DISTRIBUTED SIMULATION OF DEVS AND CELL-DEVS MODELS IN CD++ USING WEB SERVICES

Gabriel Wainer **Rami Madhoun** **Khaldoon Al-Zoubi**

Department of Systems and Computer Engineering
Carleton University
Ottawa, ON K1S-5B6 Canada

http://www.sce.carleton.ca/faculty/wainer

ABSTRACT: *DEVS is a Modeling and Simulation formalism that has been widely used to study the dynamics of discrete event systems. Cell-DEVS is a DEVS-based formalism that defines spatial models as a cell space assembled of a group of DEVS models connected together. CD++ is a modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models that has proven to be useful for executing complex models. We present the design and implementation of a distributed simulation engine, known as D-CD++, which exposes CD++ simulation utilities as machine-consumable services. In addition, we present the design and implementation of the Web-Service components which enable D-CD++ to expose the simulation functionalities to remote users. Enabling CD++ with Web-Services technology provides a solid framework for interoperating different DEVS implementations in order to achieve a standard DEVS modeling language and simulation protocols. This paves the road towards DEVS standardization, while providing a mashup approach, which can lead to higher degree of reuse and reduced time to set up and run experiments, and making sharing among remote users more effective. To prove this fact, we integrate it within larger services (such as a 3D visualization engine), showing the mechanism to incorporate to other environments (including Geographical Information Systems, web-based applications and other Modeling and simulation tools) through using standard Web-Service tools. Performance of D-CD++, major bottlenecks and communication overheads are analyzed.*

1. Introduction

Discrete Event System Specification (DEVS) [Zei00] is a modeling and simulation formalism that provides means for modeling discrete event systems as hierarchical and modular components. Cell-DEVS [Wai08, Wai02b] is an extension that allows for representing each cell in the cell space as a DEVS model that is only activated when it receives external inputs from its neighboring cells. This improves the performance of the simulation since only active cells are evaluated. DEVS and Cell-DEVS have been successfully used to model complex systems in varied field, ranging from physics, environmental sciences, traffic, defense, biology, networking, etc. [Wai08, Zei07].

CD++ [Wai02a] is a modeling and simulation toolkit developed to execute DEVS and Cell-DEVS models. It follows the definition of the DEVS abstract simulator [Zei00] in that there are two separate class hierarchies: one for representing the model and the other for representing the simulator. The simulation is carried out by processing events by the simulators via advancing the simulation clock to the timestamp of the event that is about to be processed. Different versions of CD++ have been developed to work on different platforms; the stand-alone version runs on regular workstations, PCD++ [Tro03, Liu07a] runs on high performance distributed-memory clusters, and the real time version runs on specialized hardware in a real-time environment [Yu07].

As the model complexity tends to increase, more resources are required for simulation, in which case using a single machine may be impractical. At the same time, as more systems got connected through the Internet, a framework to integrate their resources to execute complex models started to gain the attention of the research community. In recent years grid computing paradigm permitted sharing compute and storage resources in heterogeneous environments where resources reside on different platforms connected together using standard communication protocols [Tra06]. In a grid environment, resources are virtualized as services that are consumed by clients to be provided with compute and storage "services" on demand, with minimal or no limitation to the platform on which these resources reside. Therefore, grid computing offers a huge power via sharing computing power and ideas among researchers around the globe to solve problems in a fast, easy and dynamic way. Nevertheless, in most cases, grid computing environments process jobs off-line, providing speedups through replication of data and computing

resources, which is not adequate for simulations where the players are connected on-line to the platform and require interactivity. Likewise, simulation interoperation requires complex ad-hoc tailoring.

Grid computing environments are therefore not adequate in those cases in which each of the players have partial knowledge, are located remotely, and need online collaboration. The quintessential case was the SARS outbreak, in which scientists in Canada, China, the US and Europe worked independently in their labs, using local facilities. In such cases, the ability to share models, simulations and experiments used by each team could have provided a quicker solution to the problem. This type of collaboration involves two separate needs: the execution of simulations in distributed fashion (including interoperation between simulators and their combination into larger experiments), and the sharing of models and their experimental frames between the participants (from small subcomponents up to complete experiments). Service Oriented Architecture (SOA) technologies and Web-Services (WS) have proven to be useful to provide interoperability by creating a system designed to support machine-to-machine interaction over a network. Extending CD++ via enabling users to access the CD++ simulation functionalities as machine-consumable services is in fact a step towards full resources sharing. Granting only service access to those resources is acceptable in many cases, since external users (i.e. clients) only see those resources as black box. The main concern of these users is, in general, the accessibility to those valuable resources. On the other hand, power processing and speedup execution can still play a major role for other users. This has motivated our team to create a new version of CD++ (named D-CD++) to execute in distributed environments using WS technology.

By providing WS along with distributed simulation algorithms, D-CD++ can be easily integrated to other environments using a mashup approach (mashups use WS to combine data and services provided by third parties, either explicitly through open APIs, or inferred by the mashup author). This allows us to integrate D-CD++ with environments like Eucalyptus [Liu07b], Google maps [Har08], or the DoD GIG/SOA [Zei07]. In such environments (where each organization has developed its own standards) WS can help in dealing with compatibility issues. In our case, it paves the way to make different DEVS tools to work not only with each other but with other non-DEVS tools within the DEVS Standardization effort [Wai07]. Therefore, standardization and interoperability should be realized according to a shared agreement at two stages: the first one between DEVS based tools, and the second between DEVS and non-DEVS based tools. Further, interfacing DEVS tools must be done at two levels (since the DEVS theory separate modeling from simulation): at the simulator level (e.g. synchronization simulation clocks, passing control simulation messages, etc.) and at the model level (e.g., using the ideas defined by DEVS-Bus [Zei00] or the DEVS Modeling Language, DEVSML [Mit07a]). The eXtensible Markup Language (XML) is the standard data format *syntax* on the Internet; hence as a natural choice to exchange data in XML. However, to interoperate between different DEVS implementation, an agreement must also be achieved on what the data encoded in XML format refers to – *semantics* agreements. SOA can help in this context, as it is intended for exchanging raw data in XML format (in an open data-shared, interface-driven, distributed environment).

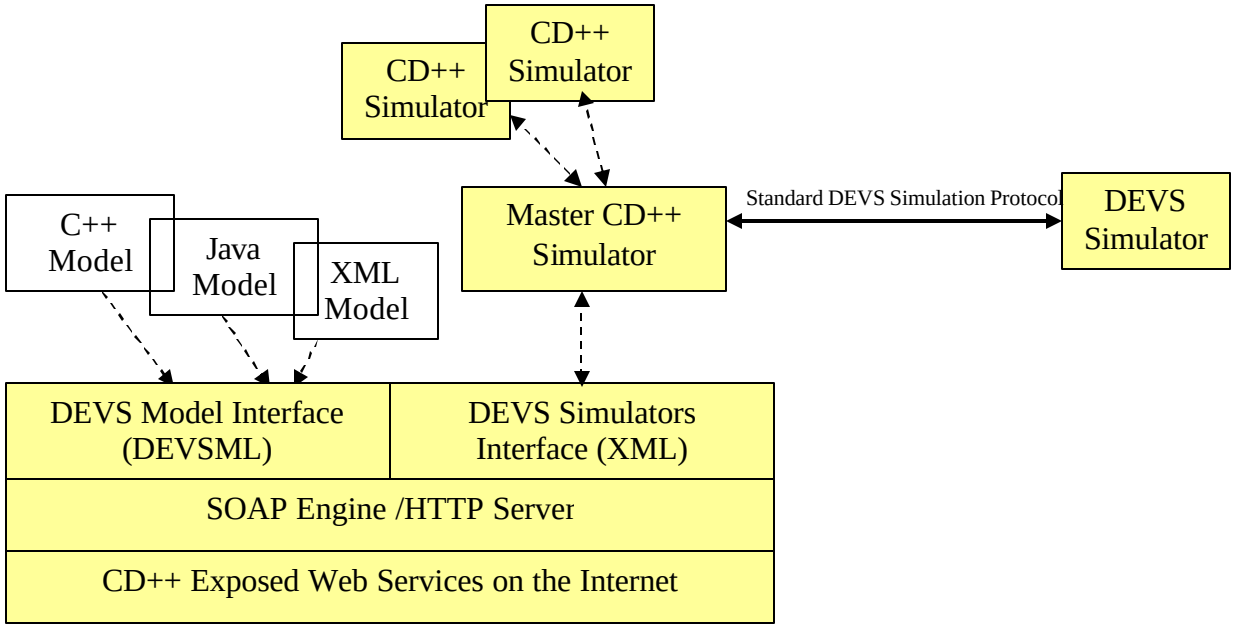

Figure 1: Framework of Interfacing CD++ according to DEVS Standard Protocol over SOA

Figure 1 illustrates the intended vision of interoperating other DEVS tools with D-CD++ to overcome incompatibilities due to different implementations at both the simulation and model levels. The figure shows a master CD++ simulator coordinating two CD++ slave simulators and another non-CD++ DEVS simulator where all of the communication is performed via exchanging XML messages (over SOAP/HTTP). D-CD++ must follow a standard XML semantics when communicating with a non-CD++ simulator. The figure also shows interoperability at the model level where a DEVS model interface (for instance, written in DEVSML) enables it to execute other DEVS models developed with other DEVS tools.

In order to allow such interoperation, CD++ was extended in two fronts: it was enabled with WS, making its services exposed and consumed using WS technology, and it was extended to execute models in a distributed environment via exchanging data in XML messages (specifically SOAP messages over HTTP). These extensions are presented in this paper, which is organized as follows. Section 2 introduces the P-DEVS and Cell-DEVS formalisms and provides the necessary background knowledge on the CD++ environment, Web-Services and commonly used WS technologies. It also gives a brief survey on the existing DEVS-based toolkits intended for parallel and distributed simulation. Section 3 describes the WS extension to CD++ environment. Section 4 discusses the new the design and implementation to the distributed simulation engine extension to the CD++ toolkit. Section 5 describes the experimental environment and the metrics we used to investigate the distributed simulation system. It also presents a detailed performance analysis.

2. Background

Discrete Event System Specification (DEVS) [Zei00] is M&S specification that is aimed to study discrete event systems. A DEVS model consists of hierarchical and modular components connected together through external port(s) where events are exchanged among models via those ports. As in any discrete-event simulation, the models being simulated change of state only at discrete points in time, upon the occurrence of an event.

The P-DEVS formalism [Zei00] expresses a system as a number of connected behavioral (atomic) and structural (coupled) components. The basic building block of DEVS models is the *atomic DEVS model*. A P-DEVS atomic model is formally defined as:

$$M = \langle X, Y, S, d_{int}, d_{ext}, d_{con}, ?, ta \rangle$$

X is the set of input values;
S is the set of states;
Y is the set of output values;
δint: S ? S is the internal transition function;
δext: Q x X^b ? S is the external transition function, where
X^b is a set of bags over elements in X,
$\delta_{ext}(s, e, f) = (s, e)$;
δ_{conf}: S x X^b ? S is the confluent transition function;
?: S ? Y^b is the output function;
ta: S ? R^+_0;
where
Q = {(s, e) | s ∈ S, 0 ≤ e ≤ ta(s)} is the total state set and e is the time elapsed since last transition;

At any given time, an atomic model is in some state $s \in S$, and it stays in this state for the period specified by the state time advance function *ta(s)*, with the assumption of not receiving external events. When the atomic model state lifetime expires, the model outputs the value $?(s) \in Y$, and it changes its state as indicated by the internal transition function $d_{int}(s)$. A P-DEVS model uses bag of inputs (X^b) to exploit parallelismin the system, hence execute multiple concurrent events simultaneously. If the atomic model receives one or more external events $x \in X$ before the expiration of *ta(s)* the model also changes its state as defined by the external transition function $d_{ext}(s,e,X^b)$. A confluent transition function (d_{con}) is used to conclude the model's next state via resolving the collisions when receiving external events and internal transitions simultaneously.

A *coupled DEVS model* consists of atomic and/or other coupled models connected together through their input and output ports. A P-DEVS coupled model is formally defined as:

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

Both X and Y define the sets of input and output events respectively. D is an index of components of a coupled model and, for each $d \in D$, M_d is a basic P-DEVS model (atomic or coupled). 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 components themselves are defined by the internal coupling (IC). Thanks to the property of closure under coupling, a coupled model can be reduced to a behaviorally equivalent atomic model, and thus be treated as a basic component in the construction of more complex hierarchical models.

Cell-DEVS [Wai08, Wai02b] describes n-dimensional cell spaces as discrete-event DEVS coupled models, where each cell is represented as a DEVS atomic model. Furthermore, it defines timing constructions rules for each cell, allowing explicit timing specification, asynchronous model execution, and integration with other types of models. A Cell-DEVS atomic model is formally defined as:

$$C = \langle X, Y, I, S, ?, N, delay, d, d_{int}, d_{ext}, t, ?, D \rangle$$

Each cell has a modular interface (I) that is composed of a fixed number of ports connected to its neighboring cells. The future state of a cell is computed by the local transition function (t) based on the cell's current state and input values. State changes are spread only after a delay given by the delay function (d). Each cell also has the computing apparatus (d_{int}, d_{ext}, and ?) as defined in P-DEVS atomic models. Cells are coupled by the neighborhood relationship to form a cell space, which can then be integrated with other DEVS and Cell-DEVS models. A cell space is formally defined as a Cell-DEVS coupled model:

$$GCC = \langle X_{list}, Y_{list}, I, X, Y, h, \{t_1, \ldots, t_n\}, N, C, B, Z \rangle$$

The cell space (C) consists of a fixed-sized n-dimensional array of cells, and the relative position between each individual cell and its surrounding neighbors is defined by the neighborhood set (N). B specifies the border of the cell space, which can be wrapped (i.e., all cells have exactly the same behavior) or non-warped (i.e., the border cells have a different behavior from others in the cell space). The translation function (Z) defines the input/output coupling between the cells.

CD++ [Wai02a, Wai08] is a modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models. CD++ provides a simulation framework that creates an executive entity for each component in the modeling hierarchy to implement the abstract simulator that is responsible for executing the simulation in line with the formalisms [Zei00]. The simulator architecture is shown in Figure 2 (a detailed description is given in Figure 17, Figure 21 and Figure 22).

