by the distributed simulation middleware, hence our presented topic here. The basic requirements of the interoperability

standards are to allow legacy systems to run their specific model representations, practical software changes (i.e. wrapper to translate messages from/to standardized protocol, see Figure 21), flexible for improvements, independent of any formalism or technology.

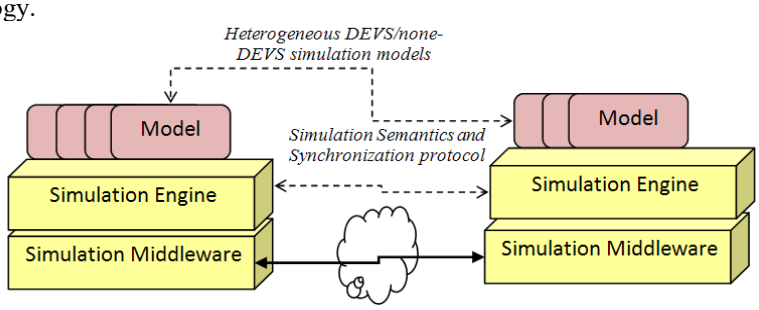

Figure 21: Concept of Standardized Distributed simulation Middleware

Plugging simulation components into RISE, enabling them to be online, hence accessed via URIs on the Web is one objective of RISE. Further, those components may need to synchronize between each other to simulate a single model within the same simulation model, hence distributed simulation session. In this case, distributed simulation synchronization is still under one team control. Thus, protocols can be customized as needed for specific simulation environment similar to DCD++ previously discussed here. On the other hand, having different independent-developed simulation environment synchronize between each other is another story of complexity. The main complex issue is to bring different teams agree on an interoperability standard. In reality, people do not support standards that require software changes that might affect an existing implementation. The preferred solution is usually by having a wrapper that translates standards from/to local messages. Further, programmers, in practice, do not like to read complex standards, particularly when they are simply evaluating standard proposals without being forced to use it. The lesson learned of the process of having DEVS interoperability standards is that standards should be simple and quickly to understand, fast to support, and without software changes to existing systems. RISE-based standards, presented here, uses the RESTful Web-services plug-and-play and dynamic interoperability style to overcome these issues. The RISE-based proposal details are described in [3] where we discussed all of the submitted proposals by the DEVS community in [30][31][32][33]. The following summarizes the RISE-based proposals in conjunction with the needed wrappers for both CD++ [34] and DEVS/SOA [33] that allow both environments to interoperate.

The RISE-based standards [3][30][31][32][33] divides the entire simulation space into domains. Each domain wraps a DEVS model and DEVS-based simulation engine to simulate that model. Each domain is accessed via three URIs (i.e. the wrapper API in Figure 21) to exchange semantics (i.e. synchronization and configuration) as standardized XML messages. The wrapper API (i.e. URIs) is created at runtime for each experiment setup. The standards completely hide interior implementation domain, avoiding software changes in existing implementation. For example,

RESTful DCD++ performs distributed simulation while DEVS/SOA uses DEVSJAVA [11] engine to perform distributed simulation using SOAP-based WS.

Interoperability is achieved at three levels: (1) the interoperability framework architecture level (API), (2) The model interoperability level, and (3) the simulation synchronization level. These aspects are summarized next.

The interoperability framework architecture level (API) provides the URI template that allows modelers to setup experiment resources across the network, as shown in Figure 22. These resources (URIs) are described as follows: (1) …/{framework}: represents a simulation environment domain. It is named by the modeler upon creation. The modeler uses this URI to submit all necessary information, including RISE XML configuration. (2) …/{framework}/simulation: represents active simulation in a domain, hence used by other domains to exchange simulation messages to synchronize a simulation session. The modeler further uses this URI to start/abort simulation, and to manipulate simulation during runtime or to retrieve online results while simulation is in progress. (3) …/{framework}/results: is automatically created by a domain upon completing the simulation successfully, maintaining simulation results and future results retrieval.

