Studying the Impact of Web-Services Implementation of Distributed Simulation of DEVS and Cell-DEVS Models

Rami Madhoun
Dept. of Systems and Computer Eng.
Carleton University, Ottawa, ON
K1S-5B6 Canada
rmadhoun@sce.carleton.ca

Gabriel Wainer
Dept. of Systems and Computer Eng.
Carleton University, Ottawa, ON
K1S-5B6 Canada
gwainer@sce.carleton.ca

ABSTRACT

DEVS is a Modeling and Simulation formalism that has been used to study the dynamics of discrete event systems. Cell-DEVS is a DEVS-based formalism that defines the cell space as a group of DEVS models connected together. This work presents the design and implementation of a distributed simulation engine based on CD++; a modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models. The proposed simulation engine follows the conservative approach for synchronization among the nodes, and takes advantage of web service technologies in order to execute complex models using the resources available in a grid environment. In addition, it allows for the integration with other systems using standard web service tools. The performance of the 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 Light Path (UCLP). UCLP is a web service-based network management tool used by grid applications to allocate bandwidth on demand.

1. INTRODUCTION

Modeling and simulation (M&S) plays an important role in studying complex natural and artificial systems. For some systems, analytical analysis is not always feasible due to the complexity pertinent to them, for others, it is too dangerous or impractical to experiment with them. One of the fields of M&S is discrete event simulation which is related to studying systems that exist in finite set of discrete states over continuous periods of time. Some examples of these systems include customer queues in a bank, computer networks, and manufacturing facilities.

Discrete Event System Specification (DEVS) [Zei00] is a modeling and simulation formalism that has been used to study discrete event systems. It depends on modeling the system as hierarchal components, each of which has input and output ports to interact with other components and with the external environment. The success of using the DEVS approach in the field of M&S has inspired researchers to define other DEVS-based formalisms. In this regard, Timed Cell-DEVS [Wai01] is an extension to the traditional cellular automata [Wol86]; it allows for representing each cell in the cell space as a DEVS model that is only activated when it receives external inputs from its neighbouring cells. This improves the performance of the simulation since only active cells are evaluated as opposed to evaluating the whole cell space as in the case of cellular automata.

CD++ [Wai02] is a modeling and simulation toolkit that was 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. Each DEVS atomic model has a simulator and each coupled DEVS model (group of atomic and/or coupled models connected together) has a coordinator to represent its behaviour. The simulation is carried out by processing events by the simulators and coordinators and 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][Gli04] runs on high performance distributed-memory clusters, and the real time version runs on specialized real-time hardware [Gli02].

As the system under study gets more complicated, the model complexity tends to increase. This causes more resources to be needed in order to execute the model, in which case using a single machine to run the simulation may be impractical. This has inspired the research in the area of parallel and distributed simulation in order to use the hardware resources in distributed environments to execute complex models. At the same time, as more and 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. Grid computing represents a new paradigm for sharing compute and storage resources in heterogeneous environments where resources reside on different platforms connected together using standard communication protocols. In a grid environment, resources are virtualized as services consumed by clients in a way similar to the way electricity is consumed in a power grid. The objective of grid computing is to provide the client with compute and storage “services” on demand, with minimal or no limitation to the platform on which these resources reside. Some of the grid middleware adopted web services to facilitate grid application development and to expose the application functionality in a platform-independent manner. The use of the parallel simulation algorithms with the emerging grid and web service technologies provides an appealing opportunity to use the resources available in a grid environment to run complex distributed simulations. In this context, the idle CPU time and memory resources in a machine can offer simulation “services”

to remote users/services while the local user is performing other tasks.

We are interested in running increasingly complex models that represent natural and artificial systems and to integrate this capability with larger systems to provide better use of the simulation results. Although other versions of CD++ have been developed to run complex models on distributed-memory clusters, they are specific in terms of the hardware, software, and network connectivity among the nodes running the simulation. We aim at providing a flexible framework for integrating resources running on commodity hardware and connected using commodity Internet connections to run complex models.

The need to integrate the simulation capabilities into larger systems is evident when the user of the simulator is not proficient in interpreting the simulation results or when it is not convenient for him to do so. Our objective of using web services is to provide standard means of interacting with the simulator taking into account the wide spread of web service technologies in grid environments. The examples in which simulation can be applied in order to better understand the system under study are countless. One of these examples is using an orchestration language such as Business Process Execution Language (BPEL) [And03] to establish a workflow between the simulation services and other services such as visualization services. These services are being integrated in a larger project in order to help architecture engineers to simulate different incidents taking place in their designs and visualize the effect of their decisions on people's behaviour in case of emergency. By being able to design a building, simulate the people's behaviour in that building, and visualize the results of the simulation, the architects can have better understanding of the consequences of their designs. The resources used for that project are located in geographically dispersed locations that are connected together using User Controlled Light Path (UCLP) [Arn03]. UCLP is a web service-based network management tool that can be easily integrated with the simulation services. This allows for on-demand connectivity between the simulation services, the visualization services, and the users, in a seamless and efficient manner.

2. BACKGROUND

Discrete Event System Specification (DEVS) [Zei00] is a M&S specification that is aimed to study discrete event systems. The formal definition of DEVS models is given as [Zei00]:

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

The model exists initially in state *s*, and it was scheduled to remain in that state for duration of *ta(s)*. However, before *ta(s)* is elapsed, the model receives an external input (x), which causes the model to execute its *external transition* function (δ_{ext}) in order to evaluate the model's new state after receiving the input. The external transition function takes into account the model's *total state (Q)*, which is defined by the model state (*s*) and the time elapsed since the model was in that state (*e*). Had the model not received an external input, it would have executed the *output function (?)* after being in state *s* for *ta(s)* time units. This would have been followed by the *internal transition* function (δ_{int}), which determines the model's next state because of an internal transition.

