Simulation Modelling Practice and Theory 55 (2015) 27–45

Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

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

Distributed simulation of DEVS and Cell-DEVS models using the RISE middleware

Khaldoon Al-Zoubi *, Gabriel Wainer

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

ARTICLE INFO

Article history:
Received 11 November 2014
Received in revised form 28 February 2015
Accepted 30 March 2015
Available online 17 April 2015

Keywords:
Interoperability
Distributed simulation
Web services
REST
SOAP
Middleware
DEVS
Cell-DEVS
CD++

ABSTRACT

With the expansion of the Web, the desire toward global cooperation in the distributed simulation technology has also been on the rise. However, since current distributed simulation interoperability methods are coupled with system implementations, they place constraints on enhancing interoperability and synchronization algorithms. To enhance simulation interoperability on the Web, we implemented the RISE (RESTful Interoperability Simulation Environment) middleware, the first existing simulation middleware to be based on RESTful Web-services (WS). RISE is a general middleware that serves as a container to hold different simulation environments without being specific to a certain environment. RISE can hold heterogeneous simulations, and it exposes them as services via the Web. One of such services is called Distributed CD++ (DCD++) simulation system, an extension of the CD++ core engine that allows executing DEVS and Cell-DEVS models. Here, we introduce a proof-of-concept design and implementation of DCD++ using the distributed simulation using the RISE environment. We show how the RESTful WS interoperability style in RISE has improved the design, implementation and the performance of the DCD++ simulator. We also discuss a substantial performance improvement of the implementation of the RISE-based DCD++ presented here, showing many advantages of the RESTful WS presented here: improved interoperability, a seamless method to be connected into a cloud computing environment, and performance improvement when compared to our SOAP-based DCD++ in a similar testing environment.

1. Introduction

Modeling and simulation (M&S) is used extensively in studying complex systems. As simulated systems become increasingly sophisticated, the simulation software becomes larger and more complex. In these cases, the resources provided by a single-processor machine often become insufficient to execute these systems. Distributed simulation can expand from a single building to global networks usually interoperating heterogeneous processors (and software) [15]. With the expansion of the Internet, the desire toward global cooperation in the distributed simulation technology has also been on the rise as indicated by a number of surveys such as [9,29]. A focal point of distributed simulation software has been on how to achieve model reuse via interoperation of different simulation components. Other benefits include [32] connecting geographically distributed simulation components (without relocating people/equipment to other locations), interoperating different vendor simulation components (allowing reuse of M&S solutions), and information hiding—including the protection of

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

http://dx.doi.org/10.1016/j.simpat.2015.03.010
1569-190X/

intellectual property rights, and simulating larger problems via exploiting more available distributed computer resources (e.g. memory).

Web-enabled simulation is increasingly becoming the norm. These simulation systems can be divided into the following categories: (1) Web *access* enabled simulations, and (2) Web-based *distributed* simulations. The *Web access enabled simulation* systems provide users access to simulation systems running on a single machine [18,27] or on multiple machines geographically distributed [38]. However, the distributed simulation, in this case, is not *synchronized* via the Web, but according to those specific architectures. The second category is the *Web-based distributed simulation*. In such systems, the simulation partitions are synchronized partially or fully using Web services. Web-based distributed simulation interoperability methods are usually based *on SOAP-based WS* (e.g. [22,28,30,37]) or an *HLA* [20] *with SOAP WS* interface (e.g. [10,19,39,40]). However, such distributed simulation frameworks still have constraints in the structural rules that are placed on the interoperability design methods. In particular, the way they exchange, structure, and use information is tied to programming, making it difficult to decouple systems implementations and design. In practice, such constraints are difficult to overcome when interoperating existing heterogeneous simulation systems to synchronize same distributed simulation run. This is because each system interface is highly coupled to its own software design and programming, hence it is complex to homogenize different systems interfaces in order to be able to synchronize a distributed simulation.

To provide better solutions to overcome such interoperability constraints, we have proposed the RESTful Interoperability Simulation Environment (RISE) middleware [2], which is the first existing RESTful WS simulation environment. The objective was to achieve improved interoperability, as explained in [2], using RESTful WS. The use of RESTful WS is on the rise, and it has become the standard interoperability method on industrial cloud computing environments such as those in IBM [17] and Cisco [12]. RISE middleware has been built as a general container of different simulation environments. RISE exposes simulations as services (as URIs), allowing them to be seamlessly interoperated with other services (simulation and other) on the Web and cloud computing environments. The Distributed CD++ simulator (DCD++) presented here is one of such services exposed by RISE. Therefore, the presented DCD++ here is one of the service types exposed via the RISE middleware. However, RISE can expose more software services in addition to DCD++ like other simulators, visualization etc. [21,35].

In fact, exposing simulation services via the RISE middleware interoperability methods have been proven on different fronts. For example, as described in [35], RISE exposed DCD++ simulation services via the Amazon elastic compute cloud (Amazon EC2) [5]. In [35] we also proposed methods for model composition using workflows via RISE. In [21] we proposed a hybrid simulation and visualization approach where a dedicated mobile application runs on an Android Smartphone and the RISE-based DCD++ simulation (whose details are presented here) runs the simulation while hosted on the Cloud.

As discussed above, RISE is a general a Web-based interoperability container (i.e. system-of-systems); hence, systems need to be plugged into RISE before being able to use its services. We extended simulator called CD++ [31] and plugged into RISE so that it can perform distributed simulation on the Web, called Distributed CD++ (DCD++) [3,4]. It is worth to note that RISE-based DCD++ is the only distributed simulation system to use RESTful WS interoperability style, hence being part of a Cloud. Further, performing RISE-based DCD++ distributed simulation comes as the first natural step toward practical DEVS standardization [34], allowing different DEVS implementation to interoperate in order to reuse each other models and resources.

In the following sections, we introduce the RISE-based DCD++ simulation design, algorithms, implementation and performance. In Section 2, we present background information and summarizes current related work with a comparison to the presented RISE-based DCD++ simulation. Section 3 presents the RISE-based DCD++ simulation design and synchronization algorithms. Section 4 presents the RISE-based DCD++ implementation and the middleware implementation with a focus on the relevant parts to the DCD++. Section 5 compares the performance of the RISE-based DCD++ (presented here) to the SOAP-based DCD++ [30].

2. Background

This section provides an overview of the RISE middleware and the DEVS formalism, and it discusses related work.

2.1. Overview of the RISE middleware interoperability

RISE (RESTful Interoperability Simulation Environment) [2] is the first existing simulation middleware to be based on the RESTful Web-services [25]. In the Representational State Transfer (REST) [14] which is used to describe the WWW architectural elements, resources hold representations (states) where these representations are transferable between resources. For example, the client (e.g. Web browser) uses the GET HTTP method to transfer the HTML representation from the resource (e.g. Web site) to the client. Therefore, RISE is an interoperability container that can hold different simulation services regardless of their differences as shown Fig. 1. The presented DCD++ here is one of those services. However, RISE can expose more software services in addition to DCD++ like other simulators, visualization etc. (Fig. 1). It is worth to note that Fig. 1 only shows one partition of the DCD++ simulation, but DCD++ is often performed between several partitions.

To summarize RISE principles [2], RISE spreads services over a number of resources (resource-oriented), and resources exchange synchronization information in form of XML messages (message-oriented) via predefined constant virtual channels (uniform interface). These resources are exposed as URI templates whose instances can be created/destroyed at runtime

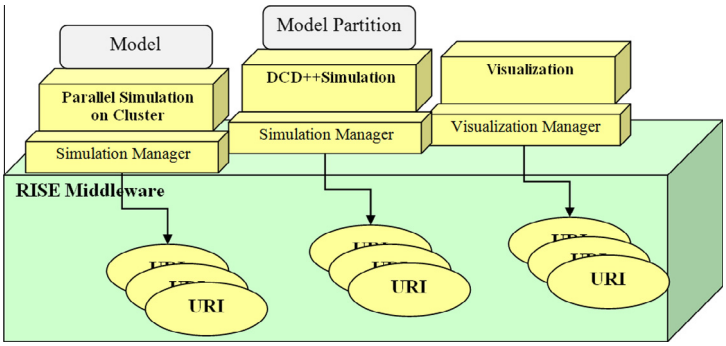

Fig. 1. RISE environment on single machine.

(by modelers). This leads to a concept of general layered interoperability where different simulation resources (URIs[1]) organized at a separate layer above the middleware. The message-oriented principle encapsulates simulation algorithms semantics, hence decoupling system implementations. Further, each URI is connected to other URIs via constant virtual channels; hence, this means that these channels are predefined in software (i.e. field in messages header). RISE channels are based on HTTP methods as follows: (1) The GET channel is used to read information from resources such as simulation status and results. (2) The PUT channel is to create a resource or update an existing data in a resource such as experiment settings. (3) The POST channel is used to append new information to an existing resource such as sending an XML synchronization message in a distributed simulation session. (4) The DELETE channel is used to remove a resource from RISE such as deleting an experiment URI.

In this style, messages are sent to URIs to manipulate URIs states, or messages are retrieved from URIs to read those URIs states. Thus, RISE interoperability goes beyond Web-based distributed simulation, but the ability to put anything attached to the Web (i.e. anything addressed by URIs) within simulation loop. Further, RISE serve as general middleware, allowing various systems to operate on the same local machine. These interoperability layers (where various simulation systems can operate at the same layer level on top of the middleware layer) allow one to have each layer using its own interoperability methods, and providing services to the layer above it. Following this concept, RISE is organized in three layers: Middleware, Simulation, and Modeling.

The Middleware Layer (discussed in [2]) provides a number of services to the Simulation layer, such as all means of communication and managing all simulation experiments lifecycle and executions. The Simulation Layer deploys different simulation environment types, each of which supports its own time management. The DCD++ simulator presented here is one of the systems supported by the RISE middleware. The Modeling Layer operates above the Simulation layer. This represents the system under study, which is simulated by a specific simulation environment.