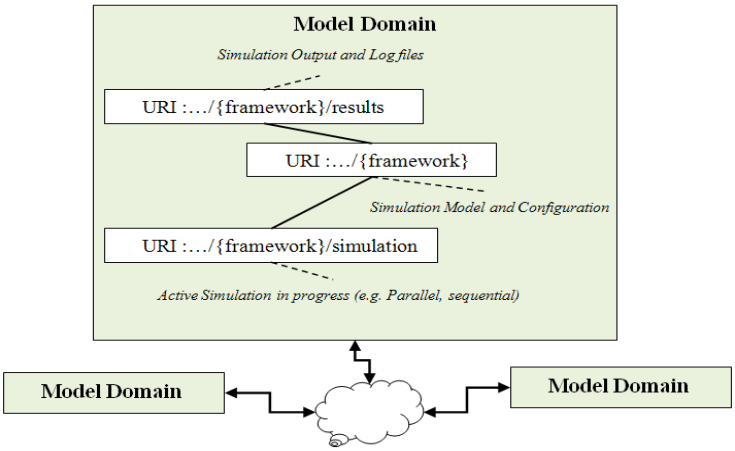

Figure 22: A Domain Wrapper Application Programming Interface (API)

The model interoperability level provides XML rules for binding different models together. This XML document is provided via PUT channel to URI …/{framework} as part of its initial configuration before a simulation is conducted. However, any dynamic changes during runtime are submitted to URI …/{framework}/simulation. This is mainly when a domain joins/disjoins a simulation session at runtime.

Connecting models across domains is a straightforward step, because of our assumption that each domain contains an entire model with external ports. For example, Figure 23 shows two models placed at two different domains. In this case, the model is wrapped in URI …/{framework}: The first model URI is …/Domain1 and the second model URI is …/Domain2. Each model, in Figure 23, has two external ports connected to the other model ports. This interconnection is shown in the XML document in Figure 24. For example, Lines 7-10 shows the connection link of port

OUT1 (at …/Domain1) to port IN1 (at …/Domain2). The XML document also shows other configuration such as "Type" at Line 3 is set to "O", indicating that the simulation will be synchronized according RISE conservative based algorithm; hence, "Type" attribute can be set to "O" to conduct optimistic synchronization. Line #5 selects the main domain, which is mainly needed to manage the conservative-based simulation. Based on this document Figure 24, each domain needs to build a routing table to identify each of its output port connections so that messages can be transmitted to their destination.

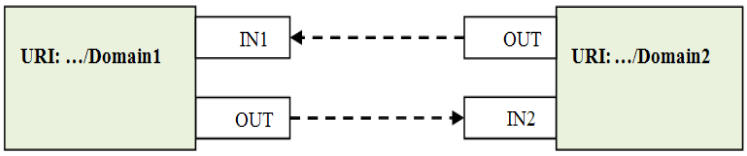

Figure 23: Models Interconnection across Domains

```
1 <ConfigFramework>
2     …
3     <RISE Version="1.0" Type="C">
4      <Domains>
5       <Main><URI>…/Domain1</URI></Main>
6        <Links>
7         <Link>
8            <From><Port>OUT1</Port><URI>…/Domain1</URI></From>
9            <TO><Port>IN2</Port><URI>…/Domain2</URI></TO>
10        </Link>
11        <Link>
12           <From><Port>OUT2</Port><URI>…/Domain2</URI></From>
13           <TO><Port>IN1</Port><URI>…/Domain1</URI></TO>
14        </Link>
15       </Links>
16      </Domains>
17    </RISE>
18    …
19 </ConfigFramework>
```

Figure 24: Model Interconnection XML Configuration

The simulation synchronization level provides high-level simulation algorithms (i.e. conservative/optimistic) and synchronization channels in order to carry simulation among different domains. In the optimistic type, XML synchronization messages are sent directly to other domains, since domains should be able to detect errors (e.g. due to straggler messages) and fix them. On the other hand, the conservative type needs to place a Time-Management component (e.g. called here RISE-TM) to synchronize all participants to satisfy local causality constraint via ensuring safe timestamp-ordered processing of simulation events within each domain. Our focus here is on the conservative-type algorithms, since it involves more work from the standards perspective.

