WEB SERVICE-BASED DISTRIBUTED SIMULATION OF DISCRETE EVENT MODELS

By

Rami Madhoun, B. Sc.

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of

the requirements of the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario

Canada

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis

Web Service-Based Distributed Simulation
of Discrete Event Models

Submitted by Rami Madhoun
In partial fulfillment of the requirements for the
Degree of Master of Applied Science

_____

Thesis Supervisor
Dr. Gabriel Wainer

_____

Chair, Department of Systems and Computer Engineering
Dr. Victor C. Aitken

Carleton University
2006

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 distributed 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.

Acknowledgments

This work is dedicated to my family for their end-less support and care. I also would like to thank Dr. Gabriel Wainer for his patience, advice, and for always being there when I needed him both on the academic and personal levels.

Table of Contents

ABSTRACT iii
List of Tables vii
List of Figures viii
Chapter 1: Introduction 1
1.1 Motivation and Goals 3
1.2 Contribution 4
1.3 Thesis Organization 6
Chapter 2: Grid Middleware for Discrete Event Modeling and Simulation 7
2.1 Discrete Event System Specification (DEVS) 8
2.2 Timed Cell-DEVS 13
2.3 The CD++ Toolkit 16
2.4 Distributed Simulation 25
2.4.1 Conservative Simulation 27
2.4.2 Optimistic Simulation 28
2.5 Web Services (WS) 29
2.6 Service Oriented Architecture (SOA) 34
2.7 User Controlled Light Path (UCLP) 35
Chapter 3: Trends in the Implementation of Distributed DEVS Simulators 37
3.1 Web Service-Based Approach for Distributed DEVS Simulation 43
Chapter 4: Web Service-Enabled CD++ 46
4.1 Design Methodology 46
4.2 Implementation Details 48
4.3 Service Architecture 53
4.4 Service Interface 60
Chapter 5: Distributed CD++ (DCD++) 65
5.1 Implementing the Parallel-DEVS Algorithms 67
5.2 Implementing the Simulation Components 73
5.3 Designing and Implementing Distributed-CD++ (DCD++) 74
5.3.1 Master and Slave Coordinators 77

5.3.2 Model Loading Mechanism 78
5.3.3 Message Passing Mechanism 80
5.4 Sample Scenario 80
5.5 Integrating Optimistic (PCD++) and Conservative (DCD++) Simulators 88
5.5.1 Interfacing DCD++ to PCD++ 89
5.5.2 Integrating DCD++ and PCD++ 91
Chapter 6: Performance Analysis 94
6.1 Experimental Models and Execution Results 95
6.2 Result Retrieval 111
Chapter 7: Conclusions 113
7.1 Future Research Work 114
References 116
Appendix-A: P-DEVS and DCD++ Simulation Algorithms 121
Appendix-B: Web Service Components 132

List of Tables

Table 1: *Atomic* class functions 16
Table 2: DEVS simulator messages 19
Table 3: Arguments of the *receiveRemoteMessage* operation 86
Table 4: Execution results of the Fire model using one machine 98
Table 5: Execution results of the Fire model using two machines (Internet) 100
Table 6: Execution results of the Fire model using two machines (UCLP) 102
Table 7: Execution results of the Sand-pile model using one machine 104
Table 8: Execution results of the Sand-pile model using two machines (Internet). 106
Table 9: Execution results of the Sand-pile model using two machines (UCLP).... 107
Table 10: Summary of the execution results of the Fire and Sand-pile models..... 109
Table 11: Percentage of remote messages in distributed simulation 110
Table 12: File transfer times via the Internet/UCLP..... 111

List of Figures

Figure 1: Main entities in a M&S environment [Zei00] 7
Figure 2: Informal definition of an atomic DEVS model [Zei00] 9
Figure 3: Coupled DEVS model 11
Figure 4: Cellular Automata 13
Figure 5: CD++ model and class hierarchies 17
Figure 6: Content and synchronization messages in CD++ 18
Figure 7: Barbershop model architecture 19
Figure 8: *BarberShop* model definition 20
Figure 9: An excerpt of the *Reception* class definition 21
Figure 10: An excerpt of the *Reception* class definition 22
Figure 11: An excerpt of the *Battlefield* model definition 24
Figure 12: An excerpt of the *Battlefield* rule definition 25
Figure 13: Causality errors in distributed simulation 27
Figure 14: An example of a SOAP message embedded in HTTP [Gud03] 32
Figure 15: Web service layers 33
Figure 16: A web service container [Glo05] 33
Figure 17: Major components of the simulation service 47
Figure 18: Implementing the simulation service using JNI and message queues 49
Figure 19: Simulation web service operation 52
Figure 20: Message queues connecting the simulation components to the *WrapperProxy* 53
Figure 21: Web service components UML diagram 56
Figure 22: A sample *grid configuration file* 58
Figure 23: A typical invocation of the simulation web service 60
Figure 24: An excerpt of the *message* definition of the simulation web service 61
Figure 25: An excerpt of the *portType* definition of the simulation web service 61
Figure 26: An excerpt of the *binding* definition of the simulation web service 62
Figure 27: An excerpt of the *service* definition of the simulation web service 63
Figure 28: Message exchange during a simulation cycle 65

Figure 29: Tie breaking using the *select* function 67
Figure 30: Concurrent model activation in Parallel-DEVS 69
Figure 31: The simulation class hierarchy 70
Figure 32: The *MainSimulator* class 72
Figure 33: Unnecessary remote messages in distributed simulation 75
Figure 34: The use of *Master* and *Slave* coordinators to avoid unnecessary messages 76
Figure 35: Master and Slave coordinator classes 78
Figure 36: DCD++ model hierarchy 79
Figure 37: The Generator-Processor-Transducer (GPT) model 81
Figure 38: GPT model partitioning on two machines 81
Figure 39: An excerpt of the log file of Machine 1 82
Figure 40: An excerpt of the log file of Machine 2 83
Figure 41: *createSlaveSession* request 85
Figure 42: An *initialization message* sent as SOAP from Machine 1 to Machine 2 87
Figure 43: *retrieveLogFile* response 88
Figure 44: Implementing the simulation web service with PCD++ 90
Figure 45: PCD++ architecture 91
Figure 46: Sending remote messages in distributed simulation 94
Figure 47: An excerpt of the Fire model definition 96
Figure 48: Fire model rule definition 96
Figure 49: Fire model simulation time using one machine 98
Figure 50: Fire model total execution time using one machine 99
Figure 51: Fire model partitions on two machines 99
Figure 52: Comparing the simulation time using 1&2 machines (Internet) 101
Figure 53: Comparing the total execution time using 1&2 machines (Internet) 101
Figure 54: Comparing the simulation time using 1&2 machines 102
Figure 55: Comparing the total execution time using 1&2 machines 103
Figure 56: An excerpt of the Sand-pile model definition 104
Figure 57: Simulation time of the Sand-pile model using one machine 104
Figure 58: Total execution time of the Sand-pile model using one machine 105

Figure 59: Sand-pile model partitions on two machines 105
Figure 60: Comparing the simulation time of the Sand-pile model using 106
Figure 61: Comparing the total execution time of the Sand-pile model using 107
Figure 62: Comparing the simulation time of the Sand-pile model using 1&2 machines (Internet, UCLP) 108
Figure 63: Comparing the total execution time of the Sand-pile model using 108
Figure 64: Relationship between remote messages and simulation times 110
Figure 65: Comparing the file transfer times via the Internet/UCLP 111
Figure 66: Simulator's reaction to a *collect messag* 121
Figure 67: Simulator's reaction to an *internal messag* 122
Figure 68: Coordinator's reaction to an *internal messag* 123
Figure 69: Coordinator's reaction to an *output message* 123
Figure 70: Coordinator's reaction to a *done message* 124
Figure 71: Coordinator's reaction to a *collect message* 124
Figure 72: The *Root* coordinator behaviour when receiving a *done message* 126
Figure 73: The *Master* coordinator's behaviour when receiving 127
Figure 74: The *Master* coordinator's behaviour when receiving 128
Figure 75: The *Master* coordinator's behaviour when receiving a *collect message* 128
Figure 76: The *Master* coordinator's behaviour when receiving a *done message* .. 129
Figure 77: The *Slave* coordinator's behaviour when receiving a *collect message* ... 129
Figure 78: The *Slave* coordinator's behaviour when receiving an *output message* 130
Figure 79: The *Slave* coordinator's behaviour when receiving a *done message* 130
Figure 80: The *Slave* coordinator's behaviour when receiving 131
Figure 81: Web service components 132

Chapter 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 model state, output, and response to external events are defined by a set of functions that define the model behaviour. 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. In addition, complex timing behaviour can be represented by introducing different time delays for different cells in the cell space. Both (DEVS and Cell-DEVS) have been successfully used to model complex systems such as fire spread in a forest [Ame01], land battlefield between two armies [Mad05], and computer networks [Ahm05].

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. In its basic version, CD++ has a one-to-one

correspondence between the model and simulator class hierarchies. 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 on 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. This process continues until the simulation time reaches the final execution time (as provided by the user) or until there is no more events to process. 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 hardware in a real-time environment [Gli02].

The decision of which version of CD++ to consider is governed by two factors; the complexity of the system to be modeled, and the kind of resources available to the modeller. 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 that are consumed by clients in a way similar to the way electricity is consumed in a power grid. The client consumes electricity by plugging his appliance in the power socket without being concerned with the details of the power generator used or the type of cables used to deliver the electrical power. Similarly, 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. Part of the motivation behind grid computing is the enormous resources available today in terms of CPU time and memory space. Organizations have compute resources either in high-end servers or in user workstations with resources not being fully used. By exposing those resources as services that can be used by remote as well as local users, better efficiency in terms of using those resources can be achieved. In addition, the complex Business-to-Business transactions taking place within large organizations usually connect different companies in different locations, traversing different security domains. By connecting the company resources using standard middleware, the interactions among them can have a robust and more secure environment of operation.

Some of the issues usually faced in grid environments include resource description and discovery, resource allocation and management, user authentication and authorization. To facilitate the development and deployment of grid applications, different grid middleware technologies have been developed. The key feature of these technologies is their reliance on standard protocols that can be used on different platforms. Web service technologies represent a means of deploying and exposing applications in standard and platform-independent form. 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.

1.1 Motivation and Goals

The motivation of this work comes from the need to run 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 distributed environments. The examples in which simulation can be applied in order to better understand the system under study are countless. One of these examples include 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.

1.2 Contribution

In this dissertation, we present our work in designing and implementing distributed simulation services based on the CD++ engine. Firstly, CD++ was *wrapped* as a web service allowing the users to submit the model, start the simulation, and retrieve the results remotely. The services were extended to run complex models in distributed environments by taking advantage of web service technologies, namely SOAP [Gud03], as the main messaging protocol. The platform depends on running the simulation as a

service on each node participating in the distributed session, and synchronizing the simulation activities through message passing among the different services. The client connects to the “master” simulation service through SOAP and supplies the model definition and partition information through an XML-based configuration file. Once started, the simulation is processed following the conservative approach for clock advancement, which is controlled by a master *Root* coordinator residing on the master node. Two types of coordinators are used; the *master* coordinator is responsible for forwarding messages among its local children and passing messages from/to the upper-level coordinator in the simulator class hierarchy. The *slave* coordinator is responsible for forwarding messages among its local child models instead of forwarding these messages to the master coordinator that might be running on a different machine. A similar approach was followed when implementing a previous version of the simulator that runs on distributed-memory clusters and it has been shown that using the slave coordinators reduces the overhead of transmitting messages over the network [Tro03]. The main advantage of the proposed simulation engine is that it provides an efficient way of using the CPU and memory resources by running the simulation as a service on commodity hardware (workstations) that can be used by other users to perform other activities, such as word processing. The resources of such machines can be used collectively to execute complex models in a distributed manner. The efficiency comes from the fact that those resources (if not utilized by local users) would have been wasted if not used to run the simulation services.

We provide a prototype for integrating the distributed simulator using SOAP as a messaging protocol and following the conservative approach with an optimistic parallel version of the simulator (PCD++) [Gli04] that uses MPI [MPI95] as a messaging protocol. The optimistic parallel simulator was *wrapped* as a web service in order to enable remote execution of models on distributed-memory clusters. In order to ensure the correctness of the simulation, changes are proposed to PCD++ to ensure that the simulation results are correct in case of rollbacks taking place within PCD++.

We present a performance analysis of the distributed simulator when running different models. Two machines were used to run the tests, one located in Ottawa and the other in Montreal. The performance of the simulator in terms of the time used to initialize and execute the models was studied using two configurations. In one configuration, the machines were connected using a commodity Internet connection, and the results showed a noticeable overhead of the distributed simulation compared to when using one machine to execute the model. In the other configuration, the machines were connected using UCLP, which showed a considerable reduction of the overhead.

1.3 Thesis Organization

This dissertation is organized in different chapters. Chapter 2 introduces the Discrete Event System Specification (DEVS) formalism as a modeling and simulation framework discussing the model definition and the different functions that control its behaviour. In addition, Cell-DEVS is discussed as an extension to the traditional cellular automata. The following section discusses CD++ as the modeling and simulation toolkit used to implement the distributed version of the simulator, followed by a section highlighting the main approaches followed for synchronization in the field of parallel and distributed simulation. The second part of chapter 2 provides an overview of some of the middleware technologies used nowadays to enable grid and distributed applications. In chapter 3, we cover some of the available DEVS simulation engines in grid and distributed environments, highlighting the main characteristics that distinguish our implementation. Chapter 4 introduces the web service components implemented in the simulation engine in order to interface its capabilities to web service technologies. Chapter 5 discusses the implementation of the distributed version of the simulator highlighting its design layout and discussing the functionality of the major components. Chapter 6 provides an experimental performance analysis of the distributed simulator when using UCLP versus regular Internet connections to connect the different nodes in the simulation session. Finally, in chapter 7, conclusions are presented and future research work that can extend the outcome of this dissertation is discussed.

Chapter 2: Grid Middleware for Discrete Event Modeling and Simulation

Discrete event M&S is concerned with studying the behaviour of systems that have finite set of discrete states during continuous periods of time. Examples of these systems include computer networks, traffic in city sections, and manufacturing facilities. Different M&S frameworks have different definitions and interpretations of the functional entities in their environments. One approach of defining the role of each entity is presented by Zeigler [Zei00], in which, the model has two kinds of relationships. The modeling relationship exists between the model and the *source system*, which in turn exists within an *experimental frame*, and the simulation relationship that exists between the model and the simulator.

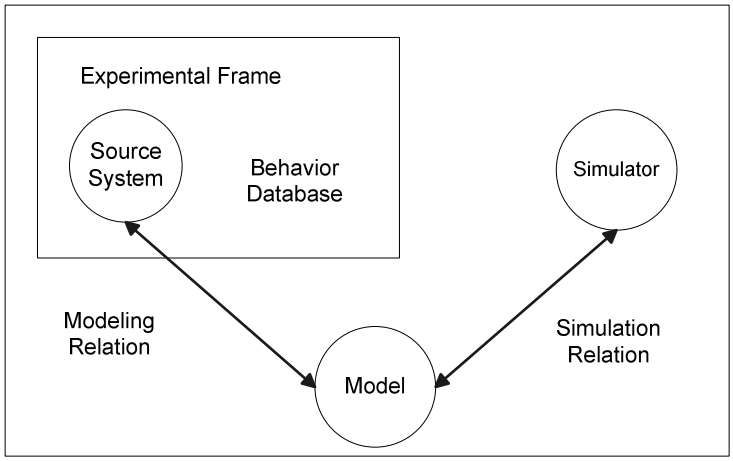

Figure 1: Main entities in a M&S environment [Zei00]

In this chapter, we will introduce the main aspects of DEVS methodology and cover some of the currently available grid middleware tools. The use of grid technologies in the implementation of distributed DEVS simulators is discussed in the following chapter. Grid computing is a computing paradigm where compute and data storage resources are shared among users in distant geographic locations and usually belonging to different security domains. This approach has gained interest in recent years due to the fact that, using the grid, complex applications can run on existing hardware infrastructure without worrying about investing in costly dedicated computer systems such as mainframes and

high-end clusters. In addition, there are large underutilized computing resources. Most desktop machines are busy less than 5%, and in some organizations, even servers can be idle most of the time [Fer03]. Grid computing provides a framework for exploiting these resources and hence has the possibility of substantially increasing the efficiency of resource usage.

The success of achieving the previous advantages largely depends on factors such as the nature of the application to run on the grid, and the kind of grid technology adopted. For example, certain types of applications can be good candidates to run on the grid, such as batch jobs that spend large amounts of time processing input data to produce output data. On the other hand, running a simple application such as word processor on the grid might introduce more overheads that make it slower than if it was run on a regular workstation. The complexity usually pertinent to the grid application may require different types of tools and technologies in order to allow the application to use the grid resources. Due to the heterogeneous nature of the grid, middleware usually depend on standard protocols to connect the grid resources together. Grid middleware can include services and utilities for resource description and discovery, resource allocation and management, and user/service authentication and authorization. In the following sections, we will introduce some of the main ideas in this area.

2.1 Discrete Event System Specification (DEVS)

Discrete Event System Specification (DEVS) [Zei00] is a M&S specification that is aimed to study discrete event systems. In DEVS, the model consists of components connected together through external port(s). Events scheduled for a model arrive through its input ports and the output generated by the model propagates to the other models (or the environment) through its output port(s). The basic building block of any DEVS model is the *atomic DEVS model*. It simulates the behaviour of the system by different functions that are defined as part of the model definition process. The *internal transition* function (δ_{int}), evaluates the next state of the model at internal state transition points. The state defined for the model remains valid for a duration specified by the *time advance* function

(ta). When the model receives external inputs through its input port(s), it examines those input(s) with its current state in order to determine its future state; this is done by executing the *external transition* function (δ_{ext}). The *output* function (λ) is executed before any internal state transition, and it generates the model output to be transmitted to the influencees of the model through its output port(s).

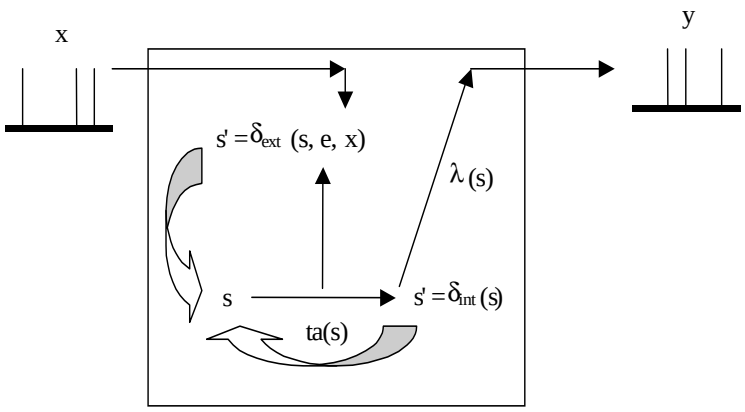

Figure 2: Informal definition of an atomic DEVS model [Zei00]

The formal definition of DEVS models is given as [Zei00]:

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

where

X is the set of input values;

S is the set of states;

Y is the set of output values;

δ_{int}: S → S is the internal transition function;

δ_{ext}: Q x X → S is the external transition function, where

Q = {(s, e) | s ∈ S, 0 ≤ e ≤ ta(s) is the total state set

e is the time elapsed since last transition;

λ: S → Y is the output function;

ta: S → $R_{0 \to \infty}$ is the time advance function;

By examining Figure 2 and the formal definition of atomic DEVS models, one can see the relationship between all the functions defining the model and their effect on its state and behaviour. In Figure 2, 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 (λ)* 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 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]:

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

where

X is the set of input values;

S is the set of states;

Y is the set of output values;

δ_{int}: S → S is the internal transition function;

δ_{ext}: Q x X^b → S is the external transition function, where

X^b is a set of bags over elements in X,

$\delta_{ext}(s, e, \varphi) = (s, e)$;

δ_{conf} : S x $X^b \rightarrow$ S is the confluent transition function;

λ: S $\rightarrow$ Y^b is the output function;

ta: S $\rightarrow$ $R_{0 \rightarrow \infty}$ is the time advance function;

where

Q = {(s, e) | s $\in$ S, 0 $\leq$ e $\leq$ ta(s) is the total state set

e is the time elapsed since last transition;

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.

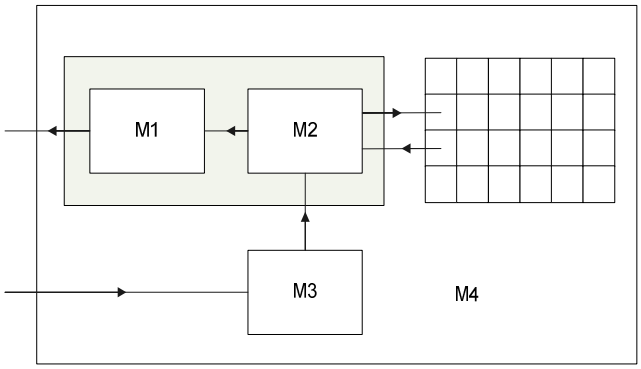

Figure 3: Coupled DEVS model

The formal definition of a coupled DEVS model is [Zei00]:

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

where

X = {(p, v) | p $\in$ IPorts, v $\in$ X_p} is the set of input ports and values, X_p is the set of external values received through port p;

$Y = \{(p, v) \mid p \in OPorts, v \in Y_p\}$ is the set of output ports and values, Y_p is the set of output values generated through port p;

D is the set of the component names;

$M_d = (X_d, Y_d, S, Y, \delta_{int}, \delta_{ext}, ta)$ is a DEVS model

where $X_d = \{(p, v) \mid p \in IPorts_d, v \in X_p\}$, and

$Y_d = \{(p, v) \mid p \in OPorts_d, v \in Y_p\}$;

External Input Coupling (EIC) connects external inputs to component inputs

$EIC \subseteq \{((N, ip_N), (d, ip_d)) \mid ip_N \in IPorts, d \in D, ip_d \in IPorts_d\}$, where

ip_N is an input port of the coupled model, and

ip_d is an input port of component d;

External Output Coupling (EOC) connects component outputs to external outputs

$EOC \subseteq \{((N, op_N), (d, op_d)) \mid op_N \in OPorts, d \in D, op_d \in OPorts_d\}$, where

op_N is an output port of the coupled model, and

op_d is an output port of component d;

Internal Coupling (IC) connects component outputs to component inputs

$IC \subseteq \{((a, op_a), (b, ip_b)) \mid a, b \in D, op_a \in OPorts_a, ip_b \in IPorts_b\}$;

No direct feedback loops are allowed, i.e. no output port of a component can be connected to one of its input ports.

$((d, op_d), (e, ip_e)) \in IC$ implies $d \neq e$;

select is the tie breaking function (not needed for the P-DEVS formalism).

A coupled DEVS model exhibits a similar behaviour to an atomic DEVS model in terms of having input and output ports connecting the model to the environment, or to other DEVS models. The connectivity between the coupled DEVS model and the other external ones is defined though the *External Input Coupling (EIC)* and the *External Output Coupling (EOC)*. The *EIC* defines the connectivity between the input ports of the

coupled model with the input ports of its components. On the other hand, the *EOC* defines the connectivity between the output ports of the coupled DEVS model components and the output ports of the model as a whole. Internal Coupling (IC) defines the connectivity among the model components themselves.

2.2 Timed Cell-DEVS

Cellular automata [Wol86] has been used to model different physical systems, were the model is represented by group of cells neighbouring each other. Each cell has a state and a local compute function. The future state of the cell is determined by its current state and by the inputs it is receiving from its neighbours. When the future state is evaluated, it is transmitted to the neighbouring cells.

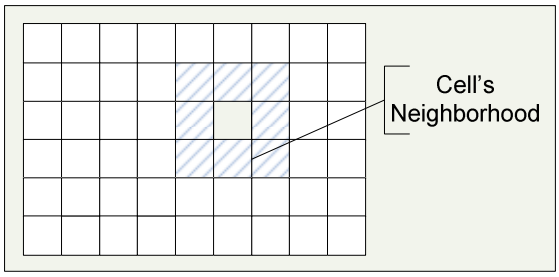

Figure 4: Cellular Automata

Cell-DEVS [Wai01] is an extension to cellular automata that depends on defining the cell as an atomic DEVS model. This adds two improvements to the traditional cellular automata approach:

i) The cells are evaluated asynchronously; i.e. only the active cells are evaluated, as opposed to the synchronous evaluation of cellular automata. This has implementation consequences in terms of requiring less memory than what is needed in the case of synchronous evaluation.
ii) The cells are only activated when they experience state change. This has an advantage of reducing the message exchange with the cell's neighbourhood and hence improving the performance of the model execution.