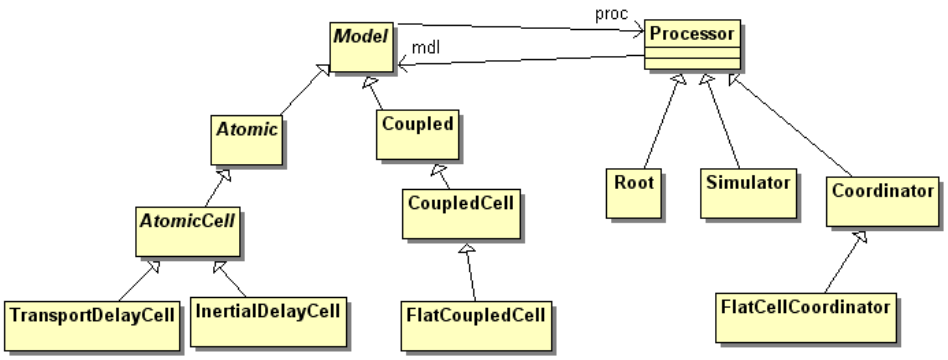

Figure 2: CD++ Model and Processor Hierarchies

As we can see, there are two interrelated object hierarchies: the *models* and the *processors* in charge of executing those models. Each *atomic* model is executed by a *simulator*, and each coupled model is executed by a *coordinator*. The simulation is driven by the *Root* processor which is responsible for (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. The *Simulator* class executes an atomic DEVS model by (1) reacting to the different types of received messages via invoking the appropriate function within the atomic DEVS class. The *coordinator* class manages a DEVS coupled model through routing messages between its children and its *parent-coordinator*. In addition, it evaluates the minimum time for the next change of its children to be passed to the *Root* coordinator. The *CellCoordinator* is responsible for message routing among the cells within a coupled Cell-DEVS model. The simulation is carried out in a message-driven fashion. Messages exchanged in a simulation fall into two classes: content messages include the external message (X, t) and output message (Y, t) that encode the actual data transmitted between the models, while control messages include the initialization message (I, t), collect message (@, t), internal message (*, t), and done message (D, t) that are used internally by the simulator to synchronize and control the simulation. Each message represents an event with an associated timestamp that indicates the simulated virtual time of the event.

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. Each atomic DEVS model uses four methods: *externalFunction()*, *internalFunction()*, *outputFunction()*, *holdIn()*, which correspond to the defined functions in the DEVS formalism. 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.

Figure 3 shows an example of a DEVS coupled model that represents a car factory including different warehouses and assembly lines. Each of the four sub-factories sends their completed components to the Final Assembly Sub-factory where the automobile is assembled.

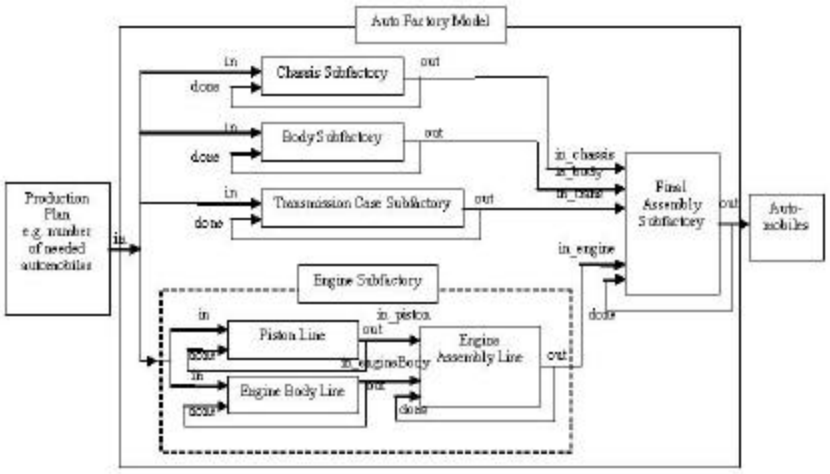

Figure 3: The auto factory coupled model example

Each atomic model must be implemented in C++, as a class derived from *atomic* (and overloading the methods just described). Once every component's atomic model is implemented, the CD++ coupled model for the structure is defined, as shown in Figure 4 (which follows the structure presented in Figure 3).

```
[top]
components : chassis@Chassis body@Body trans@Trans finalAssem@FinalAssem engineSubFact
out : out
in : in
Link : in in@chassis
Link : in in@body
Link : in in@trans
Link : in in@engineSubFact
Link : out@finalAssem out
Link : out@finalAssem done@finalAssem
```

```
...
Link : out@trans done@trans
Link : out@engineSubFact in_engine@finalAssem

[engineSubFact]
components : piston@Piston engineBody@EngineBody engineAssem@EngineAssem
out : out
in : in
Link : in in@piston
Link : in in@engineBody
...
Link : out@engineAssem out
Link : out@engineAssem done@engineAssem
```

Figure 4: The auto factory coupled model CD++ definition

As discussed in the Introduction, we are interested in building a *Web-Services* are group of standards and languages aiming to facilitate developing, publishing, and discovering web-enabled applications. In other words, a WS is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-understandable format (specifically Web Service Description Language WSDL [Chr01]). Client systems interact with the WS in a manner prescribed by its description using SOAP messages [Gud03], typically conveyed using HTTP with XML serialization in conjunction with other web-related standards [Alo03]. Although WS are usually deployed using HTTP as application layer protocol, they could similarly be used on top of other protocols such as SMTP. The reason for using HTTP is that it is familiar to most users and usually passes through company's firewalls without causing much administration or management overhead.

WSDL documents include information for the WS clients to know the operations it offers, parameters required to invoke an operation, and return type. The major elements of any WSDL document are type, message, portType, binding, port, and service elements. Some of these elements (type, message, and portType) are used to describe the functional behavior of the WS in terms of the functionality it offers. On the other hand, binding, port, and service define the operational aspects of the service, in terms of the protocol used to transport SOAP messages and the URL of the service. The former is referred to as abstract service definition, and the latter is known as concrete service definition.

SOAP plays an important role in any WS transaction. It is the messaging protocol used to convey information to and from the WS. It was designed to provide decentralized communication among multiple parties. The structure of SOAP messages is based on XML and it consists of an *Envelope* element at the root of the XML document. The *Envelope* element is composed of an optional *Header* element and a mandatory *Body* element. Figure 5 shows an example of SOAP HTTP binding.

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<m:GetLastTradePrice xmlns:m="Some-URI">
<symbol>DIS</symbol>
</m:GetLastTradePrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 5: An example of a SOAP message embedded in HTTP [Gud03]

As shown in the figure, the HTTP POST request specifies three HTTP headers: Content-Type, Content-Length and SOAPAction. The *Content-Type* defines the Multipurpose Internet Mail Extensions (MIME) type for the message and the character encoding (optional) used for the XML body of the SOAP request/response. The *Content-Length* specifies the number of bytes in the body of the SOAP request or response. In this example the namespace for the

specified function is defined in "http://www.stockquoteserver.com" address. Afterward, the SOAP Envelope element is the root element of the SOAP message where in this example it defines the request “GetLastTradePrice”.

Once the web server receives the HTTP request containing the SOAP message (i.e. as in the case of any other HTTP request), the SOAP message is extracted from the HTTP request and forwarded to a SOAP engine, which is responsible for processing messages and converting the request(s) into a method call(s) that the service implementation code can understand. This process is referred to as *unmarshalling (deserialization)*. The service implementation code is responsible for implementing the logic of the WS. Once processed, the result is handed to the SOAP engine to build response to be sent back to the client. This is referred to as *marshalling (serialization)*. The web server encapsulates the SOAP response into HTTP packets that are sent to the client. The SOAP engine (Figure 6) by itself is an application that runs within an application server installed as part of the WS deployment process.

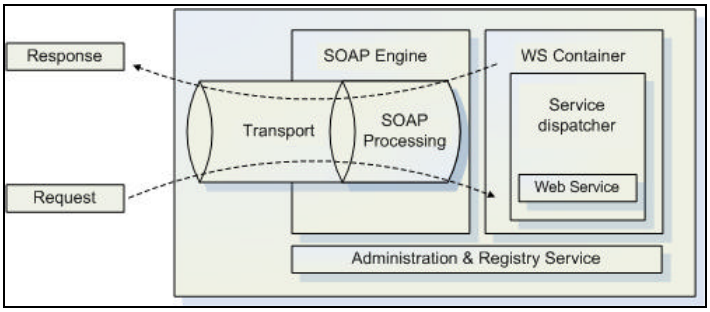

Figure 6: A Web-Service container [Glo05]

The success of the DEVS/Cell-DEVS formalism in modeling and simulating different complex systems, has attracted a lot of researchers to extend the basic abstract simulator presented in [Zei00] into a parallel/distributed one. Different groups of researchers have studied the implementation of such DEVS simulators (which are the basis for our distributed simulation engine) in varied parallel and distributed environments, including:

- **DEVS/HLA** [Zei99] is an HLA-compliant environment implemented in C++ that supports high-level model construction. It simplifies the programming effort required to establish and participate in an HLA federation.
- **DEVS/Grid** [Seo04] implements a grid-enabled DEVS simulator consisting of five layers: *application*, *modeling*, *simulation*, *middleware* and *network*. The application layer (top layer) deals with high level issues within the application domain. The modeling layer provides the required functionality for defining a model; the simulation layer is responsible for running the DEVS simulation with the support of other tools and utilities. The middleware layer is responsible for the discovery and management of the resources available in the grid. The network layer represents the hardware resources available in the grid which might include storage devices, workstations, and high-performance clusters.
- **vGrid** [Kha03] is an overall architecture for running DEVS models in grid environments. vGrid divides the model into components; the *Fine Computational Unit (FCU)* corresponds to an atomic DEVS model. Several *FCUs* can be grouped together to form a *Virtual Computational Unit (VCU)* which constitutes the basic component that can be scheduled on a single grid resource. The *vGrid Manager (VGM)* manages the resources in a grid environment with coordination with the other engines. It interacts with the *VCUs* through *Autonomous Wrappers*.
- **DEVS/P2P** [Che04] is a distributed DEVS simulator aimed to peer-to-peer networks. It exploits JXTA [JXT06] as an implementation of P2P communication middleware with the DEVS modeling and simulation capabilities. It consists of four major parts; the *Automatic Hierarchal Model Partitioning (AHMP)*, *Automatic Model Deployment (AMD)*, *Activator*, and *Generic Simulator (GS)*. *AHMP* is responsible for partitioning the DEVS model. The partitions are deployed in the host machines by the *AMD*. The A*ctivator* is responsible for receiving a model partition and creating by *GS* that runs the simulation.
- **DEVS/RMI** [Zha05] is a distributed DEVS simulator based on Java Remote Method Invocation (RMI). It provides fully re-configurable distributed simulation environment with the capability of load-balancing and fault-tolerance. A S*imulation Controller* is responsible for controlling the activities taking place during the

simulation evolution. This includes taking the partition information generated by the *Configuration Engine* and transferring it to the host machines to be executed by *Remote Simulators*. A *Simulation Monitor* collects information about the model being executed and conveys this information to the configuration engine to recreate the model partitions (if necessary).

- **DEVS/Cluster** [Kim04] is a multi-threaded distributed DEVS simulator based on CORBA. The simulator adopts a flattened approach by having one coordinator responsible for all the simulators running on a particular machine (as opposed to having a coordinator for each coupled model). In addition, it uses the facilities provided by CORBA in order to pass messages between simulators by direct remote method invocations, instead of sending/receiving explicit messages. **DEVS/Cluster-WS** [Kim05] is an extension of the DEVS/Cluster that uses WS technology such as SOAP and WSDL.
- **PCD++** [Tro 01, Liu07a] is a parallel simulation engine developed using WARPED [Rad98] middleware. It is able to execute DEVS and Cell-DEVS models. The original version of PCD++ followed a hierarchical approach for the simulation and it uses a conservative algorithm for synchronization among the nodes. An improved version of PCD++ was developed as a flat simulator dispensing with the need to have a coordinator for every coupled DEVS model, and hence improving the overall performance of the simulator. In addition, PCD++ uses Time Warp protocol [Fuj99] for synchronization among the different nodes participating in the simulation.
- **SOADEVS** [Mit07b] focuses on the interoperability at the application level, particularly, at the client level and hides the whole simulation engines. DEVS Modeling Language (DEVSML) [Mit07a] aims on standardizing DEVS models among different implementation, which an XML notation for representing models that can be shared between the different participant labs around the world. These models are subsequently realized on different implementations (based on Java, C++ or other languages), using a net-centric infrastructure.

The difference between the approaches followed by other grid-based DEVS simulation engines and our proposal, is that we aim to implement the simulation services in a modular manner to provide the flexibility required for integration with larger systems with minimal or no changes to the simulation services. A different goal is to provide a mechanism for easy mashing up of web services, which has not been implemented up to date.

3. D-CD++: a Web-Service Enabled CD++

The D-CD++ tool enables distributed simulation of DEVS models using commodity Internet connections or high-speed point-to-point optical networks. Resources are *exposed* as WS (and they are deployed as WSDL documents). Clients can then build the client stubs from the WSDL document by simply invoking the appropriate tools provided by a SOAP engine), which enables the client to *consume* the deployed services by simply calling services stubs as if they were local methods.

The global-scale distributed extension enables CD++ to run models among various machines regardless of their location. In fact by using WS, a machine can communicate with another machine in a standard, uniform way by calling it via its URL. Therefore, in the context of our modeling and simulation environment, WS are introduced to:

i) Enable CD++ to be integrated in an open market environment where services are exposed and consumed according to widely accepted standards in the Internet, allowing for executing simulations and retrieving the results through Web-Service technologies, as shown in Figure 7.
ii) Achieve distributed simulation on a global scale using WS technology.

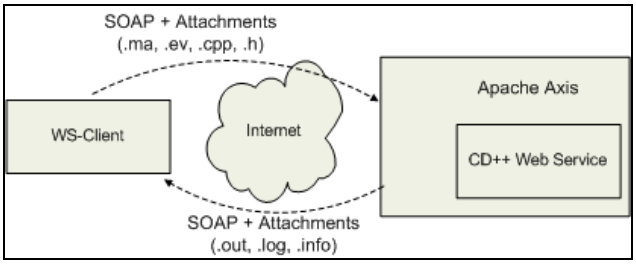

Figure 7: A typical invocation of the simulation Web-Service

Using the deployed services, the client needs to submit the required files before starting the simulation. D-CD++ is capable of executing two kinds of models, DEVS and Cell-DEVS. As discussed in Section 2, to run DEVS models the modeler needs to define each DEVS atomic model as a C++ class that is to be integrated in the class hierarchy of CD++. For coupled DEVS models, and Cell-DEVS models, the modeler needs to provide a model definition file (as in Figure 4) that includes the coupling scheme, initial values for the cells, rule definition to calculate the state of the cells, etc. In a regular invocation of CD++, the user submits the model definition and configuration files to the simulator as arguments. Once the simulation is over, the user gets the results in the form of output and log files (containing the events that were generated through the output ports of the model and detailed information about the progress of the simulation [Wai08]). D-CD++ transmits all input/output files as SOAP attachments and the simulation results are transmitted back to the client as SOAP attachments.

In the next subsection we elaborate more on the CD++ WS basic infrastructure. Afterwards, we discuss the D-CD++ provided WS, design and implementation [Mad07a].

3.1 Web-Services Architecture, DEVS standardization and Visualization Environment

CD++ distributed simulation through WS extension (presented in this paper) is intended to be provided as consumable services in on-demand provisioning environment through WS where users can share among themselves their models, experiments, etc. as shown in Figure 8.