RISE-TM executes a simulation cycle in the following steps, as shown in Figure 25: (1) Execute all events in all domains at current time. This starts a new simulation cycle with current or newly calculated RISE time. RISE-TM always starts the first phase with time zero. The domains must always execute all events with current RISE time, if any, and respond to the RISE-TM with the following information: all external messages generated for other domains stamped with RISE time (or larger), and its next time. The next time is the time of next event in a domain larger than RISE time.

(2) Once RISE-TM receives all replies from relevant domains, it calculates the next RISE time and starts a new simulation cycle.

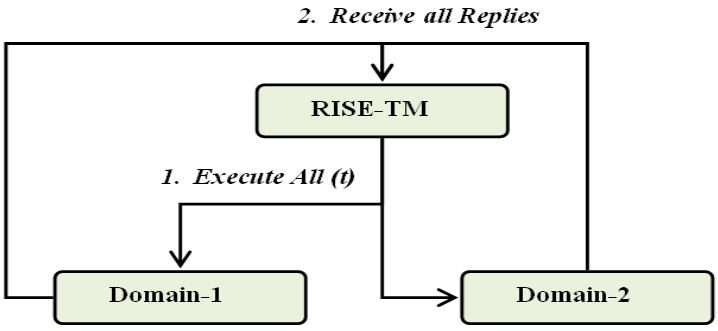

Figure 25: RISE Conservative-based Simulation Cycle at Time t

```
1 <RISE Version="1.0">
2  <Time>00:00:01:000</Time>
3  <XEvents>
4     <MessagesCount>1</MessagesCount>
5        <XEvent>
6           <Time>00:00:01:000</Time>
7           <Port>IN1</Port>
8           <Value>9</Value>
9           <URI>…/Domain1</URI>
10       </XEvent>
17  </XEvents>
18 </RISE>
```

(A) RISE-TM to Domains XML Request Message

```
1 <RISE Version="1.0">
2 <URI>…/Domain2</URI>
3  <XEvents>
4    <MessagesCount>2</MessagesCount>
5       <XEvent>
6          <Time>00:00:01:000</Time>
7          <Port>IN1</Port>
8          <Value>9</Value>
9          <URI>…/Domain1</URI>
10       </XEvent>
11       <XEvent>
12          … … …
13       </XEvent>
14    <Time>00:00:01:000</Time>
15  </XEvents>
16  <Next>00:00:03:000</Next>
17 </RISE>
```

(B) A Domain to RISE-TM XML Response Message

Figure 26: RISE-TM and Domains Exchanged Messages Example

Figure 26-A shows an example of messages sent by RISE-TM to a domain (i.e. step #1 in Figure 25). Line #2, in Figure 26-A, specifies the current RISE time, hence every event with this time, in this domain, must be executed in this cycle. Lines 3-17 enclose all collected external messages from all other domains, if any. Figure 26-B shows an example of a domain reply to RISE-TM (i.e. step #2 in Figure 25).

Line #2, in Figure 26-B, indicates the URI of the source domain. Lines 3-15 enclose all of this domain generated external messages to other domains. Line #4 specifies the count of enclosed messages. Lines 5-10 define the first external message. Line #6 specifies the execution time of this message. Line #7 specifies the model destination port (see Figure 23 and Figure 24). Line #8 specifies the message content. Line #9 indicates the destination domain (see Figure 23 and Figure 24). Line #14 specifies the minimum time of all enclosed external messages. RISE-TM must include this time when calculating next RISE time. Line #16 specifies the time of the next event of that domain. RISE-TM must include this time when calculating next RISE time. Further, it is recommended that RISE-TM does not include domains in the next simulation cycle if they have nothing to do. Note that this value must be set to "-1", indicating infinity, if there is no more events in that domain. This XML document guarantees that all of the domain events stamped with RISE time have executed. This guarantee must be ensured by the RISE-TM by ensuring that the "next" event time (i.e. Line #16 shown in Figure 26-B) is larger than the current RISE time, since it is

the time of the next event in a domain. Therefore, domains must only respond once with this XML document.