2.2. Discrete Event System Specification (DEVS)

Discrete Event System Specification (DEVS) [39] Modeling and Simulation (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 $M = \langle X, Y, S, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$. 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: $N = \langle X, Y, D, \{M_d | d \in D\}, EIC, EOC, IC \rangle$. 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*.

Cell-DEVS [31] is an extension DEVS that defines each cell as an atomic DEVS model. Cell-DEVS describes n-dimensional cell spaces as discrete-event DEVS coupled models. Furthermore, it defines timing constructions rules for each cell, allowing explicit timing specification, asynchronous model execution, and integration with other types of models.

[1] Resources and URIs are used interchangeably throughout our discussion.

Both DEVS and Cell-DEVS based modeling are implemented by the CD++ [31] simulation toolkit. CD++ [31] is an object-oriented modeling and simulation toolkit package capable of executing DEVS and Cell-DEVS models. CD++ offers various versions to execute DEVS and Cell-DEVS models on different platforms. For each DEVS atomic model, users need to implement the various functions as required by the DEVS formalism in a C++ class, which is then integrated into the modeling hierarchy during compilation. On the other hand, for DEVS coupled models and Cell-DEVS models, users can specify the coupling information as well as other attributes of cell spaces in a model configuration file using a built-in script specification language.

2.3. Related work

Web-enabled simulation is increasingly becoming the norm these days. These simulation systems can be divided into the following categories: (1) Web access enabled simulations, and (2) Web-based distributed simulations.

Web access enabled simulation systems provide users access to simulation systems. These systems might be running on a single machine (e.g. [18,27]) or multiple machines in the form of geographically distributed simulations. The distributed simulation, in this case, is not *synchronized* via the Web, but according to those systems specific architectures. For example, the work described in [38] allows users to access and execute their simulation over typical HLA systems while the work describes in [6] provides Web access to parallel simulation on local clusters. Note that RISE [2] provides seamless RESTful Web access to all services (including the DCD++ simulator presented here, which is a *Web-based distributed simulation* system).

The second category is the *Web-based distributed simulation*. In such systems, simulation partitions are synchronized partially or fully using Web services. Web-based distributed simulation interoperability methods are either purely based *on SOAP-based WS* or an *HLA with SOAP WS* interface.

Examples of such *SOAP-based WS* distributed simulation systems can be found in [22,28,30,37]. All of these systems share the same interoperability principles: SOAP-based WS simulations using a client–server model. In this case, the simulation components act as both client and server, hence, a simulator becomes a client when it sends information, while the receiver simulator becomes the server. The information exchanged uses the Remote Procedure Call (RPC) mechanism: the sending component passes all the simulation information as parameters into a subject procedure (stub), and it makes a procedure call, and a procedure (service) is invoked in the receiver simulation software. This is exactly like invoking a procedure directly at the receiving simulation software. In fact, interoperating systems via the RPC-style API can be complex, as it usually needs homogenizing different implementations. Further, this type of RPC-style API leads to challenges in the areas of systems composition scalability and dynamic interoperability. The composition scalability problem arises because a procedure stub is needed at the client (sender) side for every unique service at the server (receiver) side. Further, because these stubs are programming procedures, they need to be written and compiled with the client software. This static approach can be a problem when we need dynamic interoperability (i.e. having systems join/disjoin simulation at runtime).

The second type of Web-based distributed simulation is the *High Level Architecture – HLA* (which can also have a SOAP WS interface). Examples of such systems are described in [10,19,39,40]. In a typical HLA simulation, entities called Federates communicate with each other using a common middleware, called the Run-Time Infrastructure (RTI). The Web-based HLA method is similar to the typical HLA, but federates are able to communicate with their RTI via the Internet using SOAP-based WS. In some works like [38], the SOAP-based interface can be either at the individual federates level or at the level of multiple federates (Federation).

Table 1 summarizes the key differences between the RISE-based DCD++ and other existing Web-based distributed simulation efforts.

The first difference is that we use RESTful WS. Row 2 indicates that RISE supports layered interoperability. As previously stated, this means that the upper layers use/share lower layers services where each layer implements certain functionalities. This provides various benefits such as middleware services reuse/share by upper layers, functionalities hiding and separation. Likewise, the middleware becomes independent of specific implementations. For example, DCD++ uses/shares RISE with other additional systems that are independent of DCD++. However, all existing SOAP-based Web-based distributed implementations mix the simulation and middleware. This makes such systems specific for certain implementations (e.g. [22,28,30,37]). Row 3 indicates that since information is always exchanged via a constant number of virtual channels (specifically the four HTTP methods[2]), the growing number of partitions does not influence the overall structure in the RISE-based DCD++. However, this is not the case in the other existing systems, because they interface using programming procedures, which are variable, hence they change due to software design changes.

Row 4 indicates that DCD++ uses standardized HTTP channels for exchanging information. However, other existing systems usually use their own designed programming procedures. In practice, RPCs are usually difficult to homogenize between existing systems since they reflect internal design decisions. Row 5 indicates that DCD++ builds XML messages with the necessary simulation information and transmits them to remote partitions. This enhances heterogeneous systems implementations because systems only need to agree on "what" information to exchange in XML. However, "how" to implement handling of XML messages is left to specific systems design. It is true that SOAP WS describes procedure call as SOAP

[2] HTTP methods are actually virtual channels; hence a field in HTTP message header set upon transmission. The Term "methods" is usually used in the literature since it is the term used in the HTTP standards.

Table 1
Comparing RISE-based DCD++ to current Web-based distributed simulation.

	Characteristic	RISE-based DCD++	[22,28,30,37] SOAP-based WS distributed simulation	[10,19,39,40] HLA/SOAP based WS distributed simulation
1	Interoperability approach	RESTful WS (via RISE middleware)	SOAP WS	HLA (with SOAP WS Interface)
2	Layered interoperability	Yes	No	Yes
3	Composition scalability	Yes	No	No
4	Standardized information channels	Yes	No	No
5	Synchronization messages description	XML messages (message-oriented)	Programming parameters (RPCs in WSDL)	HLA programming attributes and RPCs described in WSDL
6	Interoperability sensitive to programming	No	Yes	Yes
7	Experiments direct access on the Web (experiments seen as URIs)	Yes	No	No
8	Experiments named with modelers choice of URIs	Yes	No	No
9	Interoperable with Web 2.0	Yes	No	No

messages, which is an XML message. However, the simulation partitions pass their simulation information as programming parameters via RPC. This procedure call is converted to XML SOAP message at the sending WS layer, which is reconverted back to a local procedure call at the receiving WS layer. Thus, simulation systems interoperate via RPC style. This makes simulation systems interoperability sensitive to programming changes (as indicated in Row 6). If a single parameter in one of the procedures is changed in a SOAP-based system, all relevant interoperating systems need to upgrade their software, recompile, and install new systems before being able to interoperate. Row 7 indicates that the experiments in RISE-based DCD++ are interfaced on the Web via a number of URIs. These URIs are exactly like any other URIs on the Web. This is important because anything attached to the Web is in the simulation loop. However, achieving this concept in other existing systems is more complicated, because experiments in other existing systems are not exposed as URIs; hence, they are not directly attached to the Web. Specifically, SOAP-based systems expose services via ports. Ports can be viewed as C++ or Java objects with a number of RPC operations. Each port is addressed by a URI, allowing remote systems to reach it. This complicates interoperability on the Web because it deviates from the Web interoperability mechanisms. In fact, by interoperating in the Web style, RISE allows DCD++ to not only expose experiments as URIs (Row 7), but also to be named and created by modelers (Row 8), and they are interoperable with Web 2.0 [24] (Row 9).

A non-related advantage is that the use of RESTful WS has substantially improved performance (discussed later). Easing the interoperability constraints provide software designers with more room to improve performance.

3. RISE-based DCD++ simulator

In this section, we discuss the distributed architecture of CD++ (DCD++) [3,4] and its model partitioning (Section 3.1), and the simulation execution and synchronization (Section 3.2). Our focus here is to present the RISE-based DCD++ simulation progress and activities once a modeler starts the simulation with and Experimental Framework (EF) [2]. The RISE EF is created by a modeler to control a simulation experiment lifecycle and to store all necessary information related to that experiment. This creation is done via creating a number of URIs of a modeler naming choice. Thus, the RISE EF is seen on the Web as a number of URIs, hence anything attached on the Web in within simulation loop. As discussed in [2], the RISE EF is designed as URI template [16], allowing modelers to have the same experiment interface regardless of the used simulation system, as shown in Fig. 2.

In this EF (Fig. 2), the modeler first creates an experiment by creating its URI "…/{systemtype}/{framework}". In this case, the {systemtype} is used to select the simulation system type. For example, setting {systemtype} to DCDpp value would instruct the RISE middleware to use the DCD++ simulation system for this experiment (modelers may select other supported simulation environments via RISE too). The {framework} template is used to name the experiment, which becomes the experiment main URI. For example, the URI "…/DCPpp/FireModel" indicates that this experiment name is "FireModel" and uses the DCD++ simulation environment. This experiment URI is used by the modeler to submit all necessary information to the experiment. For example, in case of DCD++, as discuss in the next sections, the modeler submits all of the subject CD++ model scripts and the distributed simulation configuration. This configuration is XML document to instruct the RISE middleware on how the model is partitioned over a number of machines. After that, the modeler executes the simulation of an experiment by creating the URI "…/DCDpp/{framework}/simulation" on the main RISE server[3] (i.e. the used server by modeler to create the main experiment URI). As a result, the main RISE server contacts all other servers in the network to create all simulation partitions, as discussed in the following sections. This URI "…/simulation" is further used by a simulation partition to receive all XML synchronization messages from other partitions during a distributed simulation sessions. URIs "…/debug" and "…/results" are automatically created upon a simulation run completion. The "…/debug" resource would

[3] Server and Middleware terms are used interchangeably throughout our discussion.