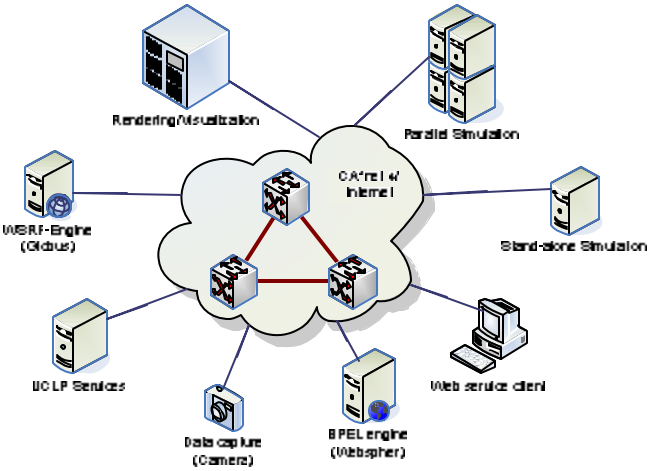

Figure 8: On Demand provisioning environment for CD++ simulation and visualization services

The overall network runs on commodity Internet connection, or using the power of User Controlled Lightpath (UCLP) which provides end-to-end lightpaths [Arn03]. UCLP can be viewed as a partition manager that treats each lightpath (and its associated network elements) in a physical network as a "service" that can then be controlled by different network users to construct their own logical IP network topologies. Eucalyptus [Liu07b] is an application built on UCLP, and it provides a WS-based solution for provisioning resources on demand. Eucalyptus divides services into two categories: task-oriented (i.e. the ability to perform a task such as starting a simulation session) and management services to support the task-oriented services. Constructing such environment (which involves different research labs in North America and Europe) was performed in an incremental development, experimental and testing. D-CD++ is a piece of this incremental construction approach which allowed distributed simulations to execute models in such heterogeneous environment.

The WS extension to CD++ also integrates the simulation capabilities of CD++ with visualization capabilities within the heterogeneous context discussed above. As a result, our team has also extended CD++ with a 3D visualization engine (which can be used along with the WS based D-CD++ parallel simulation) in the UCLP environment shown in Figure 8. The 3D visualization engine is based on Maya [Bec04] 3D visualization and its scripting language. To use Maya's with CD++, the user must use Maya facilities to create visual scene files, while an application written in Maya Embedded Language (MEL) permits to create a user interface that allows CD++ log files to interact with Maya, and to visualize the corresponding model in a 3D visual environment. In other words, this extension allows

users to animate the CD++ log files instead of going through those tedious log files. For example, a user may require to run a model using CD++ in the shown environment in Figure 8 (with the help of WS based D-CD++ extension), and at the same time visualize the simulation results.

```
X/00:000/top/in/2 to chassis
X/00:000/top/in/2 to body
X/00:000/top/in/2 to trans
X/00:000/top/in/2 to enginesubfact
D/00:000/chassis/02:000 to top
D/00:000/body/02:000 to top
D/00:000/trans/02:000 to top
X/00:000/enginesubfact/ in/2 to piston
X/00:000/enginesubfact/ in/2 to enginebody  ...
Y/02:000/chassis/out/1 to top
D/02:000/chassis/... to top
X/02:000/top/done/1 to chassis
X/02:000/top/in_chassis/1 to finalass ...
*/02:000/top to enginesubfact
*/02:000/enginesubfact to enginebody
Y/02:000/enginebody/out/1 to enginesubfact
D/02:000/enginebody/... to enginesubfact
X/02:000/enginesubfact/done/1 to enginebody
X/02:000/in_enginebody/1 to engineassem
D/02:000/enginebody/02:000 to enginesubfact
D/02:000/engineassem/02:000 to enginesubfact ...
```

Figure 9: Excerpt from the auto factory log file

As an example, when executing the auto-factory example (was shown in Figure 3 and Figure 4 in section 2) within a given framework, the simulation results are provided in a log file as shown in Figure 9 where it shows all of auto-factory activities with their stamped times. With the CD++ visualization extension, this CD++ log file can be visualized as shown in Figure 10, with the animation snapshots at time 02:000 and 04:000.

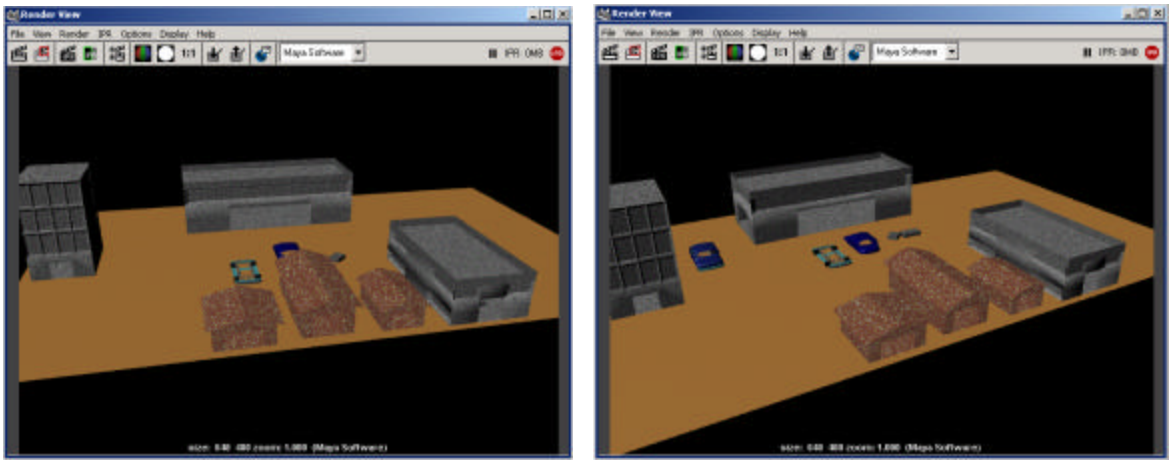

Figure 10: Auto factory animation at times 02:000 and 04:000

D-CD++ architecture is shown in Figure 11. The Client application invokes methods on generated stubs based on the contents of a WSDL description of a service. These stubs are configured with information about the CD++ WS and its endpoint. The stubs invoke remote methods available in the CD++ WS endpoint. The CD++ WS is described as a set of communication endpoints capable of exchanging messages. An endpoint is made of two parts: the abstract definition of operations (and messages); and the concrete binding of those abstract definitions to a concrete protocol (SOAP over HTTP with a message format). The compiler generates the stubs for the client and the skeleton code for the server side. JAX-RPC (Java API for XML-based RPC) is a runtime library that provides services for JAX-RPC mechanisms and APIs. CD++ can invoke functions (i.e. advertised serves via WSDL) on the server side as if were local functions. The application can connect to the CD++ web server, download or upload files, compile and simulate DEVS and Cell-DEVS models.

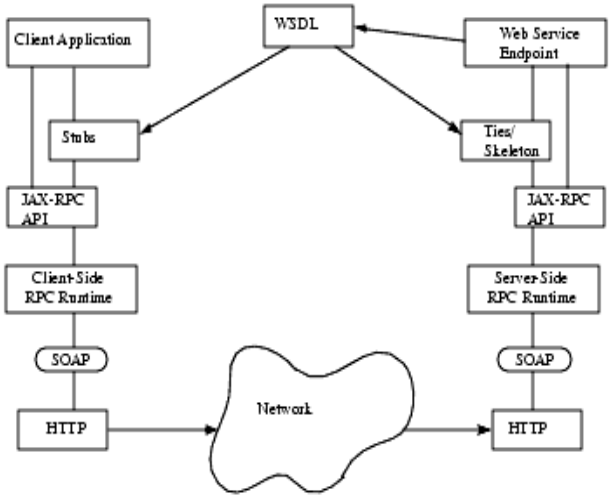

Figure 11: CD++ Web-Service Architecture [Mad07b]

3.2 Design and Implementation

In order to design and implement D-CD++, we extended CD++ in two dimensions. In one dimension, the toolkit was *wrapped* by a WS wrapper to expose its functionality to remote users/services using SOAP. We used the main WS standards such as XML [Bra04], SOAP [Gud03], Web Service Description Language (WSDL) [Chr01] for storing and parsing the configuration files used by the service, describing and exposing the service functionality, and messaging among the simulation services themselves as well as with the users, respectively. In another dimension, the simulation WS and the CD++ engine were extended to execute distributed models in a distributed environment. The model is decomposed into different partitions, each of which is assigned to a machine for execution with SOAP being used for messaging among the machines. This extension enables CD++ to be integrated in a net-centric architecture where its services are consumed by clients over the Internet.

In order to integrate the WS technologies with CD++, we created a WS wrapper to expand CD++ functionality to WS clients. The wrapper was built in Java and interfaced to the original C++ code due to the following reasons:

i) Most WS technologies are written in Java; hence implementing CD++ WS interface in Java allows better use of the WS technologies as they advance.
ii) Interfacing Java code to C++ is straightforward. Instead, interfacing C++ to WS technologies is not that simple.
iii) Separating the WS (in Java) from the simulation (in C++) provides better maintainability and modularity of the simulator. It also provides better preparation for standardization and interoperability with other DEVS tools implementation, giving the fact that all DEVS standardized models and simulation messages will be exchanged in XML format (and Java utilities clearly have the upper hand over C++ when it comes to handling XML messages).
iv) By keeping the simulation software in C++, we can achieve better performance, and we can access other simulation engines (P-CD++ and E-CD++) that run in high performance and are built using C++ as the programming language.

We used the Java Native Interface (JNI) [Lia99] to interface both Java and C++ components. JNI is a collection of APIs and is part of the Java Virtual Machine (JVM) developed by Sun Microsystems that allows Java programs to access functions written in native C/C++ code. The WS engine chosen for the implementation was Apache Axis [Axi06], an open source SOAP engine that has HTTP server functionality and runs as a web application within an application server (in our case Tomcat application server [Tom06]).

The simulation service was split into two independent parts: the *WS components* (used to handle the WS activities of the simulation service) and the *simulation components* (used to interact with CD++ by accessing and manipulating its internal objects and data structures). Both parts communicate with each other through message queues in the Linux kernel. The communication is handled by a proxy object (through the *WrapperProxy*) that is implemented as a shared C++ library plugged into the Java Virtual Machine (JVM) through the Java Native Interface (JNI) [Lia99].

A high level view of the D-CD++ software architecture is shown Figure 12. In this way, we have a separate running workspace for each simulation session, reducing potential resource contention and enabling multiple user sessions to run concurrently and independently. This also provides better maintainability and modularity, since any changes to either part (i.e. WS or simulation parts) do not affect the other one. The *WS components* of the simulation service are deployed in an Axis server, which in turn runs within an Apache Tomcat server. Axis loads all the libraries available in the directory of deployed services, which include the *JavaWrapper* (the backbone of the WS components), the server-side stubs, and the client-side stubs. In addition, when the *JavaWrapper* class is loaded, it loads the *WrapperProxy* as a shared native library into the JVM. At this point the simulation service is considered ready to receive client requests.

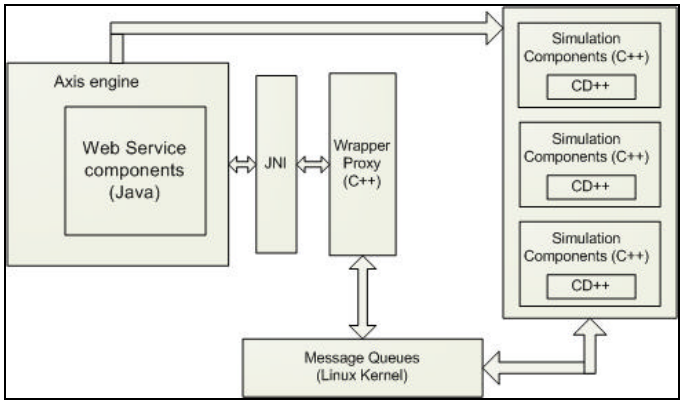

Figure 12: Implementing the simulation service using JNI and message queues

The WS components (i.e. implemented in Java, shown in Figure 13); fall into three main categories:

i) The Web service wrapper (*WS-Wrapper*) is the backbone of the *WS components* since it is the actual management of a simulation session within the Axis server for a single user's session (i.e. one thread is spawned for each simulation session and associated with one instance of the JavaWrapper class). For example, when Axis server at the server side receives a request from a client, it unpacks the request from its associated SOAP message and makes a call to the appropriate operation associated with the received request (within CDppPortTypeSoapBindingStub class) which then executes the corresponding method in the *JavaWrapper* instant attached to the client's session to fulfill the request.

ii) Utility classes: They are used to perform secondary functions required by the *WS-Wrapper* such as parsing the users and configuration files (i.e. those received as SOAP attachment in Figure 7). This takes place in two steps: when the service is started, (1) the user (i.e. client) access information (e.g. password, user name etc.) is retrieved and validated and (2) the client's model is partitioned into smaller coupled models and distributed among servers according to a grid configuration file submitted by the client. The configuration file contains the URLs of the simulation services participating in a session and the model partitioning information (which parts of the model must run on each machine in a distributed simulation session). Figure 14 shows example of a grid configuration file that partitions CD++ Cell-space between two machines.

iii) SOAP engine Interfaces: include the client side stubs (i.e. interface CDppPortTypeService) and the server-side skeleton (i.e. interface CDppPortType) to enable a D-CD++ machine to send/receive SOAP messages to/from other machines.