The formal definition of Cell-DEVS models is presented in [Wai01]:

$$TDC = < X, Y, I, S, \theta, N, d, \delta_{int}, \delta_{ext}, \tau, \lambda, D >$$

X is the set of external input events;

Y is the set of external output events;

I represent the definition of the model's modular interface;

S is the set of sequential states of the cell;

θ is the definition of the cell's state;

N is the set of states for the input events;

d is the transport/inertial delay of the cell;

$\delta_{int}: \theta \rightarrow \theta$ is the internal transition function;

$\delta_{ext}: Q \times X \rightarrow \theta$ is the external transition function, where Q is the state values defined as:

$Q = \{(s, e) / s \in \theta \times N \times d; e \in [0, D(s)]\}$;

$\tau: N \rightarrow S$ is the local computation function;

$\lambda : S \rightarrow Y$ is the output function, and

$D: \theta \times N \times d \rightarrow R_{0+} \cup \infty$, is the state's duration function;

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. Each state is associated with a time value, which gets decremented at each simulation cycle. When the time value associated with the state is equal to zero, it is assigned to the current cell state. Using *transport* delays, a state is considered valid only if it is different from the previously queued state. 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. 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. The formal definition of Coupled Cell-DEVS models is presented in [Wai01]:

$$GCC = < X_{list}, Y_{list}, I, X, Y, n, \{t1,...,tn\}, N, C, B, Z, select >$$

X_{list} is the input coupling list;

Y_{list} is the output coupling list;

I represent the interface of the modular model;

X is the set of the external input events;

Y is the set of the external output events;

n is the dimension of the cell space;

{t1,..., tn} is the number of cells in each dimension;

N is the neighbourhood set;

C is the cell space;

B is the set of border cells;

Z is the translation function; and

select is the tie breaking function;

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 DEVS 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.

2.3 The CD++ Toolkit

CD++ [Wai02] is a collection of programs and tools that are used to execute DEVS and Cell-DEVS models. The main component of the toolkit is the simulation engine (CD++). However; the toolkit includes other utilities that are used for the setup of the simulation and for the interpretation of the results.

CD++ was built following the object-orientation model using C++. CD++ executes the model by creating a collection of *model* and *simulator* classes following [Zei00]. The model classes represent the different types of models that the simulator is capable of executing. Those include *Model*, *Atomic*, *AtomicCell*, *InertialDelayCell*, *TransportDelayCell*, *Coupled*, *CoupledCell*, *FlatCoupledCell* classes. The *Atomic* is an abstract class that encapsulates the variables and methods common to all models; which include the model id, input and output ports, parent id, etc. The *Atomic* class is used to represent an atomic DEVS (or Cell-DEVS) model. In addition to the variables and methods inherited from the *Model* class, it defines the features specific to atomic DEVS models. Each atomic DEVS model has four functions associated with it, which correspond to the functions defined in the formal DEVS formalism:

CD++ *Atomic* method	**DEVS formalism function**
initFunction()	-
externalFunction()	External transition function (δ_{ext})
internalFunction()	Internal transition function (δ_{int})
outputFunction()	Output function (λ)
holdIn(state, time)	Time advance function (ta(state) = time), state = {*active*, *passive*}
passivate()	ta(state) = ∞, state = *passive*

Table 1: *Atomic* class functions

The *AtomicCell* class defines variables specific to Cell-DEVS cells, such as the cell's neighbourhood. *TransportDelayCell* and *InertialDelayCell* represent *transport* delay cells

and *inertial* delay cells, respectively. Coupled DEVS models are implemented in the simulator using the *coupled* class which encapsulates the attributes and methods necessary to define coupled DEVS models, such as establishing the parent-child relationship between the models. The *CoupledCell* represents a coupled Cell-DEVS model that has attributes such as border type, dimension, and default delay. *FlatCoupledCell* defines special kind of Coupled Cell-DEVS models where the whole cell space is executed by one processor in the simulator. Atomic DEVS models are defined through C++ classes that override the different functions defined by the abstract *Atomic* class. These models are integrated into the class hierarchy of the model and are registered by the simulator when the model is loaded and before the simulation starts.

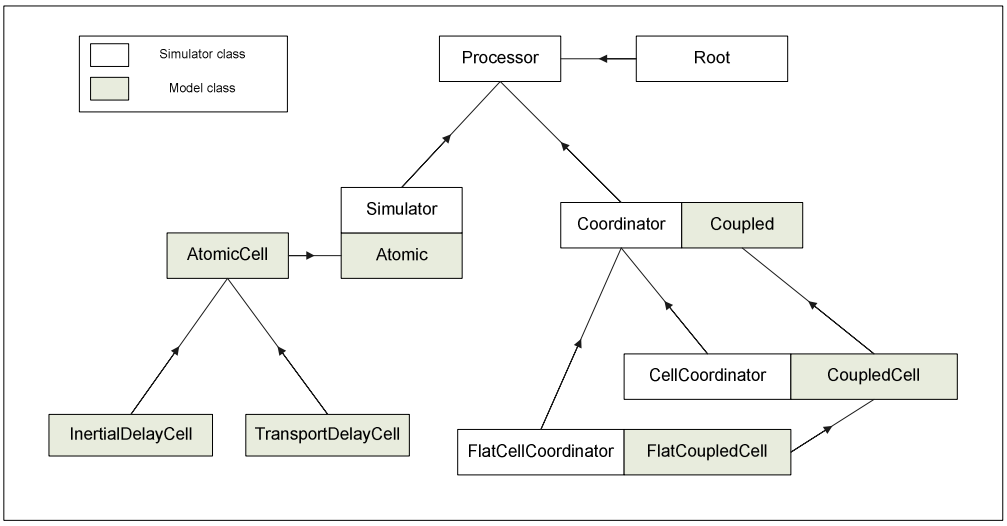

Figure 5: CD++ model and class hierarchies

The simulation is carried out by the simulation classes; those include *Processor*, *Root*, *Simulator*, *Coordinator*, *CellCoordinator*, *and FlatCellCoordinator*. The simulation is driven by the *Root* coordinator, which is responsible for starting and stopping the simulation, interfacing the simulator with the environment in terms of passing external events/output from/to the environment, and advancing the simulation clock. The *Simulator* class executes an atomic DEVS model by receiving different kinds of messages and responding by executing the corresponding function in the atomic DEVS class (*Atomic*). In addition, it maintains two important variables that are used to find the imminent models and advance the simulation clock. Those are $T_{lastChange}$, and $T_{nextChange}$;

$T_{lastChange}$ is the time of the last change of the DEVS model, and $T_{nextChange}$ is the time of the next change.

The *coordinator* class is responsible for routing the messages among its children and its *parent-coordinator*. In addition, it evaluates the minimum $T_{nextChange}$ for its children in order to report it to the *Root* coordinator. The *CellCoordinator* is derived from the *coordinator* class and is responsible for message routing among the cells in a coupled Cell-DEVS model. The *FlatCellCoordinator* class executes *flat Cell-DEVS* models.

Having separate classes for the model and simulator offers the advantage of isolating the simulator architecture from the model structure; so that changing the simulator internals does not affect the model definition. In addition, it facilitates the use of the simulator since the modeller needs only to define the model without any deep knowledge of the simulator.

The simulation is driven by passing messages among the different simulators and coordinators. The simulation continues until the simulation clock reaches a specific time or when there are no more events to process. The messages exchanged between the simulator entities are grouped into two categories: *synchronization messages*, and *content messages*:

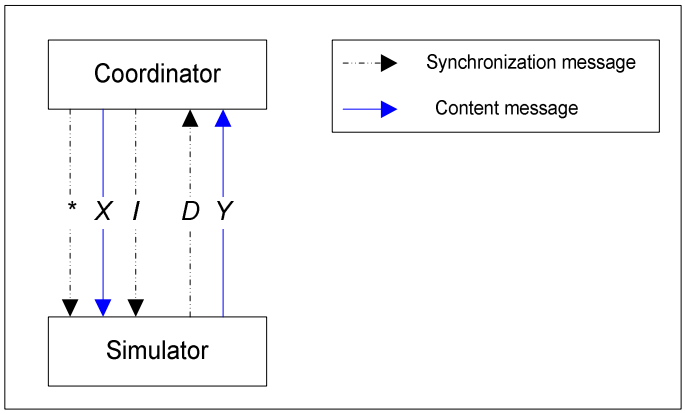

Figure 6: Content and synchronization messages in CD++

Message	Description
I	*Initialization message:* is passed by the *Root* coordinator to all of the coordinators/simulators at the initialization phase of the simulation.
*****	*Internal message*: is passed by the *Root* coordinator to the imminent models (scheduled for state change).
Done	*Done message*: is passed by the simulators to the upper-level coordinators to designate the end of state transition phase (or the processing of an external event) and report their $T_{nextChange}$.
X	*External message*: represents an external event that can be arriving from the environment or from an output message generated by another model.
Y	*Output message*: represents an output generated by the model.

Table 2: DEVS simulator messages

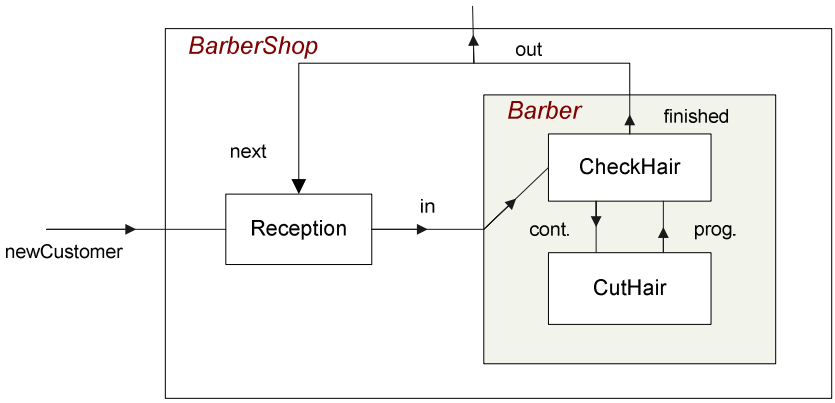

Figure 7: Barbershop model architecture

An example of a coupled DEVS model is shown in Figure 7. The *BarberShop* model represents a barbershop with three main components. The *Reception* component is an atomic DEVS model simulating the reception desk of the barbershop. The *Reception* has limited seats available for arriving customers, who are either advised to wait in the reception if the barber is already busy working on a customer, or are forwarded to the barber if he is idle. The *Barber* component is a coupled DEVS model that consists of the *CheckHair* and *CutHair* atomic DEVS models. The *CheckHair* component represents the

process of checking the customer's hair and getting the customer's preference of his hair style. The *CutHair* component represents the actual hair cut process.

The semantics of the model is defined in CD++ using the CD++ specification language. It has different constructs to define:

i) The components of the model, in a hierarchal manner;

ii) The input and output ports connecting the components, and the interconnections among those ports;

iii) The specific attributes of each model, such as the border type in the case of coupled Cell-DEVS models;

iv) Parameter values used by the model such as type and mean of stochastic distributions;

```
[top]
components : reception@Reception Barber
in : newcust next
out : cust finished

Link : newcust newcust@reception
Link : out@Barber next@reception
Link : out@Barber finished
Link : cust@reception in@Barber

[reception]
numberofChairs : 8
preparationTime : 00:00:01:000
openingTime : 09:00:00:000
closingTime : 16:00:00:000
```

```
[Barber]
components : checkhair@Checkhair
cuthair@Cuthair

in : in
out : out

Link : in cust@checkhair
Link : finished@checkhair out
Link : cutcontinue@checkhair cutcontinue@cuthair
Link : progress@cuthair progress@checkhair

[checkhair]
preparationTime : 00:00:09:000

[cuthair]
preparationTime : 00:00:11:000
```

Figure 8: *BarberShop* model definition

The *top* construct defines the overall *BarberShop* model which is composed of an instance of the *Reception* model (atomic DEVS), and the *Barber* coupled DEVS model. The input and output ports of the *BarberShop* are defined using the *in* and *out* constructs respectively. The *link* construct defines the connections between the input and output ports of the model. The different parameters used by the *Reception* model are defined within the *reception* construct. They include the number of chairs available in the reception area (*numberofChairs*), the preparation time for the customer to move from the reception to the barber chair (*preparationTime*), and the opening and closing times of the barbershop (*openingTime*, *closingTime*). The *Barber* coupled DEVS model is composed

of an instance of the *CheckHair* model, and an instance of the *CutHair* model. The input and output ports, and the links among them are defined in a similar manner to the *top* model.

In CD++, each of the atomic DEVS models needs to be defined as a C++ class overriding the main functions defined by the abstract *Atomic* class. These functions are *initFunction*, *internalFunction*, *externalFunction*, and *outputFunction*. By integrating the DEVS model class into CD++, the simulation is driven by executing these functions by the *Simulator* associated with the model. An excerpt of the definition of the *Reception* class is shown in Figure 9:

```
Model &Reception::initFunction()
{
  we_are_full = false ;
  cust_is_ready = false ;
        elements.erase( elements.begin(), elements.end() ) ;
        return *this ;
}
Model &Reception::internalFunction( const InternalMessage & )
{
        passivate();
        return *this ;
}
Model &Reception::outputFunction( const InternalMessage &msg )
{
        if (elements.size()) {
            sendOutput( msg.time(), cust, elements.front() ) ;
            elements.pop_front();
        }
        return *this ;
}
```

Figure 9: An excerpt of the *Reception* class definition

During the initialization phase of the *Reception* model (at the beginning of the simulation), the variable *we_are_full* (indicates that the reception is occupied by the maximum allowed number of customers) is reset to false. In addition, the *cust_is_ready* variable and the list of customers are reset. When the model is scheduled for an internal transition, the *internalFunction* is executed and it causes the model to be *passive* until further external input events are received. The output of the *Reception* model is generated through the *outFunction* method; in this case, the output of the model represents the customer that was waiting for the longest time.

```
Model &Reception::externalFunction( const ExternalMessage &msg )
{
        if( msg.port() == newcust ){
                if    (   (msg.time().asMsecs() - Time(openingTime).asMsecs() < 0)
                 || (msg.time().asMsecs() - Time(closingTime).asMsecs() > 0) ) {
                                cout << "rcptn  x we are closed - sorry " << "\n" ;
                } else {
                if (!(elements.size() < numberofChairs)) {
                                cout "we are full - sorry " <<  "\n" ;
                        }else{
                        elements.push_back( abs((int)msg.value()) ) ;
                        if (elements.size() == 1) {
                        holdIn( active, preparationTime ) ;
                }
        }

        if (msg.port() == next) {
        holdIn( active, preparationTime ) ;
        }
        return *this;
}
```

Figure 10: An excerpt of the *Reception* class definition

The *Reception* model has two input ports; *newCustomer* and *next*. If a message arrives through *newCustomer* port representing the arrival of a new customer, different scenarios can occur. If the message arrives with a timestamp outside the barbershop hours of operation, no action is taken and a message indicating that scenario is printed out to the modeller. If the customer arrives and there are no other customers waiting in the reception, the customer is forwarded to the *Barber* model at the time of next internal transition (which takes place once the *preparationTime* elapses). The third scenario occurs when a customer arrives while there are others waiting in the reception; in which case, the customer is added to the list of waiting customers and the one who was waiting for the longest period is forwarded to the *Barber* after *preparationTime* time units.

In order to execute Parallel-DEVS models, an abstract simulator was presented in [Cho94b]. The basic P-DEVS simulator depends on having separate representations of the model and the simulator entities. In addition, the two main components of the simulator are the *simulators* and *coordinators*. The simulators are responsible for executing atomic DEVS models, and the coordinators are responsible for executing coupled DEVS models. They both interact through messages that can be *synchronization (collect, *, done)* or *content messages (x, y)*.

When a simulator receives a *collect message* from its parent, it executes the *output* function and sends a *done message* to its parent coordinator indicating the time of the next state change. The state change of the simulator takes place when it receives an *internal message (*)* from the coordinator, in which case it executes its *internal transition* function, *external transition* function, or *confluent transition* function. The choice of which function to execute depends on different factors, which are:

i) The timestamp of the *internal message*;

ii) The time of the internal transition of the model;

iii) The status of the external message bag of the model;

The coordinator is in charge of forwarding *external* and *output messages* among the simulators and synchronizing the activities taking place during the simulation. When a coordinator receives a *collect message*, it forwards the message to its imminent child processors and reports the time of the next change to its parent coordinator. Receiving an *internal message* by a coordinator, causes it to process the messages in its external message bag, and send internal messages to its child processors scheduled for internal and/or external transitions. An *output message* generated by a simulator is sent to its parent coordinator, which in turn either forwards it to the upper-level coordinator, or translates it to *external messages* for its local receiving processors. *External messages* received by simulators and coordinators are inserted in their external message bags to be processed when they receive the next *internal message* from the parent coordinator. The algorithms defining the behaviour of the simulators and coordinators are explained in detail in Appendix-A.

In order to define Cell-DEVS models, the modeller does not need to define any C++ class; that is, CD++ already includes the *AtomicCell* classes representing the cell with *transport (TransportDelayCell)* and *inertial (InertialDelayCell)* delays. However, the modeller uses the CD++ specification language in order to define the necessary attributes of atomic and coupled Cell-DEVS models. Those include the border type, the delay type, the default delay value, the neighbourhood, etc. An example of a Cell-DEVS model is presented in [Mad05], where a battlefield between two armies is modeled. The army

consists of number of fighters, each of which has a state that can be *alive*, *injured*, or *dead*. Fighters engage in a battle and the outcome of the battle depends on a randomly assigned factor *FightingAbility*, which is assigned to the soldier at the beginning of the simulation and at the end of any engagement with his enemy.

The *Battlefield* model is composed of a 3-dimensional cell space with (10, 10, 6) cells in each dimension. The cell delay is defined using the *delay* construct to be *inertial* delay with a default value of 100 milliseconds. The border type used in the *Battlefield* model is *wrapped* indicating that the cells at the edges of the cell space are neighboured by those on the other side. The neighbourhood of the cell is defined by the *neighbors* construct. Cells are assigned a default value of zero (*initialvalue : 0*) unless they are assigned different values by the file "battle1.val" (*initialcellvalue : battle1.val*). The *zone* construct is used to assign different rules for different parts of the cell space. The layer of the cell space responsible for evaluating the soldiers' behaviour in the battlefield is the first layer ((0, 0, 0)..(9, 9, 0)); the other layers are used to store and evaluate the different variables that affect the simulation of the battlefield.

```
[top]
components : battle

[battle]
type : cell
dim : (10,10,6)
delay : inertial
defaultDelayTime  : 100
border : wrapped

neighbors : battle(0,0,-1)
neighbors : battle(0,0,-2)
neighbors : battle(0,0,-3)
neighbors : battle(0,0,-4)
neighbors : battle(0,0,-5)
.....
initialvalue : 0
initialcellsvalue : battle1.val
zone : fight-rule { (0,0,0)..(9,9,0) }
......
```

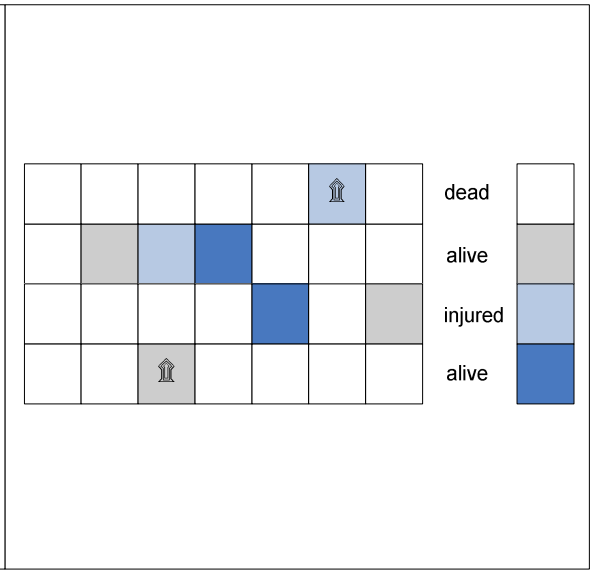

Figure 11: An excerpt of the *Battlefield* model definition

The local rule definition specifies the value each cell would take at each simulation cycle. Each rule will have a condition, delay, and a value. The condition is evaluated, and if it is true, the cell is assigned the specified value when the delay elapses. Figure 12 shows part of the rule definition of the *Battlefield* model. The first rule checks if a soldier of army A is in state *injured ((0, 0, 0) = 1)* or *alive ((0, 0, 0) = 2)* and surrounded by enemy soldiers; if so, it evaluates the fighting ability of the enemy soldiers (using the macro "fight_rule_1") and if the outcome is larger than the fighting ability of the solider, the soldier is considered *dead ((0, 0, 0) = 0)* after 100 time units. The second rule evaluates the same situation for the soldiers of army B. The third and fourth rules evaluate the status of the *flags ((0, 0, 0) = 5, (0, 0, 0) = -5)* when they get attacked by enemy soldiers.

```
rule : {  if( (0,0,0) >= abs(#macro( fight_rule_1 ) ) , (0,0,0) + #macro(fight_rule_1) , 0 ) }              100
                { cellPos(2) = 0 and ((0,0,0) = 1 or (0,0,0) =2 ) and (stateCount(-1) + stateCount(-2)) > 0}

rule : {  if( abs((0,0,0) ) >= #macro( fight_rule_2 ), (0,0,0) + #macro(fight_rule_2) , 0 ) }              100
                { cellPos(2) = 0 and ((0,0,0) = -1 or (0,0,0) =-2) and (stateCount(1) + stateCount(2)) > 0}

rule : {                                    if(  #macro(fight_rule_3) != 0 , 0 , 5 )  }                    100
                { cellPos(2)= 0 and (0,0,0) = 5 and (stateCount(-1) + stateCount(-2)) > 0 }

rule : {                                    if( #macro(fight_rule_4) !=0 , 0 , -5 ) }                      100
                { cellPos(2) =0 and (0,0,0) = -5 and (stateCount(1) + stateCount(2)) > 0 }
```

Figure 12: An excerpt of the *Battlefield* rule definition

2.4 Distributed Simulation

The complexity of the model tends to increase as the modeled system evolves or as more details need to be taken into account at a lower level of abstraction. This in turn requires more compute and memory resources when executing the model which results in a longer execution time, or in not being able to run the simulation at all due to lack of resources. The field of parallel and distributed simulation aims to study the possibilities of providing more efficient runs of complex models. This can be achieved by executing the simulation on parallel hardware that can be shared-memory multiprocessing machines, or distributed-memory clusters. In shared-memory machines, multiple processors have

access to a shared memory which might be a bottleneck if the number of processors is large. In distributed-memory clusters, different processors have different memories and sharing information takes place through message passing; in which case, the network might be the bottleneck.

To run on distributed environments, the model is usually decomposed into components that are executed by different simulators running on multiple processors. This has an advantage of utilizing the parallelism in the model, but it requires synchronization among the different processors. The synchronization among the different processors running the simulation has gained a lot of attention from the research community. The main issue is how to use the parallelism in the hardware to execute the model while maintaining the correctness of the simulation. In discrete event simulation, there are usually dependencies among the model components, such that, some events can only be processed once the events they depend on are done. This is referred to as *causal dependency* of the model components [Zei00]. For an arbitrary event x, all of the events on which it depends (either directly or indirectly) have to be processed before x gets processed, satisfying the *local causality constraint*. Failure to do so; might result in causality errors. The problem of assuring compliance with the local causality constraint is referred to as the *synchronization problem* [Fuj99].

In parallel and distributed environments, simulation is considered to be carried out by logical processors (LPs) that are mapped to physical processors. The events processed by each LP might have been received from other LPs through time-stamped message exchange or were scheduled by other local events. The correctness of the simulation is regarded as not to violate the local causality constraint. *"A discrete-event simulation, consisting of logical processes (LPs) that interact exclusively by exchanging time-stamped messages obey the local causality constraint if and only if each LP processes events in non-decreasing time stamp order"* [Fuj99]. Figure 13 shows a scenario where three logical processes are executing a model and LP1 receives an out-of-order event resulting in a causality error.

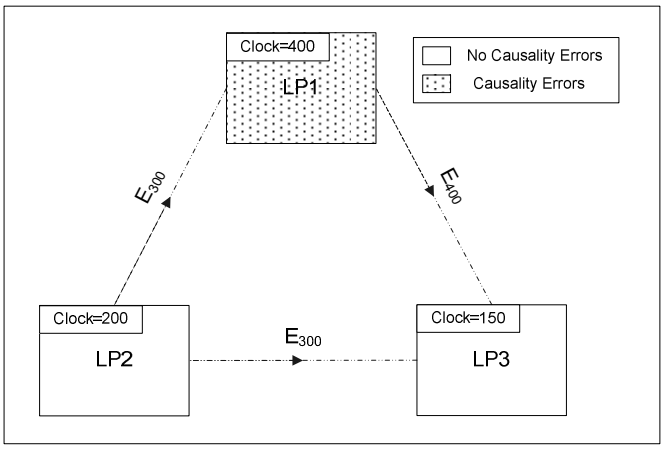

Figure 13: Causality errors in distributed simulation

Two main categories of algorithms exist to address the issue of synchronization in parallel and distributed simulation environments. The conservative approach restricts the simulation clock advancement in each logical process to the condition that no causality errors will be encountered in the future simulation time. On the other hand, optimistic approach permits causality errors to occur, but provides the means to rollback the simulation to the time of the message that caused the causality error and resumes the simulation from that point.