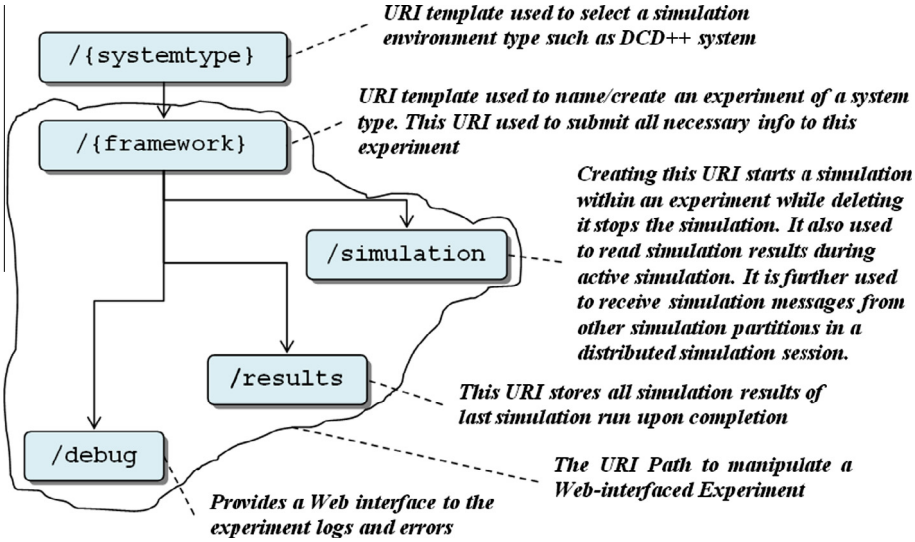

Fig. 2. Simulation experiment framework resources/URIs.

hold debugging information related to the model implementation that could help the modeler of fixing his model software issues. The "…/results" would hold the last simulation run results, which could be used for replay without running the simulation over again.

3.1. DCD++ simulation architecture

Once the simulation is started, all of the necessary software components are created in all partitions. These components are also deleted when the simulation is completed or aborted. Fig. 3 shows an example of a DCD++ experiment with two partitions during simulation. Each partition contains an instance of the CD++ engine to simulate the DEVS hierarchy models that belongs to this partition. The simulation manager is the component that manages the distributed activities on behalf of the CD++ engine with remote partitions.

Note that the CD++ (which is based on P-DEVS [13]) performs the simulation time management while the simulation manager performs the simulation data distribution on behalf of its CD++ instance. Both Simulation Manager and the CD++ engine communicate P-DEVS messages through the OS IPC queues. The use of IPC allows integrated software components to be separated in components from the RISE middleware. In this case, those components started/deleted as needed. It is still possible to implement such components within the RISE middleware, but with price of mixing implementations together. The idea here is that the CD++ executes local simulation events by dropping them directly in their local destination coordinator/simulator. However, if the destination coordinator/simulator does not exist locally, it is sent to the simulation manager (on the RISE side), which knows how to distribute it. This allows CD++ to execute the simulation locally as if it was running on a single-processor machine, while allowing the simulation manager to handle the distributed activities.

The IPC Monitors (Fig. 3) are threads found at both ends of the IPC queues, which are in charge of processing P-DEVS received from the other end. Once the CD++ IPC Monitor thread receives a message (from the Simulation Manager), it inserts it in the main CD++ external event list. This relieves the main thread (within CD++) of continually checking the IPC queue. Further, once the IPC Monitor thread (at the Simulation Manager side) receives a message from the CD++ engine, it buffers it at the Simulation Manager according to its partition destination. This allows the Simulation Manager to aggregate those remote messages in XML and transmit them together (via RISE). This not only reduces the number of remote messages through the Internet, but also avoids causality errors. This incorrect simulation could occur because P-DEVS messages may arrive in the incorrect order of their transmissions at the distant CD++ (since RISE transmits those messages concurrently via the Internet). Thus, these messages may violate the local causality constraint in the distant CD++, starting the wrong simulation phase, as discussed next in Section 3.2. The final component shown in Fig. 3 is the watchdog component. A watchdog is a thread that sends periodic messages to other simulation URIs to check their presence. If a partition is present, it responds back with the XML message <simulation>ALIVE</simulation>. Otherwise, it responds with HTTP error 401 (Not Found). The watchdog on the main partition watches all other partitions existence, while the watchdog threads on other partitions watch the main partition. Watchdog threads are necessary because a CD++ engine on a partition may fail during simulation, leading to inaccurate simulation or deadlocks. For example, assume a CD++ in a partition fails while the CD++ Root coordinator (in the main partition) is waiting for a "Done" message from that dead partition. In this case, the Root coordinator would wait forever without being able to advance the simulation (and without being aware of the problem).

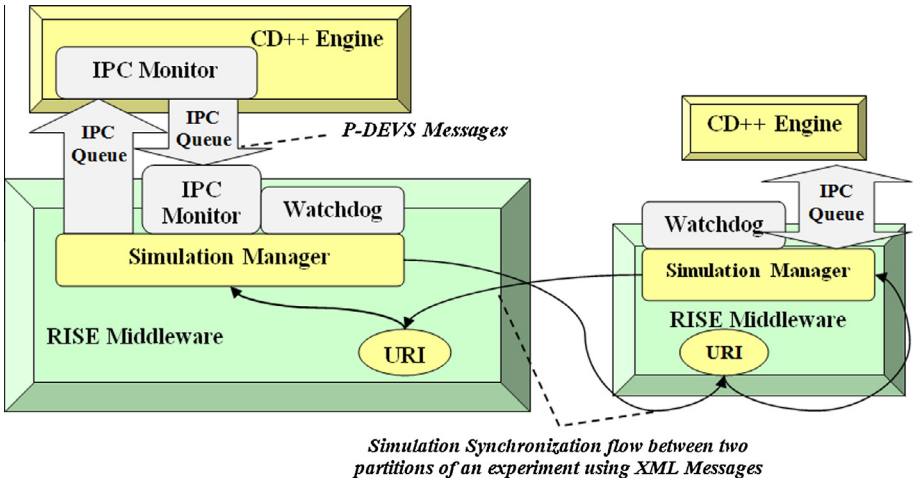

Fig. 3. Overview of DCD++ simulation architecture.

As previously discussed, once the simulation is started, the simulation manager is created, which in turns builds all components shown in Fig. 3. Afterward, the model needs to be loaded by the CD++ engine in each partition. In this case, CD++ only needs to simulate a portion of that model. To do so, CD++ in a partition initializes the model in two steps: (1) Load the model by parsing the CD++ model coupled model file (CD++ assumes that it will simulate the entire model; hence, it parses the entire model). (2) Build the processors that will simulate all submodels in the DEVS models hierarchy. Before building each sub-model simulator, CD++ requires permission from the simulation manager to do so (because it knows how the model is partitioned, as discussed shortly). If permission is granted, the sub-model simulator is built and assigned a unique ID. However, if permission is not granted, the simulator is not built, but the ID is still generated (and forwarded to the simulation manager). This is important because DCD++ maintains a unique ID for each processor (i.e. simulator/coordinator) in all partitions.

The overall DCD++ model is structured across the network based on the model partitioning requirements (which also helps the simulation manager to perform data distribution on behalf of its CD++ instance). These requirements originally come from the modeler as part of setting up the experiment. DCD++ supports fine-grained model partitioning by assigning as low as the atomic and cell models (i.e. indivisible blocks in the overall DEVS and Cell-DEVS models) to the partitions in the distributed environment. For example, the following XML document splits a 30x30 Cell-DEVS based Fire model into two partitions:

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

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

As shown in Fig. 4, the CD++ in Partition 0 (on the left) builds the Root coordinator to manage the overall simulation (the time management and synchronization according to P-DEVS algorithms [13]). Note that the "Local communication" indicates that all messages are indirectly sent via the local CD++ Event List, as described shortly. This is the case, between the Head Coordinator and its children Simulators, between the Proxy Coordinator and its children Simulators, and between the Head Coordinator and the Root Coordinator. However, the "Communication via RISE" indicates that messages are sent

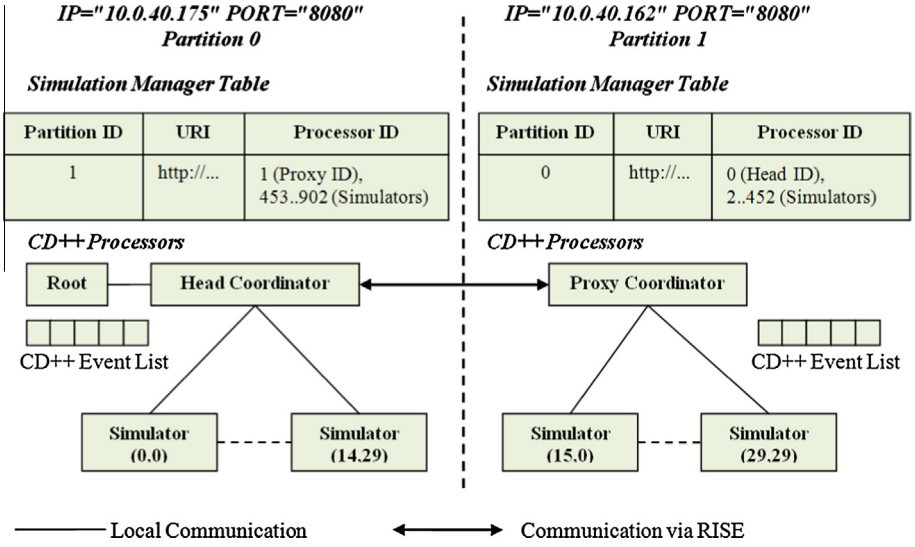

Fig. 4. DCD++ models hierarchy partitioning example.

through the Internet (which usually aggregated in XML messages as described in next section). This is the case between the Proxy and the Head Coordinators.