An exceptional case may take place if the states of two different models connected together expire at the same time. The decision of whom to evaluate next may have some implications on the correctness of the model. This situation may have a serialization effect on the model, and the decision as of which model to evaluate first is left to the modeller through the *select* function. In order to overcome this issue, Parallel-DEVS (P-DEVS) [Cho94a] formalism executes all the imminent models (models with the earliest scheduled state change) in parallel. This has a major effect on allowing the DEVS simulator to take advantage of the parallelism that might be available in the model and in the hardware resources (in the case of using parallel machines to run the model). In P-DEVS, the model has two message bags, one to store the external input messages, and the other is used to store the output messages. The formal definition of a P-DEVS model is presented in [Cho94a]:

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

The main difference between DEVS and P-DEVS formalisms is the addition of the *confluent* function (δ_{conf}), which is responsible for determining the next state of the model when an external input arrives at the same time of an internal transition. The definition of the *confluent* function is determined by the modeller so that the correct behaviour can be modeled depending on the system under study. The physical system model is created by integrating the different DEVS models together though their input and output ports; resulting in a *coupled DEVS model*. A coupled DEVS model consists of atomic and/or other coupled models connected together.

Cell-DEVS [Wai01] is an extension to cellular automata [Wol86] that depends on defining the cell as an atomic DEVS model. The asynchronous evaluation of the cells provides the modeller with powerful means to define complex temporal behaviours. Two types of delays can be defined; *transport* delay simulates queued future states. Another type of delay is *inertial* delay. Using the *inertial* delay, the newly evaluated state will pre-empt the scheduled one if they were different. Since each cell is represented as an atomic DEVS model, the cell behaviour is defined by the various functions used to define an atomic DEVS model. Once an external input arrives to the cell from one of its neighbours, it activates the *external transition* function, which calculates the next state of the model. The *time advance* function is represented by the delay associated with the cell. Once the delay expires, the *output* function is triggered to generate the cell's output, followed by the *internal transition* function, which evaluates the cell's new state. The limitation associated with the original DVES model definition, in terms of activating only one DEVS model at a time (through the *select* function) restricts the capabilities of the coupled Cell-DEVS model. The Parallel Cell-DEVS formalism [Wai00] was introduced to extend the functionality of the Cell-DEVS formalism taking advantage of the features provided by the Parallel-DEVS formalism; which include, executing imminent models in parallel avoiding the serialization problem that can lead to incorrect execution of the model. Coupled Cell-DEVS models can be formed by connecting different cells together. The cell space can take different dimensions and shapes. For example, 2D cell space can be used to model the spread of fire in a forest; 3D cell space can be used to model the spread of a specific type of viruses in a city. The

borders of the coupled cell DEVS model can be one of two types; a *wrapped* border indicates that the cells at the edge of the cell space are neighboured by the cells on the opposite side. On the other hand, *non-wrapped* border indicates that the cells at the borders have special rules that need to be defined by the modeller.

CD++ [Wai02] is an object-oriented modeling and simulation toolkit capable of executing DEVS and Cell-DEVS models. CD++ executes the model by creating a collection of *model* and *simulator* classes following [Zei00]. In order to run in distributed environments, the model is decomposed into components that are executed by different simulators running on multiple machines.

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.:

- **DEVS/Grid** [Seo04] implements a grid-enabled DEVS simulator following a layered approach.
- **vGrid** [Kha03] is an overall architecture for running DEVS and Cell-DEVS models in grid environments.
- **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.
- **DEVS/RMI** [Zha05] is a distributed DEVS simulator based on Java Remote Method Invocation (RMI). It aims at providing a fully re-configurable distributed simulation environment with the capability of load-balancing and fault-tolerance.
- **DEVS/Cluster** [Kim04] is multi-threaded distributed DEVS simulator based on CORBA [OMG02].
- **PCD++** [Tro03] [Gli04] is a parallel simulation engine developed using WARPED [War06] middleware and uses MPI [MPI95] for communications. PCD++ uses Time Warp [Jef85] protocol for synchronization among the different nodes participating in the simulation.

The methodology we followed to design and implement a distributed simulation engine depends on extending the CD++ toolkit in two dimensions. In one dimension, the toolkit was *wrapped* by a web service wrapper to expose its functionality to remote users/services using SOAP. We use the main web service 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 web service and the CD++ engine were extended to execute distributed models in a grid 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. The difference between the approaches followed by other grid-based DEVS simulation engines and our approach, 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.

Web services are group of standards and languages aiming to facilitate developing, publishing, and discovering web-enabled applications. In other words, a web service 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 web service in a manner prescribed by its description using SOAP [Gud03] messages, typically conveyed using HTTP with an XML serialization in conjunction with other standards [Alo03]. WSDL documents include enough information for the web service clients in order to know the operations it offers, the parameters required to invoke an operation, and the return type of the operation. SOAP plays an important role in any web service transaction. It is the messaging protocol used to convey information to and from the web service. It was designed in a manner that enables decentralized communication among multiple parties.

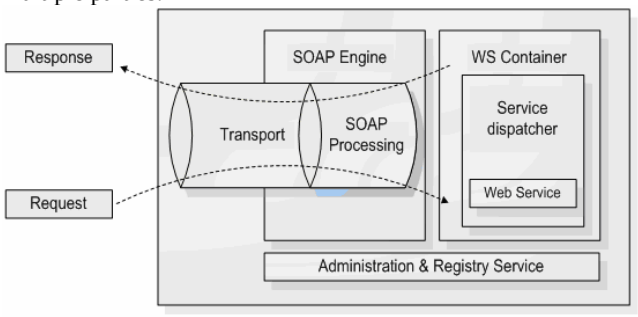

Figure 1. A web service container [Glo05]

3. WEB SERVICE-ENABLED CD++

CD++ was developed as traditional command-line application to run on Unix/Linux platform. It is capable of executing two kinds of models, DEVS and Cell-DEVS. To execute DEVS models, the modeller needs to define each atomic DEVS model as a C++ class (defined in header (h) and implementation (.cpp) files) that is to be integrated in the class hierarchy of CD++. For coupled DEVS models, and Cell-DEVS models, the modeller needs to provide a model definition file in a text format. The model definition file includes (among other things) the coupling scheme for the coupled model, 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. The output file contains the events that were generated through the output ports of the model; the log files contain detailed information about the progress of the simulation and can be used for debugging or animating the results using a visualization engine [Kha05]. In the context of our modeling and simulation environment, web services are introduced to serve two main purposes:

i) To expose the functionality of the CD++ toolkit as a web service, allowing for executing simulations and retrieving the results through web services, as shown in Figure 2.
ii) Using SOAP as a messaging protocol to enable distributed CD++ to execute complex models on multiple machines.