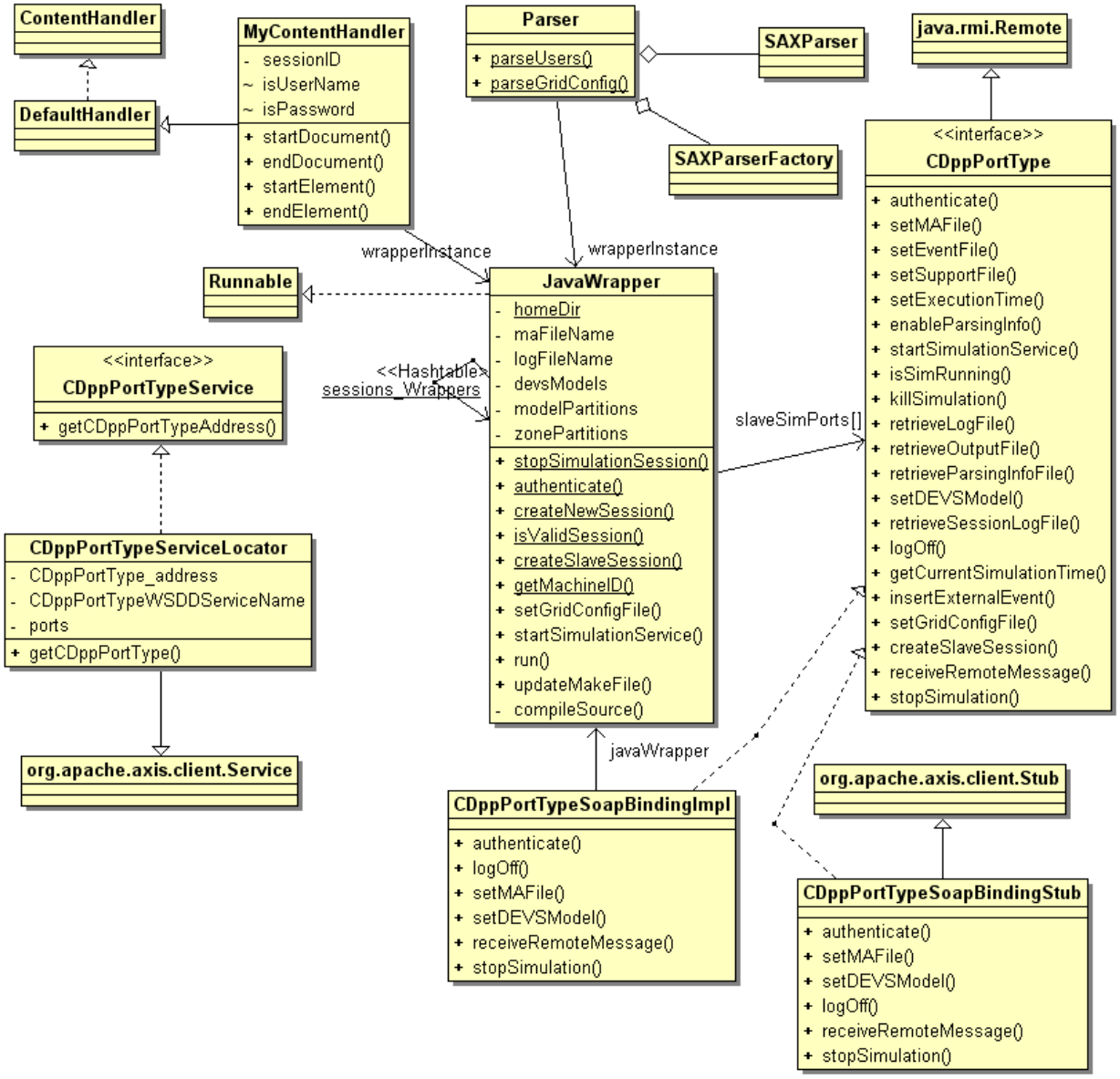

Figure 13: Web service components UML diagram

The services offered by the simulation service through its WSDL interface, are mapped into methods invoked on a *JavaWrapper* instance shown in Figure 13. Some of the operations performed by *JavaWrapper* include:

- User login: the method *authenticate* users password file stored on the server.
- Session initialization: the method *createNewSession* creates a working space for new sessions. Part of the process includes creating a *JavaWrapper* instance that will be used by the server-side for each session (deployed within the Axis server) to process messages related to that session (e.g. from the client or another D-CD++ machine participating in the simulation).
- Setting the model definition: method *setMAFile* is used to submit the model definition, *setEventFile* sets the external events file, method *setDEVSModel* sets the source and implementation files for DEVS models, and method *setSupportFile* sets the initial values file for Cell-DEVS models.
- Setting the configuration information for distributed sessions: method *setGridConfigFile* is used to submit *grid configuration file*; once the method is executed it parses the file and saves the information contained in it in the *JavaWrapper* instance created for the session.

- Starting the simulation: *startSimulationService* is used to start the simulator. This includes compiling the submitted DEVS models (if any) with the simulator, sending the model definition to slave machines, and starting the slave sessions.
- Checking the status of the simulation: *isSimRunning* is used to check the status of the simulation process since models might take long time to be executed. In addition, the *killSimulation* is used to end the simulation process (if needed).
- Retrieving the results of the simulation: *retrieveLogFileName*, *retrieveOutputFileName* are used for the log and output files retrieval. In the case of distributed simulations, the *JavaWrapper* will utilize the services on the slave machines in order to retrieve and archive all the log files into one file that can be sent to the user.
- Logging off: *logoff* is used to log the current user off and invalidate his session. This method will cause the *JavaWrapper* class to reclaim the resources used by the session and to send messages to the slave sessions to do the same.

3.3 Service Interface

Figure 7 illustrated the usual interface between a WS client and the D-CD++ simulation service using SOAP messages across the Internet. First, the client retrieves a WSDL document defining the interface of the simulation service, describing execution parameters and return values. Using the WSD document, the client can build the required stubs (normally using built-in tools with SOAP engines) for the deployed services, where the client can invoke those services as any other local calls from Java code. However, in reality when a service is called, it invokes the WS interface methods through dynamically generated SOAP messages transmitted over HTTP protocol.

The major operations provided by the D-CD++ to clients via the WS component (which routes the client requests to the simulation component) can be divided into four groups of interface methods , as follows (see class CDppPortType in Figure 13):

- Session management Interfaces:
  - ***Login /logout***: used to log current user in/out.
  - ***createSlaveSession***: initializes slave sessions for distributed simulations.

- Configuration Interfaces:
  - ***setMAFile***: sets model definition file (.ma).
  - ***setDEVSModel***: sets a DEVS model (C++ header and implementation files).
  - ***setEventFile***: sets the external events file (.ev).
  - ***setSupportFile***: sets files need for the simulator (such as initial values of Cell-DEVS models).
  - ***setExecutionTime***: sets maximum execution time of the simulation.
  - ***enableParsingInfo***: reactivates the simulator to generate parsing information file for debugging.
  - ***setGridConfigFile***: this file contains the model partitions and the addresses of the machines participating in a distributed simulation session. Figure 14 shows a grid configuration file example that partitions the fire model (Figure 28) cell-space among two machines where Machine 0 (master) contains cells range (0,0)..(14,29) and Machine 1 (slave) contains cells range (15,0)..(29,29). The conceptual model partition of this configuration file example is also shown in Figure 30. The same principle applies to the regular CD++ DEVS models where each coupled model is assigned to a machine where a coupled model may spread over a number of machines. However, atomic models are inseparable and assigned to machines by D-CD++ according to their parent couple models.

- Simulation Monitoring and Control Interfaces:
  - ***receiveRemoteMessage***: exchanges remote messages during a distributed simulation session.
  - ***stopSimulation***: used by the master machine to stop the simulation in the slave machines.
  - ***startSimulationService***: starts the simulation.
  - ***isSimRunning***: checks whether the simulation is running or not.
  - ***getCurrentSimulationTime***: checks the current simulation time.
  - ***insertExternalEvent***: inserts external events while the simulation is running.

- ***killSimulation***: aborts the simulation.

- Retrieving data Interfaces:
  - ***retreiveLogFile***: retrieves the log file(s) generated by the simulator.
  - ***retreiveOutputFile***: retrieves the output file generated by the simulator.
  - ***retrieveParsingInfoFile***: retrieves the parsing information file for debugging.
  - ***retrieveSessionLogFile***: retrieves the session bg file (including the output messages generated by the simulator).

```
<?xml version="1.0"?>

<!-- <JavaWrapper:Grid xmlns:CDppWrapper="http://www.sce.carleton.ca/ars/CDpp"> -->
<Grid>
<MACHINES>
        <MACHINE>
                <MACHINE_RANK>0</MACHINE_RANK>
                <MACHINE_URI>http://IP-Address:8080/axis/services/CDppPortType</MACHINE_URI>
        </MACHINE>
         <MACHINE>
                 <MACHINE_RANK>1</MACHINE_RANK>
                 <MACHINE_URI>http://IP-Address:8080/axis/services/CDppPortType</MACHINE_URI>
         </MACHINE>
</MACHINES>

<MODEL_PARTITIONS>
        <PARTITION machine="0">
                <ZONE>fire(0,0)..(14,29)</ZONE>
         </PARTITION>
         <PARTITION machine="1">
                 <ZONE>fire(15,0)..(29,29)</ZONE>
         </PARTITION>
</MODEL_PARTITIONS>
</Grid>
<!-- </JavaWrapper:Grid> -->
```

Figure 14: Example of Grid Configuration File Cells Partitioning

4. Distributed CD++ (D-CD++)

The WS component provides the D-CD++ interface to clients and it routes clients' requests to the simulation component. This section discusses the design and implementation of the simulation component of the D-CD++ [Mad07a]. D-CD++ arranges the coordinators in a master/slave architecture to reduce the communication overhead among nodes participating in the simulation [Tro03]. Accordingly, when a coupled model is partitioned (i.e. distributed) onto multiple nodes, one coordinator is created for each participant node and it becomes in charge of the simulators of the model portion mapped on that node. The coordinator of the first node of the partition is a master coordinator, while all the other coordinators are slaves. Further, the master coordinator is considered as the direct parent of the slave coordinators residing on remote nodes. D-CD++ executes the model by passing messages among the different *processors* in the simulation. *Coordinators* are the *processors* responsible for executing coupled models while *Simulators* are associated with atomic DEVS models and they are responsible for executing each of the functions defined by the model depending on the time and type of the received message (Figure 15). A *Root coordinator* is in charge of driving the simulation as a whole and interacting with the environment. 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.

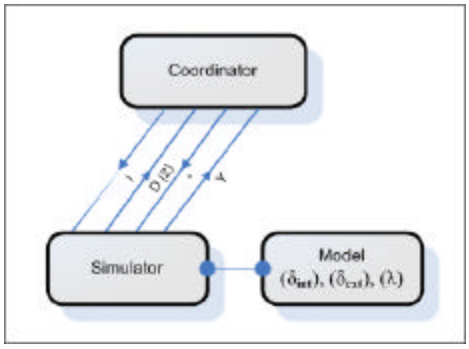

Figure 15: Message exchange during a simulation cycle

Our main focus in this paper is on executing DEVS models in parallel to speedup the simulation and to access different resources on different machines simultaneously, hence target resources may exist on different locations so that D-CD++ has to virtually assemble desired resources before the simulation starts. For example, a client desires to run experiments by using a model in a lab in California and couple it with another model in Spain. In this case, D-CD++ has to first bring those models together and execute them in parallel. After all, our main inspiration is to bring people to share resources and work together. On the other hand, D-CD++ can provide clients with other CD++ extensions such as CD++ real-time [Gli02] (i.e. simulation advances according to the real-time rather than as soon as possible) or parallel CD++ [Tro03] [Gli04] (i.e. simulation on a single machine with hundreds of nodes. In this case the CD++ on that machine divides the model among its nodes and run it in parallel). In principle, D-CD++ extension provides CD++ simulation services to clients via WS regardless of the type of the offered services.

4.1 Implementing Parallel-DEVS Algorithms

The Parallel-DEVS (P-DEVS) algorithms were introduced to solve the serialization problem with the original DEVS algorithm and to enable efficient execution of DEVS models in parallel and distributed environments. The main additions in P-DEVS are the message bags, and the *confluent transition* function (d_{conf}). Message bags are used to hold multiple input messages arriving to the model and multiple output messages generated by the model. The *confluent* function allows the modeler to define the behavior of the model when it receives an *external message* while being scheduled for internal transition. In such case, the *confluent transition* function is executed in place of the *internal* and *external transition* functions. The abstract simulator for DEVS models was extended to run P-DEVS models so that multiple imminent models can be executed together. In the P-DEVS abstract simulator, five kinds of messages are used and can be categorized into *content messages* and *synchronization messages*. Content messages include *external messages (X)* and *output messages (Y)* that are used to represent events generated by the model. Synchronization messages include *internal messages (*), collect messages (@)*, and *done messages (D)*. *Internal messages* are used by the coordinators to trigger three different transitions depending on the message arrival time and the status of the external message bag. *Collect messages* are used to trigger the *output* function of the model before any internal transition. *Done messages* are used by the simulator to report the time of the next transition to its coordinator.

By implementing the previous algorithms, CD++ is able to activate imminent models concurrently avoiding the serialization problem introduced in the original version. This is of considerable importance to the Cell-DEVS models as it allows for executing cells with zero time delay (due to the availability of message bags). In addition, it provided the possibility of extending the simulator into a distributed engine which can execute concurrent imminent models in parallel. Figure 16 shows the difference between the previous and current implementation of the CD++ engine in the case of two imminent simulators. The original implementation (left part) required the use of the *select* function in order to choose the simulator to activate first. However, when implementing the P-DEVS algorithms, the coordinator is activating both simulators at the same time (right part), solving the issue of serialization introduced in the original DEVS formalism.

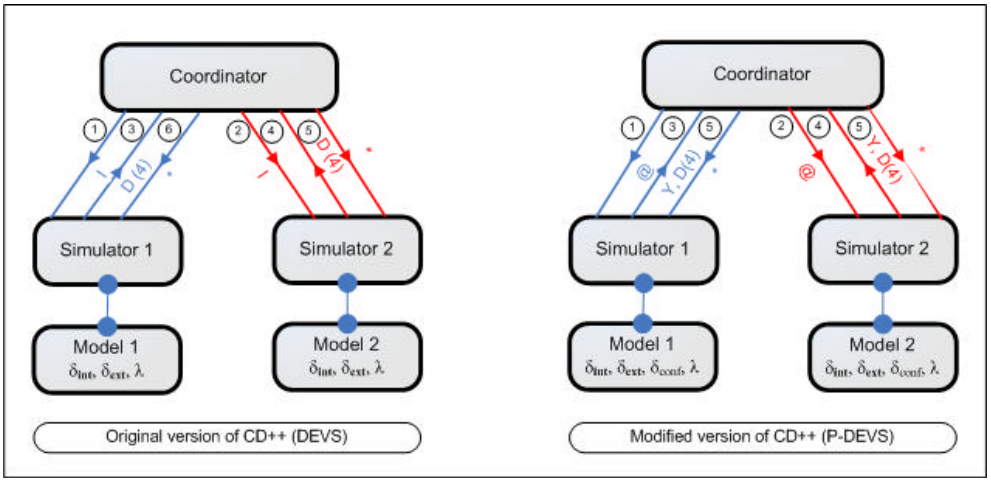

Figure 16: Concurrent model activation in Parallel-DEVS

Implementing the P-DEVS algorithms required changes to be made in simulation hierarchy of CD++, shown in Figure 17. Each processor keeps track of the model that is responsible for executing (the model hierarchy shown in Figure 22). The *processor* class is the parent of all the classes in charge of executing the model. Those include the *Simulator*, *Coordinator*, *FlatCellCoordinator*, 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* (shown in Figure 17).

The *Simulator* class extends the *Processor* class and executes the functions of the atomic DEVS model corresponding to the type of the received message. For example, when a *Simulator* receives a *collect message* from its parent coordinator, it executes the *output* function associated with its atomic model. 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 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.

Messages are implemented as separate classes, each representing a message type with all the classes inheriting the *Message* class. Different messages have different attributes; for example, the *Done Message* class has an extra field (*nextChange*) to indicate the time of the next state change.