2.4.1 Conservative Simulation

In conservative simulation, the logical process advances its simulation clock only when it is "safe" to do so. The safety is judged by the possibility of receiving a message with an earlier timestamp than the clock of the logical process. One approach for the logical process to do so is by having a queue for each link through which it receives external messages from other LPs. Then, by checking each of the queues for the earliest time stamp of the message to be processed, and comparing those with the time stamp of the event to be processed next, the logical process can determine the time of the event that won't cause any causality error if processed.

The problem with this approach is that the logical process will not be able to advance its clock if there is a link without any input in its queue, since the logical process can't

calculate the minimum time stamp of the events received through this link. This in turn may result in a deadlock in the whole simulation if there is a cyclic dependency among the logical processes. The Chandy-Misra-Bryant (CMB) [Bry77][Cha79] algorithm introduces the concept of *null* messages, which don't schedule any events, but are used to inform the logical process of the lowest time bound of any subsequent messages to be sent by the sending logical process. The time in the future before which no events will be scheduled is referred to as *lookahead*, and it depends on the system being modeled. It has been shown, that when running parallel/distributed DEVS models following the conservative approach and using *null messages* with a non-zero *lookahead* for at least one logical process, deadlock can never occur. That is, at least one logical process will be able to advance its clock and process its events [Zei00].

One of the disadvantages of the conservative simulation is that the *lookahead* property is application-dependent and may not always be easy to calculate. In addition, the parallelism in the model and hardware may not be exploited efficiently due to the conservative nature of the algorithm.

2.4.2 Optimistic Simulation

Contrary to the conservative simulation, optimistic simulation permits causality errors to occur, but provides mechanisms to rollback the simulation to an earlier time so that the local causality constraint can be satisfied. One of the most known optimistic algorithms is Time Warp, which was introduced by Jefferson [Jef85]. Time Warp introduces the concept of Global Virtual Time (GVT) and Local Virtual Time (LVT). The global virtual time is common to all logical processes and it always advances in an increasing order, the local virtual time is local to the logical process and it can be advanced in an increasing order or a decreasing order (in case of rollback). The Global Virtual Time (GVT) at a specific wall-clock time is the lowest bound on the timestamps of all the events in all the logical processes, and the messages that were sent but not received yet (in transit). Thus, no rollback can ever take place at a time equal to or less than GVT. This is an important property since it allows the Time Warp simulation to reclaim the resources used by all the

events with timestamps earlier than GVT; this process is refereed to as *fossil collection* [Jef85]. Different algorithms exist for evaluating the GVT including Samadi's GVT algorithm and Mattern's GVT algorithm [Fuj99].

Rolling back the simulation objects is done by restoring the states of the objects at the time of rollback; however, the messages sent by the simulation objects after the rollback time need to be "unsent" as well. Time Warp introduces the concept of *anti messages (negative messages)* which annihilate with the corresponding positive messages. So, in order to cancel the events that were sent by a simulation object, negative versions of the messages that were sent after the rollback time should be sent out.

The main advantage of the Time Warp algorithm is that it is able to use the parallelism in the modeled system "*optimistically*" by advancing the local simulation clock in each logical process without waiting for any safety condition to be satisfied. On the other hand, its main disadvantage is that it requires more resources to store the state and anti-message information that are needed in case of rollback. In addition, there is an overhead associated with rolling back the simulation to an earlier simulation time, however, Jefferson presents an argument that most programs follows the *temporal locality principle*, "*most messages arrive in the virtual future at their destination, not causing any rollback at all, and that those that arrive in the virtual past tend strongly to arrive in the recent past, so that few events are rolled back"* [Jef85].

2.5 Web Services (WS)

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

web-related standards [Alo03]. Web services are different from the traditional web applications in an important aspect. Web applications are hosted by application/web servers and they use the HTTP protocol to interact with the clients. Since the Internet is the largest network of resources using HTTP, they are usually embedded in the context of a service provider's webpage; the important thing about web applications is that they are used by humans in the sense that the user has to find the web application of interest and perform some tasks (such as launching an applet) to use the functionality offered by the application provider. On the other hand, web services are meant to be used by other services (and not directly humans). Although web services are usually deployed using HTTP as an 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 a lot of administration or management overhead.

The fact that web services are meant to be used by applications emphasized the need to express the functionality of the web service in machine-understandable languages. XML [Bra04] seemed to be an ideal candidate in which to develop the standard. One advantage of using XML is that it is a widely accepted language for the flexibility it offers in terms of defining the document structure. Therefore, several XML-based languages and standards have emerged to meet the needs of the web service applications:

- WSDL (Web Service Description Language) [Chr01]: is an XML-based language used to define and describe the public interface of the service. It contains enough information for the client to develop/use an application to consume the web service.
- WSDD (Web Service Deployment Descriptor): is an XML-based language used to define different deployment parameters necessary to deploy the web service. Although WSDD has not been standardized, it is widely used by different web service engines to define parameters like: the protocol used to transfer SOAP messages, the web service method signature (parameters and return types), and the methods that the user is allowed to invoke.

- UDDI (Universal Description Discovery and Integration) [Cle04]: is an XML-based language used to register and query web services (using UDDI registries).
- XML-Schema [Fal04]: is an XML-based language used to define complex data structures within XML documents.
- X-Path [Cla99]: is an XML-based language used to find different elements within XML documents.
- SOAP [Gud03]: is a messaging protocol designed to carry information between different web services. A SOAP message consists of an *envelope* which has an optional *header* and a mandatory *body*.

Among the different standards, two are of particular interest to this work: WSDL, which represents the public interface of the web service; and SOAP, since it plays an important role in message passing among web services and their clients.