This method simplifies the synchronization protocol to avoid impractical software changes for a simulation package implementation. It also intended to handle synchronization between DEVS to None-DEVS simulation environments, since it hides all details behind wrappers, including DEVS formalism. The following discusses the changes require to interoperate DCD++ and DEVS/SOA simulation environments to conduct single simulation session. We focus here is on the simulation synchronization level of the standard.

In DCD++, the Simulation manager (see Figure 7) on the main server is the RISE wrapper (Figure 21) of the entire DCD++ domain. It is worth to note that other supportive DCD++ machines are not even aware of being part of a session bridged to another heterogeneous simulation environment. The simulation manager of the main server is extended to act as RISE-TM (i.e. the coordinator of all heterogeneous domains), or as a domain wrapper (i.e. it is being coordinated by other heterogeneous domain), as shown in Figure 25. Therefore, the main simulation manager handles exchanged messages between DCD++ machines according to its specific algorithms, while treat RISE messages according to the standards. Thus, the main DCD++ modifications is in adding new synchronization level between the main simulation manager and its associated CD++ engine. This is done in three parts: (1) having the CD++ engine forward all Y (i.e. output) messages that is intended to other domains to the simulation manager. These are the Y messages received by the Root coordinator (see Figure 18). Regular CD++ considers those messages as output to the environment. (2) Having simulation-manager forward all X messages, received from other domains, to its associated CD++ engine. (3) The CD++ needs to ask the simulation-manager permission before advancing the simulation clock beyond RISE time upon starting new simulation phase (see Figure 18). These parts are discussed in the next paragraphs.

First, the CD++ Root coordinator forwards all Y (i.e. output) messages to its associated simulation manager. This message also includes the simulation timestamp and the model source port. Note that the CD++ does not know where those messages need to be sent; hence, it treats them as output messages to the simulation environment. At this point, the simulation-manager converts those Y messages into X (i.e. external) messages and stores them so that they can be transmitted altogether in single XML document. The simulation-manager also needs to add the destination port and URI. This is easily done based on routing tables constructed based on the configuration document (Figure 24). For example, Y messages received from port OUT1 in Domain-1, at Figure 23, need to be routed to port IN2 of domain-2. Once those messages need to be transmitted, the simulation-manager builds the XML message, shown in Figure 26-B, and sends them to RISE-TM. However, if this simulation manager is the acting as the RISE-TM, it merges them with other domains messages, if any, and sends them back to relevant domains, as shown in Figure 26-A. Note that this special treatment is only for RISE messages, but messages belong to the DCD++ region need to be handled according to its specific algorithms.

Second, the simulation-manger needs to filter its domain X messages upon their arrival from other domains, and forwards them to the CD++. The simulation manager receives them as the message shown in Figure 26-B, if it is the acting RISE-TM while

receives as the message shown in Figure 26-A, if it is not the acting RISE-TM. Subsequently, the CD++ saves them in special queue until the beginning of next simulation cycle where Root coordinator will insert them in the simulation event list similar to any other local events.