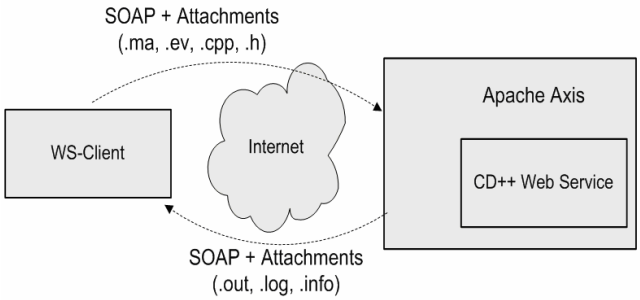

Figure 2. A typical invocation of the simulation web service

The simulation web service was redesigned to avoid the limitations of the JVM and provide a robust environment for running different simulation sessions concurrently and independently. The simulation service was split into two independent and separate parts: the *web service components* (implemented in Java) are used to handle the web service activities of the simulation service, and the *simulation components* (implemented in C++) are used to interact with CD++ by accessing and manipulating its internal objects and data structures. Both parts interact with each other though message queues maintained by the Linux kernel (through the *WrapperProxy*).

The advantages of this approach are that:

i) It provides a separate running workspace for each simulation session; the simulator runs as an operating system process independent from the simulators running other sessions.

ii) It allows for extending the functionality of each part with minimal or no change to the other part. For example, the simulation components of the service were developed to work with the parallel version of CD++ (PCD++) with minimal changes to the web service components.

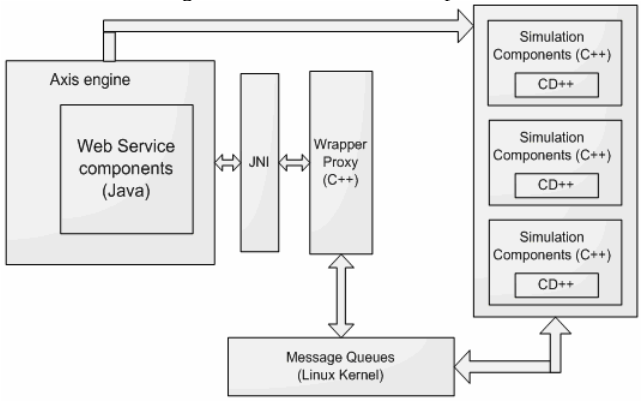

Figure 3. Simulation service using JNI/message queues

The *web service components* of the simulation service are compiled into Java archive (*.jar*) files and deployed in an Axis server, which in turn runs within an Apache Tomcat server. When the Tomcat server is started, it automatically starts the Axis engine. Axis loads all the libraries available in the directory of deployed services, which include the *JavaWrapper* (the backbone of the web service components), the server-side stubs, and the client-side stubs. In addition, when the *JavaWrapper* class is loaded, it loads the *WrapperProxy*, which is implemented as a collection of C/C++ procedures, and is loaded as a shared native library into the JVM. At this point the simulation service is considered ready to receive client requests.

3.1 Service Architecture

The web service components were developed as a collection of Java classes; they fall into three main categories:

i) The web service wrapper (*WS-Wrapper*): is responsible for most of the functionality of the web service components. This is the backbone of the *web service components* since it is linked to the server-side stubs deployed within the Axis server. When Axis receives a web service request from the client, it passes the request to the server-side stub, which in turn retrieves the instance of the *JavaWrapper* class associated with the user's session, before executing the corresponding method in the *JavaWrapper* object to fulfill the client's request.

ii) Utility classes: are used to perform secondary functions required by the *WS-Wrapper* such as parsing the users and configuration files. This takes place at two points: when the service is started, the *users file* is parsed to load the user information such as usernames, passwords, etc; and when the user submits a *grid configuration file*, the file is parsed to retrieve the model partition information as well as the addresses of the nodes participating in the simulation.

iii) Stub classes: include the client-side and server-side stubs. The server-side stub classes are required by the Axis server and are part of the code required to define and deploy the service. The client-side stubs are required by the *JavaWrapper* class to invoke the services offered by the slave nodes when running distributed simulations.

Some of the operations performed by the *JavaWrapper* include:

- User authentication.
- Session initialization: Part of the session creation process includes creating a *JavaWrapper* instance to handle the newly created session; this instance will be used by the server-side stub class deployed within the Axis server to fulfill the requests submitted by the user.
- Setting the model definition.
- Setting configuration information for distributed sessions.
- Starting the simulation: this includes some initialization to take place such as compiling the submitted DEVS models with the source code of the simulator, sending the model definition to slave machines, and starting the slave sessions.
- Checking the status of the simulation: This is used since some models might take long time to be executed; in which case, the client can start the simulation and do some other processing until the simulation is over. In addition, the user can kill the simulation process (if needed).
- Retrieving the results of the simulation: In case of running distributed simulations, the *JavaWrapper* will utilize the services running on the slave machines in order to retrieve and archive all the log files.
- Logging off: 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.

In general, the services offered by the simulation service through its WSDL interface, are mapped into methods invoked on the *JavaWrapper* class/instance. Parts of the methods defined in the

JavaWrapper class are actually native methods that were implemented in C/C++. Those constitute the *WrapperProxy* component of the service, and are implemented as procedures written in C/C++ since Java can't access the Linux message queues. These methods are interfaced to the *JavaWrapper* class using the Java Native Interface (JNI) [Lia99].