WSDL documents include enough information for the web service clients in order to know the operations it offers, what kind of parameters are required to invoke an operation, and the return type of the operation. The major elements of any WSDL document are *type*, *message*, *portType*, *binding*, *port*, and *service* elements. Some of those elements (*type*, *message*, and *portType*) are used to describe the functional behaviour of the web service in terms of the functionality it offers. On the other hand, the *binding*, *port*, and *service* (in addition to the *type*, *message* and *portType*) elements 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 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. 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. An example of a SOAP message is shown in Figure 14.

```
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 14: An example of a SOAP message embedded in HTTP [Gud03]

As per the standard specification of SOAP, the receiver of the SOAP message should perform the following tasks [Gud03]:

i) Examine the SOAP message and identify the parts that are intended for that application. The SOAP message can pass through different services, and each one might have some processing to do before forwarding the message to another service. So, it is important that the service implementation locates the parts that it has to process.

ii) Check the parts identified in step *i* to see if they are supported by the application and process them accordingly. If those parts are not supported, the SOAP message is discarded. The application may choose to ignore the optional parts of the message without violating the SOAP standard.

iii) In the case of a SOAP message not destined for the application, it should remove the parts identified in step *i* and forward it to its destination.

In a typical web service solution, different tools and standards play different roles to fulfill the application requirements. On the top layer, UDDI can be used to register the web service allowing other services and clients to discover its existence. At a lower layer, WSDL is used to describe the functionality of the service so that the client can construct proper SOAP requests knowing the kind of responses he should expect from the service. SOAP and its extensions are used as the main messaging protocol between the web service and its clients. SOAP is transported via an application layer protocol such as *Simple Mail Transport Protocol (SMTP)*, and *Hypertext Transfer Protocol (HTTP)*.

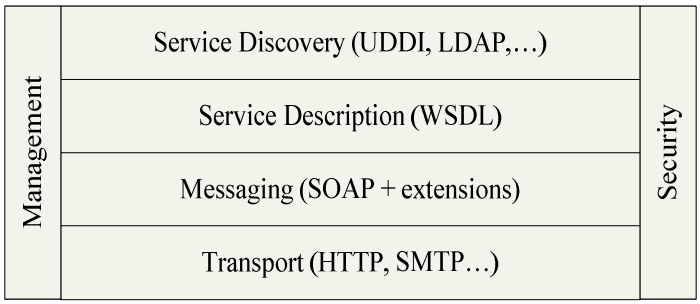

Figure 15: Web service layers

From the client perspective, the web services is seen to be no more than a SOAP message processing entity; it receives SOAP requests and generates SOAP responses after some processing time. However, It is useful to distinguish between two main components of any web service implementation; the hosting environment which provides a working space for hosting the web service, and the actual web service implementation. The hosting environment usually includes a SOAP engine, an application server, and a web server. Figure 16 shows the major components of a web service hosting environment:

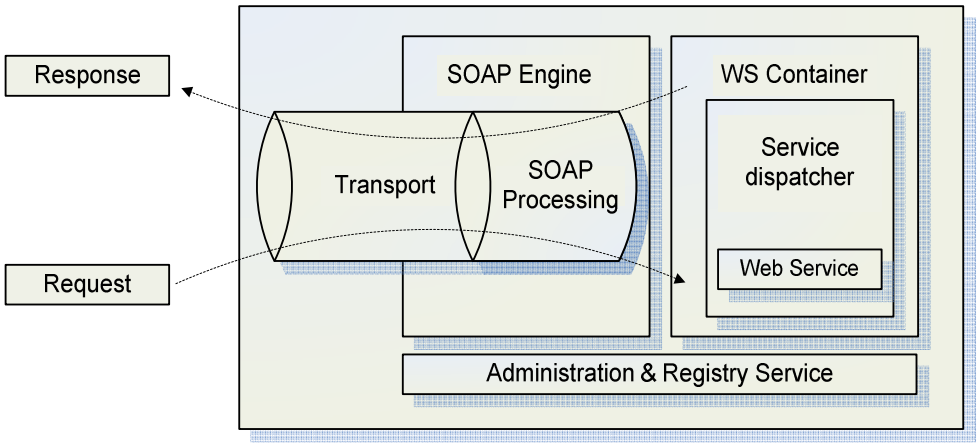

Figure 16: A web service container [Glo05]

The request is received by the service as an HTTP request containing the SOAP message. The web server is responsible for handling the HTTP traffic as in the case of any website hosting environment. Once extracted from the HTTP request, the SOAP message is forwarded to the SOAP engine, which is responsible for processing the SOAP messages and converting the SOAP 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 the entity responsible for implementing the logic of the web service. Once the processing is done by the implementation code, the result is handed to the SOAP engine to build the SOAP 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 by itself is an application that runs within an application server that is installed as part of the web service deployment process.

2.6 Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) [Erl05] refers to a new paradigm in the area of distributed application development and deployment. It depends on using standard technologies in order to split the application logic into number of components, each of which exposes its functionality in a platform-independent manner. Then, the logic of the overall application is realized by establishing some sort of workflow among the different components. Web services have been used in the implementation of SOA systems due to its wide acceptance among programmers and business leaders. The orchestration between the web services is usually implemented using standard mechanisms such as Business Process Execution Language (BPEL) [And03]. There is subtle difference between the traditional distributed systems and SOA systems in that the latter depend on standard technologies and each of the system components (services) usually implements part of the logic that communicates with the other components in a loosely coupled manner. On the other hand, traditional distributed systems typically (although not necessarily) are characterized by objects maintaining a fairly complex internal structures required to support their methods, and fine grained interaction between an object and a program using it. A Service Oriented Architecture (SOA) is typically characterized by the following properties [Alo03]:

- The service is abstracted by its logical view, which might represent actual programs, business processes, and databases, and defines what it does rather than how it does it.

- The service is defined by the type of messages it receives as an input and the messages it generates as an output (message orientation). The implementation details of the service such as programming language, process structure, or even the database structure are hidden from the web service consumer. This has an advantage of allowing the interoperability between different legacy systems that were developed using different technologies. Those systems can be "wrapped" by web service wrappers that operate together using SOAP without revealing their internal complexities.
- Services in SOA tend to use small number of operations with relatively large and complex messages.
- The services tend to be used in networked environments (network orientation).
- The services are platform-independent. They receive and send XML-based SOAP messages that can be interpreted and processed in a platform-neutral manner.

"It is argued that these features can allow service-oriented architectures to cope more effectively with issues that arise in distributed systems, such as problems introduced by latency and unreliability of the underlying transport, the lack of shared memory between the caller and object, problems introduced by partial failure scenarios, the challenges of concurrent access to remote resources, and the fragility of distributed systems if incompatible updates are introduced to any participant" [Alo03]. Web service technologies in general can be used to implement service-oriented architectures and distributed-object systems. The design approach to be followed depends on different factors such as the platforms used to host the application, the nature of the application, and expected future evolution.

2.7 User Controlled Light Path (UCLP)

User Controlled Light Path (UCLP) [Arn03] is a project initiated by CANARIE, a non for profit organization that promotes collaboration through high-speed networks, to develop management software to be used in high-bandwidth fibre networks to enable users to allocate and manage the bandwidth they require to achieve their business goals. The objective of UCLP is to enable the user to manage the bandwidth without the

intervention of network and system engineers, which saves time and money usually associated with managing large fibre networks. UCLP depends on encapsulating the network resources and components (such as switches) using web service-enabled wrappers that the user can interact with instead of using management protocols such as *Simple Network Management Protocol (SNMP), Transaction Language 1 (TL1)*, etc. This introduces the possibility of integrating the network resources within the user application (provided certain security constraints are adhered to), and being able to create and lease a light path for a specific period of time after which the light path is destroyed and its bandwidth made available to other users. As a distributed application based on web services, UCLP makes heavy use of the different web service technologies. Specifically, it uses BPEL at the orchestration layer to manage the different network resources in order to form Articulated Private Networks (APNs). An APN is a logical group of light paths that are managed as a single entity.

Chapter 3: Trends in the Implementation of Distributed DEVS Simulators

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. Chow, Zeigler, and Kim [Cho94b] have defined the semantics of an abstract simulator for the parallel DEVS formalism. The advantage of the parallel abstract simulator is that it takes advantage of the parallelism introduced in the P-DEVS formalism [Cho94a] in terms of activating all the imminent components of the model at the same time dispensing with the need for the *select* function in the original DEVS formalism. Different groups of researchers have studied the implementation of DEVS simulators in parallel and distributed environments; each followed a distinct approach in terms of the middleware tools adopted to implement the simulator and the functionality it offers. Some of the implementations have emphasized the dynamic aspect of M&S in a grid environment. That is, they provide a platform for registering and activating the simulation entities in a dynamic manner based on some partitioning scheme. They make heavy use of the tools provided by grid middleware for resource allocation and management, user authentication and authorization, and communication among the simulation nodes. Other implementations of DEVS simulators put more emphasis on the performance of the engine. They try to take advantage of the parallelism available in distributed environments in order to achieve higher speedups. In this regard, the implementation of optimistic simulation algorithms was considered by some in order to allow the nodes to advance their clocks independently as opposed to the conservative approach for synchronization. In this chapter we provide an overview of some of the major implementations of distributed DEVS simulators, highlighting their design approach and the functionality they offer. Then, we introduce some of the differences between those implementations and the design we propose in this dissertation in terms of the design methodology we followed, the middleware used for the implementation, and the advantages it offers when operating in a distributed environment.

- **DEVS/Grid**

DEVS/Grid [Seo04] implements a grid-enabled DEVS simulator following a layered approach. The system consists of five layers; *application*, *modeling*, *simulation*, *middleware* and *network* layers. The application layer is the top layer and it deals with high level issues within the application domain. The modeling layer provides the required functionality for defining the model; the simulation layer is responsible for running the actual DEVS simulation with the support of other tools and utilities. The middleware layer represents the grid-middleware layer (implemented using Globus [Glo05]) 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. The main components in the system are the model *partitioner*, which is responsible for dividing the model into a set of partition blocks. Each partition block contains one or more components of the model. The partitioning is done following a cost-based criterion and the resulting partitions are transferred by the model *deployer* to the host machines for execution. In the host machine, the *activator* receives the model partitioning information and creates a *simulator* to execute the model. In addition, DEVS/Grid provides the following functionality:

  - *Grid Index Information Service (GIIS)*: it is a M&S directory service used to resolve the names of the different simulation entities and publish/subscribe the resources available to the modeller.
  - *Static/Dynamic model deployment*: the available hosts are identified using the services offered by *GIIS* which allow for dynamic assignment of the model partitions. Once the host is identified, the *deployer* sends the model partitioning information to the host machine for execution.
  - *Remote activation*: the model is activated remotely through the *activator*, which resides on the hosting machine. It receives the partitioning information through the *deployer* and creates a simulator for each component of the model. The information about the created simulators and the models they execute is published

in *GIIS*. This information includes the addresses of the simulators, and the input/output ports that are used to examine the model coupling scheme and establish communication channels among the different simulators.

- *Communication channels*: they are formed dynamically by examining the coupling scheme and simulator addresses published through *GIIS*. There are two types of communication channels:

  - *User Channels*: they are used to route the messages among the different simulators representing the events scheduled during the execution of the model.
  - *System Channels*: they are used to send synchronization information required for advancing the simulation time and implementing barriers during the simulation.

- **vGrid**

vGrid [Kha03] is an overall architecture for running DEVS and Cell-DEVS models in grid environments. vGrid divides the model into components; the *Fine Computational Unit (FCU)* is the most basic component that corresponds to an atomic DEVS or Cell-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, such as workstation. Different engines play different roles in the vGrid architecture; the *vGrid Manager (VGM)* is responsible for managing all the resources in a grid environment with coordination with the other engines. It interacts with the *VCUs* through *Autonomous Wrappers (AW)*, which maintain operational, functional and control information about the *VCUs*. The *Monitoring Engine (ME)* is responsible for monitoring the resources in the grid and maintaining this information to be accessible by the *VGM*. *Analysis Engine (AE)* generates the *Work Capability Index (WCI)* (which is a measure of the complexity of the task) from the *FCU* and generates the *Resource Capability Index (RCI)* (which is a measure of the capability available to a single resource in the grid); both are used by the *Planning Engine (PE)* to partition the cell space into *VCUs*. The

architecture distinguishes between inactive and active *FCUs* by including the latter into *Working Sets (WS)*, which get checked by the *Execution Engine (EE)* to pick a *VCU* for execution. The vGrid architecture provides the following functionality:

- *Dynamic model partitioning*: it is achieved by moving the *FCUs* among the *VCUs* so that a specific load threshold is adhered to.
- *Distributed communication Service*: provides a flexible communication, event notification, and access control for the different entities in the simulation.

- **DEVS/P2P**

DEVS/P2P [Che04] is a distributed DEVS simulator aimed to peer-to-peer networks. Its architecture is similar to DEVS/Grid except that it uses JXTA [JXT06] as an implementation of P2P communication middleware instead of using Globus as a grid middleware. 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 according to a cost-based partitioning algorithm. The partitions are deployed in the host machines through *AMD*. The A*ctivator* is responsible for receiving a model partition and creating *GS* that runs the simulation. The message communication among the different nodes is handled by the JXTA system. DEVS messages from one simulator to another remote one are converted to XML-based messages that get sent by JXTA to the receiving machine. At the receiving end, the messages are converted back into DEVS messages to be processed by the receiving simulator(s). JXTA uses virtual communication channels (pipes) among the machines, which get mapped to the DEVS model ports to assure correct routing of messages during the simulation.

The simulator doesn't depend on a master coordinator to handle the synchronization and clock advancement. Instead, each simulator blocks once it publishes its time advance value waiting for all the other simulators to do the same, then, the one with the minimum value is allowed to proceed and advance the clock. Since all the simulators are working

simultaneously, there is chance for internal and external transitions to take place at the same time. In this case, the user has the option of selecting which one to consider first, with the default being executing the internal transition function followed by the external transition function.

- **DEVS/RMI**

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. The use of RMI has allowed for the synchronization of local and remote objects without additional simulation time management to the one used in a stand-alone version of the simulator. In addition, Java provides a platform-independent environment for the execution of DEVS models. Different components in the engine play different roles during the execution of the model. The S*imulation Controller* is responsible for controlling the activities taking place during the simulation. This includes taking the partition information generated by the *Configuration Engine* and transferring it to the host machines to be executed by the *Remote Simulators*. In addition, the configuration engine may decide that a new partitioning is required during the execution of the model; in this case, the current execution is stopped and the simulation environment is reconfigured before the simulation is resumed. The *Simulation Monitor* collects information about the model being executed and conveys this information to the configuration engine to recreate the model partitions (if necessary). The partitioning of the model can be one of two types:

  - *Static partitioning*: in this case, the model is partitioned at the creation phase and is attached to the corresponding simulator.
  - *Dynamic partitioning*: the model is dynamically partitioned in a manner that allows for the re-partitioning during the execution of the model.

Zhang, Zeigler, and Hammonds [Zha05] show that using two or more machines to run relatively simple models introduces communication overhead that slows the simulation

down. However, when running complex models, the distribution of the model on two or more machines improves the performance which translates into shorter execution time.

- **DEVS/Cluster**

DEVS/Cluster [Kim04] is multi-threaded distributed DEVS simulator based on CORBA [OMG02]. The simulator was developed using Visual C++ following the optimistic approach for synchronization among the nodes. It uses Time Warp [Jef85] algorithms in order to achieve speedup by advancing the clock in each machine independently. In addition, DEVS/Cluster adopts a flattened simulation hierarchy for the execution of hierarchal DEVS models. This improves the performance of the flat simulator compared to the case of having a hierarchal one.

CORBA is used to allow for a location-transparent environment for distributed simulation. The synchronization of the simulation is handled by the *coordinators* that exchange messages with each other and with the simulators using the services provided by CORBA. Message passing is implemented as direct remote method invocations on the receiving simulator/coordinator instead of sending and receiving explicit messages.

- **PCD++**

PCD++ [Tro03] [Gli04] is a parallel simulation engine developed using WARPED [War06] middleware and uses MPI [MPI95] for communications. It is based on the CD++ simulation engine [Wai02], and is able to execute DEVS and Cell-DEVS models. WARPED is a middleware that provides basic functionality usually requited in a M&S environment. It implements the concept of *Logical Processors (LPs)* as the execution entities of the model. Each node has a *logical processor (LP)* that has one or more *simulation objects*. Messages sent between two *processors* are *wrapped* into WARPED messages before they get *unwrapped* (at the receiving end) into the original DEVS messages used for the CD++ engine. In addition, WARPED provides the data structures and utilities that can be used for the implementation of Time Warp algorithms [Jef85].

The original version of PCD++ [Tro03] followed a hierarchical approach for the simulator and it uses a conservative algorithm for synchronization among the nodes. It has been shown that the performance of the engine is dependent on the nature of the model and the partitioning scheme used to split the model on the different nodes. If the model partitions are loosely-coupled in a way that minimizes the remote messages sent among the nodes, the simulator performs well in terms of the speedup achieved compared to using one machine to execute the model. However, in the case of tightly-coupled partitions, the overhead can be significant, which in turn, may degrade the performance of the simulator.

An improved version of PCD++ [Gli04] 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 [Jef85] protocol for synchronization among the different nodes participating in the simulation. The performance of this version is much better than the original one; however, it requires more resources in order to save and restore the states of the model and simulator during the execution/rollback phases.

3.1 Web Service-Based Approach for Distributed DEVS Simulation

We follow a different approach for the design and implementation of distributed simulation engine based on CD++. The design methodology we follow depends on implementing web service-based simulation services, able to expose the functionality of CD++ in a standard way, and to execute complex models in distributed environments using SOAP as a messaging protocol. The design approach we propose has the advantage of providing several features that either were absent or partially provided by the other implementations:

- ***Efficiency:*** We aim at avoiding the shortcomings of some of the available distributed DEVS engines. For example, DEVS/Grid [Seo04] and DEVS/P2P [Che04] depend on synchronizing the simulation by each *processor* sending the time of its next change to all of the other *processors*. The *processor* running in one machine blocks until it

receives the values from all the *processors* in the other machines, and then the one with the earliest value unblocks by processing the next event. Although the authors did not provide any results to examine the performance of the simulator in a distributed environment, it is expected that the number of messages sent among the *processors* in each simulation cycle is causing a considerable overhead, especially when there is large number of machines involved in the simulation. We argue that the design we present in this dissertation, which uses a *coordinator* to schedule the *processors* for execution, limits the number of synchronization messages sent among the *processors* and hence improves the performance of the simulation. In addition, the implementation of *Master* and *Slave coordinators* allows the *processors* to exchange messages locally if the sender and receiver are running on the same machine without the need to send any remote messages; this in turn, reduces the overhead of exchanging remote messages in distributed environments.

- ***Flexibility***: The flexibility pertinent to our design in terms of having separate, yet related, simulation and web service components, has proven to be useful when porting the services to a different simulation engine. In this regard, the services that were developed to work with the stand-alone version of the simulator (CD++) were extended to work with a parallel version (PCD++) running on a high-end distributed-memory cluster with minimal changes and short development time. Although PCD++ performs better than the distributed engine proposed here, it is not as flexible in terms of the network connectivity among the nodes participating in the simulation. PCD++ uses MPI for messaging and it requires that the machines be located in close proximity to each other. On the other hand, the distributed engine we propose is able to function irrespective to the network infrastructure used to connect the nodes, which can be regular Ethernet connections, or high-speed fibre optic links.

- ***Web Service Integration***: The main web service standards such as XML, WSDL, and SOAP were used 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. This allows the modeller to execute the model, check the status of the simulation, and retrieve the

results remotely irrespective to the platform used by the client. In addition, the use of web services without restricting the implementation to any particular grid middleware, such as Globus, provides the flexibility required for integration with different systems using standard orchestration languages such as Business Process Execution Language (BPEL) [And03]. One of systems that can be integrated with the simulation services is a visualization service that allows the modeller to examine the simulation results in a user-friendly manner.

Chapter 4: 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 service technologies.
ii) Using SOAP as a messaging protocol to enable a distributed version of CD++ to execute complex models on multiple machines.

4.1 Design Methodology

In order to integrate the web service technologies with the CD++ toolkit, a web service wrapper was developed to interact with the CD++ toolkit and wrap its functionality to be accessed by web service clients. Two main design approaches were considered at the beginning. One is to develop the wrapper in C++ since this will allow for better integration with the original code of the toolkit; another is to develop the wrapper in Java

and interface the Java classes to the original C++ code of the toolkit when it is necessary to do so. The second approach was adopted due to the following reasons:

i) Many of the web service technologies and middleware available in the market today are well supported by Java and some of them are actually written in Java. So, using Java allows for better use of the web service tools and technologies as they advance.
ii) Building the simulation web service in a modular manner consisting of different C++ components (to interact with the simulator) and Java components (to interact with the web service clients) helps to develop different versions of the service to work with the different versions of CD++ with minimal changes.

One disadvantage of this design approach is that interfacing the Java and C++ parts of the simulation service is inevitable, since the service needs to access and manipulate the data structures and objects used by the simulator. The Java classes are mainly responsible for handling the web service part of the service functionality. On the other hand, the C++ classes are responsible for accessing and manipulating the data structures and objects used by the simulator. To integrate the two parts, Java Native Interface (JNI) [Lia99] was used. JNI is a collection of APIs and is part of the Java Virtual Machine (JVM) developed by Sun Microsystems. It allows Java programs to access functions written in native C/C++ code. In addition, it allows programs written in C/C++ to execute and access Java objects. The following diagram shows an overview of the service layers.

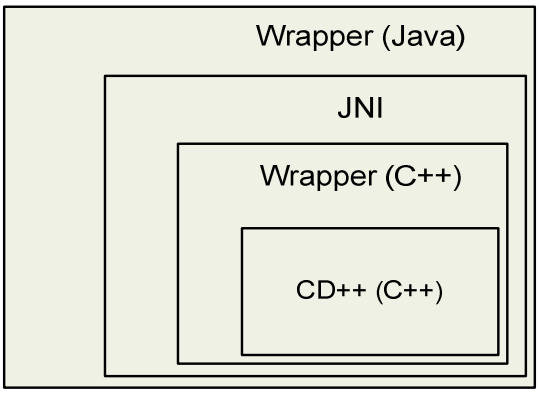

Figure 17: Major components of the simulation service

The simulation service acts as a web service interface to the CD++ toolkit. The main activities performed by the service are:

- Receiving the required files to define the model and execute the simulation. These files include: C++ and header files (in the case of DEVS models), a model definition file (.ma), and an external input file (.ev).
- Executing the simulation providing the client with the ability to check the progress of and kill the simulation (if needed).
- Sending the results of the simulation to the client in the form of text files. These files include: an external output file (.out), a simulation log file (.log), and a debug information file (.info).

The web service engine chosen for the implementation is Apache Axis [Axi06]. Axis is an open source SOAP engine that has an HTTP server functionality and runs as a web application within an application server, in our case Tomcat application server [Tom06].

4.2 Implementation Details

The wrapper was originally designed to load the simulator as a shared library that can be used to execute the simulation and return the results to the client. The advantage of this approach is that loading the simulator by the wrapper as a shared library, provides a straightforward way of accessing and manipulating the data structures of the simulator, since both (the web service and simulator) will be running as one operating system process. In addition, since the same simulator can be used to execute more than one Cell-DEVS model, this can save memory and storage space. However, designing and implementing the service in this approach has revealed two main issues that had to be resolved:

i) Loading the same simulator as a shared library may cause the web service and the simulator to crash if one of the running sessions generated an exception. This is not acceptable since the CD++ web service should be able to run multiple sessions concurrently without having one of the sessions affecting the others.

ii) The Java Virtual Machine (JVM) can not load the same native shared library more than once during the lifetime of the class loader used to load the library. In addition, the same library can't be loaded by two different class loaders. This restriction was imposed on the JVM as of Java 1.2 to avoid class name conflicts since the class loader is considered part of the class full name used within the JVM [Lia99].

Considering the previous points, 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*).

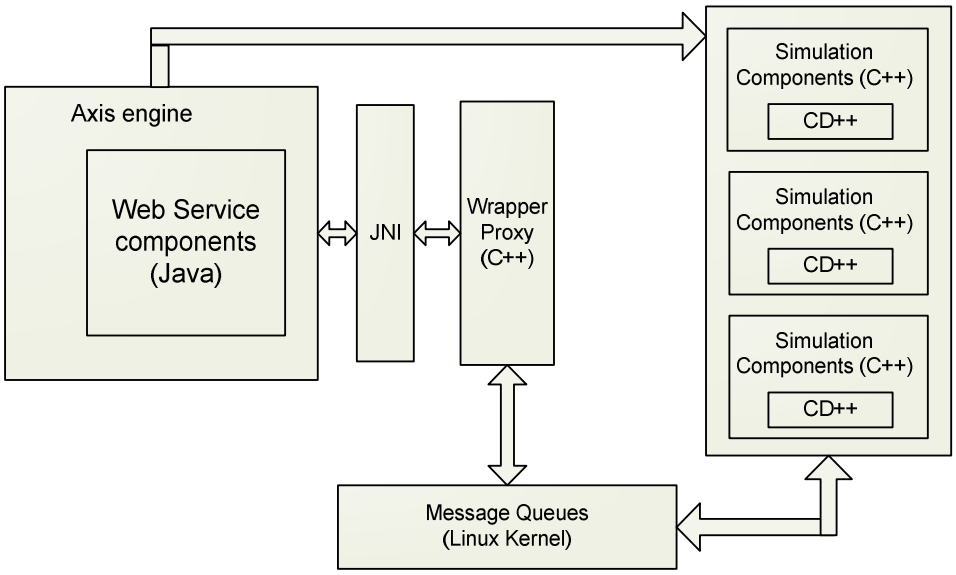

Figure 18: Implementing the simulation service using JNI and message queues

The advantages of this approach are that:

i) It provides a separate running workspace for each simulation session; the simulator is running 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.

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 in C/C++ and loaded as a shared native library into the JVM. At this point the simulation service is considered ready to receive client requests. The exact behaviour of the web service components depends on the type and sequence of requests submitted by the client; however, a typical sequence of operations is depicted in Figure 19:

- The user is authenticated and if logged on successfully, a new session is initialized for him.
- A new folder is created on the server to provide a working space for the new session. The executables and source files of the simulator are copied to the new session folder.
- The web service components invoke a method in the *WrapperProxy* to initialize a new session. The *WrapperProxy* is responsible for the communications between the web service and simulation components of the simulation service. The *WrapperProxy* is implemented as a shared library and is loaded only once during the lifetime of the Axis server, hence avoiding the constraint of JVM not being able to load a particular native library more than once.

- The *WrapperProxy* creates two message queues through the Linux kernel. One queue will be used to send messages from the web service components to the corresponding CD++ session, and the other will be used to receive messages from CD++.
- Once the initialization steps are over, the user can submit the different files and parameters necessary to define the model.
- If the user chooses to set DEVS models by sending C++ header and implementation files, the wrapper will update the *make* file (used to compile the simulator and the models) to incorporate the newly added models. In addition, part of the source code of the simulator is updated to register the new DEVS models.
- When the user starts the simulation, if the user has defined at least one DEVS model, the wrapper will compile the source code of the simulator with the newly added models. In addition, the web service components will initialize the slave sessions in case of running distributed simulation; slave sessions will be running on other machines (other than the first machine that the user is connected to). Then, the simulation will be started.
- On the CD++ side, two additional parameters are provided to the simulator. These are the full path of the session directory, and the session ID that was assigned to the simulation session.
- Once the user invokes the *startSimulationService* operation and before actually starting the simulation, CD++ will invoke a method to initialize the session (from the CD++ side) through the simulation components. CD++ will use the full path of the session to query the Linux kernel for the message queues created by the *WrapperProxy*. These queues are used to communicate with the web service components associated with the current simulation session.
- When the simulation ends, and in the case of distributed simulations, the web service components will retrieve the log files from the slave machines and archive them into a single file to be retrieved by the user.

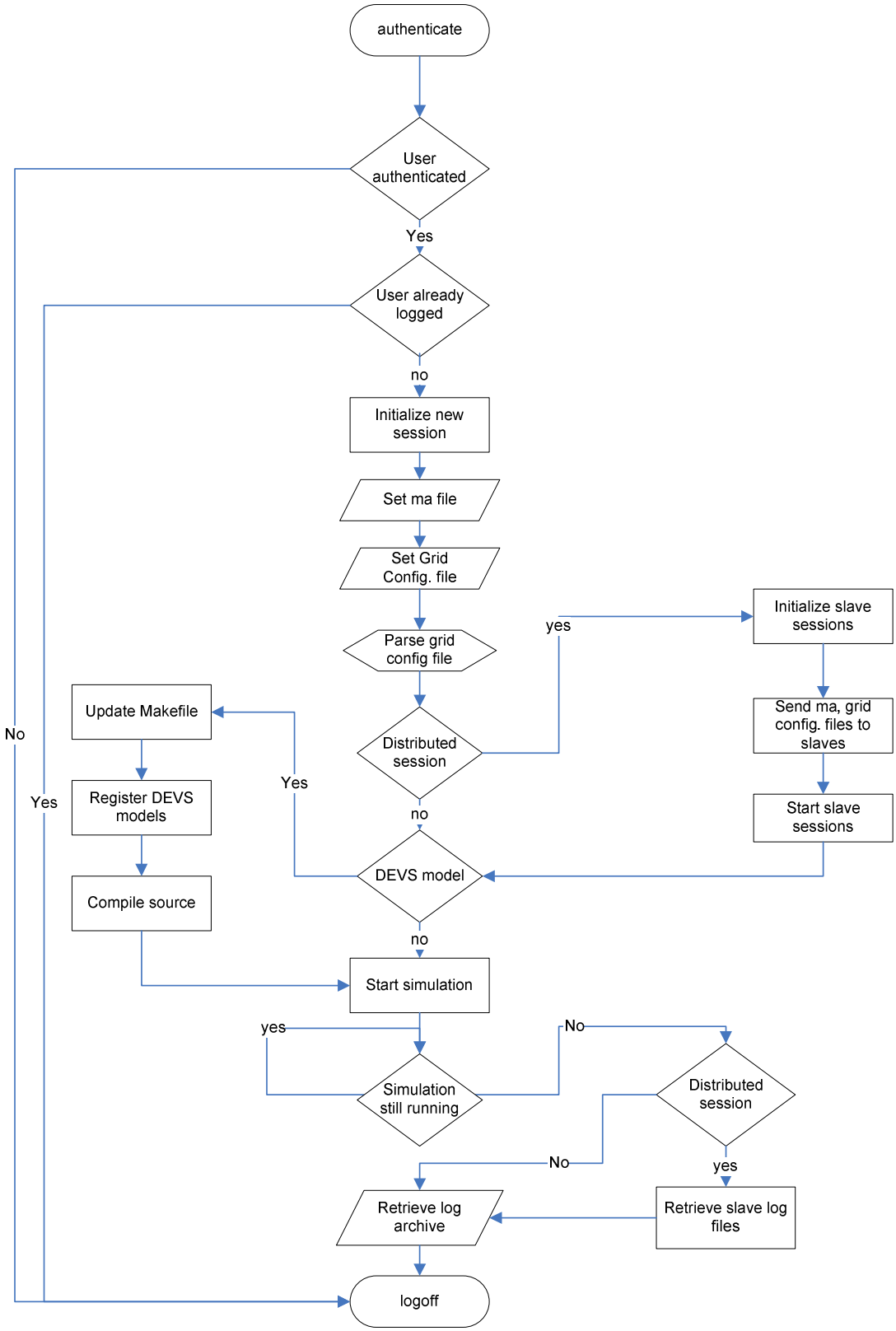

Figure 19: Simulation web service operation

The simulation process is started as an external command executed by the web service components and not through the message queues. In addition, for each session there will be three Java threads and two Linux-POSIX threads. One Java thread is responsible for executing the CD++ simulator and streaming its output into the session's log file, another thread is responsible for responding to the web service client requests, and the third thread is responsible for monitoring the message queues (though the *WrapperProxy*). On the CD++ side, one is the main simulation thread, and the other thread is used to monitor the message queues for an incoming message from the web service components.

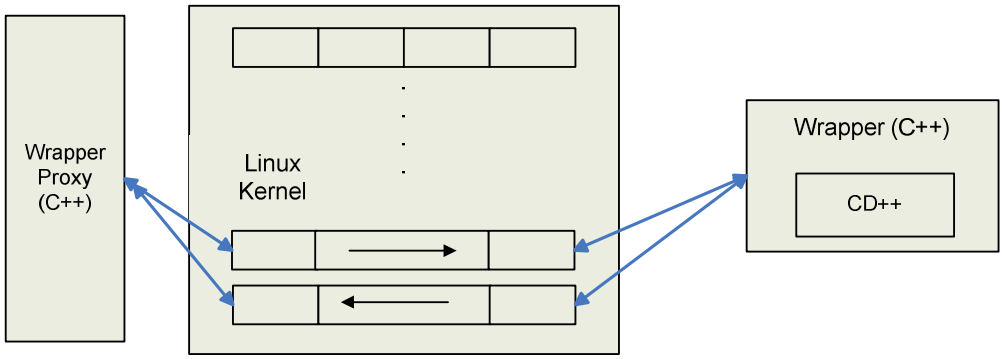

Figure 20: Message queues connecting the simulation components to the *WrapperProxy*

4.3 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.

Figure 21 shows a UML diagram of the web service components of the simulation service. The *JavaWrapper* class is the backbone of the web service components; and it includes the attributes and methods necessary to handle most of the operations offered by the service. Some of the operations performed by the *JavaWrapper* class include: (a detailed description of the web service components is presented in Appendix-B)

- User authentication: the method *authenticate* is used to authenticate users through a password file stored on the server.
- Session initialization: the method *createNewSession* creates a working space for new sessions. 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. In addition, the method *initialize* is used to initialize the resources needed for the session, such as the message queues, to communicate with the simulator.
- Setting the model definition: the methods *setMAFile, setEventFile, setDEVSModel,* and *setSupportFile* are used for defining the model. The *setMAFile* is used to submit the model definition, *setEventFile* sets the external

events file, *setDEVSModel* sets the source and implementation files for DEVS models, and *setSupportFile* sets the initial values file for Cell-DEVS models.

- Setting the configuration information for distributed sessions: the method *setGridConfigFile* is used to send the *grid configuration file* by the user; once the method is executed it causes the parser to parse the file and save the information contained in it in the *JavaWrapper* instance created for the session.
- Starting the simulation: the method *startSimulationService* is used to start the simulator. This includes some initialization to take place such as compiling the submitted DEVS models (if any) 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: the method *isSimRunning* is used to check the status of the simulation process. 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 method *killSimulation* is used to kill the simulation process (if needed).
- Retrieving the results of the simulation: the methods *retrieveLogFileName*, *retrieveOutputFileName* are used for the log and output files retrieval, respectively. 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 into one file that can be sent to the user.
- Logging off: the method *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.

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

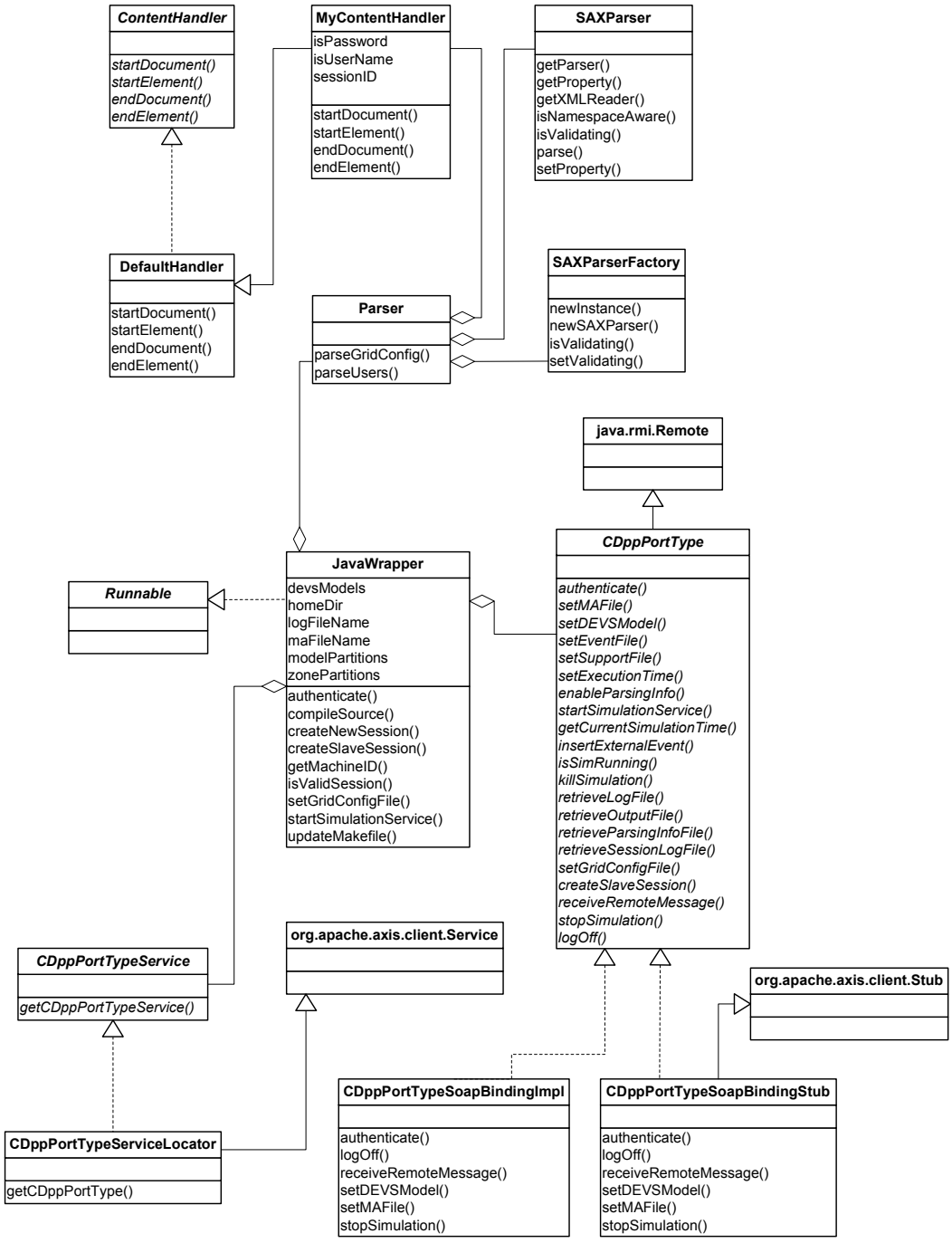

Figure 21: Web service components UML diagram

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 (see Figure 18), 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]:

- *initializeNewSession*: it creates two message queues for each session to act as a communication channel between the web service and simulation components of the service.
- *getCurrentSimulationTime*: it is used to query the simulator for the current execution time.
- *insertExternalEvent*: it inserts external events in the simulation while the simulation is running.
- *startMessageMonitor*: it starts the message monitor that keeps checking for any message coming from the simulator. This is started as a separate thread from the Java side.
- *getMachineID*: it gets the id of the machine running the simulation. This executes the *getMachineID* method in the *JavaWrapper* class, which in turn checks the address of the running service and compares it with the ones available in the *grid configuration file* to find the machine id.
- *machineForModel*: it returns the id of the machine running a particular session. It is used in distributed simulation sessions. This information is retrieved from the *JavaWrapper* class which keeps the information supplied to the service through the *grid configuration file*.
- *sendRemoteMessage*: it is used to send remote messages between machines in distributed simulation sessions. It takes a C++ message and passes it to the web service components to be sent as a SOAP message.
- *receiveRemoteMessageByProxy*: it is used to receive remote messages when running distributed simulations. It gets a SOAP message contents from the web service components and passes it to the simulator.
- *stopSimulationSession*: it is used to stop the simulation session and to deallocate any used resources.
- *addZonePartition*: it is used to define the Cell-DEVS model partitions when running distributed simulations.

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. When parsing XML documents, there are normally two approaches that can be adopted; using a *SAX (Simple APIs for XML)* parser or a *DOM (Document Object Model)* parser. *SAXParser* is an event-driven parser that calls specific methods in the *ContentHandler* class (or one of its children) at specific points of the parsing process, such as the beginning and end of each element in the XML document. The programmer can then override the functions defined in the *ContentHandler* class in order to implement the required functionality. Another option would be using a *DOM* parser that loads the entire document into memory and allows the programmer to manipulate the document. 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) The URLs of the simulation services participating in a session;

ii) The model partitioning information which includes the parts of the model running on each machine in a distributed simulation session;

```
<Grid>
<MACHINES>
        <MACHINE>
                <MACHINE_RANK>0</MACHINE_RANK>
                <MACHINE_URI>http://192.168.1.146:8080/axis/services/CDppPortType</MACHINE_URI>
        </MACHINE>
        <MACHINE>
                <MACHINE_RANK>1</MACHINE_RANK>
                <MACHINE_URI>http://192.168.1.144:8080/axis/services/CDppPortType</MACHINE_URI>
        </MACHINE>
</MACHINES>
<MODEL_PARTITIONS>
        <PARTITION machine="0">
                <MODEL>particlegenerator</MODEL>
        </PARTITION>
        <PARTITION machine="1">
                <ZONE>sandpile(0,0)..(9,9)</ZONE>
        </PARTITION>
</MODEL_PARTITIONS>
</Grid>
```

Figure 22: A sample *grid configuration file*

Figure 22 shows a sample *grid configuration file*. It consists of two main elements: the *MACHINES* element and the *MODEL_PARTITIONS* element. The *MACHINES* element includes two sub-elements for each machine participating in the simulation session. The *MACHINE_RANK* is the machine id, and the *MACHINE_URI* is the URL used to access the service. The *MODEL_PARTITIONS* element contains one *PARTITION* element for each model partition in the machine. Each model partition can be a *MODEL* designating a DEVS model or a *ZONE* designating a Cell-DEVS zone (group of cells). The id of the machine running the model partition is set as an attribute of the *PARTITION* element.

The 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. This client-side stub is used within the simulation service to access and setup slave sessions while running distributed simulations. When the user connects to one machine to start a distributed simulation session, the web service components examine the *grid configuration file* in order to extract the addresses of the services participating in the simulation. Then, it uses the *CDppPortTypeServiceLocator* class in order to locate the slave machines and create instances of the *CDppPortTypeSoapBindingStub* class that are

used to send the model definition files and the *grid configuration file*, and to initialize new sessions in the slave machines.

4.4 Service Interface

In order for the client to "consume" the simulation service, he needs to have access to the WSDL document defining the service interface. Then, the client can choose one of two options: either to generate client-side stubs, in which case he can deal with the operations offered by the simulation service as if they were local object methods; or he can invoke the services by dynamically creating SOAP requests.

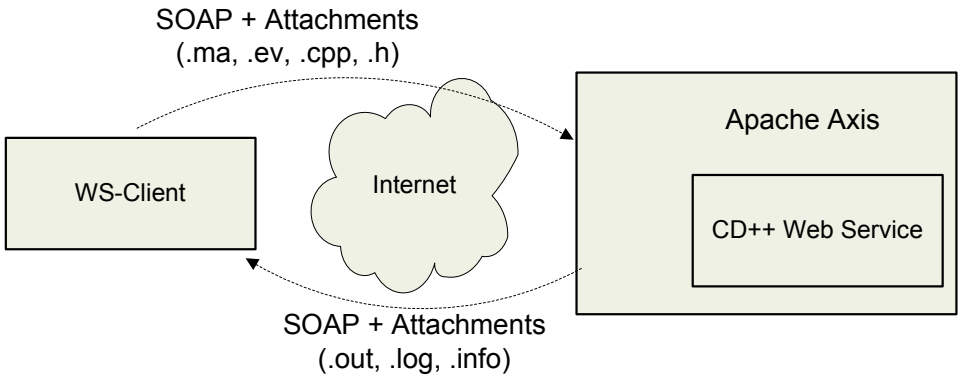

Figure 23: A typical invocation of the simulation web service

WSDL documents usually contain a *type* element to define non-standard parameter types of the messages exchanged between the web service and the client. This element does not exist in our implementation since the types are defined within the message itself. The *message* element defines the request and response SOAP messages. Figure 24 shows the request and response messages for the *setDEVSModel* operation; the *setDEVSModel* operation takes four arguments (through the message *setDEVSModelRequest*): the name of the header file defining the DEVS model class, a *DataHandler* object representing the file (sent as a SOAP attachment), the name of the C++ file containing the class implementation, and a *DataHandler* object representing the C++ file. *DataHandler* is a Java class that provides a consistent interface to data available in many different formats,

in our case the *DataHandler* represents a file that gets serialized by the client into SOAP attachment and gets deserialized to a file on the server side. The *setDEVSModelResponse* message represents the return type of the *setDEVSModel* operation, which is a string stating whether the operation was successful or not.