In this example (Fig. 4), this Root coordinator becomes the parent of the Head coordinator with ID 0, which simulates cells zone (0,0) locally to (14,29). Since CD++ simulates each cell as an atomic model, CD++ will then run 450 simulators (with IDs 2...452). In the same way, the CD++ in Partition 1 (in the right) starts the Proxy coordinator with ID 1. This Proxy coordinator locally simulates cells zone (15,0)...(29,29) on behalf of the Head coordinator in the other partition. In this case, the CD++ in Partition 1 runs 450 simulators with IDs 453...902. The CD++ processors pass to each other the P-DEVS messages by first inserting those messages in the CD++ External Event List (to be executed later by the Administrator). The Administrator (not shown in the figure) picks the message at the front of the CD++ Event List and checks the destination of the message; if the destination is a local processor, the message is directly delivered to that processor. However, if the destination processor does not locally exist, the message is then sent the Simulation Manager, which maintains the necessary partitions information to be able to transmit remote messages. Specifically, it stores the remote partitions URIs and the CD++ processors IDs on those partitions.

DCD++ extends the concept of original DEVS coordinators into a head/proxy structure [30]. The idea of the head/proxy depends on using two kinds of coordinators for each coupled model: (1) Head Coordinator: is responsible for synchronizing the coupled model execution, interacting with upper level coordinators and message routing among the local and remote processors. (2) Proxy Coordinator: is responsible for message routing among the local processors. The advantage of using proxy coordinators is to avoid remote message transmission between local processors. For example, assume that Simulator (15,0) needs to send Simulator (29,29) a message. In this case, this message will be routed through the Proxy coordinator. On the other hand, if the Proxy coordinator were not used, the message would be first sent to the Head coordinator in Partition 0, which would then route back to Partition 1.

3.2. DCD++ simulation synchronization

Once each CD++ is initialized with the model partitions as described in Section 3.1, models simulation and synchronization between partitions starts. During simulation, the CD++ processors discussed in the previous section send each other P-DEVS messages via inserting them first in the local CD++ External event list. The CD++ administrator processes messages in the list by dropping them directly in the local CD++ processors or by sending them to the simulation manager (to be forwarded to remote CD++ processors in remote partitions).

P-DEVS messages can be categorized as follows: (1) Content messages represent events generated by a model. Content simulation messages include External messages (X) and Output messages (Y). X and Y messages that are exchanged within a simulation phase are simultaneous, since they are stamped with the same simulation time. (2) Synchronization messages cause the simulation to move into another simulation phase. In other words, synchronization messages mark the beginning and the end of each simulation phase. Synchronization messages include the Initialization message (I), the Internal message ($*$) (to start a transition), the Collect message (@) (to start a collection), and the Done message (D) (to end a phase). In this case, the I, $*$, and @ messages are sent from the parent coordinator downward throughout the model hierarchy. On the other hand, the D messages are generated from simulators upward throughout the model hierarchy. Therefore, there are three phases: (1) The Initialization phase initializes all models in the hierarchy. (2) The Collection phase to collect Y messages

generated to ensure their execution at the same time with internal events (in the transition phase). (3) The Transition phase executes all X messages (which originally generated as Y messages) alongside the simultaneous internal events (by simulators). Therefore, at each simulation time, there is at least one mandatory Transition phase and an optional Collection phase. This means that multiple Transition phases may be executed multiple times (since additional internal events may continually be generated). However, the initialization phase only exists at the beginning of the simulation.

Because each phase is started by a message (I, @, or $*$) and ends with a D message, the messages must then be received at the destination processor in the correct order (to ensure correct simulation progress). However, the RISE middleware transmit all messages concurrently (i.e. each in its own thread). This is necessary because the RISE middleware layer is shared by various simulation components (that belong to different types and experiments) operating on top of it. Thus, RISE is expected to balance the workload as well as possible between those components (our RISE workload testing is not included here). In order to overcome the message-order problem, DCD++ aggregates simultaneous remote P-DEVS messages and sends them via RISE together (in single XML messages). This solution applies parallelism during the message transmission, ensures the correct order of P-DEVS messages at destination, and ensures messages execution in the correct simulation phase.

The simulation phases without aggregation progress as follows (Fig. 5A): (1) the first phase is Init (t0) (initialization phase at time 0). A Message I is sent to the Head coordinator, which passes it to Simulator 1 and the Proxy coordinator. Consequently, the Proxy routes message I to Simulator 2 and Simulator 3. Each Simulator 2 and 3 reply with the D(2) message. This means that both have scheduled an internal change in two time units from now. Further, Simulator 1 replies with D(∞). This means that Simulator 1 has no more events to execute. (2) The second phase is Collection (t2). In this phase, Root advances time to 2 and starts the collection phase by sending message @ to the Head coordinator, which only sends it to the proxy. This message is not sent to Simulator 1 because it did not schedule a change in the previous phase; hence, it becomes irrelevant in this phase. Consequently, the Proxy passes message @ to Simulator 2 and 3, which causes them to send two jobs (i.e. each sends a Y message) to Simulator 1 (via the Head and Proxy coordinators). The Head coordinator converts those Y messages to X messages and send them to Simulator 1. Simulator 1 then collects them to be executed in the next phase. Simulator 2 and Simulator 3 ends this phase by sending D(0) message upward in the hierarchy. This also means that they

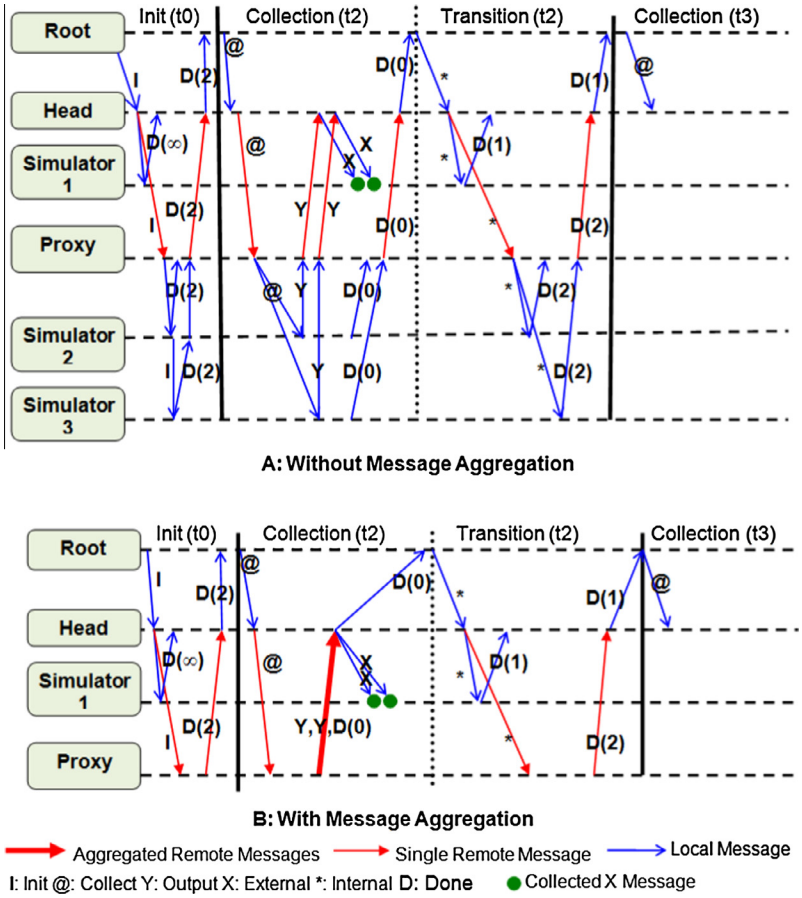

Fig. 5. Example of DCD++ simulation phases and time advancement for the same simulation scenario.

will be involved in the next phase. (3) The third phase is Transition (t2). The Root starts transition phase (by sending * message downward) causing Simulator 1 to execute the two previously collected X messages. It further schedules a change at one time unit from now. In addition, Simulators 2 and 3 schedule a change at two time units from now (when they will produce their next jobs as Y messages). As shown in Fig. 5B, the only messages that were aggregated and sent together are in the Collection (t2) phase. These messages are the two Y messages and the D(0) sent from the proxy coordinator to the head coordinator. However, other remote messages were sent individually. This is because other remote messages must not be delayed to allow simulation to progress. For example, the @ message sent by Head coordinator to the Proxy coordinator in Collection (t2) phase must be transmitted to trigger the collection phase on the Proxy portion of the hierarchy. Otherwise, the simulation does not advance.

The basic idea behind aggregation is that content messages (Y and X) are sent to processors within a simulation phase. Thus, they are simultaneous messages (messages exchanged within the same timestamp). On the other hand, synchronization messages (I, @, *, and D) are sent to start/end a phase. Therefore, content messages that are heading to same processor can be buffered until the first synchronization message is received to that processor. Further, messages that are heading to processors in the same remote partition can also be sent together. This is because those messages are heading to the same destination. Therefore, dispatching and aggregating simulation messages are only performed to remote messages (which were originally forwarded from the CD++ to its simulation manager). This algorithm is listed in Fig. 6, and an example described in Fig. 7. In this scheme (Fig. 6), the CD++ (in the local partition) maintains unprocessed events in a Least-Time-Stamp-First (LTSF) list. The CD++ Administrator executes the first event in the list by sending to a local processor or by sending it to the simulation manager. The Aggregator (in the simulation manager side), maintains the aggregation message queues. These queues are built and destroyed dynamically as needed. The aggregation queues are organized by their destination partitions and processors.