The *JavaWrapper* class uses utility classes to handle tasks such as parsing the *users* and *grid configuration* files. The *Parser* class is the main class used for parsing and it uses the *SAXParser*, *SAXParserFactory*, and *MyContentHandler* classes to do so. The users file is used for authentication and it contains the usernames, passwords, and roles for all the users that are authorized to use the service. The grid configuration file is an XML file that contains:

i) URLs of the simulation services participating in a session;
ii) Model partitioning information, which includes the parts of the model running on each machine in a distributed session.

Client and server-side stubs are required for the deployment and utilization of the simulation service. While the client stubs are not a must for using the simulation service, the client can create the SOAP requests dynamically, the server stub classes are required by the Axis server in order to properly deploy the service. The *CDppPortTypeSoapBindingImpl* represents the server-side stub; when the Axis server receives a request from the client in the form of a SOAP message, it does some processing on the SOAP message and extracts the attributes necessary to execute the service. Once the attributes are extracted, it invokes a method in the *JavaWrapper* class corresponding to the operation requested by the client. The *CDppPortTypeService* and *CDppPortTypeServiceLocator* are used to locate the web service using its Unified Resource Locator (URL). The former is an interface that is implemented by the latter and it is usually used at the beginning of any web service invocation process. The *CDppPortTypeSoapBindingStub* is a client-side stub that can be used by the program accessing the simulation service. It defines the attributes and methods that allow the client to deal with the web service as if it was local classes residing on his machine.

4. DISTRIBUTED CD++ (DCD++)

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. 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. The Parallel-DEVS (P-DEVS) algorithms [Cho94a] 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 modeller to define the behaviour 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 5 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.

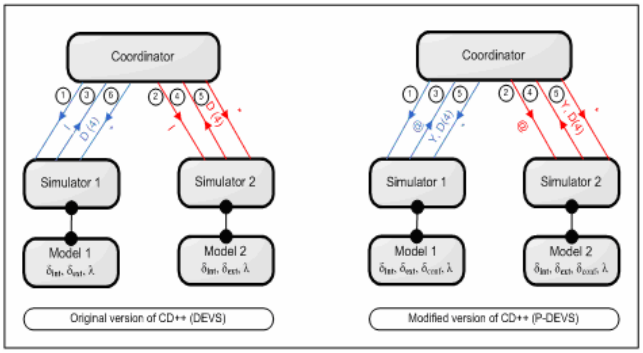

Figure 5. Concurrent model activation in Parallel-DEVS

Implementing the P-DEVS algorithms required changes to be made in the class and model hierarchies of CD++. 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. Those include the *receive* methods, which are responsible for receiving and processing the different simulation messages. The messages are sent among *processors* through the *MsgAdmin* class. The sending *processor* would send the message

to the *MsgAdmin* through the *send* method, which will cause the message to be queued until it gets sent. Sending a message is done by executing the *receive* method on the receiving *processor*.