Third, the CD++ needs to consult the simulation manager before advancing to new cycle (Figure 18). The entire DCD++ simulation cycles are driven by the Root coordinator, specifically, upon a DONE message arrival, as described in Figure 14. In this case, the Root checks DCD++ next event time against last known RISE time, it then proceeds if they are equal to each other. Otherwise, (1) it requires RISE Time update from simulation manager, (2) Insert any received external messages from other domains into the simulation event list, (3) calculate next event time, and (4) report next time to simulation manager. Based on the next event time and the current RISE time, the simulation manager knows the end of the current simulation cycle. These steps are handled in the following algorithm:

```
While (RISE Time != DCD++ Next Time) {
       Get RISE Time from Simulation Manager;
       Insert Other Domains Collected X messages;
       Calculate new DCD++ Next Time;
       Report Next Time to Simulation manager;
}
```

DEVS/SOA [33] uses DEVSJAVA [11] simulation engine to perform distributed simulation using SOAP-based WS. As illustrated in Figure 27, the DEVS/SOA protocol is executed as following (shown in Figure 27): (Step #1 and #2) the highest coordinator (i.e. Root) requests the next event time of each of its children simulators and coordinators. Messages nextTN and outTN are performed in a single RPC invocation. (2) The Root requests each of its children to compute its output messages to other simulators (i.e. getOut and outTN). (3) Finally, each simulator executes its ApplyDeltFunc method, which computes the combined effect of the received messages and internal scheduling on its state.

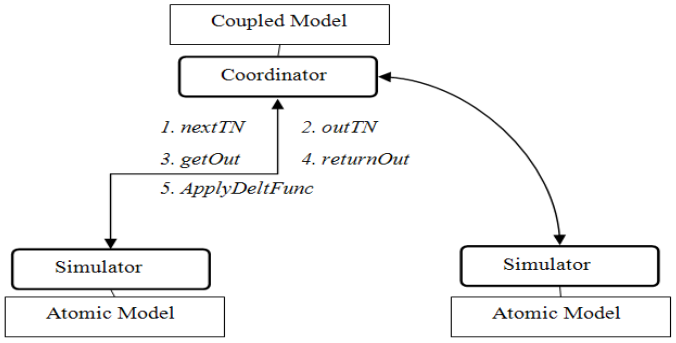

Figure 27: DEVS/SOA Distributed Simulation Protocol

DEVS/SOA needs to have wrapper (see Figure 21) to translate internal DEVS/SOA commands, shown in Figure 27, into RISE messages. This wrapper, similar to DCD++ simulation manager, needs to exchange RISE XML messages in HTTP envelopes. Further, the Root coordinator should not advance beyond current

RISE time. The major requirements of this wrapper is to translate DEVS/SOA RPC internal commands into RISE XML messages (Figure 26) and vice versa, as follows.

RISE XML message (Figure 27-A) corresponds to DEVS/SOA "nextTN", "getOut", "ApplyDelta" calls. Upon this message arrival from RISE-TM, all DEVS/SOA simulators must execute all internal/external events at this cycle time (i.e. element <Time>). Further, RISE-TM forwards previous output messages from previous cycles, if any, in this message.

RISE XML message (Figure 27-B) corresponds to DEVS/SOA "returnOut" and "OutTN" calls: (1) returnOut (i.e. output message) is RISE external message, defined in element <XEvent>. (2) OutTN (i.e. next time) defined in element <Time>.

References

[1] Al-Zoubi K. Wainer, G.: Performing Distributed Simulation with RESTful Web-Services Approach. In: Proceedings of the Winter Simulation Conference (WSC 2009), pp. 1323 - 1334. Austin, TX. (2009)

[2] Al-Zoubi K.; Wainer, G.: Using REST Web Services Architecture for Distributed Simulation. In: Proceedings of Principles of Advanced and Distributed Simulation (PADS 2009). pp. 114-121. Lake Placid, New York, USA. (2009)

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

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

[5] Amazon Web-services: Security Best Practices. <http://awsmedia.s3.amazonaws.com/Whitepaper_Security_Best_Practices_2010.pdf>. (2010). Accessed June 2010.