For example, the Aggregator (Fig. 7) opened queues for Partition 1 and Partition 2. In this case, Partition 1 has two queues one for Processor 5 while the other for Processor 8. As shown in the figure, Processor 8 queue is complete since it has already received the synchronized message D(tN). This means that CD++ is not going to generate any messages to Processor 8. However, Processor 8 messages are not dispatched yet because Processor 5 still has more expected messages. On the other hand, the Dispatcher sends Processor 2 messages to Partition 2, since all of Partition 2 queues have been completed. Consequently, the Dispatcher walks over all messages and sends them in single XML message.

On the other hand, the aggregation scheme (Fig. 7) needs to answer two possible situations: (1) The Dispatcher may send messages to a partition before other processors (belong to that partition) queues even started. In this case, new queues are

```
DispatchAndAggregateRemoteMessage (Msg) {

   Dispatch_XML_Msg = false;

   Find_Remote_Partition_ID (Msg.DestinationProcessorID);

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

   Insert Msg in the Processor's bag;

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

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

Fig. 6. Dispatching and aggregating simulation messages in XML.

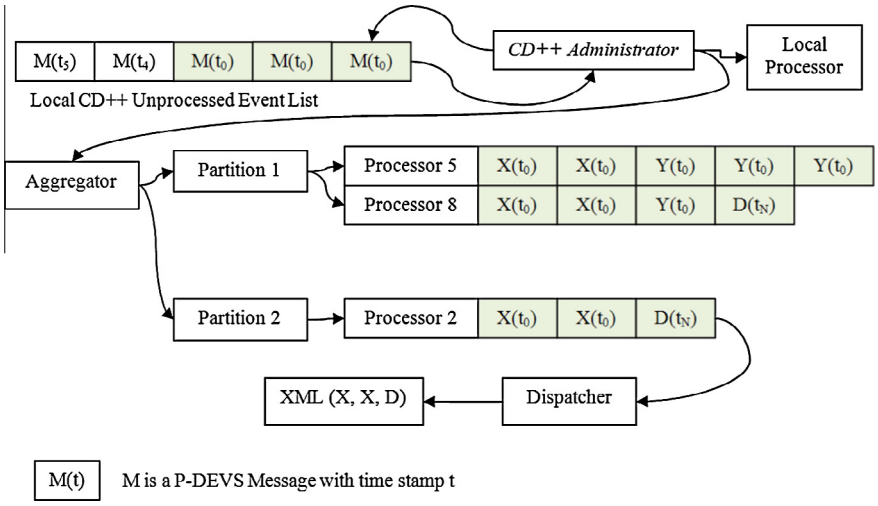

Fig. 7. DCD++ aggregation message queues.

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

4. Implementation

In this section, we will provide the implementation of the RISE-based DCD++ (Section 4.2). However, because DCD++ operates on top of the RISE middleware, we also present the RISE middleware implementation, but with a focus on the most relevant subsystems to the DCD++ (Section 4.2).

4.1. RISE subsystems

Our main interest here is two of RISE subsystems: Resources subsystem and SimulationAdmin subsystem: Resources subsystem (Fig. 8) processes received messages to RISE URIs while the SimulationAdmin subsystem (Fig. 9) manages active simulations.

RISE starts a thread (from a pool) to handle each incoming request to a URI. Afterward, the following steps are taken: (1) an instance of a Java class (Fig. 8) is created based on the destined URI template (previously discussed in Section 3). Thus, there is a Java class (in Fig. 8) corresponds to each URI template in RISE, as shown in Table 2. (2) The appropriate virtual channel operation of the subject resource class is invoked depending on the message access channel (previously discussed in Section 3). In this case, channels are indicated by the listed HTTP method in the HTTP envelop header. The channels are implemented by the following Java operations (if supported) in each Java class: (1) GET channel is handled by the represent operation. (2) PUT channel is handled by the storeRepresentation operation. (3) POST channel is handled by the acceptRepresentation operation. (4) DELETE channel is handled by the removeRepresentations operation.

As discussed in Section 3.1 (Fig. 3), the DCD++ XML synchronization messages (wrapped within HTTP envelope) are sent to the active simulation URI of a partition. Afterward, this URI resource parses the XML message content and forwards the information to the simulation manager of that partition (which in turn forwards the DEVS messages to its correspondent CD++ engine via the IPC queues). To put these steps in the implementation context, let us suppose, for example, an HTTP message is just received to URI "…/cdpp/sim/workspaces/Bob/DCDpp/FireModel/simulation". To process this message, RISE starts a Java thread and creates for that thread an instance of Java class SimulationResource (Fig. 8). This is because the destined URI instance matches the URI template associated with class SimulationResource (as shown in Table 2). Part of the class initialization (i.e. in the class constructor), URI templates variables are assigned as follows: {userworkspace} = Bob, {servicetype} = DCDpp, and {framework} = FireModel. This information allows RISE to retrieve the Java object associated with this experiment instance. Further, the password associated with username Bob is also retrieved. Note that database objects are cached upon the first access (for simplicity, we do not discuss database implementation here). Once the class is initialized, the acceptRepresentation operation of that class is invoked by RISE, since POST is the listed channel in the HTTP header. This operation first authenticates the received message (based on HTTP Basic Authentication). It then retrieves and parses the embedded XML in the HTTP envelope.

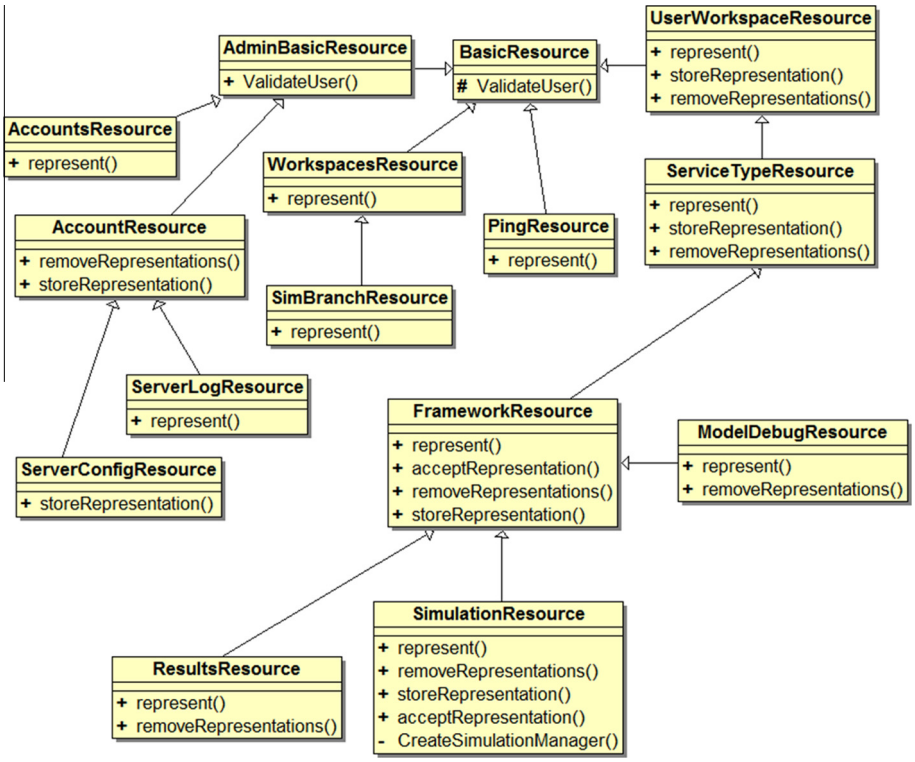

Fig. 8. Resources subsystem implementation overview.

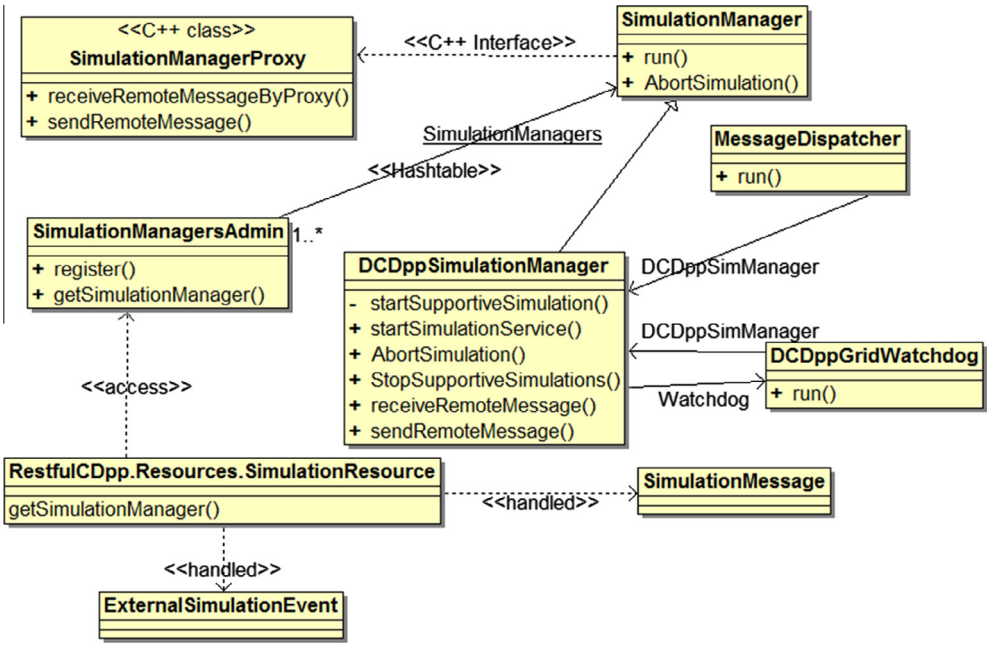

Fig. 9. SimulationAdmin subsystem implementation overview.

This parsed XML information are stored in SimulationMessage (Fig. 9) objects and passed to operation receiveRemoteMessage of class DCDppSimulationManager (Fig. 9). Of course, the information needs to be passed to the correct DCDppSimulationManager object associated with the subject experiment. In this case, RISE maintains all simulation managers (with unique IDs) in array structures within class SimulationManagersAdmin (Fig. 9). Since RISE also stores each

Table 2
Resources URI templates mapping to Java classes.

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

manager ID in the experiment framework object, the SimulationResource class can then use this ID to locate the correct manager object. Note that the SimulationManager ((Fig. 9) object of a simulation run is created upon simulation start by method CreateSimulationManager in SimulationResource class (Fig. 8).

The DCDppSimulationManager class is extended from the generic SimulationManager class to handle DCD++ simulation management in geographically distributed environment. The DCDppSimulationManager class keeps track of all information related to other remote simulations and their model partitions. Some of these tasks are summarized as follows: (1) it passes received messages from remote simulation to its correspondent CD++ engine. This is done with the help of the SimulationManagerProxy class. The SimulationManagerProxy class (written in C++) handles the IPC communication between a CD++ engine and its associated simulation manager. Thus, there is an instance of the SimulationManagerProxy class for each DCDppSimulationManager instance. (2) It handles the message transmissions with all remote simulation partitions (as discussed in Section 3). This is done with the help of class MessageDispatcher. The MessageDispatcher class is used to dispatch messages where each message transmission lives in a separate thread. In this case, a simulation manager is able to transmit many messages simultaneously. (3) It starts/stops watchdog thread to watch the participant in the distributed simulation environment. This is done with the help of the DCDppGridWatchdog class. (4) If it is the main simulation, it starts and stops simulation on all support partitions (via operation startSupportiveSimulation). It further collects results and debugging data from support entities. Note that the RISE middleware uses the Restlet API [26] (which is realized by the Noelios Restlet Engine (NRE) implementation [23]) to provide set of APIs, mainly allowing the RISE resources to access the received HTTP information in a consistent way.

4.2. Interfacing CD++ With RISE

In Section 4.1, we indicated that the SimulationManagerProxy class (Fig. 9) forwards all messages via the IPC queues to the CD++ engine within a partition. This class also forwards received messages from the CD++ to class DCDppSimulationManager (Fig. 9) to be distributed remotely.

In a similar way, the CPPManager class (Fig. 10) is responsible for interfacing the CD++ simulation engine with the SimulationManagerProxy class via IPC queues. The CPPManager class creates two message queues: send_queue_id (for sending messages to the simulation manager) while receive_queue_id (for receiving messages from the simulation manager). The functionality of the CPPManager includes: (1) Initializing the message queues used for communication with the simulation manager (i.e. operation initializeMessageQueues). (2) Querying and retrieving the model partitions from the simulation manager in RISE side (i.e. operations machineForModel, and addZonePartition). (3) Querying the current execution time and inserting external events while the simulation is running (i.e. operations getCurrentSimTime, and insertExternalEvent). (4) Sending remote messages while running distributed simulations (i.e. operation sendRemoteMessage). This method takes a CD++ message and sends it to the simulation manager to be sent to the remote machine. (5) Receiving remote messages while running distributed simulations (i.e. operation receiveRemoteMessage). This method receives a message from the simulation manager and constructs a CD++ message to be processed by the simulator. (6) Stopping the simulation when receiving a stop message from the simulation manager (i.e. operation stop).