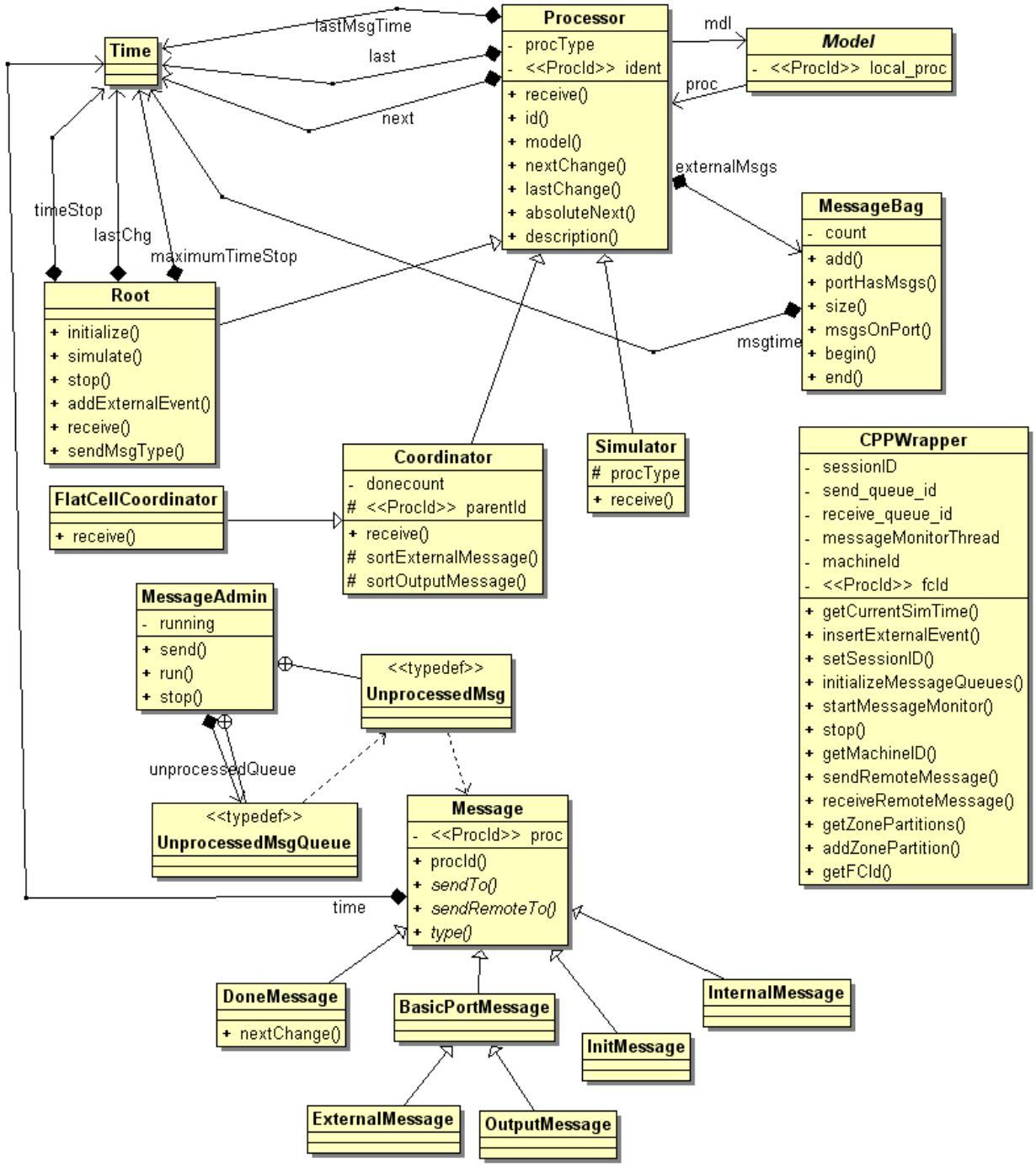

Figure 17: The simulation class hierarchy

4.2 Interfacing Web-service and Simulation Components

The *CPPWrapper* class (shown in Figure 17) is responsible for interfacing CD++ simulation components with the wrapper proxy via Linux inter-process communication system (see Figure 18). The proxy object creates two message queues in the Linux kernel for each client session to implement a bidirectional communication channel between the WS component and the corresponding simulation component, as shown in Figure 18. Further, the

Wrapper Proxy interfaces with the WS using the Java Native Interface (JNI) [Lia99]. The functionality of the *CPPWrapper* includes:

- Initializing the message queues used for communication with the WS components (*initializeMessageQueues*).
- Querying and retrieving the model partitions from the WS components (*machineForModel*, *addZonePartition*).
- Querying the current execution time and inserting external events while the simulation is running (*getCurrentSimulationTime*, *insertExternalEvent*).
- Sending remote messages while running distributed simulations (*sendRemoteMessage*). This method takes a C++ message and sends it to the WS components to be sent to the remote machine.
- Receiving remote messages while running distributed simulations (*receiveRemoteMessage*). This method receives a message from the WS components and constructs a C++ message to be processed by the simulator.
- Stopping the simulation when receiving a stop message from the WS components (*stop*).

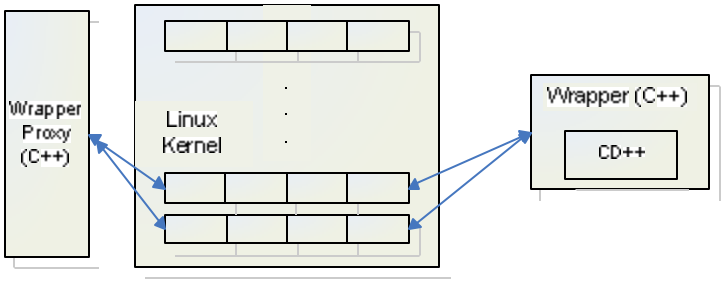

Figure 18: Message queues created for each user session [Mad07b]

4.3 Designing and Implementing Distributed-CD++ (D-CD++)

Researchers have dealt with the "synchronization and causal dependency problem" in two ways: (1) Conservative approach: restraining models from advancing until all models are in sync. This method is implemented usually by having a global simulation clock (i.e. this method is currently adopted by this paper) and (2) Optimistic approach: advancing local models optimistically without worrying about other models progress. However, if a model is received a message from a remote model that causes a causality error, the subject model has to roll back simulation to correct the just discovered error. Obviously, the second approach can trigger many messages which can be very expensive in the distributed environment due to the communication overhead. Therefore, three main approaches were investigated:

i) Implementing an optimistic simulation engine using the Time Warp algorithm. Although Time Warp unties the different machines in distributed simulations by allowing each machine to advance its clock independently from the other machines, its performance depends on the communication overhead strength (i.e. synchronization messages) to handle rollbacks. It was observed that speedup achieved by the Time Warp algorithm might be compromised by the delay of the SOAP messages in the distributed environment.

ii) Implementing a conservative simulation engine by allowing each machine to advance its clock when it can guarantee that causality errors will not occur. This can be accomplished by sending *lookahead* values using *null messages*. This approach has the disadvantage of adding to the overhead of the engine by the time required to transmit *null messages* using SOAP. In addition, deadlock might occur if there is a cyclic dependency between the models with zero lookahead. This in turn, requires implementing deadlock detection and recovery mechanisms.

iii) Implementing a conservative engine by handling clock advancement in one machine to minimize the synchronization messages among the machines participating in the simulation.

The third approach was adopted in order to limit the synchronization messages among the machines to those required by the PDEVS algorithms. Implementing the distributed engine required two major changes to the simulator. On one side, the model definition classes had to be extended to allow the partitioning of the model on multiple machines. On the other side, the model execution mechanism had to be extended in order to handle

message routing and synchronization on multiple machines. In principle, executing the model on multiple machines requires:

i) Loading the model hierarchy and model partition information in each machine participating in the simulation. This is required in order to check the causal dependencies among the model components when an event needs to be sent from one model to another. In addition, having the model partition information is needed to distinguish the local model components from the remote ones.

ii) Running simulators and coordinators on each machine for local models in order to handle message passing and model execution.

The model partitioning information is provided to the simulation through the *grid configuration file* (an XML file containing the addresses of the machines executing the model and the parts of the model running on each machine). Using the original implementation of the *Coordinator* class will add unnecessary overhead if two child *processors* want to exchange messages and are running in a machine different than the coordinator. As shown in Figure 19, *Simulator 3* sends an *output message* that is to be translated into an *external message* to *Simulator 2*. When sending the message to the coordinator, it ends up being transmitted twice as remote messages due to the fact that the coordinator is running on a different machine than the source and destination of the message.

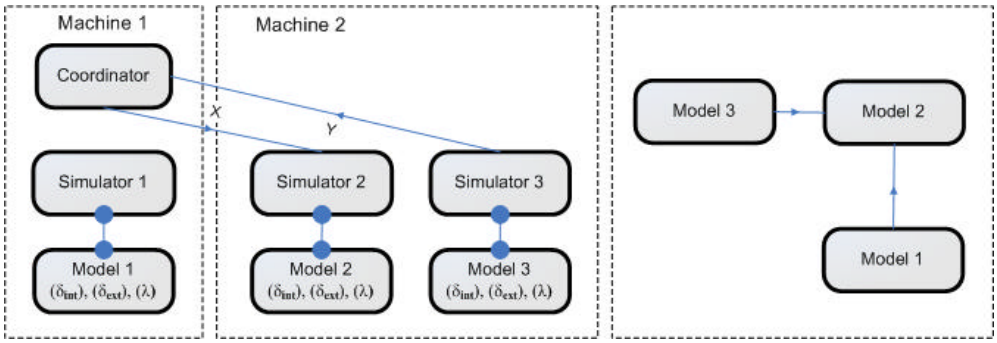

Figure 19: Unnecessary remote messages in distributed simulation

This problem could have been avoided if there is a *processor* responsible for message routing locally in each machine. One approach to solve this issue is to use one coordinator in each machine for message routing among the local *processors*; this was initially adopted by PCD++ [Tro03] in order to minimize the remote message transmission among the machines. The idea depends on using two kinds of coordinators for each coupled DEVS/Cell-DEVS model:

i) *Master Coordinator:* is responsible for synchronizing the model execution, interacting with upper level coordinators and message routing among the local and remote model components.

ii) *Slave Coordinator:* is responsible for message routing among the local model components dispensing with the need to send remote messages if the master coordinator is residing on a different machine than that used to run the sending and receiving *processors*.

Having a slave coordinator in *Machine 2* (as shown in Figure 20), causes the message from *Simulator 3* to *Simulator 2* to be sent locally improving the performance of the simulator.

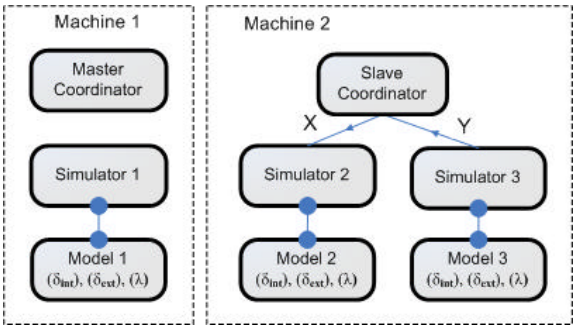

Figure 20: The use of Master and Slave coordinators to avoid unnecessary messages

Implementing the distributed simulator includes extending CD++ in three main aspects:

i) The simulation mechanism is implemented mainly using the master and slave coordinators (discussed in subsection 4.3.1),
ii) The model loading mechanism is extended to maintain the partitioning information (discussed in subsection 4.3.2), and
iii) The message passing mechanism is extended to handle local and remote message passing (discussed in subsection 4.3.3).

4.3.1 Master and Slave Coordinators

The master and slave structure was used to reduce the number of exchanged messages among participant machines in the simulation which can be very expensive in the distributed environment. It was observed that the performance degrades according to the number of exchanged remote messages among distributed simulators and the distance that those messages have to travel to reach their destinations. Evidently, the amount of exchanged messages is relative to the distributed model being executed and usually varies from a model to another. Therefore, an exact performance measurement of master/slave structure can be very challenging. Further, in our approach, the master/slave structure is transparent from the modeler whereas the modeler only defines how the model is partitioned among distributed machines.

The master and slave coordinators are implemented by extending the functionality of the *Coordinator* class. The reactions of the master and slave coordinators when receiving messages differ from those of the original coordinator.

When a master coordinator receives a *collect message* from its parent coordinator, it forwards it to its imminent child *processors*; those can be *Simulators*, *Master Coordinators*, or *Slave Coordinators*. The *external messages* in the master coordinator's bag are processed when it receives an *internal message*. This, results in sending *internal messages* to the child *processors* scheduled for internal and/or external transitions. The *output messages* are processed depending on their destinations; they could be translated into *external messages* for local child *processors* or *output messages* to be sent to the parent *coordinator*.

The slave coordinator handles the messages in a similar way to the original coordinator in the stand-alone version of CD++ (discussed in section 4.1). The main difference between the two is in the interaction with the upper level coordinator; the slave coordinator interacts with the master coordinator instead of sending messages directly to the upper level coordinator.

The master and slave coordinators implemented by extending the *Coordinator* class as shown in Figure 21. Both override the *receive* function used to process the different messages received by the *processors*. In addition, they implement operation *sortExternalMessages* (triggered when the coordinator receives an *internal message* from its parent coordinator), operation *sortOutputMessages* (triggered when receiving an *output message* from a child *processor*), operation *sortExternalMessages* (triggered when the coordinator receives an *internal message* from its parent coordinator which causes the coordinator to process all external messages in its bag by forwarding them to their destinations either locally or remotely), and operation *calculateNextChange* (used to evaluate the imminent child *processors* where its action differs depending on the coordinator type: In the case of the master coordinator, it considers the local child *processors* in addition to the remote slave coordinators; while in the case of the slave coordinator, it only considers the local child *processors*).