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

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

[8] Bryant, R. E.: Simulation of packet communication architecture computer systems". Massachusetts Institute of Technology. Cambridge, MA. (1977)

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

[10] Cheon, S.; Seo, C.; Park, S.; Zeigler, B.P.: Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System. In: Proceedings of the Advanced Simulation Technologies Conference, Arlington Virginia. (2004)

[11] DEVSJAVA: < http://www.acims.arizona.edu/SOFTWARE/software.shtml>. Accessed June 2010

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

[13] 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.
[14] Fielding R., Gettys J., Mogul J., Frystyk H., Masinter L., Leach P., Berners-Lee T.: Hypertext Transfer Protocol -- HTTP/1.1. RFC 2616. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>. Accessed October 2008.
[15] Fujimoto, R. M.: Parallel and distribution simulation systems. John Wiley & Sons. New York. (2000)
[16] Gregorio J: "URI Templates". <http://bitworking.org/projects/URI-Templates/>. Accessed October 2008
[17] Henning M.: The Rise and Fall of CORBA". Communications of the ACM. Vol. 51, No. 8, August 2008. Also available at <http://queue.acm.org/detail.cfm?id=1142044>. (2008). Accessed March 2010.
[18] Henning M., Vinoski S.: Advanced CORBA programming with C++. Reading, MA: Addison–Wesley. (1999)
[19] IBM Mashup Center. <http://www-01.ibm.com/software/info/mashup-center/>. Accessed June 2009.
[20] IBM Software Group: Why Mashups Matter. <ftp://ftp.software.ibm.com/software/lotus/lotusweb/portal/why_mashups_matter.pdf>. Accessed June 2009
[21] Khul F., Weatherly R., Dahmann J.: Creating Computer Simulation Systems: An Introduction to High Level Architecture. Prentice Hall (1999).
[22] Mandel L.: Describe REST Web services with WSDL 2.0. < http://www.ibm.com/developerworks/webservices/library/ws-restwsdl/>. Accessed May 2009
[23] Mittal S., Risco-Martín J.L., and Zeigler B.P.: DEVS-based simulation web services for net-centric T\&E. In: Proceedings of the 2007 summer computer simulation conference, San Diego, California, USA. 2007
[24] NetBeans IDE <http://www.netbeans.org/)>. Accessed June 2009.
[25] O'Reilly T.: What Is Web 2.0. <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>. (2005). Accessed May 2009
[26] Richardson L., Ruby S.: RESTful Web Services. O'Reilly Media, Inc., Sebastopol, California. (2007)
[27] Shahbazian E.: Introduction to DF: Models and Processes, Architectures, Techniques and Applications. in Multisensor Fusion. Kluwer Academic Pulishers, pp. 71-97. (2000)
[28] Strassburger S., Schulze T., Fujimoto R.: Future trends in distributed simulation and distributed virtual environments: results of a peer study. In: Proceedings of Winter Simulation Conference (WSC 2008), pp. 777-785. Miami, FL. (2008)
[29] Wainer, G.; Madhoun, R.; Al-Zoubi, K. "Distributed Simulation of DEVS and Cell-DEVS Models in CD++ using Web Services". Simulation Modelling Practice and Theory. Vol. 16, No. 9, pp. 1266-1292. October 2008.
[30] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. "DEVS Standardization: Foundations and Trends". Chapter 15, "Discrete-Event Modeling and Simulation: Theory and Applications." G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. October 2010 (expected publication).
[31] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. "An Introduction to DEVS Standardization". Chapter 16, "Discrete-Event Modeling and Simulation: Theory and Applications." G. Wainer, P. Mosterman (Editors). CRC Press. Taylor and Francis. October 2010 (expected publication).
[32] Wainer G., K. Al-Zoubi, S.Mittal, J.L. Risco Martín, H. Sarjoughian, B. P. Zeigler. "Standardizing DEVS Model Representation". Chapter 17, "Discrete-Event Modeling and

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

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

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

[35] Wainer, G. and Al-Zoubi, K.: An Introduction to Distributed Simulation. Chapter X in: Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains. Banks C., Sokolowski J. Editors. Wiley. New Jersey, (2010)

[36] Web Services AXIS. <http://ws.apache.org/axis/>. Accessed October 2008

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

[38] WSDL 2.0: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. < http://www.w3.org/TR/wsdl20/>. Accessed July 2010.

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