During CD++ simulation of a DEVS model, each processor (in Fig. 10) keeps track of the atomic/coupled model that is responsible of executing. The Processor class is the parent of all the classes in charge of executing the model. Those include the Simulator, Coordinator, and Root classes. The Processor class implements the basic functionality required by all simulation classes: (1) Receiving and processing the different simulation messages, (2) Sending messages and scheduling simulation events via class MessageAdmin. The Simulator class extends the Processor class and executes the functions of the atomic DEVS model corresponding to the type of the received message. The Coordinator class is responsible for forwarding messages among the Simulators and for synchronizing the events taking place during the simulation. The FlatCellCoordinator class is in charge of executing flat Cell-DEVS models, which differ from Cell-DEVS models in that they

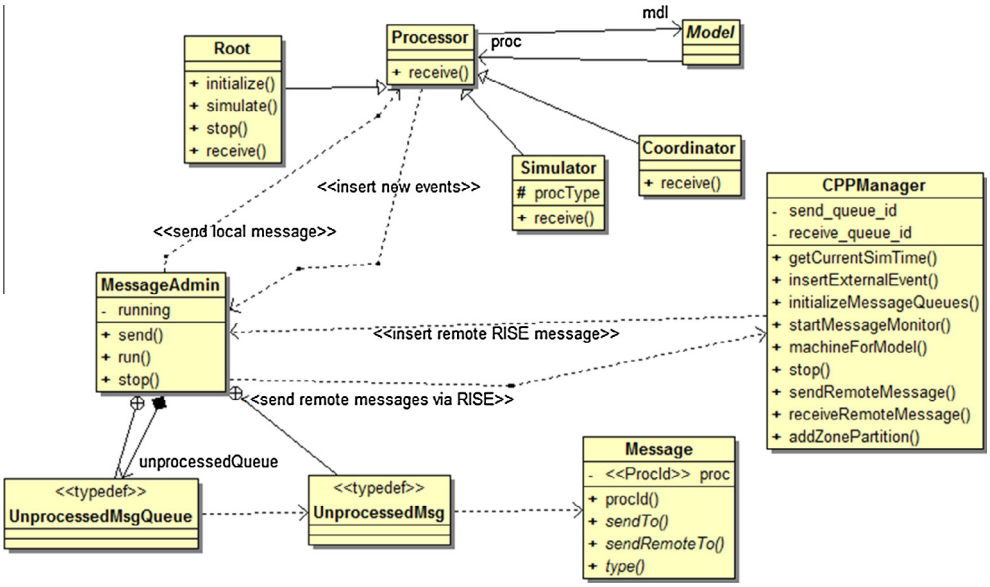

Fig. 10. CD++ processors hierarchy.

are executed by one processor instead of using a processor for each cell in the cell space. The Root coordinator is in charge of starting and stopping the simulation, interacting with the environment, and clock advancement. Example of simulation phases and messages flow between coordinators and simulators was previously described in Fig. 5.

The MessageAdmin class executes events from the UnprocessedMsgQueue by forwarding local events directly to the appropriate processers (i.e. Simulators or Coordinators). On the other hand, the MessageAdmin sends remote events through the CPPManager, which in turns sends it to the RISE middleware via the operating system IPC queues. The CPPManager also inserts received remote messages from RISE (via IPC queues) directly into the UnprocessedMsgQueue (to be later processed by the MessageAdmin).

5. Performance

The main objective of the RISE middleware is the improvement of simulation access and interoperability on the Web. Particularly, decoupling systems design and implementations while allowing composition scalability (i.e. any number of systems can join the distributed environment) and dynamicity (i.e. systems can be created and destroyed at runtime). However, performance still matters, particularly in RISE-based distributed simulation like the presented DCD++ here. This is because distributed simulation is usually performed on the Internet between different partitions over wide geographical area. It is worth to note that the RISE middleware (as a general container) is expected to manage various simulation experiments of different types simultaneously. Thus, managing and balancing the workload (at the RISE middleware level) is highly important to provide best practical performance for those competing sessions above it.

Our approach here is to compare the RISE-based DCD++ (discussed in Section 3) against the SOAP-based DCD++ (discussed in [30]) over a number of different CD++ models using different experiment environment settings. Of course, when comparing two systems always the issue of fairness comes in mind. In our case, because both systems development is under our control, we tried as much as possible to make this comparison as fair as possible, as shortly discussed. It is worth to note that the use of RESTful WS interoperability principles have indirectly contributed to the presented results because REST does not place restrictions on implementations, allowing the programmers to introduce their ideas freely. In REST (which is the Web method), senders build a message and transmit to a URI on the Web, according to the Web standards (without restricting implementation to a certain style). On the other hand, the use of SOAP-based WS is tied to existing software implementations, which usually makes it difficult to introduce new improvements without major software changes. Therefore, it is worth to note that the performance *bottleneck* in both REST and SOAP based distributed simulation system are the same, which is the network messages latency. In both systems, network messages are still transmitted in HTTP. So, both should perform almost the same in similar environments. However, our point is the flexibility of REST interoperability opens the door for more algorithm enhancement, which lead to better performance.

Both systems deployment (i.e. simulation partition on a machine) are shown in Fig. 11. Fig. 11A shows the RISE middleware running as a Servlet (i.e. a program running within an HTTP container) inside the Apache Tomcat HTTP server container [7]. In this case, Tomcat forwards all of the HTTP messages to RISE while RISE processes the messages according to their destination URIs (as discussed in Section 3). In these tests, the RISE middleware is deployed as a Servlet within a Tomcat

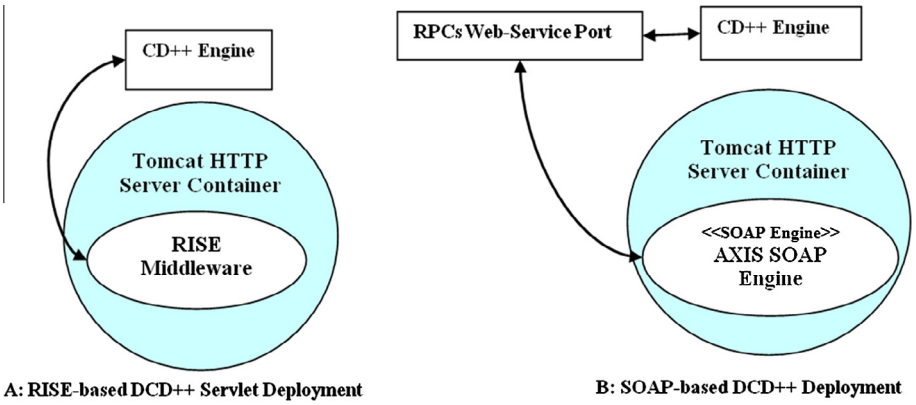

Fig. 11. RISE-based and SOAP-based partition on a machine.

HTTP server. Such deployment is similar to the SOAP-based DCD++, as shown in Fig. 11. It is worth to note that RISE can also be deployed as a standalone HTTP server without the use of HTTP container. Fig. 11B shows the Apache AXIS SOAP engine [36] running as a Servlet inside Apache Tomcat [7]. The SOAP engines translate RPCs to SOAP messages and vice versa, hence they implement the SOAP standards [32]. In this case, Tomcat forwards the HTTP messages to the AXIS engine, which translates the contained SOAP message into a local procedure call in the RPCs Web-service port, which then communicates the passed-in parameters to the CD++ engine.

The experiments in this section used the following five CD++ models (see [11] for more details): (1) Barbershop is a DEVS model simulates a retail barbershop store activities. In this model, customers arrive to the store and have their hair cut by the available barber in First Come First Serve (FCFS) order. (2) Fire is a 2-D 30×30 Cell-DEVS model used to simulate forest fire. The simulation allows foreseeing the propagation and intensity of the fire. Three parameters are involved in the ratio of spread: (A) particles properties (amount of heat, minerals and density), (B) type of fuel (includes the size of the vegetation) and (C) values involved with the natural environment (wind speed, territory inclination and humidity). (3) Ship evacuation is a 2-D 49×27 Cell-DEVS model used to simulate the evacuation of a ship in an emergency. This model has two phases. In Phase 1, each cell calculates its shortest path toward the exit. In Phase 2, people run in their initial direction until they encounter another person or an obstacle (e.g. wall) causing them to change direction. The simulation is completed once all persons leave the ship. (4) Cancer is a 2-D 20×20 Cell-DEVS model is used study cancer spreading on different types of tissue. In this model, cancer tumors invade normal tissues (replacing healthy cells with cancer cells) until cancer is spread over other parts of the body. (5) Battlefield is 3-D $10 \times 10 \times 6$ Cell-DEVS model used to simulate a battle between two armies trying to capture the other flag. This model also simulates different activities such as soldiers' injuries, deaths, movements, and fight engagements.

The presented results were conducted using three different distributed environment setups, as shown in Table 3. In each setup, each model is partitioned into two or more partitions where each partition is assigned to a machine; hence, each partition is simulated by a single CD++ engine on a machine. This allows each run of simulation experiments performance metrics to be collected independently. Note that the RISE-based DCD++ model partitioning mechanism is discussed in Section 3 while the SOAP-based DCD++ partitioning is discussed in [30].

Both systems were compared by setting up the same simulation experiments where a simulation is repeated alternately between both systems over a number of runs. The same experiment setup means that the CD++ model under simulation is partitioned in the same way over the same physical machines. The performance results were then collected based on the following metrics: (1) Number of Remote Messages (NRM) exchanged in a simulation run. This counts the messages that travel through the network within HTTP envelopes. The NRM is the same over the same experiment setup using the same system, because the simulation always executes the same events (deterministic simulation). However, because the RISE-based DCD++ system aggregates remote messages in XML (Section 3), the NRM values are usually different between RISE-based and SOAP-based systems. (2) Total Execution Time (TET) is the average time to complete the simulation of an experiment. The simulation is repeated over a number of runs (usually 25 or more) to achieve at least a 95% Confidence