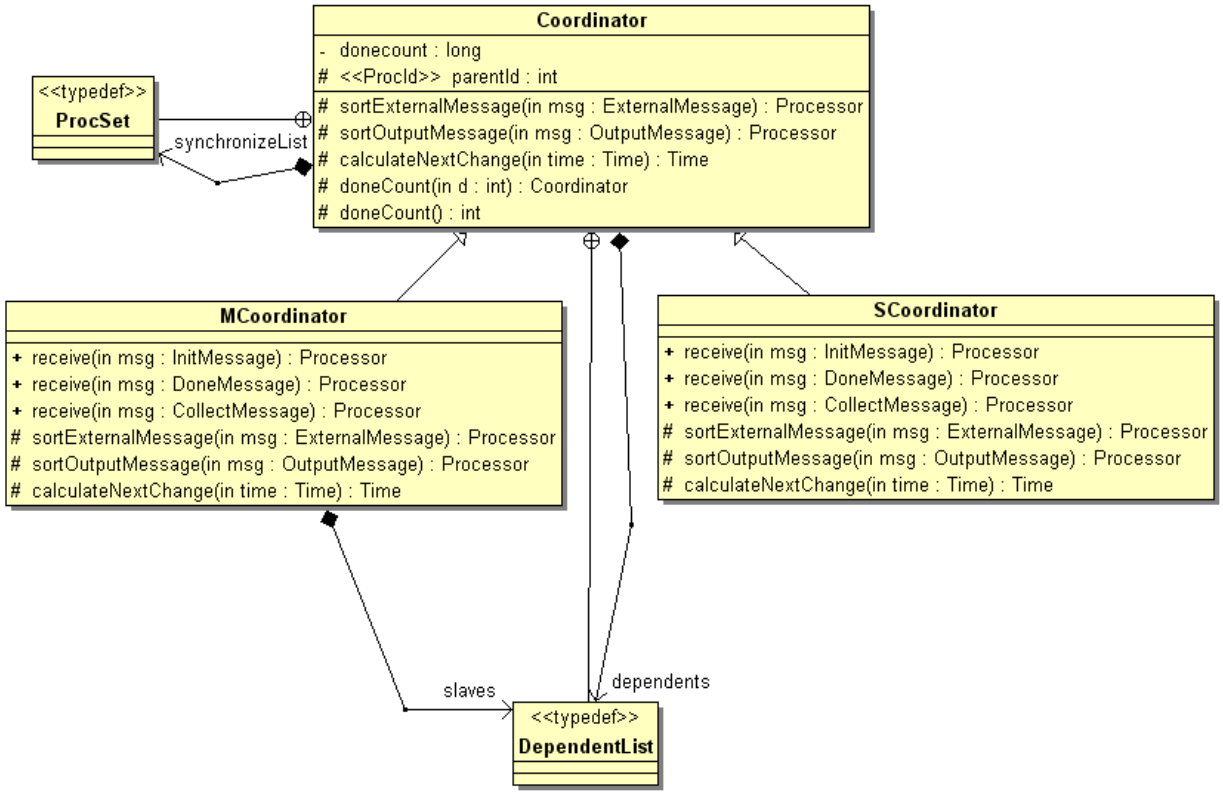

Figure 21: Master and Slave coordinator classes

4.3.2 Model Loading Mechanism

The model loading mechanism in the stand-alone CD++ was based on parsing the model definition files and creating the corresponding simulator/coordinator for each of the model components. Those components can be atomic DEVS models, coupled DEVS models, atomic Cell-DEVS models, coupled Cell-DEVS models, and flat coupled Cell-DEVS models. After implementing D-CD++, the model loading mechanism includes loading the partitioning information as part of the model loading process; the partitioning information is retrieved from the WS components through the *CPPWrapper* class (was shown in Figure 17). Atomic models are assigned to run on a specific machine and a coupled model can span different machines with each of its components running on an individual machine.

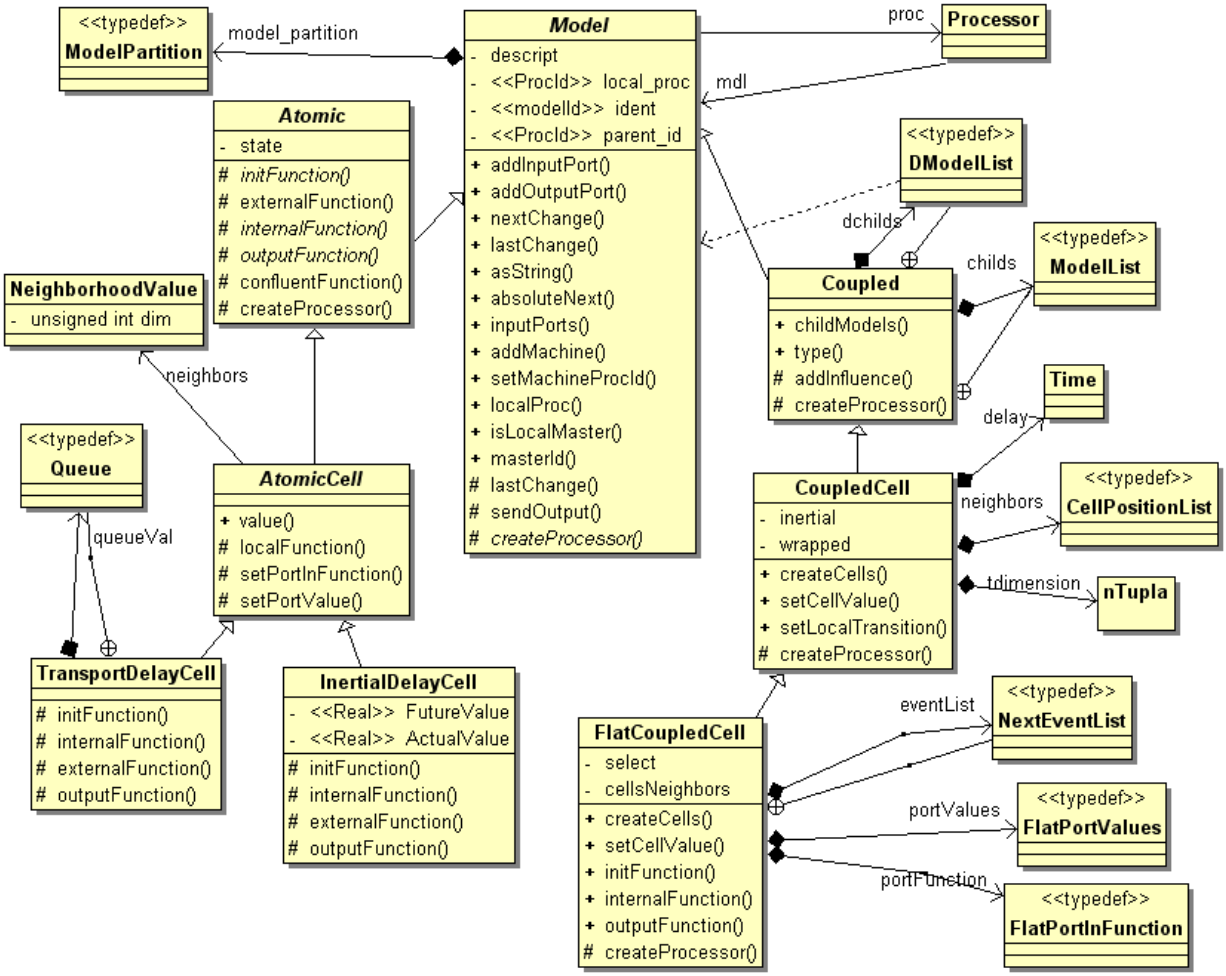

Figure 22: D-CD++ model hierarchy

Figure 22 shows the relationship between the different classes representing the model hierarchy in D-CD++. The figure also shows that each model keeps track of the processor that is responsible of executing it (the processor hierarchy shown in Figure 17). During the initialization stage of the simulation, the simulation loads all required models (on all participant machines in case of the distributed simulation), create processors to executed the loaded models (via *createProcessor* method) along with the organization of child-parent relationship among the created processors, and it then assigns those processors to the appropriate machines according to the models that they responsible to execute (via invoking *addMachine* method of the appropriate model). Further, a processor of type "Simulator" (Figure 17) is created and assigned to the local machine, if it is created for an atomic one. On the other hand, if the model is a coupled model, a master coordinator is created for it in order to be run on the local machine (given its first sub-model is assigned to run on local machine); otherwise, a slave coordinator is created and linked with the coupled model.

4.3.3 Message Passing Mechanism

Both the *MessageAdmin* and *CPPWrapper* class (was shown in Figure 17) are responsible for message passing mechanism locally and remotely. The *MessageAdmin* class is activated when the simulation is started, and as long as there is at least one message in the *unprocessedMessages* queue. The *MessageAdmin* class picks the message at the front of the *unprocessedMessages* queue and checks the destination of the message; if the destination is a local *processor*, the message is delivered to the *processor* by invoking the *receive* function of that processor. Otherwise, the message is passed to *CPPWrapper* which in turn passes it to the WS components to be sent remotely.

The WS encapsulates the remote message within a SOAP message and transmitted it to the distant machine. Subsequently the WS components (at the distant machine) extract the information from the received SOAP message and pass it to the *CPPWrapper*, which invokes it as a local message (via class *MessageAdmin*). This approach makes involved simulators treats passing messages the same regardless of being remote or local messages. It also increases CD++ simulation engines maintainability if transmitting remote message technology is changed, since the communication module in the system is separated from the simulation part.

4.4 Sample Scenario

In this section we introduce a sample in order to show the overall operation of the simulator in a distributed environment (i.e. a coupled DEVS model executed using two machines). The model, shown in Figure 23, consists of four DEVS models: the *generator* is an atomic model producing jobs to be processed by the *processor*, the *queue* is used to store arriving jobs before they get processed, the *processor* is responsible for processing the jobs, and the *transducer* is in charge of calculating statistics such as the throughput of the *processor.*

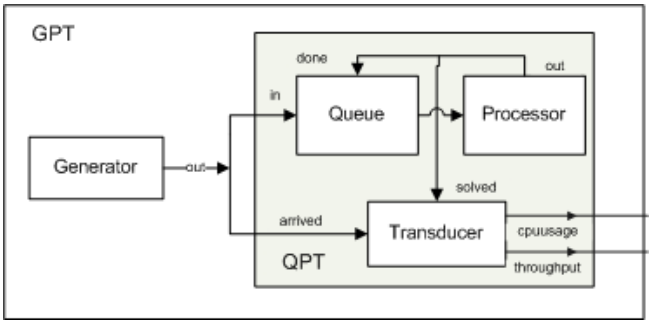

Figure 23: The Generator-Processor-Transducer (GPT) model

Two machines were used to execute the model, one located in Ottawa and the other in Montreal. They were connected using a commodity Internet connection. The *generator* component of the model was set to run on *Machine 1(Ottawa)*, and the QPT coupled model was set to run on *Machine 2(Montreal)*.

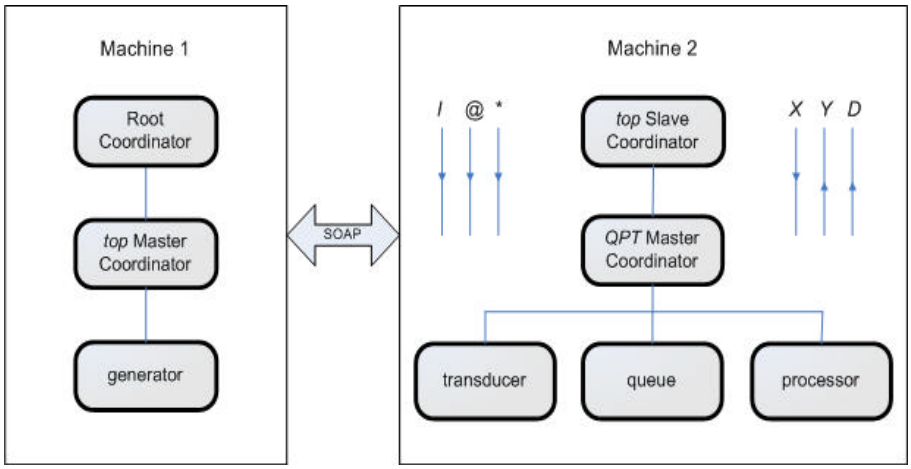

Figure 24: GPT model partitioning on two machines (GPT) model

In this case Machine 1 loads three *processors*: the *Root coordinator*, the *top master coordinator*, and the *generator*. Machine 2 loads the *top slave coordinator*, the *master coordinator*, and simulators for the *QPT*: *transducer*, *queue*, and *processor*. The simulation starts by the *Root* coordinator sending an *initialization message (I)* to the *top master coordinator*, which in turn forwards it to its child *processors* (*generator* and *top slave coordinator*). The message to the *top slave coordinator* is sent remotely using a SOAP message and upon reception, it is forwarded to its child *processor* (*QPT*). The *initialization message* causes the simulators to initialize their models and report their next

state change to their parent coordinators. The D-CD++ simulator saves the progress of the simulation in each machine into a log file that includes an entry for each message received by the *processors* running on that machine.

```
0 / L / I / 00:00:00:000 / Root(00) for top(06)
0 / L / I / 00:00:00:000 / top(06) for generator(01)
0 / L / D / 00:00:00:000 / generator(01) / 00:00:00:000 for top(06)
0 / R / D / 00:00:00:000 / top(07) / 00:00:02:000 for top(06)
0 / L / D / 00:00:00:000 / top(06) / 00:00:00:000 for Root(00)
0 / L / @ / 00:00:00:000 / Root(00) for top(06)
0 / L / @ / 00:00:00:000 / top(06) for generator(01)
0 / L / Y / 00:00:00:000 / generator(01) / out /    0.00000 for top(06)
0 / L / D / 00:00:00:000 / generator(01) / 00:00:00:000 for top(06)
0 / L / D / 00:00:00:000 / top(06) / 00:00:00:000 for Root(00)
0 / L / * / 00:00:00:000 / Root(00) for top(06)
0 / L / * / 00:00:00:000 / top(06) for generator(01)
0 / L / D / 00:00:00:000 / generator(01) / 00:00:09:000 for top(06)
0 / R / D / 00:00:00:000 / top(07) / 00:00:00:001 for top(06)
0 / L / D / 00:00:00:000 / top(06) / 00:00:00:001 for Root(00)
```

Figure 25: An excerpt of the log file of Machine 1

Figure 25 shows an excerpt of the log file of Machine 1. The first field in a log entry is the machine id, followed by the source of the message (L: local, R: remote), then the timestamp of the message is listed, followed by the source and destination *processors*. In the case of *external* and *output messages*, two extra fields are listed: the port name and message value sent through the port. As we can see in Figure 25, the initial message was transmitted to the generator. After sending the *initialization message*, the *top master coordinator* receives *done messages* from its child *processors*. This includes the *done message* sent from the *generator* (line 3 in Figure 25) reporting the time of the next change as "00:00:00:000"; in addition, it includes a remote *done message* from the *top slave coordinator* (line 4 in Figure 25) running on Machine 2 reporting the minimum time of the next change as "00:00:02:000". The *top master coordinator* sends the minimum time of next state change to the *Root* coordinator (line 5 in Figure 25). In the next simulation cycle, the *Root* coordinator sends a *collect message* at time "00:00:00:000" to the *top master coordinator* that in turn forwards it to the *generator*. The *collect message* causes the *generator* to execute its *output* function to generate the output that is forwarded to its parent coordinator. Line 8 in Figure 25 shows the *output message* sent from the *generator* to the *top master coordinator* through the *out* port carrying a value of *zero*. No *collect message* is sent to the *top slave coordinator* at this point, since its next transition occurs at time "00:00:02:000".

```
1 / R / X / 00:00:00:000 / top(06) / out /     0.00000 for top(07)
1 / R / * / 00:00:00:000 / top(06) for top(07)
1 / L / X / 00:00:00:000 / top(07) / in /     0.00000 for qpt(05)
1 / L / X / 00:00:00:000 / top(07) / arrived /     0.00000 for qpt(05)
1 / L / * / 00:00:00:000 / top(07) for qpt(05)
1 / L / X / 00:00:00:000 / qpt(05) / arrived /     0.00000 for transducer(04)
1 / L / X / 00:00:00:000 / qpt(05) / in /     0.00000 for queue(02)
1 / L / * / 00:00:00:000 / qpt(05) for queue(02)
1 / L / * / 00:00:00:000 / qpt(05) for transducer(04)
1 / L / D / 00:00:00:000 / queue(02) / 00:00:00:001 for qpt(05)
1 / L / D / 00:00:00:000 / transducer(04) / 00:00:02:000 for qpt(05)
1 / L / D / 00:00:00:000 / qpt(05) / 00:00:00:001 for top(07)
```

Figure 26: An excerpt of the log file of Machine 2

Figure 26 shows an excerpt of the log file of slave. The *output message* generated by the *generator* is translated by the *top master coordinator* into an *external message* that is sent to the *top slave coordinator* via SOAP (line 1 in Figure 26). The *top slave coordinator* saves the message into its external message bag until it receives an *internal message* from the *top master coordinator* (line 2 in Figure 26); at which point, it forwards the message to the *QPT master coordinator* through the *in* and *arrived* ports. This causes the *QPT master coordinator* to send the *external messages* in its bag to the *transducer* and *queue* models (lines 6, 7 in Figure 26). The *internal message* sent to the *QPT master coordinator* is forwarded to the *queue* and *transducer* models (lines 8, 9 in Figure 26). This results in the *queue* and *transducer* models executing their *external transition* functions and reporting the time of the next

change as "00:00:00:001" and "00:00:02:000", respectively (lines 10, 11 in Figure 26). The *done message* (generated by the *top slave coordinator*) is forwarded to the *top master coordinator* using SOAP (line 14 in Figure 25). Then the *top master coordinator* evaluates the minimum time of the next change ("00:00:00:001") and sends it to the *Root* coordinator. The *Root* coordinator advances the clock of the simulation to "00:00:00:001" and the simulation continues until at leas one of the following conditions holds: there are no more events/messages scheduled by any of the *processors*; or, the simulation clock reaches the maximum execution time as provided by the user.