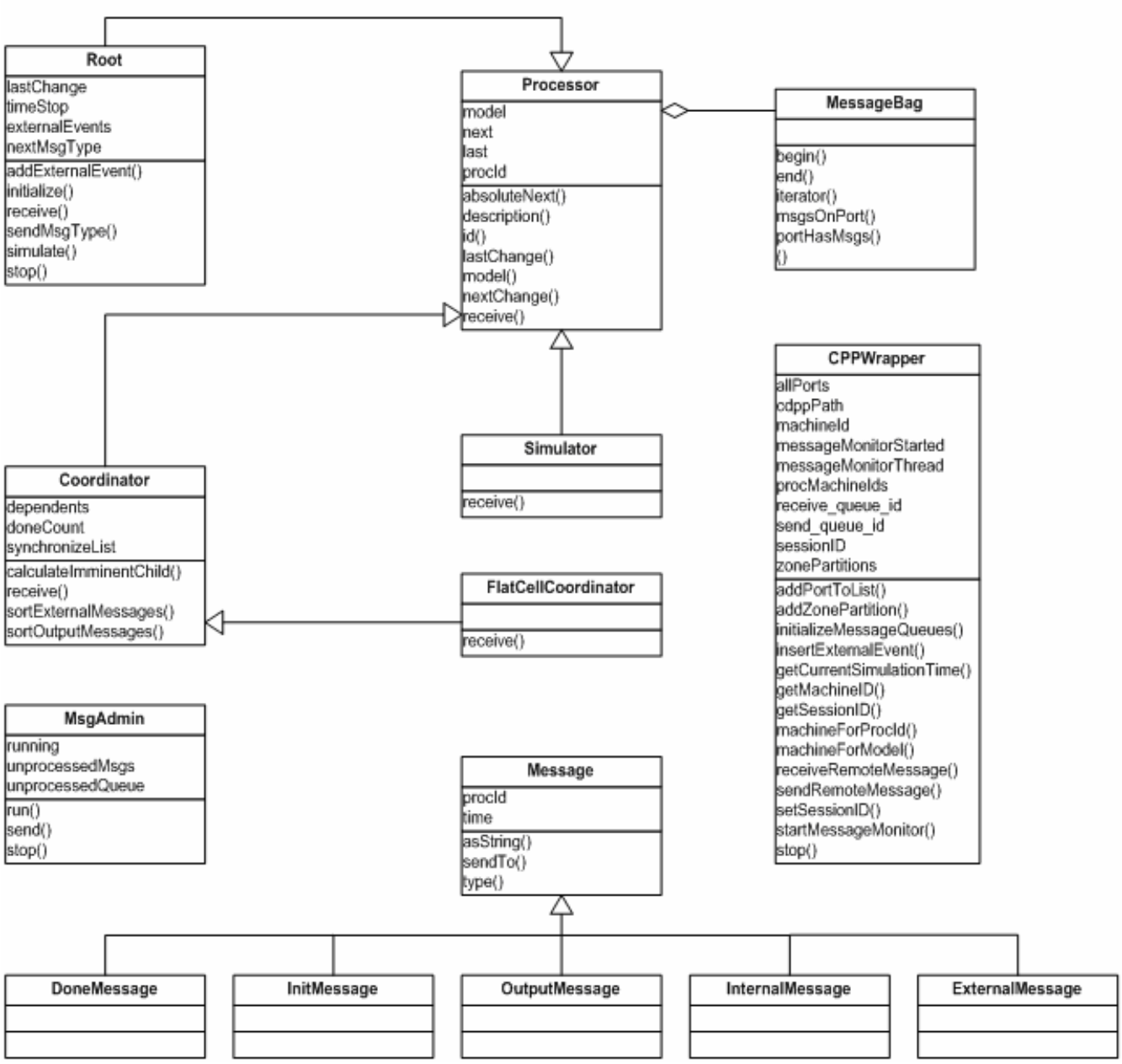

Figure 6. The simulation class hierarchy

The *Simulator* class extends the *Processor* class and overrides the *receive* function in order to execute the function of the 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 model in order to generate the model output. This is followed by the *Simulator* sending a *done message* to the coordinator reporting the time of the next change of the model. The *Simulator* receives only specific types of messages; no *done* or *output messages* are received by the *Simulator*.

The *Coordinator* class is responsible for forwarding messages among the *Simulators* and for synchronizing the events taking place during the simulation. The *receive* method has the same functionality as in any *processor* class, but the behaviour of the method is different from that in the *Simulator* class. That is, to implement the P-DEVS algorithms, the *coordinator* receives all kinds of synchronization and content messages and reacts accordingly. The message bag associated with the *coordinator* is processed through the *sortExternalMessages* method which gets invoked at the time of receiving an *internal message (*).* This causes the messages in the bag to be forwarded to their

destinations (*Simulators* and/or *Coordinators*). The *sortOutputMessages* method is invoked whenever a child *Simulator* or *Coordinator* sends an *output message* to its parent coordinator. This, results in the message either being translated into *external message(s)* sent to the local destination(s), or an *output message* being forwarded upward in the class hierarchy. The *calculateImminentChild* is responsible for evaluating the imminent child *processors* by examining the minimum time of the next state change. The *FlatCellCoordinator* 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.

Model partitioning information is provided 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 7, *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.

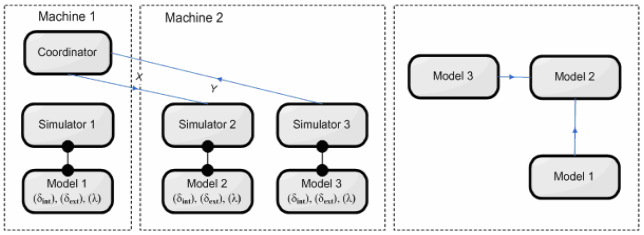

Figure 7. Unnecessary remote messages

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:* it is responsible for synchronizing the model execution, interacting with upper level coordinators and message routing among local and remote 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 8), causes the message from *Simulator 3* to *Simulator 2* to be sent locally improving the performance of the simulator.

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

i) The simulation mechanism is implemented mainly using the master and slave coordinators;
ii) The model loading mechanism is extended to maintain the partitioning information;
iii) The message passing mechanism is extended to handle local and remote message passing;

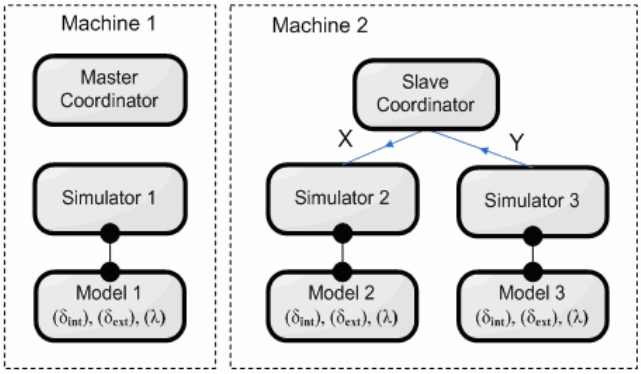

Figure 8. *Master* and *Slave* coordinators

4.1 A Sample Scenario

In order to present the overall operation of the simulator in a distributed environment, a coupled DEVS model is executed using two machines. The model consists of four DEVS models; the *generator* is an atomic DEVS model producing jobs to be processed by the *processor*, the *queue* is used to queue the 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*. The structure of the model is shown in Figure 9:

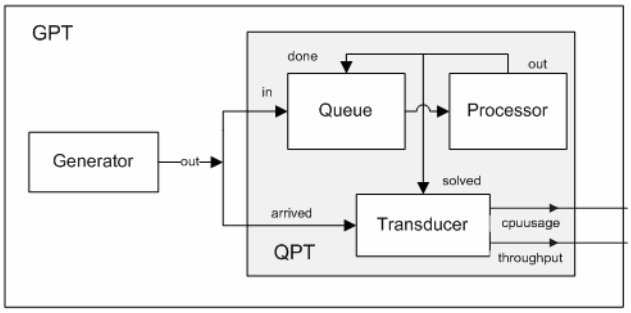

Figure 9. The Generator-Processor-Transducer (GPT) model

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 *queue*, *processor*, and *transducer* models were running on *Machine 2(Montreal)*. When loading the models and simulators, Machine 1 loads three *processors*: the *Root coordinator*, the *top master coordinator*, and the *generator*. Machine 2 loads the *top slave coordinator*, the *QPT* (coupled DEVS model consisting of the

Queue, *Processor*, and *Transducer* models) *master coordinator*, the *transducer*, the *queue*, and the *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. When the *top slave coordinator* receives the *initialization message*, it forwards it 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. DCD++ 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.