Table 3
Test environments settings.

Test environment	No. of machines (partitions)	Machines geographical locations
First	2, 3 and 4	All machines attached to the same Ethernet
Second	2 and 3	Placed machines at different locations within the city of Ottawa, Canada
Third	2	One is placed in Ottawa, Canada while the other is placed in Amman, Jordan

Interval (CI). The CI is calculated as the following [8]: the model follows the normal distribution with mean θ and variance σ^2 and the goal is to estimate θ. The natural estimator of θ is the overall mean of R independent replications (runs), that is:

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

where $\overline{Y}$ is the sample mean (average). However, $\overline{Y}$ is not θ but an estimate within ($\pm\mu$). Thus, the usual CI, which assumes the Y_i values are normally distributed, is:

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

Here, σ is the standard deviation of all runs while $t_{\alpha/2,R-1}$ is the quantile of the t normal distribution with $R-1$ degrees of freedom that cuts off $\alpha/2$ of the area of each tail (with probability α). For example, suppose 120 runs have been repeated with $\overline{Y}$ of 5.80 and σ of 1.6. Since, our objective is a 95% of CI, thus, $t_{0.025,\ 119}$ = 1.98 (this value is based on the normal distribution tables in [8]). Therefore, in this case the CI is 5.80 ± 0.29.

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

The TET results clearly show substantial performance improvement for DCD++ simulation via RISE comparing to using SOAP-based WS regardless of the used model or environment setup. For example, in the first environment settings results (Table 5), RISE Speedup ranges from 1.31 to 60.55 times (with 2 partitions setup), 1.56 to 60.40 times (with 3 partitions setup), and 6.62 to 65.15 times (with 4 partitions). Further, the second and third environments showed similar results as shown in Tables 6 and 7 respectively.

Tables (Tables 5–7) show the simulation Total Execution Time (TET) results obtained for the first, second, and third environment settings respectively (see Table 3). The tables results show the model used in the simulation, the number of partitions applied to the model where each partition is assigned to a machine, and both RISE and SOAP TET averages of all repeated runs (in seconds). In this case, the CI (with 95%) is presented in addition to the standard deviation (σ) of all repeated runs. The tables also show the RISE Speedup, which is calculated as SOAP TET ÷ RISE TET. The tables further show the New Partitions Effect for both systems, which is calculated as (Current Partition TET − Previous Partition TET) ÷ Current Partition TET.

The TET results also showed that adding more partitions into the simulation has slowed down the simulation in most cases, but the effect on RISE was much less. For example, adding a third partition to the Ship model simulation (for the

Table 4
Number of Remote Messages (NRM) values.

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

Table 5
First environment test setting TET results.

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

Table 6
Second environment test setting TET results.

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

Table 7
Third environment test setting TET results.

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

second environment shown in Table 6) slowed down the simulation by 0.36 and 0.97 for the RISE and SOAP systems respectively. This simulation slowdown is mainly because the communication synchronization overhead is larger than the local computation overhead by machines.

However, few cases showed that adding another partition (machine) sped up the simulation. For example, adding a third partition for the Cancer model has speeded up the simulation a little bit in the RISE-based simulation as shown in Tables 6 and 7. This means that the synchronization overhead is not large enough to overweigh the benefits of adding more computation power via splitting the model between more machines. Particularly that the Cancer model was able to reduce the NRM value via aggregation by a factor of 21 times as shown in Table 4. Thus, speeding up simulation is possible in distributed simulation but it depends on the environment and the model under simulation characteristics.

Indeed, these results compare both systems implementations and distributed simulation algorithms. As previously mentioned, we tried to make this compression as far as possible by deploying both systems in Tomcat, by using exactly the same CD++ engines, and the same physical machines. However, we are still using the AXIS engine to process the SOAP messages

(not needed in RISE), since we need software to realize the SOAP standards. We further tried to introduce multithreading to the SOAP system by invoking each RPC stub within a thread. This not only proved to be difficult because of the way RPC stubs are structured in AXIS, but also because it requires rebuilding the entire DCD++ WS component. This new implementation would also need to aggregate simulation messages to ensure accurate simulation because of the reasons discussed in Section 3. Consequently, this message aggregation also needs to be sent via the RPC as an array (which proved to be non-trivial to implement), or within an XML message (sent as an attachment). These solutions depend on the use of AXIS; therefore, replacing AXIS with a different vendor implementation would affect our WS solutions. It is worth to note that the implementation coupling between different software solutions in the RPC-style based SOAP WS is a none-trivial issue in practice when introducing new interoperability protocols and algorithms. For example, we had to redesign a new SOAP WS solution in [1] (and put aside the original SOAP WS described in [30]) to ease interoperability between CD++ and other DEVS-based implementations using SOAP WS, as part of DEVS standardization process [33]. These issues show one of the REST major contributions: REST does not place restrictions on implementations, allowing the programmers to introduce their ideas freely. In REST (which is the Web method), senders build a message and transmit to a URI on the Web, according to the Web standards (without restricting implementation to a certain style). On the other hand, the use of SOAP-based WS is tied to existing software implementations, which usually makes it difficult to introduce new improvements without major software changes.

6. Conclusion

We presented the design and implementation of the distributed CD++ (DCD++) simulation, which operates via the RISE middleware. RISE is a general middleware that serves as a container to hold different simulation environments without being specific to a certain environment. We discussed that the RISE middleware three design principles (i.e. general resource-oriented, uniform-interface, and messages-oriented) can enhances distributed simulation interoperability and experimentation. Particularly, hiding interoperating systems heterogeneities (by decoupling systems APIs from internal implementations), composition scalability (by advocating uniform-interface), and dynamicity (since information channels automatically exist). We also showed how these principles have been used in the DCD++ design, particularly, enhancing algorithms performance via simulation message aggregation in XML.

Thus, the RISE-based DCD++ presented here is a proof-of-concept showing that RISE principles ease interoperability (when compared to existing Web-based distributed simulation tools). Those principles include:

(1) Decoupled simulation algorithms from the middleware versus the mixed approach in other existing systems. The RISE middleware is a general container that is not specific to any system [2].
(2) All of the exchanged information between simulation partitions is transmitted via uniform, universal standardized, constant virtual channels. Each component in the distributed environment is always connected with other component with the same number of channels. This is important to achieve scalability when the number of simulation partitions increases. Further, virtual channels are represented as fields in the exchanged message headers, hence putting the channels' representations outside the components' implementations, which is important for decoupling the component implementations. In contrast, existing approaches use Remote Procedure Call (RPCs) channels to exchange information. RPCs are usually a part of the systems implementation, making it difficult to decoupling the components from their implementations. They are also variable and heterogeneous since they are designed by different programmers.