5. Performance Analysis

Our objective, in this section, is to first put our WS based distributed simulator extension to CD++ (D-CD++) to the test by making CD++ distributed simulators cooperate among each other to execute one single model between two cities. By doing so, we can perform more testing on the D-CD++ between more cities to make it ready for final integration with a web-based heterogeneous sharing environment, as was discussed in section 3. The second objective is to study the long-distance communication overhead. Clearly, as we going to see in the section, the remote communication overhead is the performance bottleneck as expected. However, in our case in this section, we used small models to focus more on the communication between simulators, but many factors can change which affects the model execution time. For example, performance will clearly improve if we use a model that requires a great deal of computation, and one target parallel machine with hundreds of nodes, hence the remote communication is only limited to sending model files to the target machine and receiving results back to the client.

The distributed simulation engine introduces overhead that affects the simulation execution time. The time it takes for a local message (implemented as a C++ object) to be transmitted between two local processors is much shorter than the time it takes for a remote message (i.e. SOAP, in our case) carrying the same information to be transmitted between two remote processors. The overhead is contributed to by two main parts of the message path between two remote processors: (1) the time it takes to transmit a message between the simulator and the WS components through the Linux kernel, and (2) the time it takes to transmit a remote message between the two simulation WS. Figure 27 illustrates the path for both the remote and local messages.

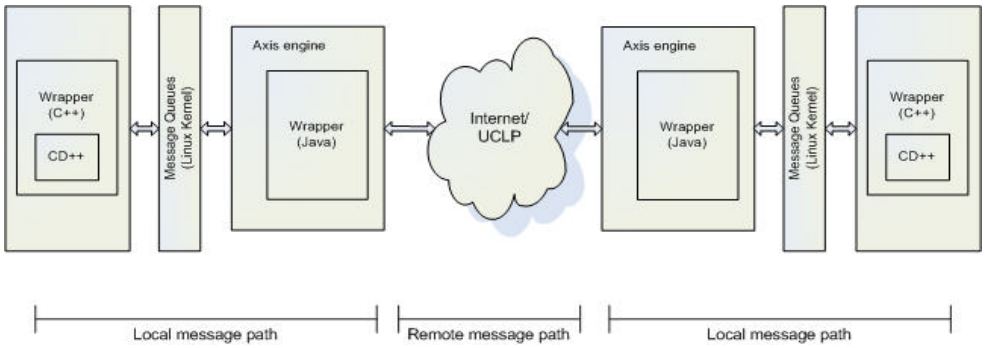

Figure 27: Sending remote messages in distributed simulation

In order to study the performance of the simulator, different distributed simulation sessions were executed using two machines; one of the machines was located in Montreal, and the other was in Ottawa (i.e. those distributed simulators were connected via WS). Two different models were executed using two different connections between the machines. In the first group of runs, the machines were connected using a commodity Internet connection; in the second group, UCLP was used to create a point-to-point (P2P) connection between the Montreal and Ottawa sites. The results of these two groups were compared to each other as well as to the results obtained when executing the models using a single machine. The readings obtained during the runs include:

i) The simulation time required to execute the models;
ii) The average time it takes in each run to transmit a SOAP message from Ottawa to Montreal.
iii) The average time to transmit a message within the Linux kernel using message queues.
iv) The average time to transfer a local message within a single machine.
v) The bandwidth available for the simulator when using the Internet and UCLP connections.

The model we will discuss for performance analysis is a fire spread in a forest and it is implemented as 30x30 coupled Cell-DEVS model [Ame01]. Each cell represents a square area of the forest which is considered to be burned if its temperature exceeds a threshold. Figure 28 shows an excerpt of the model definition with possible initial values of the cells. As the figure shows, the cell-DEVS coupled model definition uses a cell space with *inertial* delay. The cell's neighborhood is composed by the immediate neighboring 8 cells. *Fire(-1,-1)* represents the cell in the North West side (NW), *Fire(0, -1)* represents the cell in the west (W), etc. The rules that define the state of the cells in each simulation cycle are defined using the *localtransition* construct.

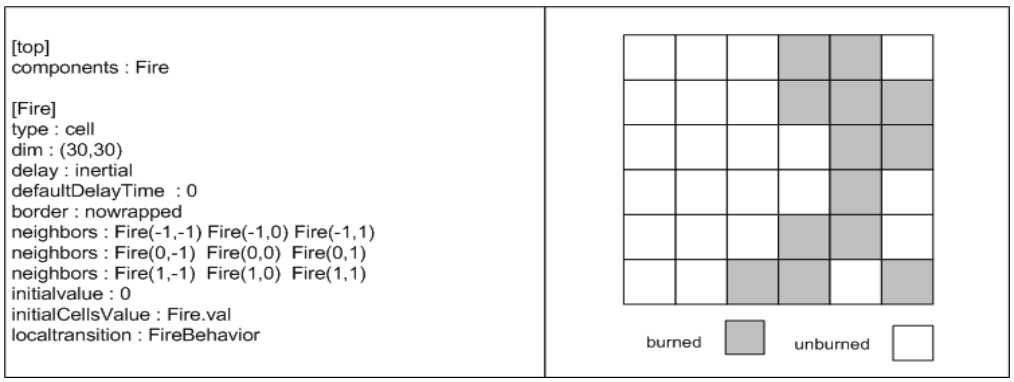

Figure 28: An excerpt of the Fire model definition

In order to study the performance of the distributed simulator, three types of experiments were performed. The first one was carried out using one machine to estimate the simulation time without the overhead incurred by sending remote messages (using SOAP). The second one was conducted by splitting the fire model into two equal partitions; each of which was assigned to a machine connected using a commodity Internet connection. In the third experiment, the two machines were connected using a dedicated fiber optic link using UCLP. The simulation service records (the required metrics) at different stages an accurate measure of the duration of each stage with a precision of microseconds.

We computed the average time and the standard deviation of a number of runs whereas tests were repeated as needed to obtain a confidence interval of 95%. The confidence interval was computed as follows [Ban01]: a confidence interval of $100(1-\alpha)$ %, allowing us to obtain:

$\bar{q} - t_{\alpha/2,f}\, \bar{s}(\bar{q}) \leq q \leq \bar{q} + t_{\alpha/2,f}\, \bar{s}(\bar{q})$;

Where $\bar{q}$ is the point estimator of q, $\bar{q} = \frac{1}{R}\sum_{r=1}^{R} \bar{q}r$;

$\bar{s}(\bar{q})$ is an estimate of the variance of q, $\bar{s}^2(\bar{q}) = \frac{1}{R(R-1)} \sum_{r=1}^{R} (\bar{q}r - \bar{q})^2$;

f = R -1 is the degrees of freedom, R is the number of replications, α is the confidence coefficient [Ban01].

Time	Average	Std. Deviation	Confidence Interval 95%
Local Msg. (us)	3.655	0.16843255	3.562 = X = 3.748
Init. Time (ms)	99.811	24.03019409	86.534 = X = 113.089
Simulation Time (s)	2.695	0.008052211	2.691 = X = 2.7
Total Exec. Time (s)	2.795	0.022725378	2.782 = X = 2.808

Table 1: Execution results of the Fire model using one machine

Table 1 shows the execution results of the fire model using one machine. In this table, the *Local Message time* is the time required to transmit a message from one *processor* to another in the same machine (i.e. invoking the *receive* method of the receiving *processor*), which explains the short time required to communicate among local *processors* (average of 3.655 microseconds). The *Initialization Time* is the time required by the simulator to load the model into memory, parse the configuration files, etc; this is done before starting the simulation process. The *Simulation Time*

is the total time spent running the simulation which begins before processing the first event and ends after processing the last event.

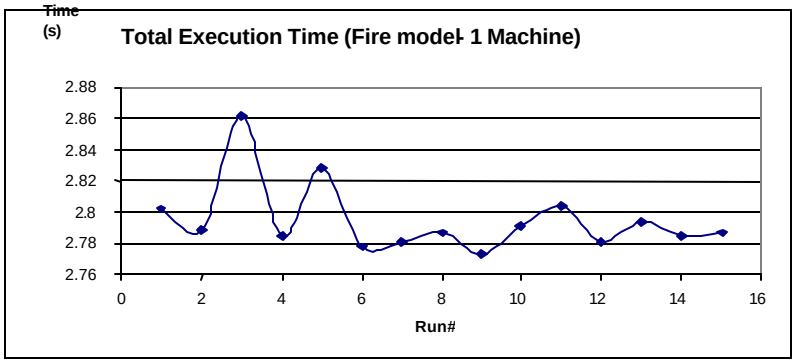

Figure 29: Fire model total execution time using one machine

Figure 29 shows the total execution time of the model for a number of runs. Although there are variations in the execution time of the fire model in one machine, the variations are very small compared to the average value of the total execution time (standard deviation of 0.022725378 with an average of 2.795 seconds, shown in Table 1). These variations are the result of the different processes running by the operating system on the machine.

In the second experiment, the cell space was split into two equal parts and each part was assigned to a different machine, as shown in Figure 30. Due to the nature of the Internet, the bandwidth between the machine in Ottawa and Montreal was not constant since the connection speed was dependant on the Internet usage in both sites and the load of the involved routers in between due to other traffic. In order to estimate the bandwidth available for the machines during the simulation runs, a separate software utility (Iperf [Gat06]) was run concurrently with the simulation.

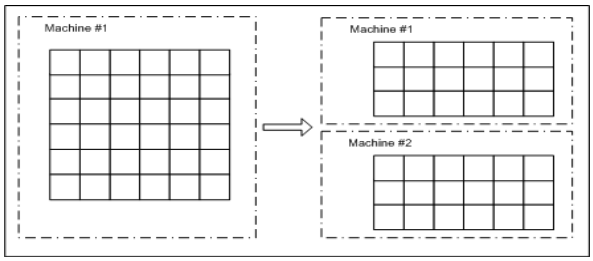

Figure 30: Fire model partitions on two machines

Table 2 shows the obtained results for this experiment. As we can see, the local message transfer is similar to single machine case (was shown in Table 1), since the messages are sent between local *processors*. However, sending a message from a processor to a remote one involves sending it through the Linux kernel (to reach the WS components), then distantly sending it as a SOAP message (through the Internet), and finally from the web receiving it (through the Linux kernel).

Time	Average	Std. Deviation	Confidence Interval (95%)
Local Msg. (us)	3.988	0.113841996	3.9251 = X = 4.051
Kernel Msg. (ms)	0.862	0.792427302	0.424 = X = 1.3
SOAP Msg. (ms)	892.631	177.5010084	794.553 = X = 990.708
Init. Time (ms)	315.006	352.3675322	120.307 = X = 509.705
Simulation Time (s)	98.977	5.17287701	96.119 = X = 101.835
Total Exec. Time (s)	99.292	5.191	96.424 = X = 102.161
Bandwidth (KB/s)	811.221	29.6063781	794.863 = X = 827.581

Table 2: Execution results of the Fire model using two machines (Internet)

The average time for message transfer through the kernel is 0.862 milliseconds. On the other hand, the time for SOAP transfer from one machine to another is much longer time, and the communication overhead over a long distance is the main contributing factor to the overhead associated with the distributed simulator. Another point to notice is that the *initialization time* is longer when running distributed simulation; this is due to the extra *processors* created to manage message passing among multiple machines (*master* and *slave* coordinators). However, this is an acceptable trade off between reducing the number of exchanged remote messages among distributed simulators (when adopting the master/slave structure) and increasing the initialization time which occurs only once during the simulation warm-up time. By comparing the execution time when using one and two machines, the overhead introduced by the distributed simulator can be visualized, as shown in Figure 31.

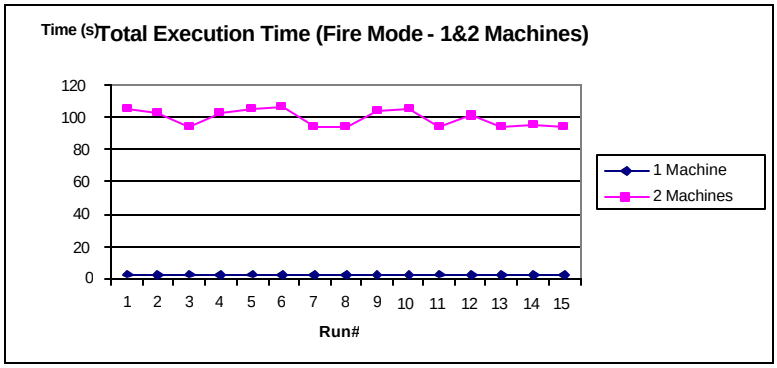

Figure 31: Comparing the total execution time using 1&2 machines (Internet)

To minimize the overhead (introduced by the standard Internet connection), the two machines were connected through a P2P connection using UCLP. Iperf [Gat06] was used to estimate the average bandwidth as 241.13 M Bit/second. Table 3 shows the obtained results of this experiment. In this case, we can see that performance is improved through a dedicated high path connection. Another point to notice is that the variation in execution time when using UCLP is less than that when using a regular Internet connection (external traffic is controlled).