```
<wsdl:message name="setDEVSModelRequest">
        <wsdl:part name="in0" type="soapenc:string" />
        <wsdl:part name="in1" type="apachesoap:DataHandler" />
        <wsdl:part name="in2" type="soapenc:string" />
        <wsdl:part name="in3" type="apachesoap:DataHandler" />
</wsdl:message>
<wsdl:message name="setDEVSModelResponse">
        <wsdl:part name="setDEVSModelReturn" type="soapenc:string" />
</wsdl:message>
<wsdl:message name="isSimRunningRequest" />
<wsdl:message name="killSimulationRequest" />
```

Figure 24: An excerpt of the *message* definition of the simulation web service

The *portType* element defines a collection of operations, each operation has an input and output. In this case (Figure 25), the input is the ***setDEVSModelRequest*** message and the output is the ***setDEVSModelResponse*** message. The *portType* element is analogous to the *Interface* concept in the Java programming language.

```
<wsdl:portType name="SimulationPortType">
        <wsdl:operation name="setDEVSModel" parameterOrder="in0 in1 in2 in3">
                <wsdl:input message="impl:setDEVSModelRequest"
                        name="setDEVSModelRequest" />
                <wsdl:output message="impl:setDEVSModelResponse"
                        name="setDEVSModelResponse" />
        </wsdl:operation>
```

Figure 25: An excerpt of the *portType* definition of the simulation web service

The *binding* element defines the binding of the web service SOAP messages to an actual protocol (HTTP or SMTP). In addition, it defines the encoding style (RPC/message) and encoding type (encoded/literal). Figure 26 shows a partial definition of the binding of the simulation service to HTTP (*http://schmas.xmlsoap.org/soap/http*). The *binding* element lists the operations implemented in the service with the input and output messages for each one.

```
<wsdl:binding name="SimulationServiceSoapBinding" type="impl:SimulationPortType">
<wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="setDEVSModel">
                <wsdlsoap:operation soapAction="" />
                 <wsdl:input name="setDEVSModelRequest">
                        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                        namespace="http://www.sce.carleton.ca/ARS/SimulationService"
                                                use="encoded" />
                </wsdl:input>
                 <wsdl:output name="setDEVSModelResponse">
                        <wsdlsoap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                                namespace="http://www.sce.carleton.ca/ARS/SimulationService"
                                        use="encoded" />
                </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
```

Figure 26: An excerpt of the *binding* definition of the simulation web service

The *service* element groups a number of ports together. Each port links a *binding* definition of a specific *portType* to a Uniform Resource Identifier (URI) to be used to access the service. In Figure 27, the simulation service binding (*SimulationServiceSoapBinding)* is linked to the *SimulationService* port, which in turn is assigned the URL (*http://localhost:8080/axis/Service/SimulationService)*. The URL is necessary in order for the clients to access and utilize the simulation service.

```
<wsdl:service name="SimulationPortTypeService">
        <wsdl:port binding="impl:SimulationServiceSoapBinding" name="SimulationService">
        <wsdlsoap:address location="http://localhost:8080/axis/Service/SimulationService" />
        </wsdl:port>
</wsdl:service>
```

Figure 27: An excerpt of the *service* definition of the simulation web service

The operations offered by the simulation web service are:

- ***authenticate***: it is responsible for authenticating users and initializing a new session for each successful login.
- ***setMAFile***: it is used to set the model definition file (.ma).
- ***setDEVSModel***: it is used to set a DEVS model by C++ header and implementation files.
- ***setEventFile***: it is used to set the external events file (.ev).
- ***setSupportFile***: it is used to set support files that need to be available to the simulator such as a file containing the initial values of the cells (in the case of Cell-DEVS models).
- ***setExecutionTime***: it is used to set the execution time of the model.
- ***enableParsingInfo***: it is used to inform the simulator to generate a parsing information file that can be used to debug Cell-DEVS models.
- ***setGridConfigFile***: it is used to set the *grid configuration file* which contains the model partitions and the addresses of the machines participating in a distributed simulation session.
- ***createSlaveSession***: it is used to initialize slave sessions when running distributed simulations.
- ***receiveRemoteMessage***: it is used to exchange remote messages during a distributed simulation session.

- ***stopSimulation***: it is used by the master machine to stop the simulation in the slave machines at the end of a distributed simulation session.
- ***startSimulationService***: it is used to start the simulation.
- ***isSimRunning***: it is used to check whether the simulation is running or not.
- ***getCurrentSimulationTime***: it is used to check the current simulation time.
- ***insertExternalEvent***: it is used to insert external events to the model while the simulation is running.
- ***killSimulation***: it is used to kill the simulation.
- ***retreiveLogFile***: it is used to retrieve the log file(s) generated by the simulator.
- ***retreiveOutputFile***: it is used to retrieve the output file generated by the simulator.
- ***retrieveParsingInfoFile***: it is used to retrieve the generated parsing information file that can be used to debug Cell-DEVS models.
- ***retrieveSessionLogFile***: it is used to retrieve the session log file which includes the output messages generated by the simulator.
- ***logOff***: it is used to log the current user off and to invalidate his session.

Chapter 5: 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.

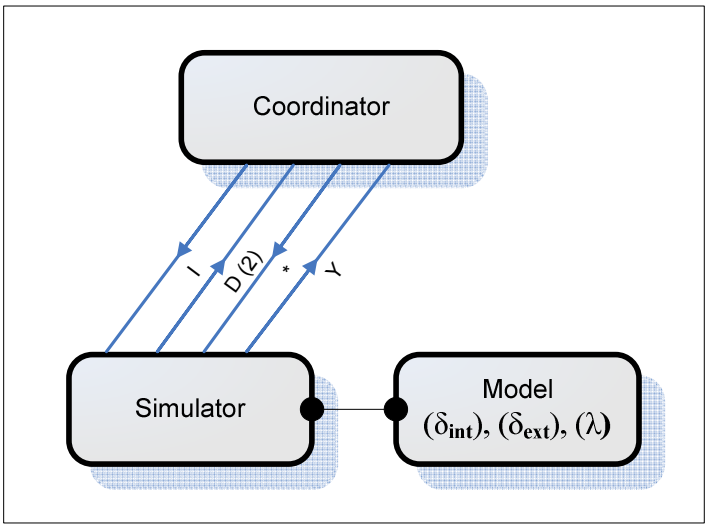

Figure 28: Message exchange during a simulation cycle

When the simulation is started, the *Root coordinator* sends *initialization* messages (*I*) to all of its child coordinators which in turn forward them to their child coordinators and simulators. When the simulator receives an *initialization message*, it calculates the time of the next state transition and it reports it to its parent coordinator through a *done message (D).* When the *Root coordinator* receives all the *done messages* from its child *processors*, it advances the simulation clock to the time of the next state transition, and it sends an *internal message (*)* to the simulators of the imminent child models starting a new simulation cycle. When the simulator receives an *internal message* from its parent

coordinator, it executes the *output function (*λ*)* of its model and sends an *output message (Y)* to the parent coordinator. Then, it executes the *internal transition* function *(*δ_{int}*)* of the model in order to evaluate the next state. The final step of state transition would be sending a *done message* to the parent coordinator reporting the time of the next state change of the mode. If an external event is forwarded to the simulator though an *external message (X)* (from the environment, or translated from an *output message* from another model), the simulator executes the *external transition* function *(*δ_{ext}*)* of the model and reports the time of the next state change to its parent coordinator. The previous steps continue until there are no more messages/events to process or until the simulation clock reaches the maximum execution time as provided by the modeller.

CD++ was developed originally to run on a single workstation; by implementing the original CD++ algorithms, it was able to run DEVS and Cell-DEVS models as long as the modeller defines the *select* function for tie breaking. Whenever two models are scheduled for state transitions at the same time (as shown in Figure 29), CD++ would pick the one specified by the *select* function to execute first, followed by the other imminent models. Although this might be acceptable for some models, it has two limitations:

i) It introduces a serialization problem that may lead to incorrect model execution.

ii) It prohibits the modeller from defining complex Cell-DEVS models taking advantage of the zero-delay permissible by the Parallel Cell-DEVS formalism.

Figure 29 shows an example of two simulators in two scenarios; with and without tie. The left part of Figure 29 shows message exchange sequence between the coordinator and the two simulators. The coordinator sends an *initialization message (I)* to the simulators followed by two *done messages (D)* sent by the simulators reporting the times of their next state changes. *Simulator 1* is scheduled for internal transition after two time units, and *Simulator* 2 is scheduled for internal transition after six time units. This results in the coordinator activating *Simulator 1* first (by sending an *internal message (*)*) followed by *Simulator 2*. The second case (shown in the right part of Figure 29), shows

the two simulators scheduled for internal transition at the same time (after four time units). This results in the coordinator examining the *select* function in order to decide which simulator to activate first, in this case *Simulator 2*.

In order to expand CD++ into a distributed engine, able to execute complex models in distributed environments, the serialization issue with CD++ had to be resolved. That is, partitioning the model on different components while using the original CD++ algorithms doesn't allow parallel execution of the model.

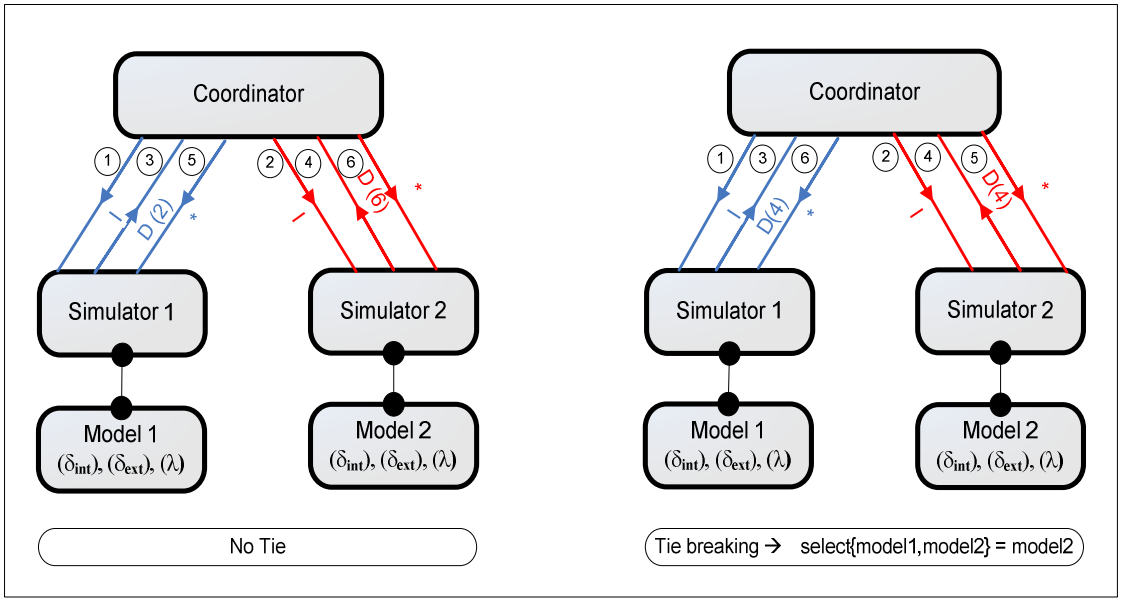

Figure 29: Tie breaking using the *select* function

5.1 Implementing the Parallel-DEVS Algorithms

The Parallel-DEVS (P-DEVS) algorithms [Cho94a] were introduced to solve the serialization problem with the original DEVS algorithm and to enable the execution of DEVS models in parallel and distributed environments. The main additions in P-DEVS are the message bags, and the *confluent transition* function (δ_{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.

CD++ was redesigned in order to implement the P-DEVS algorithms. As in the original version, *Simulators* are used to execute atomic DEVS and Cell-DEVS models, while *Coordinators* handle message passing and event synchronization between the different models. A *Root coordinator* is used for starting/stopping the simulation, clock advancement, and interfacing with the environment. CD++ executes the model by creating a simulator/coordinator hierarchy that matches the model hierarchy; for each atomic DEVS/Cell-DEVS model there is a simulator, and for each coupled DEVS/Cell-DEVS model there is a coordinator. The simulators and coordinators behave differently to each of the messages received. The simulators receive *initialization messages (I)*, *collect messages (@)*, *internal messages (*)*, *and external messages (X)*. However, coordinators receive *initialization messages (I)*, *collect messages (@)*, *internal messages (*)*, *external messages (X)*, *done messages (D)*, and *output messages (Y)*. The details of the algorithms that define the behaviours of the simulators and coordinators are presented in Appendix-A.

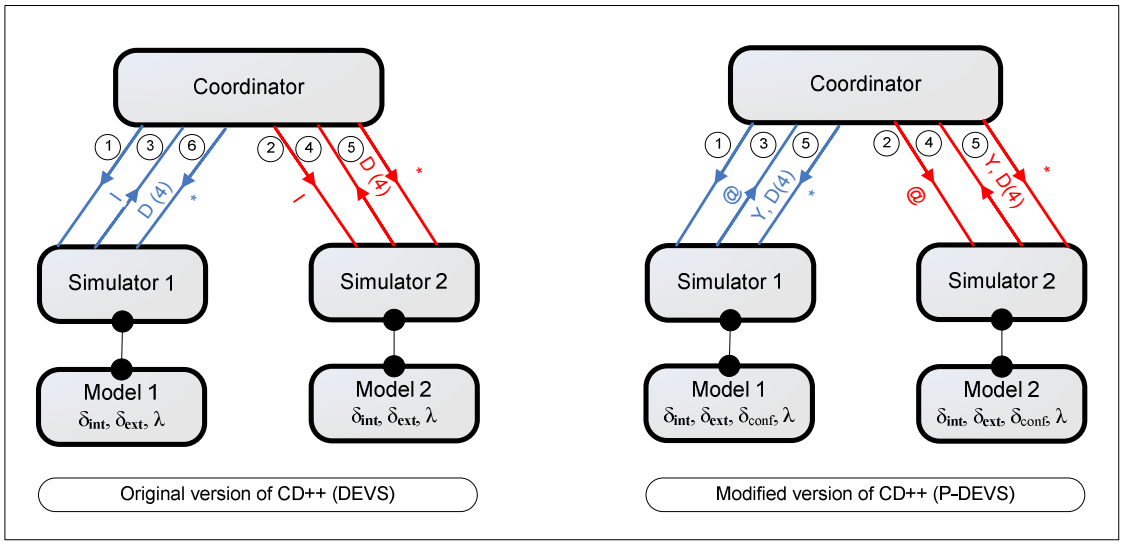

Figure 30: Concurrent model activation in Parallel-DEVS

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 30 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 solving the issue of serialization introduced in the original DEVS formalism.

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*. In addition to the *receive* method, the *processor* class implements three important methods for the execution of the model, those are:

- *lastChange():* it reports the time of the last state change;
- *nextChange():* it reports the time of the next state change;
- *absoluteNext():* it reports the absolute time of the next change (lastChange() + nextChange());

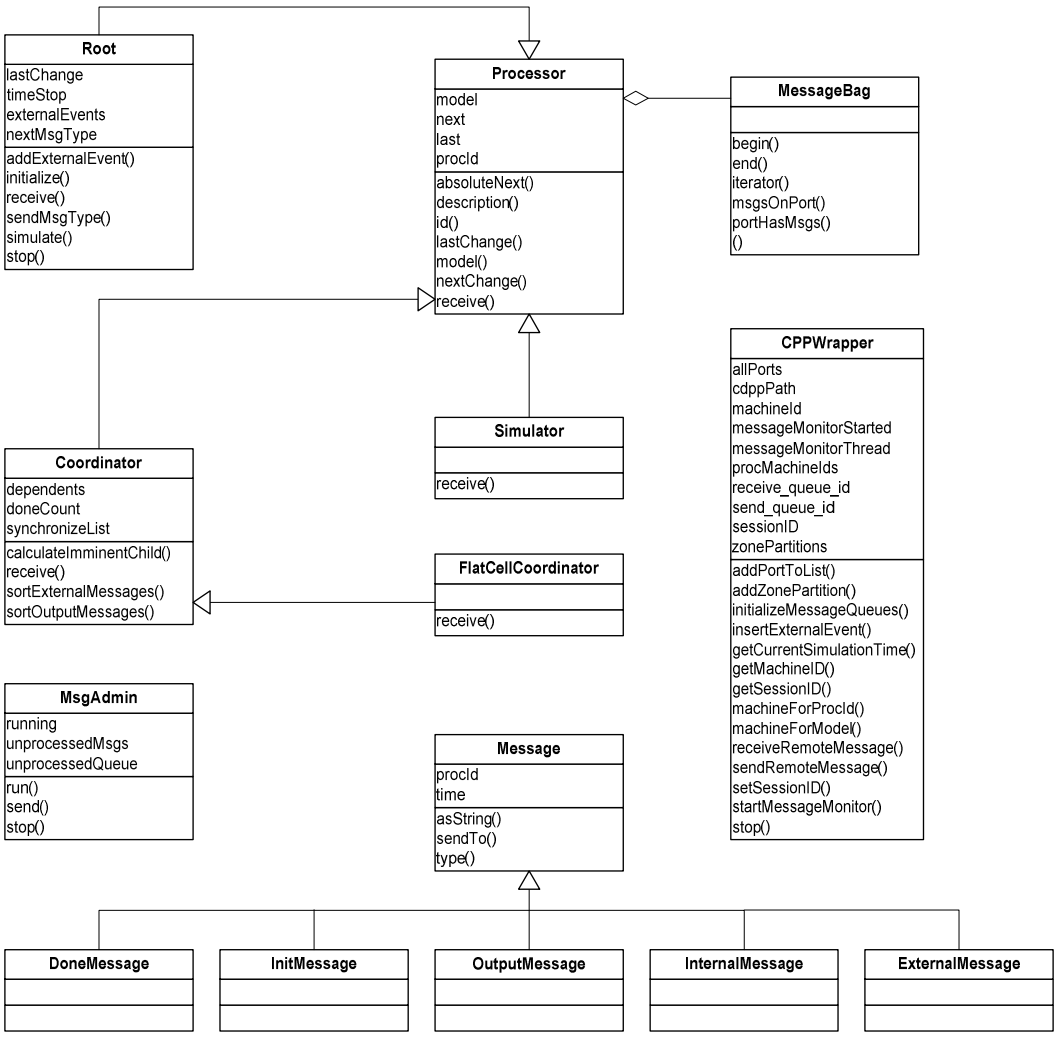

Figure 31: The simulation class hierarchy

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 (detailed description of the coordinator algorithms is provided in Appendix-A). 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* class is the main *processor* in the simulation and it is in charge of:

- Starting the simulation though the *simulate* method;
- Stopping the simulation through the *stop* method;
- Interacting with the environment in terms of loading the external events and generating the model output;
- Advancing the clock of the simulation;

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.

In addition to the simulation class hierarchy, other classes play an important role in driving the simulation. The *SimLoader* class (shown in Figure 32) is responsible for loading the model definition and execution options when the simulator is started and before executing the model. This includes loading the model definition and external events as input streams and loading the simulation log and output as output streams. The S*imLoader* is used by the *MainSimulator* class during the initialization phase of the simulation. The main method in the *MainSimulator* class is the *run* method, which organizes the activities handled by the *MainSimulator*. Those include loading the model hierarchy in the memory, loading the initial values of the cells, loading the external events, and creating the simulators to execute the model.

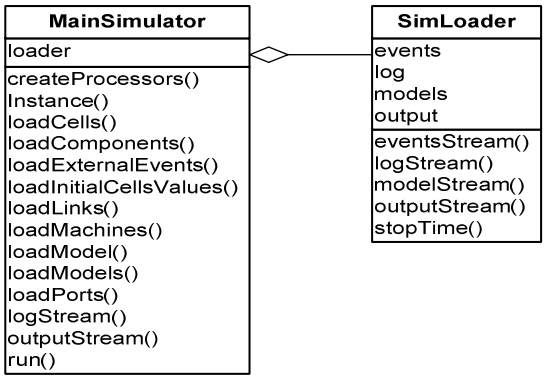

Figure 32: The *MainSimulator* class

5.2 Implementing the Simulation Components

As discussed in Chapter 3, the design of the simulation service depends on developing the service as a set of independent, yet related, components that interact by message passing through the Linux kernel. The major parts of the service are: *web service components*, *simulation components*, and the *WrapperProxy* which is used to pass messages between the two. The simulation components are responsible for executing the model and interacting with the web service components to receive the model partitions, fulfill any client request while the simulation is running, and retrieve the results when the simulation is over. They consist of two main parts, the modified version of the CD++ engine which is in charge of executing the simulation (discussed in the previous section), and the *CPPWrapper* class (see Figure 31), which is responsible for interfacing CD++ to the web service components.

The functionality of the *CPPWrapper* class includes:

- Initializing the message queues used for communication with the web service components (*initializeMessageQueues*).
- Querying and retrieving the model partitions from the web service 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 web service components to be sent to the remote machine.
- Receiving remote messages while running distributed simulations (*receiveRemoteMessage*). This method receives a message from the web service components and constructs a C++ message to be processed by the simulator.
- Stopping the simulation when receiving a stop message from the web service components (*stop*).

5.3 Designing and Implementing Distributed-CD++ (DCD++)

When considering the design and implementation of the distributed simulation engine, different approaches were considered to assess the integration of web service technologies with the algorithms used in the field of parallel and distributed simulation. The objective of the design was to take advantage of the web service capabilities while minimizing the overhead incurred on the simulator as a result of adopting a new middleware. 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, it depends on exchanging synchronization messages to handle rollbacks. When considering the overhead of transmitting SOAP messages embedded in HTTP packets, it was noticed that the speedup achieved by the Time Warp algorithm might be compromised by the delay of the SOAP messages.

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 P-DEVS 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 33, *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.

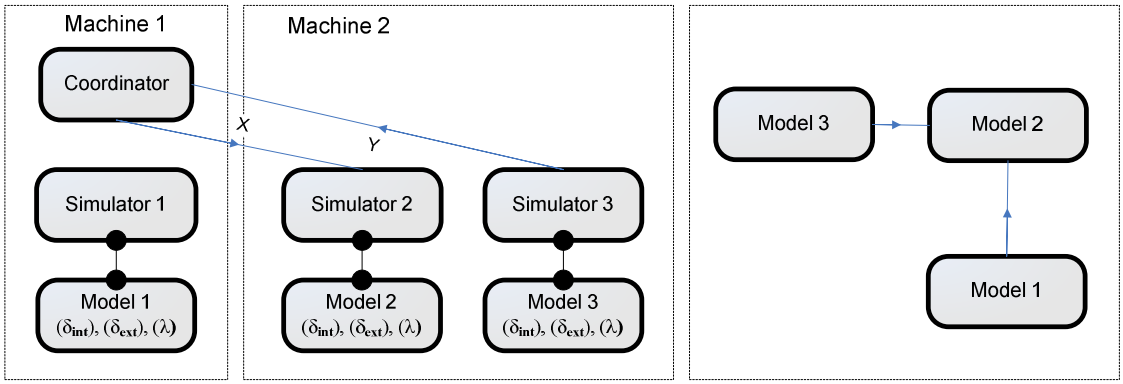

Figure 33: Unnecessary remote messages in distributed simulation

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 34), causes the message from *Simulator 3* to *Simulator 2* to be sent locally improving the performance of the simulator.

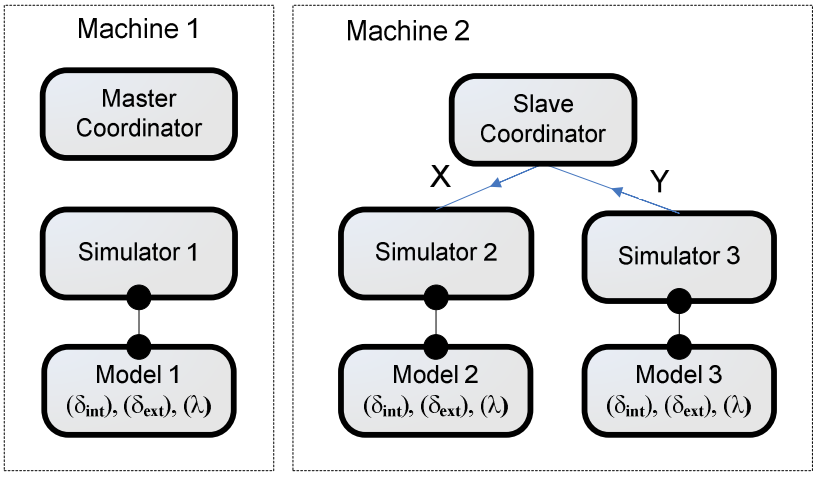

Figure 34: The use of *Master* and *Slave* coordinators to avoid unnecessary messages

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

i) The simulation mechanism is implemented mainly using the master and slave coordinators;
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;

5.3.1 Master and Slave Coordinators

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. A detailed description of the behaviour of the master and slave coordinators is presented in Appendix-A.

Figure 35 shows a partial definition of the master and slave coordinators, which are implemented by extending the *Coordinator* class and integrating them into the simulator class hierarchy. Both override the *receive* function used to process the different messages received by the *processors*. In addition, they implement the *sortExternalMessages* and *sortOutputMessages*. The *sortOutputMessages* method is triggered when receiving an *output message* from a child *processor*. The *sortExternalMessages* method is triggered when the coordinator receives an *internal message* from its parent coordinator. It causes the coordinator to process all the messages in its bag by forwarding them to their destinations either locally or remotely. The *calculateNextChange* method is used to evaluate the imminent child *processors* and its behaviour is different for each coordinator. 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*.

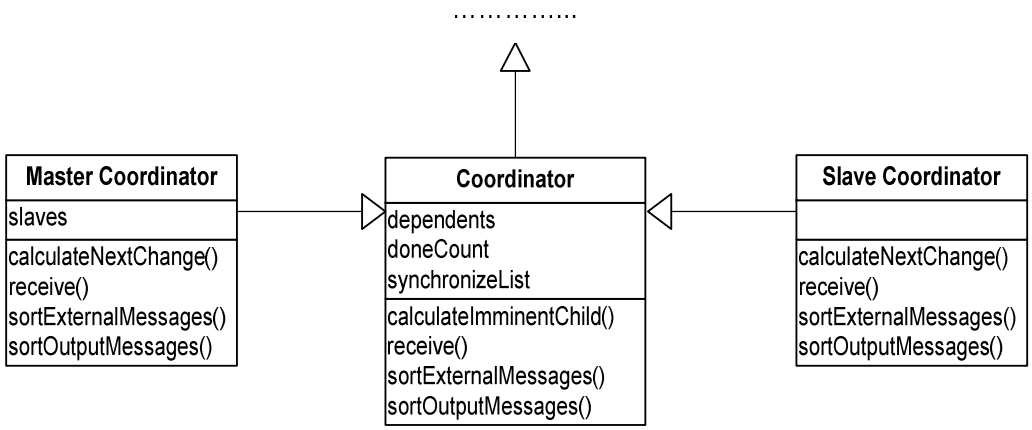

Figure 35: Master and Slave coordinator classes

5.3.2 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 DCD++, the model loading mechanism includes loading the partitioning information as part of the model loading process; the partitioning information is retrieved from the web service components through the *CPPWrapper* class. 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.

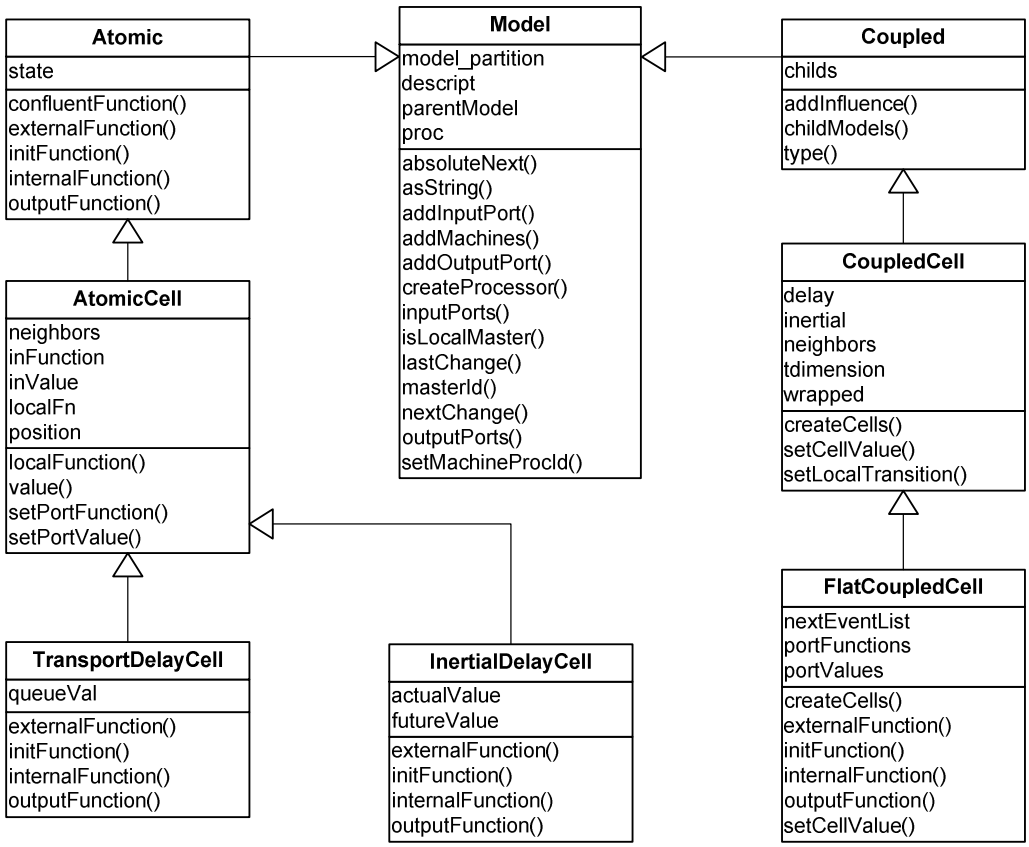

Figure 36: DCD++ model hierarchy

Figure 36 shows the relationship between the different classes representing the model hierarchy in DCD++. During the model loading process, the *MainSimulator* class (shown in Figure 32) executes the model's *addMachines* method, which is common to all the models. The *addMachines* method queries the *CPPWrapper* for the model partitioning information in order to store that information within the models; this information is used when the *createProcessor* method is invoked. The *createProcessor* method checks the model partitioning information to see if the model has a local component on the local

machine; if so, it creates the corresponding *processor* for the model. If the model is an atomic one, and is assigned to run on the local machine, a simulator is created. On the other hand, if the model is a coupled model with the first component assigned to run on the local machine, a master coordinator is created; otherwise, a slave coordinator is created and associated with the coupled model.

5.3.3 Message Passing Mechanism

The message passing mechanism was extended to handle local and remote messages. The *MsgAdmin* class is responsible for forwarding messages in coordination with the *CPPWrapper* class. The *MsgAdmin* class is activated when the simulation is started, and as long as there is at least one message in the *unprocessedMessages* queue. The *MsgAdmin* 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 executing its *receive* function. Otherwise, the message is passed to *CPPWrapper* which in turn passes it to the web service components. The web service components extract the message information and encapsulate it into a SOAP message that is sent to the receiving machine. When the SOAP message arrives to the destination machine, the web service components extract the information and pass it to the *CPPWrapper*. The *CPPWrapper* builds a C++ message and hands it over to the *MsgAdmin* class, which forwards the message like any other local message. This approach was followed to keep message passing transparent to the simulator in the case of local and remote messages. The message communication between the *CPPWrapper* class and the web service components takes place through the Linux kernel using the *WrapperProxy*.

5.4 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 37:

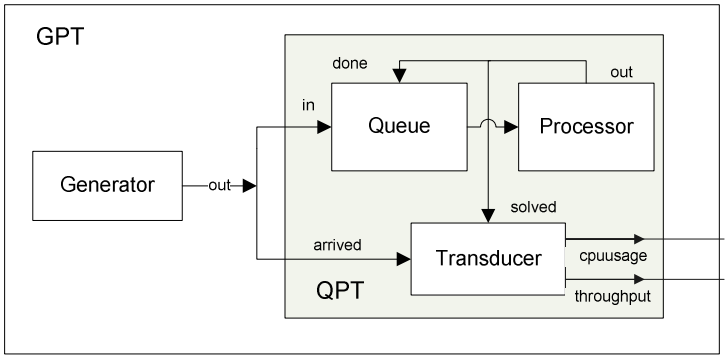

Figure 37: The Generator-Processor-Transducer (GPT) model

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)*.

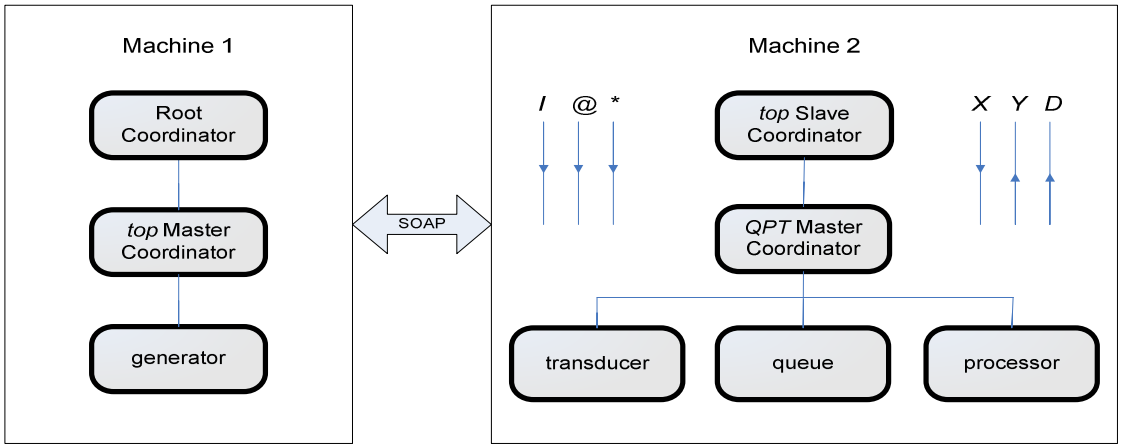

Figure 38: GPT model partitioning on two machines

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.