(3) All of the exchanged information is described in Extensible Markup Language (XML) messages while other approaches exchange information as programming parameters. XML is important to enhance systems decoupling and make it less sensitive to programming.
(4) Experiments are dynamically created and attached to the Web. This is because experiments are interfaced with standard URIs where those URIs are created on the fly with any value. In fact, easing interoperability by decoupling implementations from algorithms and exchanging semantics in XML has also showed substantial performance improvements when compared to other interoperability approaches as we show here.

To summarize, easing interoperability constraints open the way for further simulation algorithms enrichment even when interoperating similar systems. RISE-based DCD++ simulation system is the first system to perform distributed simulation using the RESTful Web-services (WS) interoperability style, hence being part of a Cloud. It does so by operating as a service on top of the RISE (RESTful Interoperability Simulation Environment) middleware. RISE is a general middleware that serves as a container to hold different simulation environments without being specific to a certain environment. RISE-based DCD++ shows as a "proof-of-concept" that RISE easing interoperability principles are not only achievable but also can open the way for more algorithms innovation, which can lead to better performance. These principles are important in the area of DEVS standardization because they decouple implementations, leaving standards to focus only on semantics. It is worth to note that this is an article on the extensions to the CD++ tool in order to be able to execute distributed simulation models using the RISE middleware. This is the first (and as far as we know) the only system to perform distributed simulation based on RESTful WS style. The other system named DCD++ described in [30] is one of our previous work from 2008 to 2010, and this was a completely different middleware, based on SOAP-based WS.

Finally, as we discussed throughout this paper that current SOAP-based WS interoperability constraints can complicate interoperability in Web-based distributed simulation. On the other hand, using the RESTful WS (which is the Web interoperability style) can achieve Web-based distributed simulation, but without such constraints.

References

[1] K. Al-Zoubi, G. Wainer, Interfacing and coordination for a DEVS simulation protocol standard, in: Proc. 12th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications (DS-RT2008), 2008, pp. 300–307.
[2] K. Al-Zoubi, G. Wainer, RISE: a general simulation interoperability middleware container, J. Parallel Distrib. Comput. Elsevier 73 (5) (2013) 580–594.
[3] K. Al-Zoubi, G. Wainer, Performing distributed simulation with RESTful Web-services, in: Proc. 2009 Winter Simulation Conference (WSC 2009), 2009, pp. 1323–1334.
[4] K. Al-Zoubi, G. Wainer, Using REST Web services architecture for distributed simulation, in: Proc. 23rd ACM/IEEE/SCS Proceedings of Principles of Advanced and Distributed Simulation (PADS2009), 2009.
[5] Amazon elastic compute cloud (Amazon EC2), <http://aws.amazon.com/ec2/> (accessed February 2015).
[6] K. Anderson, J. Du, A. Narayan, A. El Gamal, GridSpice: a distributed simulation platform for the smart grid, IEEE Trans. Industr. Inf. 10 (4) (2014).
[7] Apache Tomcat, <http://tomcat.apache.org/> (accessed February 2015).
[8] J. Banks, J. Carson, B. Nelson, D. Nicol, Discrete-Event System Simulation, Pearson Prentice Hall, Upper Saddle River, NJ, 2009.
[9] C. Boer, A. Bruin, A. Verbraec, A survey on distributed simulation in industry, J. Simul. 3 (1) (2009) 3–16.
[10] A. Boukerche, F. Iwasaki, R. Araujo, E. Pizzolato, Web-based distributed simulations visualization and control with HLA and Web services, in: Proc. 12th IEEE Int'l Symp. Distributed Simulation and Real-Time Applications (DS-RT '08), 2008, pp. 17–23.
[11] CD++ toolkit, <http://cell-devs.sce.carleton.ca> (accessed January 2015).
[12] Cisco Cloud Computing, <http://www.cisco.com/c/en/us/td/docs/routers/csr1000/software/restapi/restapi.html> (accessed June 2014).
[13] A. Chow, B. Zeigler, Parallel DEVS: a parallel, hierarchical, modular modeling formalism, in: Proc. 1994 Winter Simulation Conference (WSC1994), 1994, pp. 716–722.
[14] R. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD dissertation, Dept. of Computer Science, Univ. of California, Irvine, CA, USA, 2000.
[15] R. Fujimoto, Parallel and Distribution Simulation Systems, John Wiley & Sons, New York, 2000.
[16] J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, URI Templates, <http://tools.ietf.org/html/draft-gregorio-uritemplate-04. 2010> (accessed June 2011).
[17] IBM cloud Computing, <http://www.ibm.com/developerworks/cloud/library/cl-RESTfulAPIsincloud/index.html?ca=dat> (accessed June 2014).
[18] D. Kimmig, T. Brenner, K. Bittner, A. Schmidt, Towards a Web based modelling and simulation tool for research, engineering and education in the field of hydrogen and fuel cell technology, in: Proc. 2014 IEEE Computational Science and Computational Intelligence (CSCI2014), 2014, pp. 193–196.
[19] P. Ke, S. Turner, C. Wentong, L. Zengxiang, A service oriented HLA RTI on the grid, in: Proc. 2007 IEEE International Conference on Web Services (ICWS 2007), 2007, pp. 984–992.
[20] F. Khul, R. Weatherly, J. Dahmann, Creating Computer Simulation Systems: An Introduction to High Level Architecture, Prentice Hall, 1999.
[21] E. Mancini, G. Wainer, K. Al-Zoubi, O. Dalle, "Simulation in the cloud using handheld devices, in: Proc. 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid2012), 2012.
[22] M. Jarrah, B. Zeigler, A modeling and simulation-based methodology to support dynamic negotiation for web service applications, J. Simul. 88 (3) (2012).
[23] Noelios Restlet Engine (NRE), <http://www.noelios.com/products/restlet-engine> (accessed January 2015).
[24] T. O'Reilly, What Is Web 2.0, <http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html> (accessed January 2015).
[25] L. Richardson, S. Ruby, RESTful Web Services, O'Reilly Media Inc, Sebastopol, California, 2007.
[26] Restlet API, <http://www.restlet.org/> (accessed January 2015).
[27] D. Rorich, M. Bernhard, T. Handte, S. Brink, Webdemos: an interactive, web-based visualization and simulation framework for open access, in: Proc. of the 2014 IEEE International Conference on Web and Open Access to Learning (ICWOAL2014), 2014, pp. 1–6.
[28] Y. Shi, D. Zhang, M. Xiao, M. Lu, SOA-based simulation framework: a way to simulation composability, in: Proc. 2012 IEEE International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2012, pp. 232–236.
[29] S. Strassburger, T. Schulze, R. Fujimoto, Future trends in distributed simulation and distributed virtual environments: results of a peer study, in: Proc. 2008 Winter Simulation Conference (WSC2008), 2008, pp. 777–785.
[30] G. Wainer, R. Madhoun, K. Al-Zoubi, Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web services, Simul. Model. Pract. Theory 16 (9) (2008) 1266–1292.
[31] G. Wainer, Discrete-Event Modeling and Simulation: A Practitioner's Approach, CRC Press, Taylor & Francis Group, Boca Raton, Florida, 2009.
[32] G. Wainer, K. Al-Zoubi, An introduction to distributed simulation, in: C. Banks, J. Sokolowski (Eds.), Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains, Wiley, New York, 2010, pp. 373–402.
[33] G. Wainer, K. Al-Zoubi, S. Mittal, J. RiscoMartín, H. Sarjoughian, B. Zeigler, in: G. Wainer, P. Mosterman (Eds.), Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press. Taylor and Francis, 2010, pp. 389–494. Chapters 15–18.
[34] G. Wainer, K. Al-Zoubi, S. Mittal, J. RiscoMartín, H. Sarjoughian, B. Zeigler, in: G. Wainer, P. Mosterman (Eds.), Discrete-Event Modeling and Simulation: Theory and Applications, CRC Press. Taylor and Francis, 2010, pp. 389–494.
[35] S. Wang, G. Wainer, A simulation as a service methodology with application for crowd modeling, simulation and visualization, Simulation 9 (1) (2015) 71–95.
[36] Web-services AXIS, <http://ws.apache.org/axis/java/user-guide.html> (accessed February 2015).
[37] Q. Xiang, G. Chen; Y. Wang, Distributed simulation based on web enabling HLA, in: Proc. 2nd IEEE International Conference on Artificial Intelligence, Management Science and Electronic Commerce, 2011.
[38] W. Xiong, W. Tsai, HLA-based SaaS-oriented simulation frameworks, in: Proc. of the 2014 IEEE 8th International Symposium on Service Oriented System Engineering (SOSE2014), 2014, pp. 376–383.
[39] B. Zeigler, H. Praehofer, T. Kim, Theory of Modeling and Simulation, Academic Press, San Diego, CA, 2000.
[40] S. Zhu, Z. Du, X. Chai, GDSA: a grid-based distributed simulation architecture, in: Proc. 6th IEEE Int'l Symp. on Cluster Computing and the Grid Workshops (CCGRID2006), 2006, pp. 66–71.