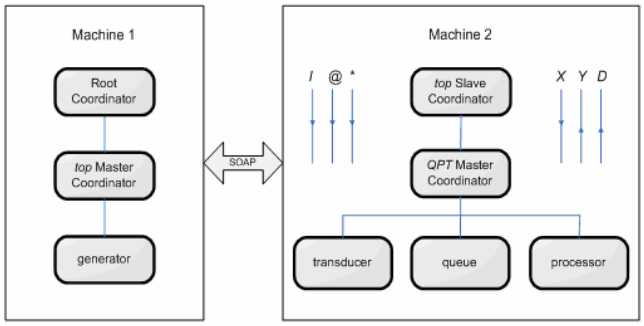

Figure 10. GPT model partitioning on two machines

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, which are the port name and message value sent through the port. Figure 11 shows an excerpt of the log file of Machine 1 while executing the *GPT* model. 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 11) 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 11) 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 11). 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 11 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".

```
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 11. An excerpt of the log file of Machine 1

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 12). 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 12); 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 12). The *internal message* sent to the *QPT master coordinator* is forwarded to the *queue* and *transducer* models (lines 8, 9 in Figure 12). 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 12). The *done message* (generated by the *top slave coordinator*) is forwarded to the *top master coordinator* using SOAP (line 14 in Figure 11). 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.

```
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 12. An excerpt of the log file of Machine 2

5. PERFORMANCE ANALYSIS

Using web services to implement the distributed simulation engine has allowed for the execution of complex models in grid environments. However, it introduced some overhead that affects the execution time of the models. That is, 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 SOAP message 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. The first part is the time it takes to transmit a message between the simulator and the web service through the Linux kernel; the other part is the time it takes to transmit the SOAP message between the two simulation web services.

In order to study the performance of the simulator, different sessions were executed using two machines; one of the machines was located in Montreal, and the other in Ottawa. 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, User Controlled Light Path (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 it takes in each run to transmit a message within the Linux kernel using message queues.
iv) The average time it takes in each run to transfer a local message within a single machine.
v) The bandwidth available for the simulator when using the Internet and UCLP connections.

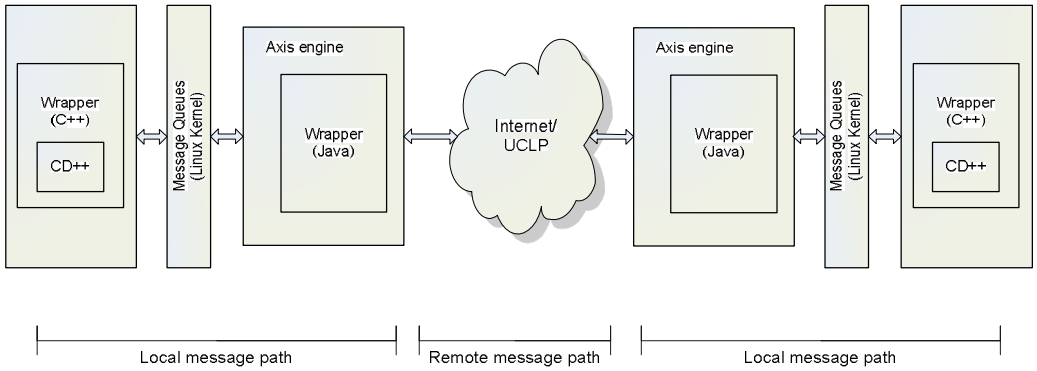

Figure 13. Sending remote messages in distributed simulation

The model used for performance analysis is a fire spread in a forest and it is implemented as 30x30 coupled Cell-DEVS model [Ame01]. It is composed of 30x30 cell space; each cell represents a square area of the forest. The cell is considered to be burned if its temperature exceeds a specific value. Figure 14 shows an excerpt of the model definition and initial values of the cells.

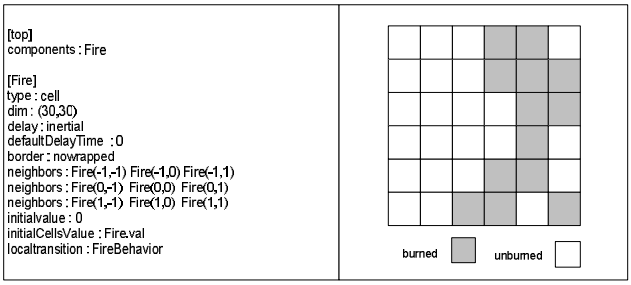

Figure 14. An excerpt of the Fire model definition

The cell space is 30x30 using *inertial* delay. The neighbourhood of the cell (defined by the *neighbors* construct), is defined by the 8 cells from all sides. *Fire(-1,-1)* represents the cell in the North West side (NW), *Fire(0, -1)* represents the cell to the W, etc.

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

	Average	Std. Deviation	Confidence Interval 95%
Local Msg. (us)	3.655	0.16843255	3.562=X= 3.748
Init. Time (ms)	99.811	24.0301940	86.53=X=113.0
Simulation Time(s)	2.695	0.00805221	2.691 = X = 2.7
Total Exec.Time(s)	2.795	0.02272537	2.782=X= 2.808

In order to study the performance of the distributed simulator, three types of experiments were performed. The first experiment was carried out using one machine in order to estimate the simulation time without the overhead incurred by sending remote messages using SOAP. The second experiment was conducted by splitting the fire model into two equal partitions; each of which was assigned to one machine that is connected to the other machine using a commodity Internet connection. In the third experiment, the two machines were connected using a dedicated P2P fibre optic link created using UCLP, as we discuss following. The Local Message time is the time required to transmit a message from one simulation *processor* to another in the same machine. The transmission of a local message in a single machine is implemented as a method call *(receive)* in the receiving *processor*, which explains the short time required to communicate between two 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 time of running the simulation which begins before processing the first event and ends after processing the last event.

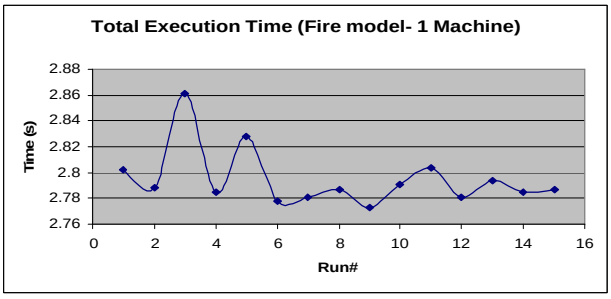

Figure 15. Fire model total execution time using one machine

Figure 15 shows the total execution time of the model. Although it shows variations in the execution time of the model in one machine, they 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). These variations are the result of the different processes and daemons running on the machine. In the second experiment, the cell space was split into two equal parts (15x30) and each part was assigned to run on a different machine.