```
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 39: 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, which are the port name and message value sent through the port. Figure 39 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 39) 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 39) 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 39). 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 39 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 40: An excerpt of the log file of Machine 2

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 40). 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 40); 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 40). The *internal message* sent to the *QPT master coordinator* is forwarded to the *queue* and *transducer* models (lines 8, 9 in Figure 40). 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 40). The *done message* (generated by the *top slave coordinator*) is forwarded to the *top master coordinator* using SOAP (line 14 in Figure 39). 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:

i) There are no more events/messages scheduled by any of the *processors*.
ii) The simulation clock reaches the maximum execution time as provided by the user.

The actions taken by the simulator when receiving an *internal message* depend on the timestamp of the *internal message*, the time of the next internal transition of the model, and the status of the external message bag. If the *internal message* arrives when there are messages in the bag and no *internal transition* is scheduled, the *external transition* function is executed. If the *internal message* arrives when there are no messages in the bag and the internal transition is scheduled to take place, the *internal transition* function is executed. If the *internal message* arrives when there are messages in the bag and the model is scheduled for internal transition; in this case the *confluent transition* function is executed. In the *GPT* example, when the *transducer* and *queue* received the *internal messages* from the *QPT master coordinator* (lines 8, 9 in Figure 40) they were not scheduled for any internal transitions; hence they executed their *external transition* functions as a response to the *internal messages*.

SOAP plays an important role in distributed simulation sessions; it is not only used for sending remote simulation messages between two *processors*. Rather, it is also used for the initialization, and the control of the remote sessions. When a user connects to the simulation service to start a distributed session, he connects to the first machine which is considered as the *master node* throughout the session. Once the model and configuration files are submitted to the service and before actually starting the simulation, it initializes the slave sessions running on the other nodes (those are referred to as *slave nodes*). To do so, the master node uses the services offered by the simulation services running on the slave nodes in order to send the model and configuration files. In the GPT example, this is done by submitting different requests to Machine 2 in order to initialize the distributed simulation session.

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/
                                    2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>

   <ns1:createSlaveSession soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="http://
          www.sce.carleton.ca/ars/CDpp">
    <in0 href="#id0"/>
    <in1 xsi:type="soapenc:string" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">Rami</in1>
    </ns1:createSlaveSession>
    <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                  xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">153999</multiRef>

</soapenv:Body>
</soapenv:Envelope>
```

Figure 41: *createSlaveSession* request

Figure 41 shows a SOAP request for invoking the *createSlaveSession* operation in the service running on Machine 2. This operation initializes a new session on the machine bypassing the process of user authentication, since authentication took place when the user connected to the master node. It takes two arguments: the username of the user who initiated the session and the session id assigned by the master node. The SOAP request consists of an *envelope*, which contains a *body* (and an optional *header*). The envelope defines the different namespaces that are normally used in SOAP messages such as the namespace of the SOAP envelop itself, *XMLSchema*, and *XMLSchema-Instance*. The body contains the arguments of the operation; those include the username *("Rami")* and session id *(153999)*. After finishing the initialization process, which included sending the model definition and *grid configuration* files to Machine 2, the execution of the GPT model was started and SOAP was used to exchange remote messages between the two machines. Figure 42 shows the SOAP message used to send the *initialization message* from the *top master coordinator* (Machine 1) to the *top slave coordinator* (Machine 2). The envelope and body attributes list the namespace definitions that are usually part of SOAP messages. The operation responsible for sending remote messages is *receiveRemoteMessage* running as part of the simulation service in Machine 2; the arguments submitted in the SOAP request include:

Argument	**Description**	***I Message*** **(Figure 42)**
sessionID	Session id	153999
MessageTime	Timestamp of the message	00:00:00:000
MessageType	The type of the message	9 *(I Message)*
NextChange	Time of the next change (used for *done messages*)	Null
SendingProcessor	The id of the sending *processor*	6 (*top* master)
PortId	The id of the port (used for *X* and *Y messages*)	-1
Value	The value of the *X* and *Y messages*	-1
SenderModelId	The id of the original sender of the message	-1
isFromSlave	True if the sender is a *slave coordinator*, false otherwise	False
ReceivingProcessor	The id of the receiving *processor*	7 (*top* slave)

Table 3: Arguments of the *receiveRemoteMessage* operation

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
      www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
   <ns1:receiveRemoteMessage soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
           xmlns:ns1="http://www.sce.carleton.ca/ars/CDpp">
   <in0 href="#id0"/>
   <in1 href="#id1"/>
   <in2 xsi:type="soapenc:string" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/  \
         ">00:00:00:000</in2>
   <in3 href="#id2"/>
   <in4 xsi:type="soapenc:string" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"/>
   <in5 href="#id3"/>
   <in6 href="#id4"/>
   <in7 href="#id5"/>
   <in8 href="#id6"/>
   <in9 href="#id7"/>
   </ns1:receiveRemoteMessage>
   <multiRef id="id0" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
        153999</multiRef>
   <multiRef id="id1" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
         9</multiRef>
   <multiRef id="id2" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
         6</multiRef>
   <multiRef id="id3" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
        encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          -1</multiRef>
   <multiRef id="id4" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
        encoding/" xsi:type="xsd:double" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          -1.0</multiRef>
   <multiRef id="id5" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          -1</multiRef>
   <multiRef id="id6" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
       encoding/" xsi:type="xsd:boolean" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding">
          false</multiRef>
   <multiRef id="id7" soapenc:root="0" soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/
         encoding/" xsi:type="xsd:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
          7</multiRef>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 42: An *initialization message* sent as SOAP from Machine 1 to Machine 2

At the end of the simulation, the master node retrieves all the log files generated by the slave nodes and makes them available for the user to retrieve. The files are sent from the slave nodes to the master node as SOAP attachments. The SOAP message doesn't actually include the file; rather, it includes an id necessary for the receiving service (master node) to retrieve the attachment. This is shown in Figure 43:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://
    www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:retrieveLogFileResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:ns1="http://www.sce.carleton.ca/ars/CDpp">
    <retrieveLogFileReturn href="cid:0251D55ABD69CA54CDFF90768DB0FF79"
        xsi:type="ns2:DataHandler" xmlns:ns2="http://xml.apache.org/xml-soap"/>
  </ns1:retrieveLogFileResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Figure 43: *retrieveLogFile* response

5.5 Integrating Optimistic (PCD++) and Conservative (DCD++) Simulators

Distributed CD++ (DCD++) represents an efficient means of exploiting unused resources (CPU time and memory resources) in order to execute complex models. By exposing the simulator functionality as a service, unused resources could be used in a productive manner. However, the middleware tools used to realize DCD++ have added some overhead in terms of the execution time of the simulation. On the other hand, PCD++ [Gli04] was developed following the optimistic approach using WARPED [War06] as a simulation middleware and MPI [MPI95] as a messaging protocol. PCD++ is able to execute models in shorter execution times due to the algorithms used in the simulation, and the fact that the delay associated with sending MPI messages is mush less than the delay associated with sending SOAP messages. The plus point that DCD++ has over PCD++ is that the connectivity between the machines can be anything ranging from commodity Internet connections, to high-speed point-to-point fibre links. As demonstrated with the *GPT* model, one of the machines used for the tests was located in Ottawa, while the other in Montreal. On the other hand, MPI is usually used for networked workstations within close proximity in terms of the geographic locations. Integrating the two simulators together has an appealing objective of attaining the speedup provided by PCD++ while making efficient use of unused resources through DCD++. In order to do so, two major issues need to be taken care of:

i) A messaging and coordination mechanism needs to be established since the two simulators use totally different middleware and algorithms for their operations.

ii) Synchronization mechanisms need to be in place to ensure the correctness of the simulation.

In the next two sub-sections, we describe a solution for the first issue, followed by a proposed solution for the second issue.

5.5.1 Interfacing DCD++ to PCD++

PCD++ uses WAPRED [War06] as a simulation middleware in order to implement the Time Warp algorithms for parallel simulations. The messages sent by the simulator are encapsulated into WARPED messages that get sent from one simulation object to another. When receiving a WARPED message, the simulation object extracts the information carried by the message as a regular PCD++ message that gets processed by one of the simulators and coordinators in the system. On the other hand, DCD++ uses the original CD++ messages for local communications among the simulation objects (*processors*), and uses SOAP for remote communication. In DCD++, when a *processor* needs to send a message to a remote one, it sends the message to the simulation components of the service which pass the message to the web service components to construct a SOAP message. When received at the destination, the SOAP message is used to construct a CD++ message that gets processed by the receiving *processor*.

In order to Interface DCD++ to PCD++, the simulation services were adapted in order to work with PCD++. As discussed in Chapter 3, the simulation services consist of two major components: the *web service components* responsible for the web service functionality, which are developed in Java (except the native methods developed in C/C++); and the *simulation components*, which are responsible for running the simulation and interacting with the web service components. The simulation components were modified and integrated with the code of PCD++, without any major change on the web

service side. The modular approach for developing the simulation services was flexible enough to be used with PCD++ while maintaining the redesign and reimplementation time to minimum.

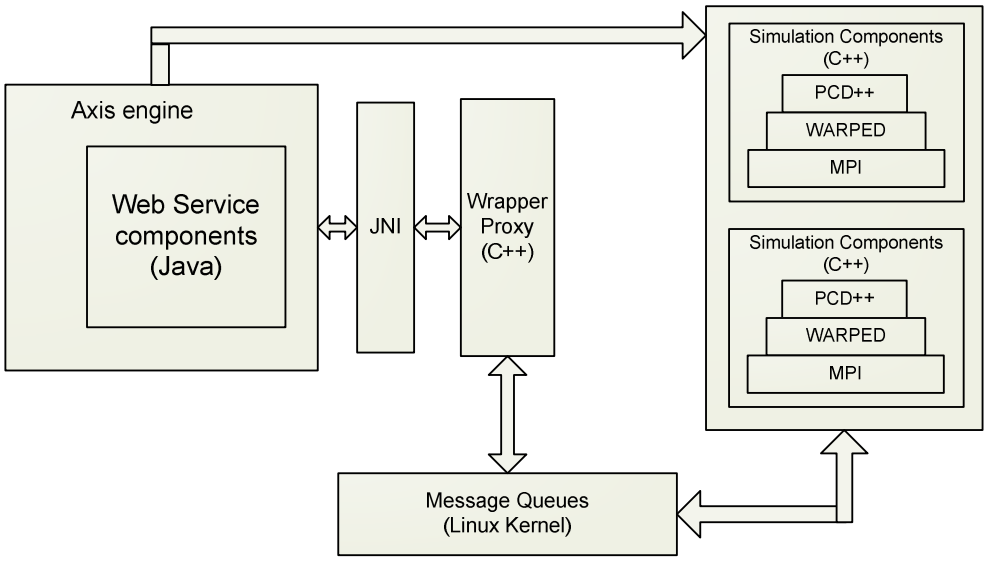

Figure 44: Implementing the simulation web service with PCD++

After implementing the simulation services using PCD++, the client is able to run parallel simulations remotely as a web service. PCD++ runs on a high performance distributed-memory cluster consisting of 32 Linux machines. The services available for the client are identical to the ones offered by DCD++ (discussed in Chapter 3) with two exceptions:

i) The *createSlaveSession* operation is not available. That is, PCD++ does not function as slave node(s) even in the proposed architecture for integration with DCD++ (discussed in the following sub-section);

ii) New operation is implemented (*setPartitionFile*) in order to allow the user to set the partition file of the simulation. The partition file is a text file that defines the model partition on the cluster nodes.

5.5.2 Integrating DCD++ and PCD++

The proposed architecture for integrating DCD++ and PCD++ depends on integrating the *CPPWrapper* class (part of the simulation components developed in C++) within the simulation hierarchy of PCD++. *CPPWrapper* will function as an interface between DCD++ and PCD++ in order to hide the details and complexities of DCD++ from PCD++, and vice versa. PCD++ was developed as a flat simulator [Gli04]; the simulation hierarchy does not match the model hierarchy in terms of having a coordinator for each coupled DEVS/Cell-DEVS model running on the machine. Each node has two main coordinators and one simulator for each local atomic DEVS/Cell-DEVS model. The *node coordinator* is responsible for interacting with the other nodes in the simulation and with the environment. In addition, it is in charge of advancing the simulation clock independently (*optimistically*) from the other nodes. The *flat coordinator* lies under the *node coordinator* in the simulator hierarchy and is responsible for interacting with the simulators and forwarding messages upward and downward the simulation hierarchy depending on the type and destination of the message. The simulators are responsible for executing the atomic DEVS and Cell-DEVS models in the simulation.

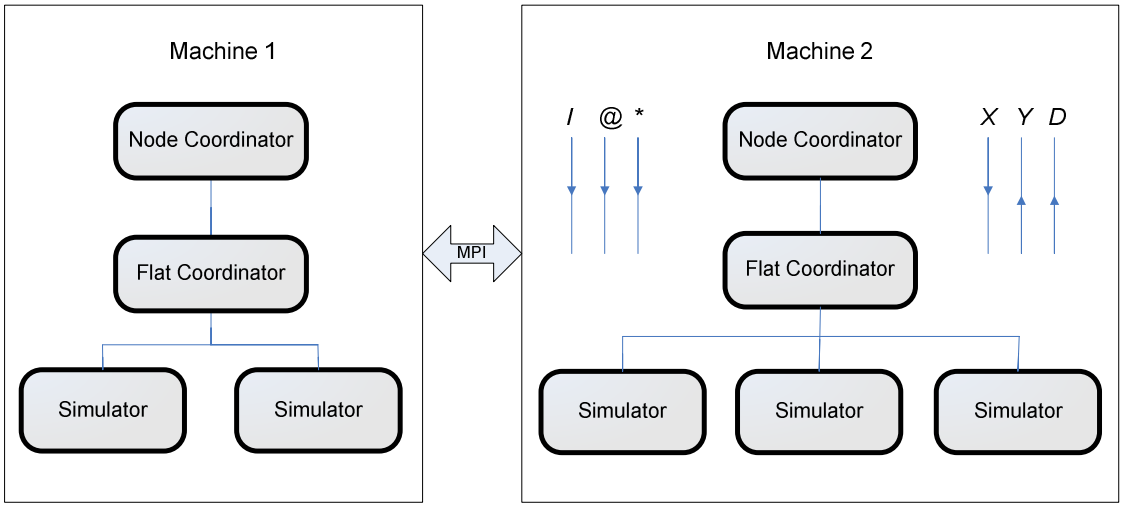

Figure 45: PCD++ architecture

Both DCD++ and PCD++ implement the P-DEVS algorithms for the model execution. However, PCD++ uses *anti messages* during the rollback phase of the simulation if a *node coordinator* receives a message with timestamp earlier than the local clock of the machine (*straggler message*). On the other hand, DCD++ neither uses nor can handle *anti messages*. So, the main issue in integrating the two is concerned with properly synchronizing the *optimistic* behaviour of PCD++ with the *conservative* behaviour of DCD++. The idea presented here depends on distinguishing between two kinds of information in the optimistic simulation: *conditional knowledge/information* and *unconditional knowledge/information*. Conditional knowledge is the simulation transactions that took place after the Global Virtual Time (GVT); since those transactions could be rolled back if the node in which they are running has received a straggler message. Unconditional knowledge, on the other hand, represent all the transactions that were completed with timestamps less than or equal to the current GVT value since those won't be rolled back during the simulation.

The idea depends on integrating the *CPPWrapper* as one of the simulation objects under the *flat coordinator* of *Node 0* (the first node in the cluster running PCD++), and changing the behaviour of the *flat coordinator* accordingly to forward any message destined to remote simulation objects (those assigned to run within DCD++) to the *CPPWrapper*, which in turn does one of the following:

i) If the message timestamp is larger than the GVT value, the message is inserted in a queue maintained by the *CPPWrapper* class to be sent out when the GVT is re-evaluated and reaches the timestamp of the message.

ii) If the message timestamp is equal to the GVT value, the message is forwarded to the remote simulation object (running within DCD++) to be processed as if it was running on *Node 0*.

The implementation of the previous mechanism requires changes to be made to the mechanisms used for loading the model in both simulators. That is, it is important that both use identical ids for the same models in order to handle message routing among the

models properly. Otherwise, some mapping/translation would be required by the *CPPWrapper* in order to ensure correct message passing between the two simulators. Since the *CPPWrapper* is integrated under the *flat coordinator* of *Node 0*, the *node coordinator* won't be able to advance the clock on *Node 0* in the same pace as the other nodes of the cluster running PCD++. This is due to the processing taking place in DCD++ and the delay associated with sending SOAP messages. In other words, *Node 0* has to "wait" for DCD++ to finish processing the messages that were sent to it. This has an effect of slowing the overall time for executing the model compared with the case of running PCD++ alone. However, the parallelism available on the other nodes (other than *Node 0*) can be exploited to achieve the speedup provided by the optimistic algorithms of PCD++. As a result, the performance of the two simulators working together is expected to be worse than running PCD++ alone but better than running DCD++ alone.

Chapter 6: Performance Analysis

The web service capabilities introduced to CD++ have extended its functionality in two aspects. In one aspect, it enabled the simulator to be invoked remotely and interfaced with other larger systems using web service standards. In another aspect, it allowed the simulator to run complex models in distributed environments using SOAP as a messaging protocol. However, the extended functionality has introduced some overhead when running distributed simulation. 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 components through the Linux kernel; the other part is the time it takes to transmit the SOAP message between the two simulation web services.

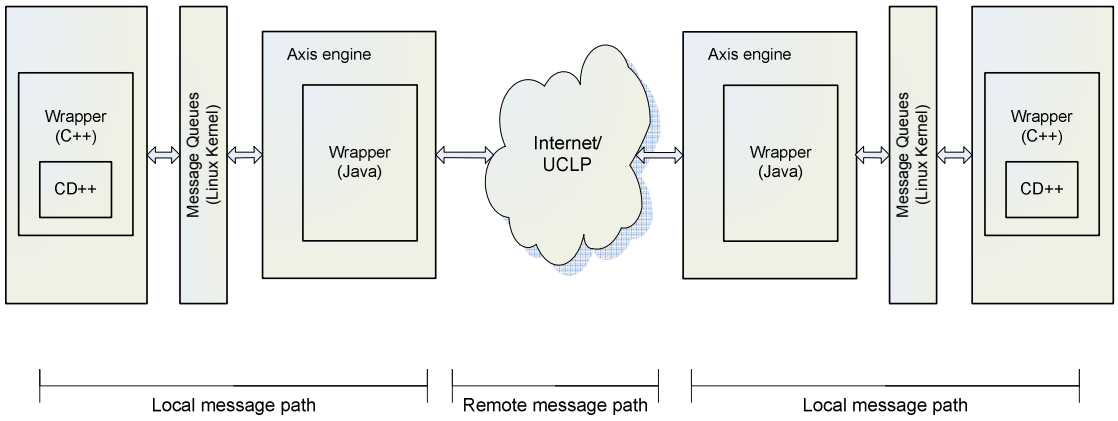

Figure 46: Sending remote messages in distributed simulation

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. 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.

In addition, the average time it takes to retrieve the results of the simulation was measured using files of different sizes.

6.1 Experimental Models and Execution Results

Two types of models were used during the performance analysis. One of the models is fire spread in a forest and it is implemented as 30x30 coupled Cell-DEVS model [Ame01]. The other model is a sand-pile model [Saa03], which consists of DEVS and Cell-DEVS models. The DEVS model is a sand particle generator connected to a coupled Cell-DEVS model representing the formation of a sand-pile.

The fire model 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 47 shows an excerpt of the model definition with possible initial values of the cells.