Time	Average	Std. Deviation	Confidence Interval (95%)
Local Msg. (us)	3.856	0.285877096	3.698 = X = 4.014
Kernel Msg. (ms)	0.709	0.516410394	0.424 = X = 0.995
SOAP Msg. (ms)	489.343	178.9398125	390.470 = X = 588.215
Init. Time (ms)	256.101	349.078392	63.219 = X = 448.983
Simulation time (s)	27.622	0.44313255	27.377 = X = 27.867
Total Exec.Time (s)	27.878	0.539100354	27.580 = X = 28.176

Table 3: Execution results of the Fire model using two machines (UCLP)

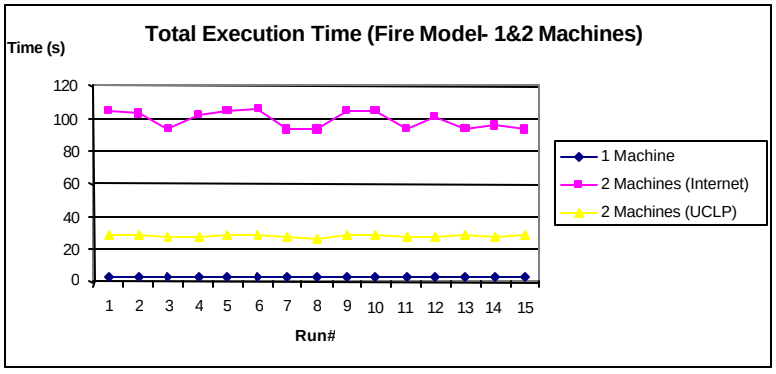

Figure 32: Comparing the total execution time using 1&2 machines (Internet, UCLP)

The results of another model executing with the same configurations are discussed next. This model represents the formation of a sand-pile [Saa03]; it consists of a DEVS model generating particles and a coupled Cell-DEVS model representing the sand-pile formation. A comparison of the results of both the fire and sand-pile models is summarized in Table 4. As the table shows that our previous discussion about the fire model results still applies to the sand-pile results.

Time	Fire#1	Fire#2 (Int.)	Fire#2 (UCLP)	Sand-pile#1	Sand-pile#2(Int.)	Sand-pile#2(UCLP)
Init. Time (ms)	99.811	315.006	256.101	25.925	46.597	19.259
Sim. Time (s)	2.695	98.977	27.622	0.1091	50.439	8.117
Total Exec. Time (s)	2.795	99.292	27.878	0.135	50.485	8.136
SOAP Delay (ms)	0	892.631	489.343	0	846.544	483.525
Total No. of Messages	45974	47770	47770	3710	4191	4191
Local Messages (%)	100	96.24	96.24	100	88.52	88.52
Remote Messages (%)	0	3.76	3.76	0	11.48	11.48

Table 4: Summary of the execution results of the Fire and Sand-pile models

In order to study the contribution of the remote messages sent between remote *processors* to the overhead introduced by the distributed simulator, we compared the percentage of remote messages sent in each case. This slowdown of the simulation was compared with the percentage of the remote messages sent during the simulation in order to examine the relationship between them as shown in Table 5.

Model	Remote Msgs. (%)	Sim_Time2(Internet)/ Sim_Time1	Sim_Time2(UCLP)/ Sim_Time1
Fire model	3.76	36.73	10.25
Sand-pile model	11.48	462.32	74.4

Table 5: Percentage of remote messages in distributed simulation

As shown in Table 5, the more the remote messages exchanged among distributed simulators, the more it takes a model to complete the simulation regardless of the connection type. Therefore, we conclude that if speeding up the execution of simulation time of a model is the main purpose for using the WS-based distributed simulator, the modeler should carefully assess this decision which may depend on a number of factors such as how the model is partitioned among machines, the connection type (e.g. UCLP), the amount of computation in the model and the target machine(s) (e.g. a parallel machine with hundreds of nodes, hence less communication overhead).

6. Conclusions

We presented the design and implementation of the distributed simulation engine, known as D-CD++, which is an extension of CD++ to expose the simulation utilities as machine-consumable services. In addition, we presented the design and implementation of the WS components that enable D-CD++ to expose the simulation functionalities to remote users. The distributed simulator (D-CD++) enabled CD++ users to take advantage of the benefits provided by the grid environment such as sharing grid computing power, simulation models, experiments, etc. on a global scale. The D-CD++ extension (distributed simulation using WS) moved CD++ to a standard context which prepared CD++ to provide a shared standard for interoperability among different DEVS tools implementation in order to make sharing DEVS resources (simulators and models) among different teams around the world more effective. In addition, it allowed for integrating the simulation services into larger systems to form a complex workflow including visualization services. Another goal was to implement a distributed simulation engine using WS that provides users access to remote resources (such as remote existing simulation models and experiments, improving the sharing work and ideas), and enables them to execute complex models using multiple machines in a distributed manner. By establishing network connectivity among the machines, different simulators can exchange messages during the distributed session. The advantage of using SOAP is that it can be embedded into HTTP traffic which in turn can be used on different network infrastructures (e.g. LAN, WAN, Ethernet, fiber optic, etc.). Likewise, by providing WS along with distributed simulation algorithms, D-CD++ can be easily integrated to other environments using a mashup approach (by combining data and services provided by third parties, either explicitly

through open APIs, or inferred by the mashup author). We have already interfaced D-CD++ with environments like Eucalyptus [Liu07b], and Google maps [Har08], showing the usefulness of the approach.

D-CD++ manages the participant machines in the simulation via a master/slave coordinator structure. The master coordinator is responsible for passing messages between its child models and the upper level components in the model hierarchy. On the other hand, the slave coordinator is responsible for passing messages among local children, reducing the remote message traffic among the machines when running distributed simulations.

The WS components added to CD++ have introduced some overhead that is mostly apparent when running distributed simulations (the time of transferring a SOAP message from one machine to another is by far longer than the time it takes to exchange messages locally). This is especially true when the machines are connected using commodity Internet connections. The advancement in the area of application-controlled networks where the network management can be handled at an upper layer (the application layer), has enabled grid applications to take control on their needs of the network bandwidth. User Controlled Lightpath (UCLP) is a WS-based management services for fiber optic networks that were used in conjunction with CD++ in order to establish the connectivity between different machines in a distributed environment. Having a point-to-point connection between the machines running distributed simulation has improved the performance of the simulator in terms of shorter execution time of the model. In addition, the bandwidth could be relinquished when the application is not needed anymore, which results in an efficient use of the network resources.

We could see that the performance of the D-CD++ engine depends on the network connectivity among the nodes; which can be commodity Internet connections, or dedicated point-to-point links created using User Controlled Lightpath (UCLP). The major bottleneck in the system was identified as the communication overhead, particularly the long delay of the exchanged messages over long distances. However, in the performance analysis section we took a close look at a case study between Ottawa and Montreal which showed that the major performance bottleneck in the system is in the remote communication overhead (for obvious reasons and also because of the model under study was a small one, hence, it doesn't require a great deal of computation). However, the more the computation of a model is increased, the more the communication overhead decreases.

References

[Alo03] Alonso, G. *Web-Services : concepts, architectures and applications*. Springer. 2003.

[Ame01] Ameghino, J.; Troccoli, A.; Wainer, G. "Models of complex physical systems using Cell-DEVS". Proceedings of the 34th Annual Simulation Symposium. Seattle, WA. USA. 2001.

[Arn03] Arnaud, B.; Wu, J.; Kalali, B. "Customer Controlled and Managed Optical networks ". *IEEE/OSA Journal of Lightwave Technology, special issue on Optical Networks*. Vol. 21(11), pp. 2804-2810. November, 2003.

[Axi06] Web-Services-Axis. Available via <http://ws.apache.org/axis/>. [Accessed February, 2006].

[Ban01] Banks, J.; Carson, J.; Nelson, B.; Nicol, D. *Discrete-Event System Simulation*. Prentice Hall. 2001.

[Bec04] Patricia Beckmann, Scott Wells, Scott Wells. "Exploring 3D Modeling with Maya 6: Design Exploration Series". Cengage Delmar Learning 2004.

[Bra04] Bray, T.; Paoli, J.; Sperberg-McQueen, C.M.; Yergeau, F. "Extensible Markup Language, XML 1.0 (Third Edition)". February, 2004. Available via <http://www.w3.org/TR/2004/REC-xml-20040204/>. [Accessed October, 2005].

[Che04] Cheon, S.; Seo, C.; Park, S.; Zeigler, B.P. "Design and Implementation of Distributed DEVS Simulation in a Peer to Peer Network System". Advanced Simulation Technologies Conference, Arlington Virginia. April, 2004

[Chr01] Christensen, E; Curbera, F.; Meredith, G.; Weerawarana, S." Web Service Desctiption Language (WSDL) 1.1". March, 2001. Available via < http://www.w3.org/TR/wsdl>. [Accessed December, 2005].

[Fuj99] Fujimoto, R.M. *Parallel and Distribution Simulation Systems*. Wiley. 1999.

[Gat06] Gates, M.; Warshavsky, A. "Iperf version 1.1.1". February, 2000. Available via <http://dast.nlanr.net/Projects/Iperf1.1.1/>. [Accessed July, 2006].

[Glo05] "A Globus Primer". Available via <http://www.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf>. [Accessed January, 2006].

[Gud03] Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.; Nielsen, H. "SOAP Version 1.2 Part 1: Messaging Framework". June, 2003. Available via <http://www.w3.org/TR/soap12-part1/>. [Accessed November, 2005].

[JXT06] www.jxta.org. [Accessed June, 2006]

[Har08] Y. Harzallah, V. Michel, Q. Liu, G. Wainer. "Distributed Simulation and Web Map Mash-Up for Forest Fire Spread". Proceedings of IEEE ICWS. Honolulu, HI. 2008.

[Kha03] Khargharia, B.; Hariri, S.; Parashar, M.; Ntaimo, L.; Kim, B. "vGrid: A Framework for Building Autonomic Applications". International Workshop on Challenges for Large Applications in Distributed Environments (CLADE 2003), pp. 19-26. June, 2003.

[Kim04] Kim, K.; Kang, W. "CORBA -Based, Multi-threaded Distributed Simulation of Hierarchical DEVS Models: Transforming Model Structure into a Non-hierarchical One". International Conference on Computational Science and Its Applications (ICCSA). Assisi, Italy. 2004.

[Kim05] Kim, K. and Kang, W., "A Web Service Based Distributed Simulation Architecture for Hierarchical DEVS Models". LNCS 2005.

[Lia99] Liang, S. *Java Native Interface (JNI), Programmer's Guide and Specification*. Addison-Wesley. 1999

[Liu07a] Q. Liu, G. Wainer. "Parallel Environment for DEVS and Cell-DEVS Models". *Simulation, Transactions of the SCS.* Vol. 83, No. 6, 449-471 (2007).

[Liu07b] Liu, S; Liang, Y.; Xu, B.; Zhang, L.; Spencer, B.; Brooks, M. "On Demand Network and Application Provisioning ThroughWeb-Services". IEEE International Conference on Web-Services (ICWS). Salt Lake City, USA. 2007.

[Mad07a] Madhoun, R. and G. Wainer. "Studying the Impact of Web-Services Implementation of Distributed Simulation of DEVS and Cell-DEVS Models". Proceedings of the 2007 DEVS Integrative M&S Symposium (DEVS'07), Norfolk, VA, USA. 2007.

[Mad07b] Madhoun, R., B. Feng, and G. Wainer. "On the Creation of Distributed Simulation Web-Services in CD++". Proceedings of the 14th AI, Simulation and Planning in High Autonomy Systems (AIS 2007), Buenos Aires, Argentina. 2007.

[Mit07a] S. Mittal, J.L. Risco. "DEVSML: Automating DEVS Execution over SOA Towards Transparent Simulators". In Special Session on DEVS Collaborative Execution and Systems Modeling over SOA, DEVS Integrative M&S Symposium DEVS'07, March 2007.

[Mit07b] Mittal, Risco and Ziegler. B. "DEVS-Based Simulation Web-Services for Net-Centric T&E". Proceedings of SCSC 2007. San Diego, CA. 2007.

[OMG02] Object Management Group. The common object request broker: architecture and specification. Revision 3.0. OMG Technical report. June, 2002. 492 Old Connecticut Path, Framingham, MA. USA.

[Rad98] Radhakrishnan, R., D.E. Martin, M. Chetlur, D.M. Rao, and P.A. Wilsey. 1998. "An Object-Oriented Time Warp Simulation Kernel". In Proceedings of the 2nd International Symposium: Computing in Object-oriented Parallel Environments (ISCOPE 98), Santa Fe, NM, LNCS 1505: 13-23.

[Saa03] Saadawi, H.; Wainer , G. "Modeling a sand pile application using Cell-DEVS". Proceedings of the 2003 Summer Computer Simulation Conference. Montreal, QC. Canada. 2003.

[Seo04] Seo, C.; Park, S.; Kim, B.; Cheon, S.; Zeigler, B. "Implementation of Distributed high-performance DEVS Simulation Framework in the Grid Computing Environment". Advanced Simulation Technologies conference (ASTC). Arlington, VA. USA. 2004.

[Tom06] Apache Tomcat. Available via <http://tomcat.apache.org/>. [Accessed February, 2006].

[Tra06] Travostino F., Mambretti J., Karmous E.G. "Grid Networks: Enabling Grids with Advanced Communication Technology". Wiley. 2006.

[Tro03] Troccoli, A., Wainer, G. "Implementing Parallel Cell-DEVS". Proceedings of 36th IEEE/SCS Annual Simulation Symposium. Orlando, FL. USA. 2003.

[Wai00] Wainer, G. "Improved Cellular Models with Parallel Cell-DEVS". *Transactions of the Society for Computer Simulation International.* Vol. 17(2), pp. 73-88. June, 2000.

[Wai02a] Wainer, G. "CD++: a toolkit to develop DEVS models". *Software - Practice and Experience.* vol. 32, pp. 1261-1306. 2002.

[Wai02b] Wainer, G. and N. Giambiasi. "N-dimensional Cell-DEVS Models". *Discrete Event Dynamic Systems 12(2),* (April 2002): 135-157.

[Wai07] Wainer, G.; Zeigler, B.; Nutaro, J.; Kim, T. et al. "DEVS Standardization Study Group Interim Report". http://www.sce.carleton.ca/faculty/wainer/standard/.

[Wai08] G. Wainer. "Discrete-Event Modeling and Simulation: a practitioner's approach". Taylor and Francis. In Press. 2008.

[Yu07] J. Yu, G. Wainer. "E-CD++: a tool for modeling embedded real-time applications". In *Proceedings of the 2007 SCS Summer Computer Simulation Conference.* San Diego, CA. 2007.

[Zei99] Zeigler, B., and H. S. Sarjoughian. "Support for Hierarchical Modular Component-based Model Construction in DEVS/HLA". *Proceedings of the 1999 Spring Simulation Interoperability Workshop,* Orlando, FL, USA, (March 1999).

[Zei00] Zeigler, B.; Kim, T.; Praehofer, H. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press. 2000.

[Zei07] Zeigler, B. and Hammonds, P. “Modeling & Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange”. Elsevier Academic Press. 2007.

[Zha05] Zhang, M.; Zeigler, B.; Hammonds, P. "DEVS/RMI-An Auto-Adaptive and Reconfigurable Distributed Simulation Environment for Engineering Studies". *ITEA Journal*. July. 2005.