Table 2: Results of Fire model using two machines (Internet)

	Avg.	Std. Dev.	Confidence Interval (95%)
Local Msg. (us)	3.98	0.113	3.9251 = X = 4.051
Kernel Msg.(ms)	0.86	0.792	0.424 = X = 1.3
SOAP Msg. (ms)	892	177.5	794.553= X = 990.708
Init. Time (ms)	315	352.3	120.307= X = 509.705
Simulation Time (s)	98.9	5.172	96.119 = X = 101.835
Total Exec.Time (s)	99.2	5.191	96.424 = X = 102.161
Bandwidth (KB/s)	811	29.60	794.863= X = 827.581

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. 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.

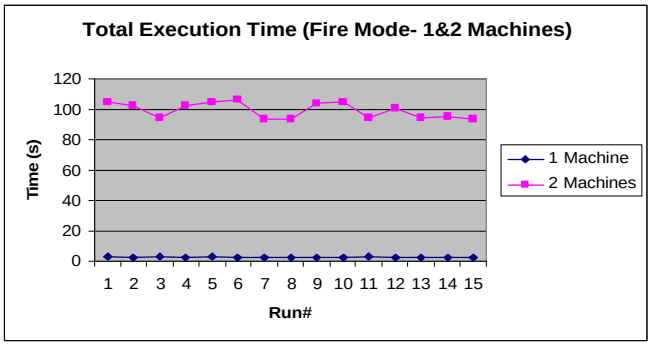

Figure 16. Comparing total execution time (Internet)

The local message transfer is close to that when using a single machine since the messages are sent between local *processors*. When two machines are used to run distributed simulation, sending a message from one processor to another remote one involves sending it through the Linux kernel first to reach the web service components of the simulation service, then sending it as a SOAP message through the network (Internet), and finally from the web service components to the simulator at the receiving end (through the Linux kernel). The average time for message transfer through the kernel is .862 milliseconds. On the other hand, the time for SOAP transfer from one machine to another is much longer than the kernel message transfer time, and it 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). 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 16.

To minimize the overhead incurred by the distributed simulator, the two machines were connected through a P2P connection using UCLP as opposed to using a commodity Internet connection. In order to estimate the bandwidth available to the simulator, Iperf [Gat06] was used to estimate the average bandwidth as 241.13 M Bit/second.

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

	Avg.	Std. Deviation	Confidence Interval (95%)
Local Msg.(us)	3.856	0.28587709	3.698 = X = 4.014
Kernel Msg.(ms)	0.709	0.51641039	0.424 = X = 0.995
SOAP Msg. (ms)	489.3	178.939812	390.470=X=588.215
Init. Time (ms)	256.1	349.078392	63.219=X = 448.983
Simul. time (s)	27.62	0.44313255	27.377 = X = 27.867
Total Exec. (s)	27.87	0.53910035	27.580 = X = 28.176

By examining the execution time when using UCLP, it was noticed that the performance is much better than that when using a regular Internet connection. That is, UCLP provides a dedicated P2P connection that is solely used for the simulation session. Another point to notice is that the variation in execution time when using UCLP is less than that when using a regular Internet connection.

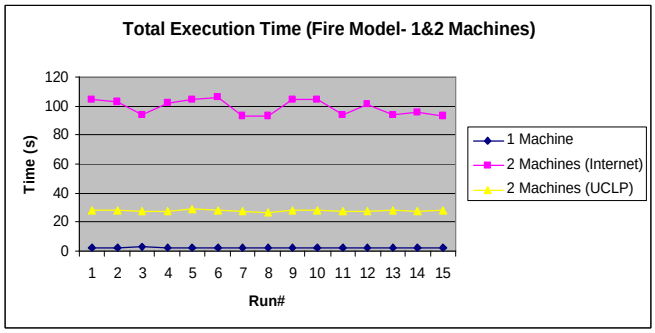

Figure 17. Comparing total execution time (Internet, UCLP)

In order to confirm the previous results, another model was executed following the same configurations: one machine, two machines connected via the Internet, and two machines connected via UCLP. The model is a sand-pile model [Saa03] consisting of a DEVS model simulating a sand particle generator and a coupled Cell-DEVS model representing the sand-pile formation. A summary of the results is shown in Table 4:

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

	Sand-pile#1	Sand-pile#2(Int.)	Sand-pile#2(UCLP)
Init. Time (ms)	25.925	46.597	19.259
Sim. Time (s)	0.1091	50.439	8.117
Total Exec. Time (s)	0.135	50.485	8.136
SOAP Delay (ms)	NA	846.544	483.525
No. of Messages	3710	4191	4191
Local Msg. (%)	100	88.52	88.52
Remote Msg. (%)	0	11.48	11.48

The overall results show few points that are worth emphasizing. The time to execute the model in one machine is usually shorter than that when using two machines. This is due to the overhead incurred by sending remote messages as SOAP, which seems to be the major contributor to the overhead. There are other factors affecting the overhead such as the time required to send messages through the Linux kernel (message queues); however, it is insignificant compared to the delay caused by SOAP. The initialization time for the Fire model was longer when running the simulation on two machines due to the extra coordinators required for message passing and synchronization (*master* and *slave* coordinators).

In order to study the contribution of the remote messages sent between remote *processors* to the overhead introduced by the distributed simulator, the average simulation times when using two machines were divided by those when using a single machine. The results are compared with the percentage of remote messages sent in each case. By dividing the simulation time when using two machines by the time when using one, a measure of the slowdown of the simulation can be obtained. This measure is compared with the percentage of the remote messages sent during the simulation in order to examine the relationship between the two.