```
[top]
components : Fire

[Fire]
type : cell
dim : (30,30)
delay : inertial
defaultDelayTime  : 0
border : nowrapped
neighbors : Fire(-1,-1) Fire(-1,0) Fire(-1,1)
neighbors : Fire(0,-1)  Fire(0,0)  Fire(0,1)
neighbors : Fire(1,-1)  Fire(1,0)  Fire(1,1)
initialvalue : 0
initialCellsValue : Fire.val
localtransition : FireBehavior
```

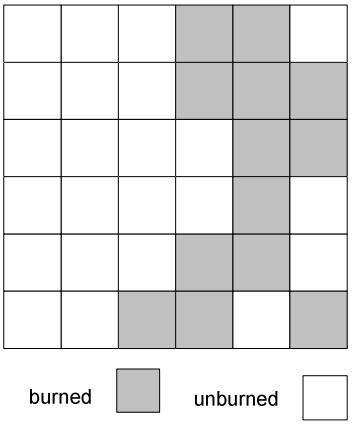

Figure 47: An excerpt of the Fire model definition

The cell space is 30x30 using *inertial* delay. The neighbourhood of the cell is defined by the *neighbors* construct, the cell is neighboured by 8 cells from all sides. *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; those rules are shown in Figure 48:

```
[FireBehavior]
rule : {  (1,-1)+(21.552615/17.967136)   } {       (21.552615/17.967136)*60000  }
                  {       (0,0)=0 and (1,-1)!=? and 0<(1,-1)       }
rule : {  (1,0)+(15.24/5.106976)         } {       (15.24/5.106976)*60000       }
                  {       (0,0)=0 and (1,0)!=? and 0<(1,0)         }
rule : {  (0,-1)+(15.24/5.106976)        } {       (15.24/5.106976)*60000       }
                  {       (0,0)=0 and (0,-1)!=? and 0<(0,-1)       }
rule : {  (-1,-1)+(21.552615/1.872060)   } {       (21.552615/1.872060)*60000   }
                  {       (0,0)=0 and (-1,-1)!=? and 0<(-1,-1)     }
rule : {  (1,1)+(21.552615/1.872060)     } {       (21.552615/1.872060)*60000   }
                  {       (0,0)=0 and (1,1)!=? and 0<(1,1)         }
rule : {  (-1,0)+(15.24/1.146091)        } {       (15.24/1.146091)*60000       }
                  {       (0,0)=0 and (-1,0)!=? and 0<(-1,0)       }
rule : {  (0,1)+(15.24/1.146091)         } {       (15.24/1.146091)*60000       }
                  {       (0,0)=0 and (0,1)!=? and 0<(0,1)         }
rule : {  (-1,1)+(21.552615/0.987474)    } {       (21.552615/0.987474)*60000   }
                  {       (0,0)=0 and (-1,1)!=? and 0<(-1,1)       }
rule : {                  (0,0)          } 0 {                     t            }
```

Figure 48: Fire model rule definition

The rules define the time it takes for the cell to be burned if one of its neighbours is burned. For example, the first rule dictates that if the cell in the south west side of the cell is burned (0 < (1,-1)), the cell will take (21.552615/17.967136)*60000) milliseconds to

be burned. The value of (21.552615) represents the diagonal distance of each cell (measured in meters), and the value of (17.967136) is the speed of the fire spread (measured in meters/minute) as presented in the model definition [Ame01]. By dividing the distance that the fire has to spread through by the speed of the fire spread, the time it takes for fire spread is evaluated in minutes and by multiplying it with 60,000 the time in milliseconds is obtained as the delay of the cell. If the condition in the first rule holds, the cell state is updated to the value of Fire(1,-1) + (21.552615/17.967136) when the delay elapses.

In order to study the performance of the distributed simulator, three types of experiments were performed using two identical machines (each with dual PIV 3.2 GHz processors, and 512 MB of RAM). 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 P2P fibre optic link created using UCLP, as we discuss following. In order to measure the required metrics, different pieces of code have been inserted in the simulation service at different stages to record the current time, and by comparing the times at these stages, an accurate measure of the duration of each stage could be obtained. The function used to record the time was the C++ *gettimeofday*, which returns the time since midnight January 1, 1970 in seconds with a precision of microseconds. In order to evaluate the confidence interval, the approach presented in [Ban01] was followed; a confidence interval of 100(1-α) % can be calculated as follows:

$\bar{\theta} - t_{\alpha/2,f}\,\bar{\sigma}(\bar{\theta}) \le \theta \le \bar{\theta} + t_{\alpha/2,f}\,\bar{\sigma}(\bar{\theta})$;

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

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

f = R -1 is the degrees of freedom, R is the number of replications, α is the confidence coefficient;

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

Table 4: Execution results of the Fire model using one machine

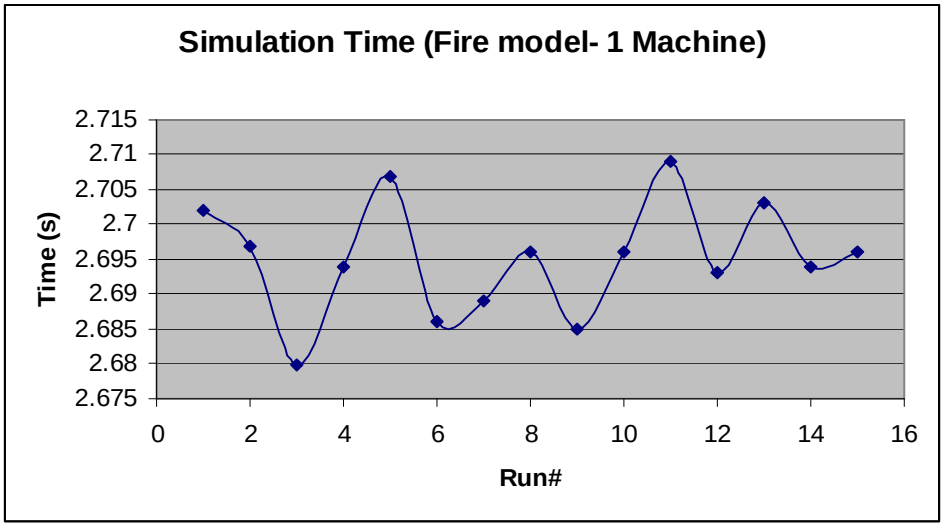

Figure 49: Fire model simulation time using one machine

The Local Message time is the time requited to transmit a message from one *processor* (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.

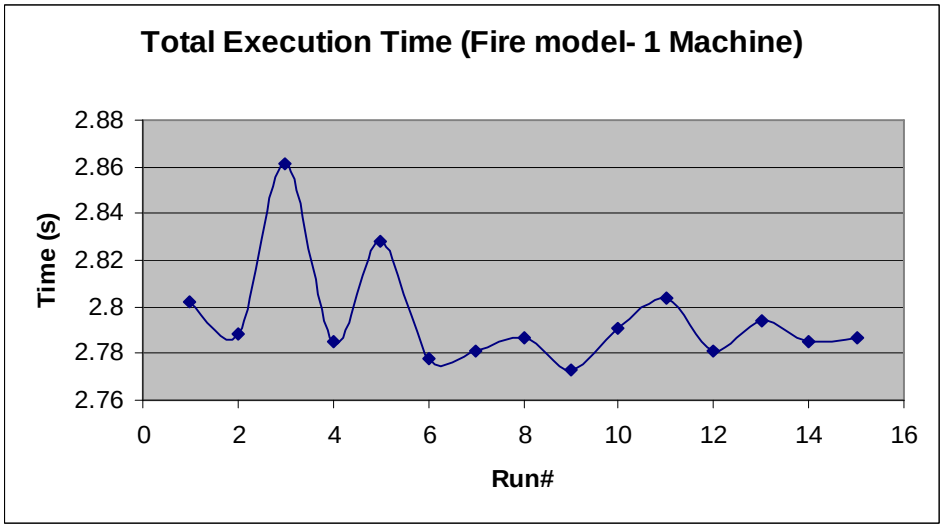

Figure 50: Fire model total execution time using one machine

Although the graphs in Figures 49 and 50 show variations in the simulation and total execution times 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). 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, as shown in Figure 51.

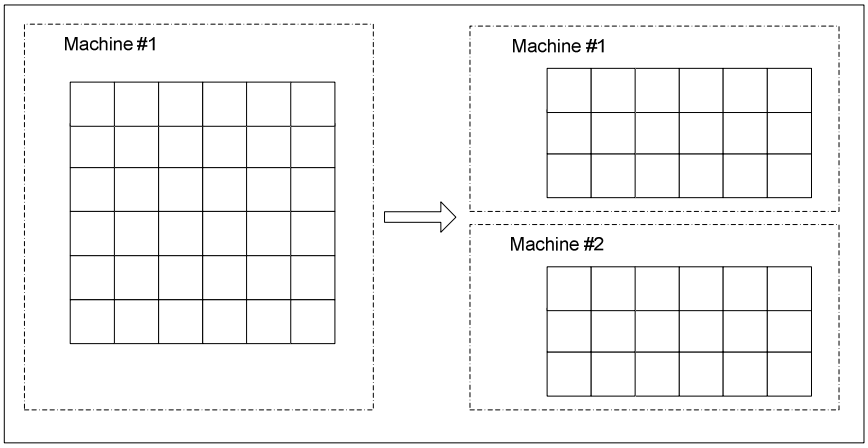

Figure 51: Fire model partitions on two machines

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:

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

Table 5: Execution results of the Fire model using two machines (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 simulation time when using one and two machines, the overhead introduced by the distributed simulator can be visualized:

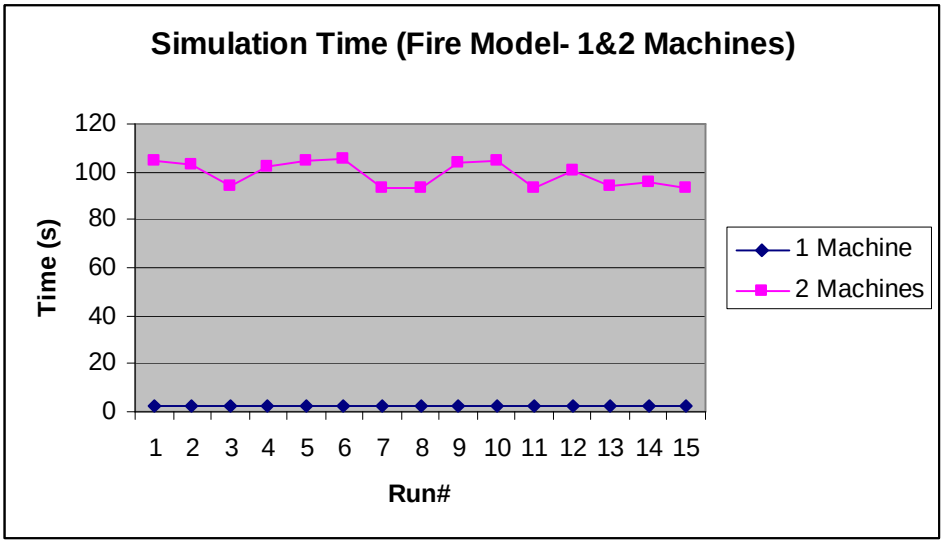

Figure 52: Comparing the simulation time using 1&2 machines (Internet)

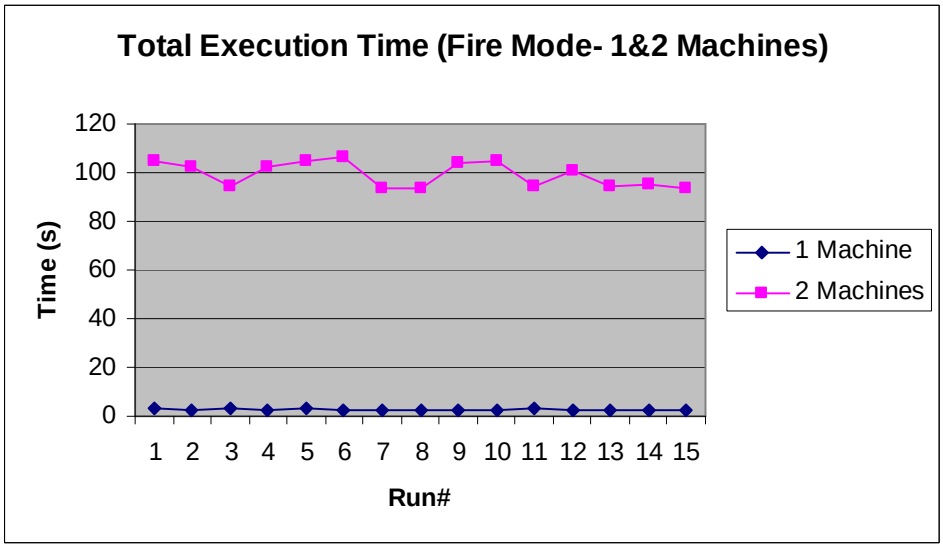

Figure 53: Comparing the total execution time using 1&2 machines (Internet)

Comparing Figures 52 and 53, shows that the simulation and total execution times of the model are almost identical. The difference between the two is the time necessary to initialize the model which is insignificant compared to the time required to execute the model (average of 315 milliseconds compared to an average of 99.292 seconds). It is worth mentioning that the initialization time is measured for Machine 1 since the model in Machine 2 is loaded before starting the simulation in Machine 1. In addition, the time

for loading the model in Machine 1 is very close to that in Machine 2 due to the symmetric partitioning of the model.

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.

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

Table 6: Execution results of the Fire model using two machines (UCLP)

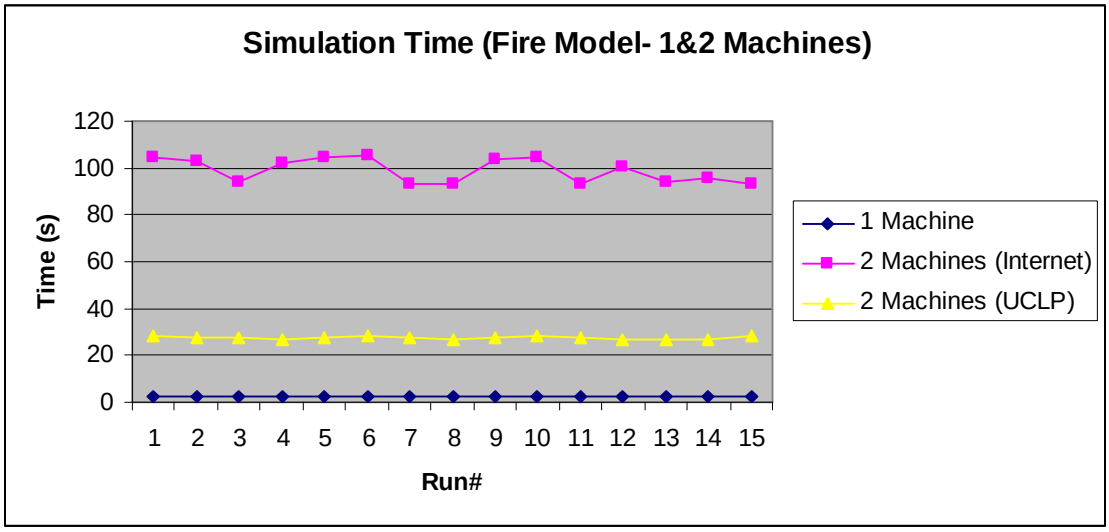

Figure 54: Comparing the simulation time using 1&2 machines (Internet, UCLP)

By examining the simulation 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 simulation time when using UCLP is less than that when using a regular Internet connection.

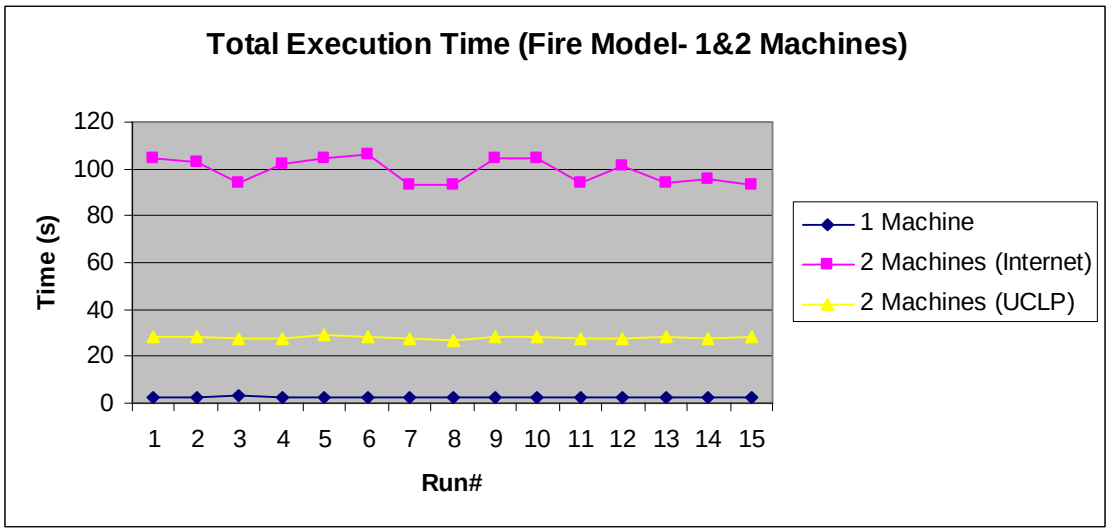

Figure 55: Comparing the total execution time using 1&2 machines (Internet, UCLP)

Examining the total execution time of the simulation in Figure 55 shows the same behaviour as in Figure 54. That is, the initialization time is insignificant compared to the time required to execute the model.

The sand-pile model [Saa03] consists of a DEVS model representing a sand particle generator and a coupled Cell-DEVS model that simulates the sand-pile formation. The output of the generator is connected to the input of the coupled Cell-DEVS model, which in turn is connected to the input of one of cells (*sandpile(5, 5)*). An excerpt of the definition of the sand-pile model is shown in Figure 56.