Table 5: Percentage of remote messages

	Remote Msgs.(%)	Sim_Time2(Int.)/Sim_ Time1	Sim_Time2(UCLP)/Sim_ Time1
Fire	3.76	36.73	10.25
Sand-pile	11.48	462.32	74.4

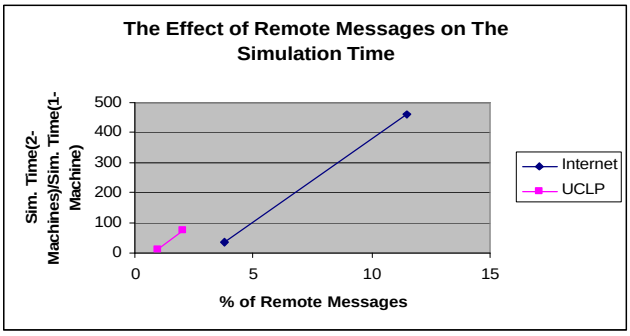

Figure 18. Remote messages/simulation times relation

Figure 18 shows the effect of the remote messages on the execution times of the models in distributed simulations. The effect is more evident when using regular Internet connections than when using UCLP. The curve in pink represents the slowdown of the model execution versus the percentage of remote messages when using commodity Internet connections. The curve in blue represents the slowdown when connecting the machines using UCLP.

6. CONCLUSIONS

Discrete event simulation plays an important role in studying complex systems, especially those that are not feasible for analytical studies. The nature of discrete event models tends to be more complex as the modeled system evolves or more information needs to be considered when developing the model. This has required more efficient simulation engines that are able to execute complex models in a reasonable amount of time. CD++ is a simulation engine that was developed to execute DEVS and Cell-DEVS models on different platforms. In this dissertation, a framework of using web services with CD++ was presented in order to accomplish two main goals.

The first goal is to interface the original version of the simulator to web service technologies using web service wrappers. This has enabled the modeller to execute the simulation, check the progress of the model execution, and retrieve the results remotely using SOAP (and its extensions) protocol. In addition, it allowed for integrating the simulation services into larger systems to form a complex workflow. Business Process Execution Language (BPEL) can be used in this context to integrate the simulation services with visualization services that enable the modeller to study the results of the model execution in a user-friendly manner. The other goal achieved through using web services, is the implementation of distributed simulation engine that is able to execute complex models using multiple machines. The model can be split into different partitions, each of which is assigned to run on a different machine. By establishing network connectivity among the machines, the different simulators can exchange messages during the distributed session using SOAP. The advantage of using SOAP is that it can be embedded into HTTP traffic which in turn can be used on different network infrastructures, such as LAN, WAN, Ethernet, fibre optic, etc.

The approach followed for implementing the distributed simulator depends on having *master* and *slave* coordinators. 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 its local children instead of involving the master coordinator that might be running on a different machine. This has a considerable effect of reducing the remote message traffic among the machines when running distributed simulations. This minimizes the overhead incurred with sending and receiving SOAP messages and hence improves the performance of the simulator.

The web service 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 Light Path (UCLP) is a web service-based management services for fibre 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 a lot in terms of shorter execution time of the model. In addition, the bandwidth could be relinquished when the application doesn't need it anymore, which results in an efficient use of the network resources.

7. REFERENCES

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

[And03] Andrews T.; Curbera, F.; Dholakia, H.; Goland, Y.; Klein, J.; Leymann, F.; Liu, K.; Roller, D.; Smith, D.; Thatte, S.; Trickovic, I.; Weerawarana, S. "Business Process Execution Language for Web Services version 1.1". May, 2003. Available via <http://www-128.ibm.com/developerworks/library/specification/ws-bpel/>. [Accessed February, 2006].

[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.

[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

[Cho94a] Chow, A.; Zeigler, B. "Parallel DEVS: A parallel, hierarchical, modular modeling formalism". Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.

[Cho94b] Chow, A.; Kim, D.; Zeigler, B. "Abstract Simulator for the parallel DEVS formalism". AI, Simulation, and Planning in High Autonomy Systems. Gainesville, FL. USA. 1994.

[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].

[Gli02] Glinsky, E.; Wainer, G. "Performance Analysis of Real-Time DEVS models". Proceedings of 2002 Winter Simulation Conference. San Diego, U.S.A. 2002.

[Gli04] Glinsky, E. "New Techniques for Parallel Simulation of DEVS and Cell-DEVS Models In CD++". Master Thesis. Carleton University 2004.

[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].

[Jef85] Jefferson, D.R. "Virtual time". ACM Transactions on Programming Languages and Systems. vol. 7(3), pp. 404-425. July, 1985.

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

[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.

[Kha05] Khan, A.; Wainer, G. "A visualization engine based on Maya for DEVS models". Proceedings of SISO Fall Interoperability Workshop. San Diego, CA. U.S.A. 2005.

[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.

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

[MPI95] Message Passing Interface Forum. MPI: A Message-Passing Interface standard (version 1.1). Technical report. Available via: <http://www.mpi-forum.org >. [Accessed May, 2006].

[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.

[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.

[San06] Sandy, L.; Liang, Y.; Spencer, B. "Eucalyptus: A Service-oriented Participatory Design Studio Supported by UCLP". Available via < http://www.cs.unb.ca/itc/ResearchExpo/posters/2006/abs20a.pdf >. [Accessed February, 2006].

[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].

[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.

[Wai01] Wainer, G.; Giambiasi, N. "Timed Cell-DEVS: modelling and simulation of cell spaces". Invited paper for the book Discrete Event Modeling & Simulation: Enabling Future Technologies. Springer-Verlag. 2001

[Wai02] Wainer, G. "CD++: a toolkit to develop DEVS models". Software - Practice and Experience. vol. 32, pp. 1261-1306. 2002.

[War06] Warped: A Time Warp Simulation Kernel. Warped Documentation for version 1.0. Available via <www.ececs.uc.edu/~paw/warped/>. [Accessed April, 2006.]

[Wol86] Wolfram, S. Theory and applications of cellular automata. Advances Series on Complex Systems. World Scientific. Singapore. 1986.

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

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