```
[top]
components : sandpile particleGenerator@Generator
link : out@particleGenerator in@sandpile
out :  out
link : out@particleGenerator out

[sandpile]
type : cell
dim : (10, 10)
delay : inertial
defaultDelayTime  : 100
border : nowrapped
neighbors : sandpile(0,1)   sandpile(1,0)  sandpile(0,-1)
neighbors : sandpile(-1,0)  sandpile(0,0)
in : in
link : in in@sandpile(5,5)
initialvalue : 0
localtransition : sandpile-rule
portInTransition : in@sandpile(5,5) NewParticle-rule
```

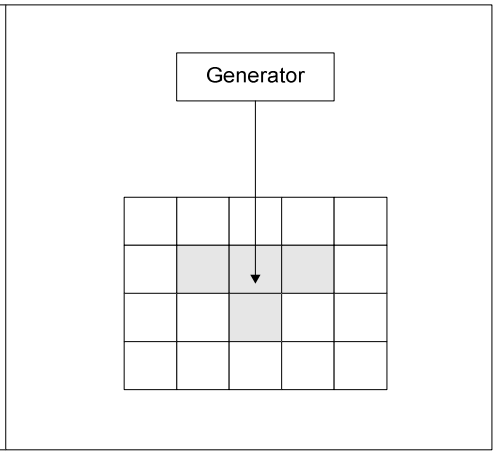

Figure 56: An excerpt of the Sand-pile model definition

The sand-pile model was first executed using a single machine:

	Average	Std. Deviation	Confidence Interval 95%
Local Msg. (us)	3.764	0.253230556	$3.624 \leq X \leq 3.904$
Init. Time (ms)	25.925	3.168856641	$24.174 \leq X \leq 27.676$
Simulation Time (s)	0.1091	0.000589388	$0.1087 \leq X \leq 0.1094$
Total exec. Time (s)	0.135	0.003209	$0.1332 \leq X \leq 0.1368$

Table 7: Execution results of the Sand-pile model using one machine

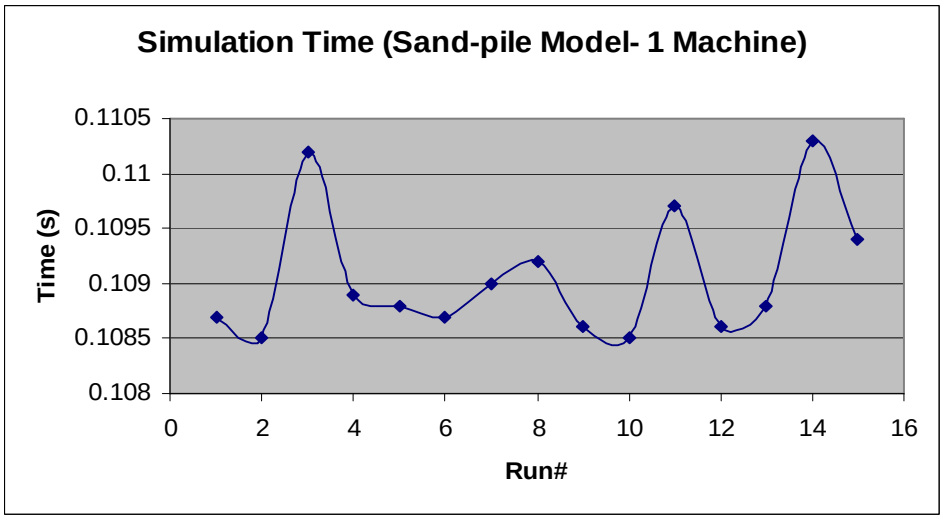

Figure 57: Simulation time of the Sand-pile model using one machine

The initialization time was less than that for the fire model due to the smaller cell space used, which resulted in smaller number of models to be initialized. However, the time required to load the models seems to be significant compared to the simulation time (the average initialization time is 25.925 milliseconds, and the average simulation time is 109.1 milliseconds), which resulted in a longer execution time as in Figure 58. On the other hand, the variations in the simulation and execution times are insignificant (the standard deviation of execution time is 0.003209 seconds with an average of .135 seconds) and are due to the different processes and daemons running on the machine.

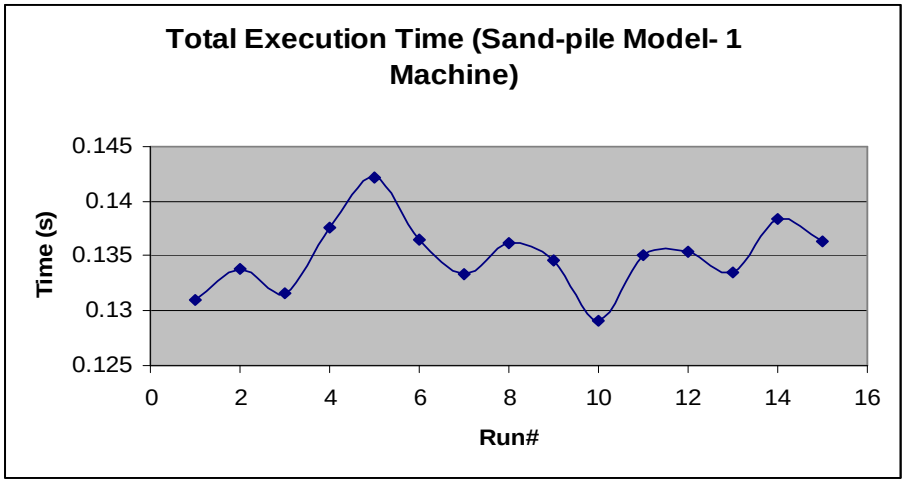

Figure 58: Total execution time of the Sand-pile model using one machine

When running the distributed simulation, the model was split into two parts. The first part contained the sand particle generator (DEVS) and the second included the sand-pile formation model (Cell-DEVS). Each part was assigned to run on one machine and the two machines were connected using a commodity Internet connection.

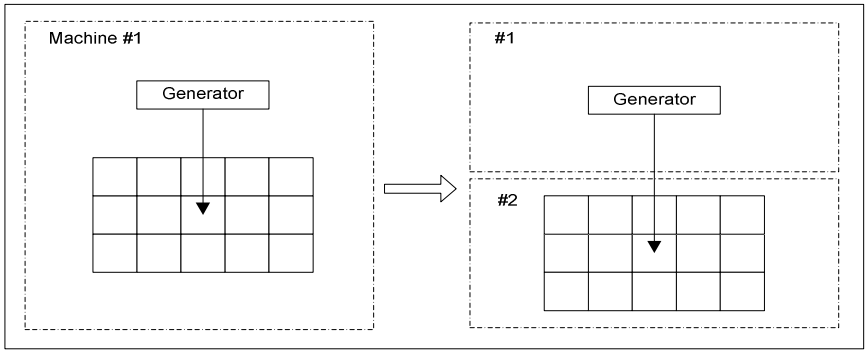

Figure 59: Sand-pile model partitions on two machines

	Average	Std. Deviation	Confidence Interval (95%)
Local Msg. (us)	4.429	0.355597418	$4.233 \leq X \leq 4.626$
Kernel Msg. (ms)	0.494	0.059172226	$0.461 \leq X \leq 0.527$
SOAP Msg. (ms)	846.544	195.5588008	$738.489 \leq X \leq 954.600$
Init. Time (ms)	46.597	31.54870575	$29.165 \leq X \leq 64.029$
Simulation Time (s)	50.439	0.905780553	$49.939 \leq X \leq 50.939$
Total exec. Time (s)	50.485	0.922040301	$49.976 \leq X \leq 50.995$
Bandwidth (KB/s)	810.947	29.5132616	$794.639 \leq X \leq 827.254$

Table 8: Execution results of the Sand-pile model using two machines (Internet)

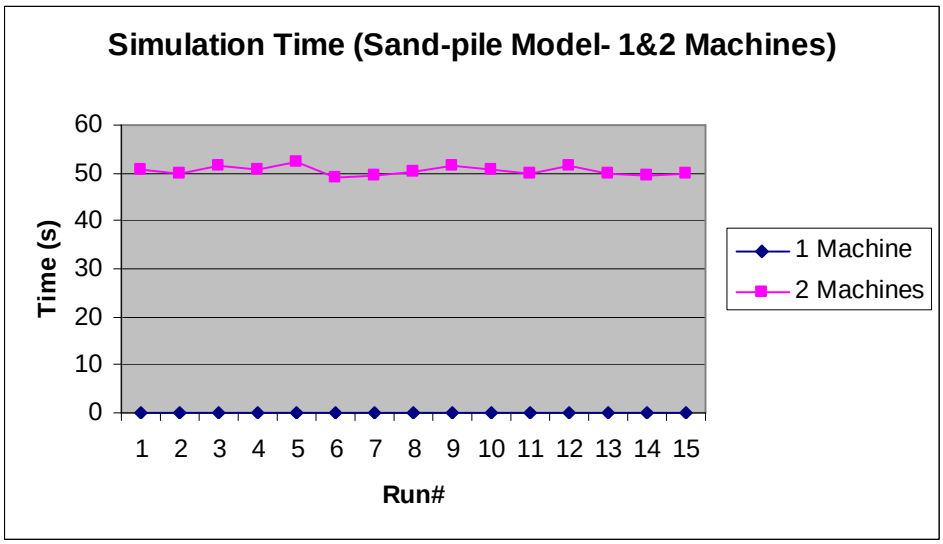

Figure 60: Comparing the simulation time of the Sand-pile model using 1&2 machines (Internet)

The results obtained are consistent with the ones obtained when running the fire model. The initialization time is longer when running distributed simulation since more *processors* need to be initialized. The simulation time is longer than that for a single machine due to the delay caused by sending SOAP messages between the remote *processors*.

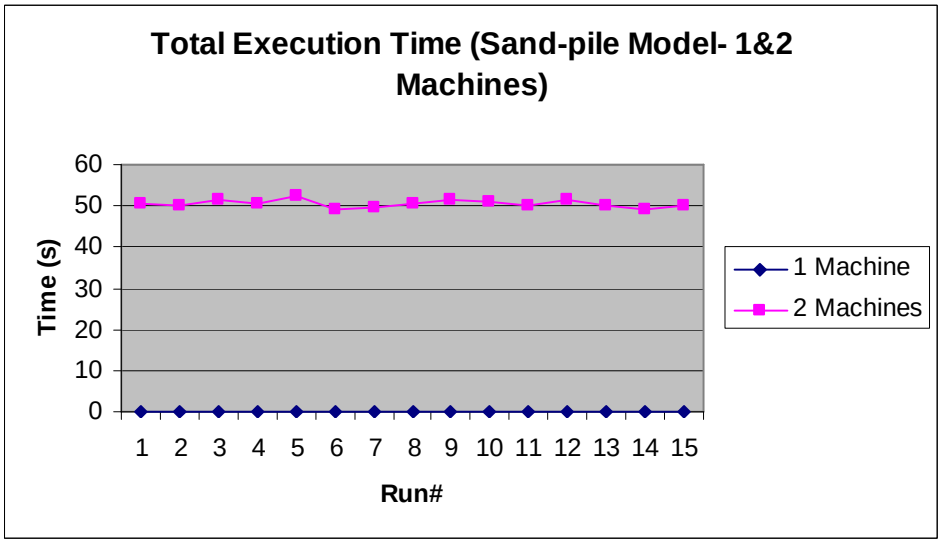

Figure 61: Comparing the total execution time of the Sand-pile model using 1&2 machines (Internet)

The behaviour of the execution time is almost identical to the behaviour of the simulation time due to the insignificance of the initialization time (the average initialization time is 46.597 milliseconds, and the average simulation time is 50.439 seconds). The following table shows the execution results when connecting the machines using UCLP:

	Average	Std. Deviation	Confidence Interval (95%)
Local Msg. (us)	4.413	0.088694231	$4.364 \leq X \leq 4.462$
Kernel Msg. (ms)	0.414	0.048226722	$0.387 \leq X \leq 0.440$
SOAP Msg. (ms)	483.525	133.2349746	$409.907 \leq X \leq 557.143$
Init. Time (ms)	19.259	1.708194042	$18.315 \leq X \leq 20.203$
Simulation Time (s)	8.117	0.081470209	$8.0719 \leq X \leq 8.1619$
Total Exec. Time (s)	8.136	0.081264089	$8.091 \leq X \leq 8.181$

Table 9: Execution results of the Sand-pile model using two machines (UCLP)

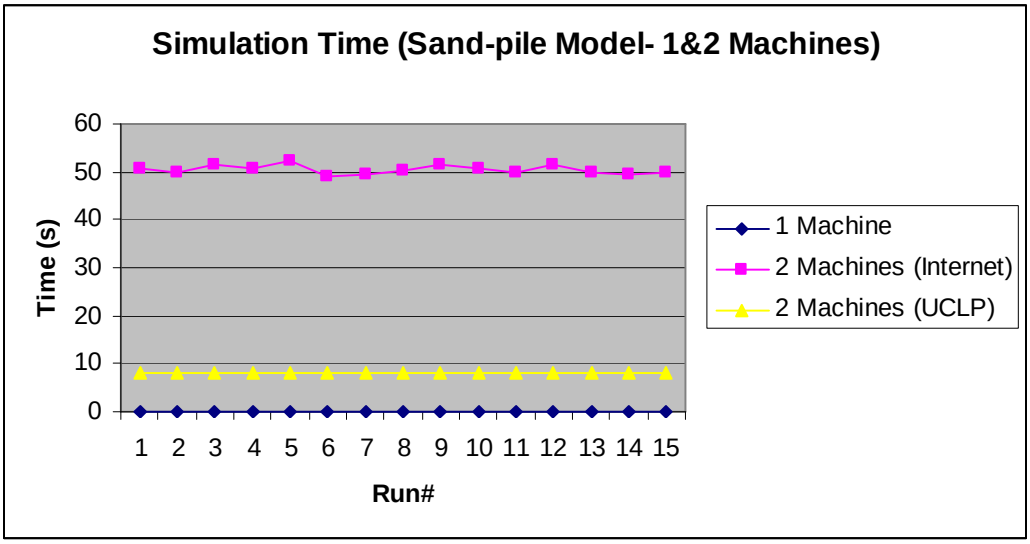

Figure 62: Comparing the simulation time of the Sand-pile model using 1&2 machines (Internet, UCLP)

When using a dedicated link between the two machines, the simulation time improved from an average of 50.439 to an average of 8.117 seconds. In addition, the variation in simulation time when using UCLP is less than that when using a commodity Internet connection.

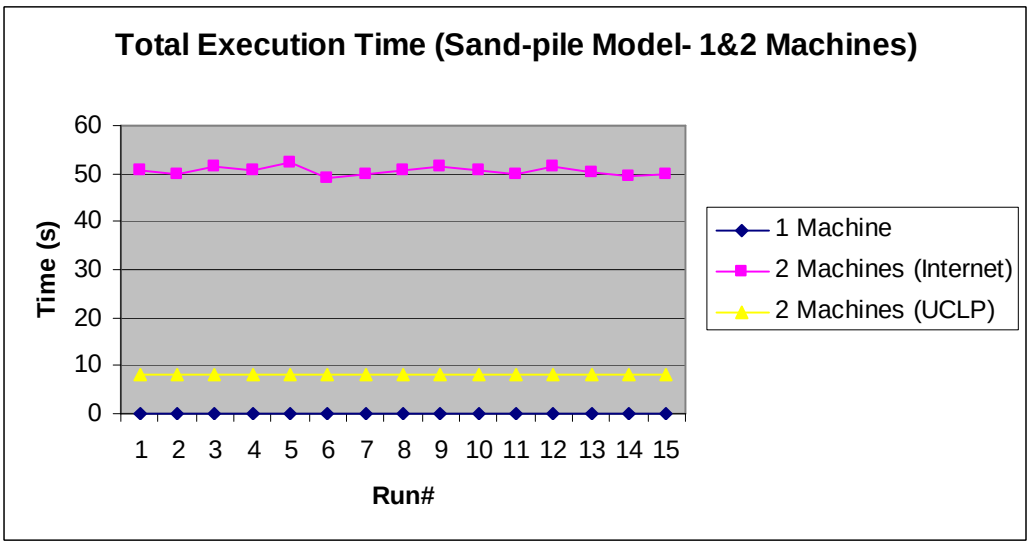

Figure 63: Comparing the total execution time of the Sand-pile model using 1&2 machines (Internet, UCLP)

As in the case of the fire model, the execution time follows the behaviour of the simulation time since the initialization time is much less than the simulation/execution time. Table 10 shows a summary of the three experiments performed on each one of the models (the Fire and Sand-pile models):

	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)	NA	892.631	489.343	NA	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 10: Summary of the execution results of the Fire and Sand-pile models

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). This was not the case for the sand-pile model (using UCLP) due to the fact

that the initialization time was measured for Machine 1 which only had one of the model components running (the *generator* model) as shown in Figure 59.

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.

	Remote Msgs. (%)	**Sim_Time2(Int.)/ 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 11: Percentage of remote messages in distributed simulation

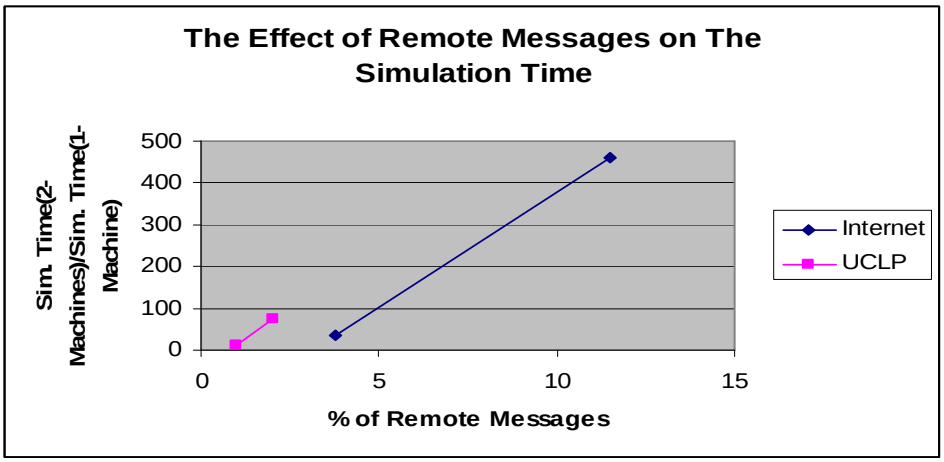

Figure 64: Relationship between remote messages and simulation times

Figure 64 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.

6.2 Result Retrieval

In addition to measuring the performance of the simulator, different experiments were performed in order to assess the performance of result retrieval when using UCLP compared to when using a commodity Internet connection. Three log files generated by the Fire model were used in the experiments. The sizes of the files were (file1 ~ 1MB, file2 ~2.5 MB, file3 ~5MB).

	Average	Std. Deviation	Confidence Interval (95%)
File1 (Internet) (s)	8.117	0.232105244	$2.316 \leq X \leq 13.918$
File1(UCLP) (s)	0.066	0.001081409	$0.0188 \leq X \leq 0.1129$
File2(Internet) (s)	20.878	0.698528493	$5.957 \leq X \leq 35.798$
File2(UCLP) (s)	0.129	0.003675746	$0.0369 \leq X \leq 0.2215$
File3(Internet) (s)	36.070	1.008546831	$10.292 \leq X \leq 61.849$
File3(UCLP) (s)	0.235	0.024736163	$0.0671 \leq X \leq 0.4031$

Table 12: File transfer times via the Internet/UCLP

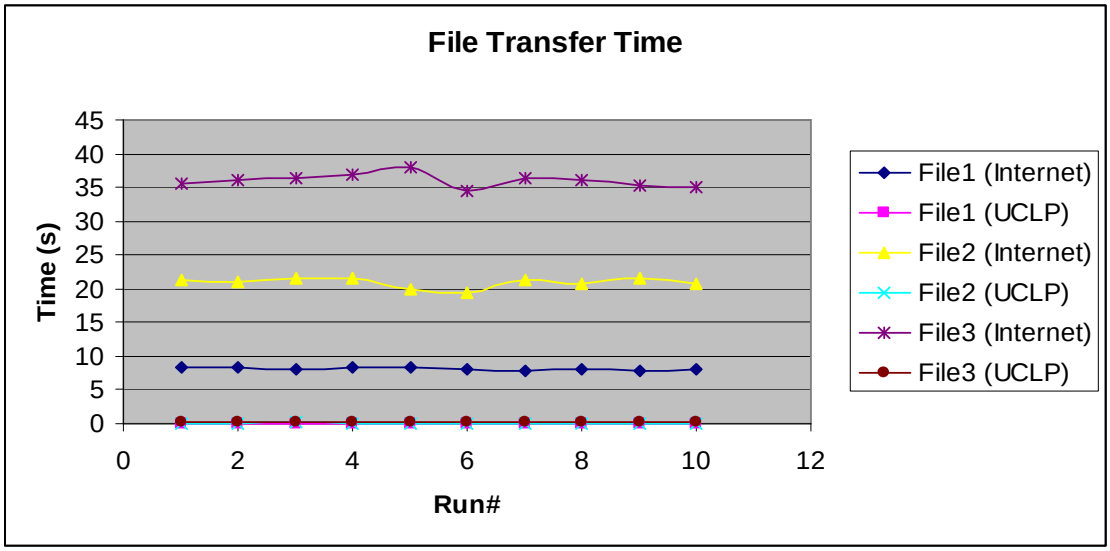

Figure 65: Comparing the file transfer times via the Internet/UCLP

Figure 65 shows a big difference between the times needed to retrieve the results when using UCLP and those needed when using commodity Internet connections. This is due to the larger bandwidth provided by UCLP (average of 241.13 M Bits/second) compared to that provided by regular Internet connections (less than 1M Bit/second).

Chapter 7: 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.

7.1 Future Research Work

Implementing DCD++ using web services has answered a lot of questions about the feasibility, advantages, and disadvantages of the approach presented in this dissertation. However, it kept a lot to be investigated in future research work and left some room for improvement of the features already implemented in DCD++:

One of the main advantages of using web services is its ability to be interfaced and integrated with other systems. The simulation services developed could be integrated with larger systems such as the Participatory Design Studio (PDS) [San06]. PDS is a

project aiming at building collaborative environment of different resources using web services. The resources available include visualization services; image capture devices such as cameras, camcorders, and network management services such as User Controlled Light Path (UCLP). The project is to provide an environment for architecture engineers to facilitate the process of designing buildings in a collaborative manner. Business Process Execution Language (discussed in chapter 2) can be used to establish a workflow between the simulation and visualization services. This allows the engineers to simulate different incidents in the buildings they design and to visualize the results of the simulation in real time.

The user authentication process in the simulation service is based on a password file stored on the server; this is done when the user first connects to the service. When establishing slave sessions, no authentication is performed since the service nodes are considered to trust each other. This can be improved by having the authentication process based on digital certificates. In order for the user to connect to the service, the user would need to have a trusted digital certificate; in addition, each node would have its certificate in order to be used when establishing slave sessions on the slave nodes.

The success of a distributed simulation session depends on the network connectivity among the nodes; if any network failure happens during the simulation, the slave sessions may end up running and consuming resources without doing any useful processing. This can be avoided by implementing some synchronization mechanism among the nodes in order to detect any network problems and kill the session (and reclaim its resources) accordingly after raising the proper exception to the user.

Integrating DCD++ and PCD++ into one framework has an appealing objective of taking advantage of both engines, the speedup offered by PCD++ and the web service capabilities offered by DCD++. In addition, this would provide proof of concept of the approach presented in chapter 5 for integrating optimistic and conservative simulations together.

References

[Ahm05] Ahmed, M.; Yonis, K.; Elshafei, M.; Wainer, G. “Building a tool for modeling and simulation of computer networks”. Proceedings of the 38th IEEE/SCS Annual Simulation Symposium. San Diego, CA. U.S.A. 2005.

[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.

[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].

[Bry77] Bryant, R.E. Simulation of Packet Communication Architecture Computer Systems. Massachusetts Institute of Technology, Cambridge, MA. USA. 1977.

[Cha79] Chandy, K.; Misra, J. “Distributed Simulation: A Case Study in Design and Verification of Distributed-Programs”. *IEEE Transactions on Software Engineering*, pp. 440-452. 1979.

[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].

[Cla99] Clark, J.; DeRose, S. “XML Path Language (XPath),Version 1.0”. November, 1999. Available via <http://www.w3.org/TR/xpath>. [Accessed February, 2005].

[Cle04] Clement, L.; Hately, A.; Riegen, C.; Rogers, T. “UDDI Version 3.0.2, UDDI Spec Technical Committee Draft”. October, 2004. Available via < http://uddi.org/pubs/uddi_v3.htm>. [Accessed March, 2006].

[Erl05] Erl, T. *Service-Oriented Architecture, Concepts, Technology, and Design*. Pearson Education, Inc. 2005.

[Fal04] Fallside, D.; Walmsley, P. “XML Schema Part 0: Primer Second Edition”. October, 2004. Available via <http://www.w3.org/XML/Schema>. [Accessed November, 2004].

[Fer03] Ferreira, L.; Bursitis, V.; Armstrong, J.; Kendzierski, M.; Neukoetter, A.; Masanobu T.; Bing-Wo, R.; Amir, A.; Murakawa, R.; Hernandez, O.; Magowan, J.; Bieberstein, N. ”Introduction to Grid Computing with Globus”. Available via <http://www.redbooks.ibm.com/redbooks/SG246778/wwhelp/wwhimpl/java/html/wwhelp.htm>. [Accessed January, 2006].

[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

[Mad05] Madhoun, R.; Wainer, G. “Modeling battlefield scenarios in Cell-DEVS”. Proceedings of SISO Fall Interoperability Workshop. San Diego, CA. U.S.A. 2005.

[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].

[Sei04] Seidner, R. “A BPEL Primer”. July ,2004. Available via <http://www.webservicespipeline.com/trends/23902103>. [Accessed March, 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.]

[Web06] WebSphere. Available via <http://www-128.ibm.com/developerworks/websphere/newto/>. [Accessed February, 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.

Appendix-A: P-DEVS and DCD++ Simulation Algorithms

In this appendix, the algorithms governing the behaviour of the simulators and coordinators in CD++ and DCD++ are presented. For the following discussion, T_L represents the time of the last state change of the model, T_N represents the time of the next state change, *s* is the model state, *e* is the time since the last state transition, and *processor* refers to a simulation processor (not physical processor).

When a simulator receives an *external message (X)* at time t, it simply adds it to the external message bag to be processed when the next *internal message (*)* is received. Figure 66 shows the behaviour of the simulator when receiving a *collect message (@)* at time *t*, it executes the *output* function and returns a *done message (D)* indicating the time of the next change:

```
Simulator ( @, t )

if t = T_N
{
        T_L = t ;
        y := λ( s ) ;
        send ( y, t) to parent coordinator;
        send done (0) to parent coordinator;
}
else
{
        raise exception;
}
```

Figure 66: Simulator's reaction to a *collect message (@)*

When a simulator receives an *internal message* at time *t*, depending on the message time and the message bag status, one of the transition functions ($\delta_{ext,}$ δ_{int}, and δ_{conf}) is triggered as shown in Figure 67. If the message arrives when there is no internal transition scheduled, and the message bag is not empty, the *external transition* function is executed. If the message arrives at the time of an internal transition and the message bag is empty, the *internal transition* function is executed. The third case is when the internal message arrives at the time of an internal transition and the message bag is not empty; in this case, the *confluent transition* function is executed:

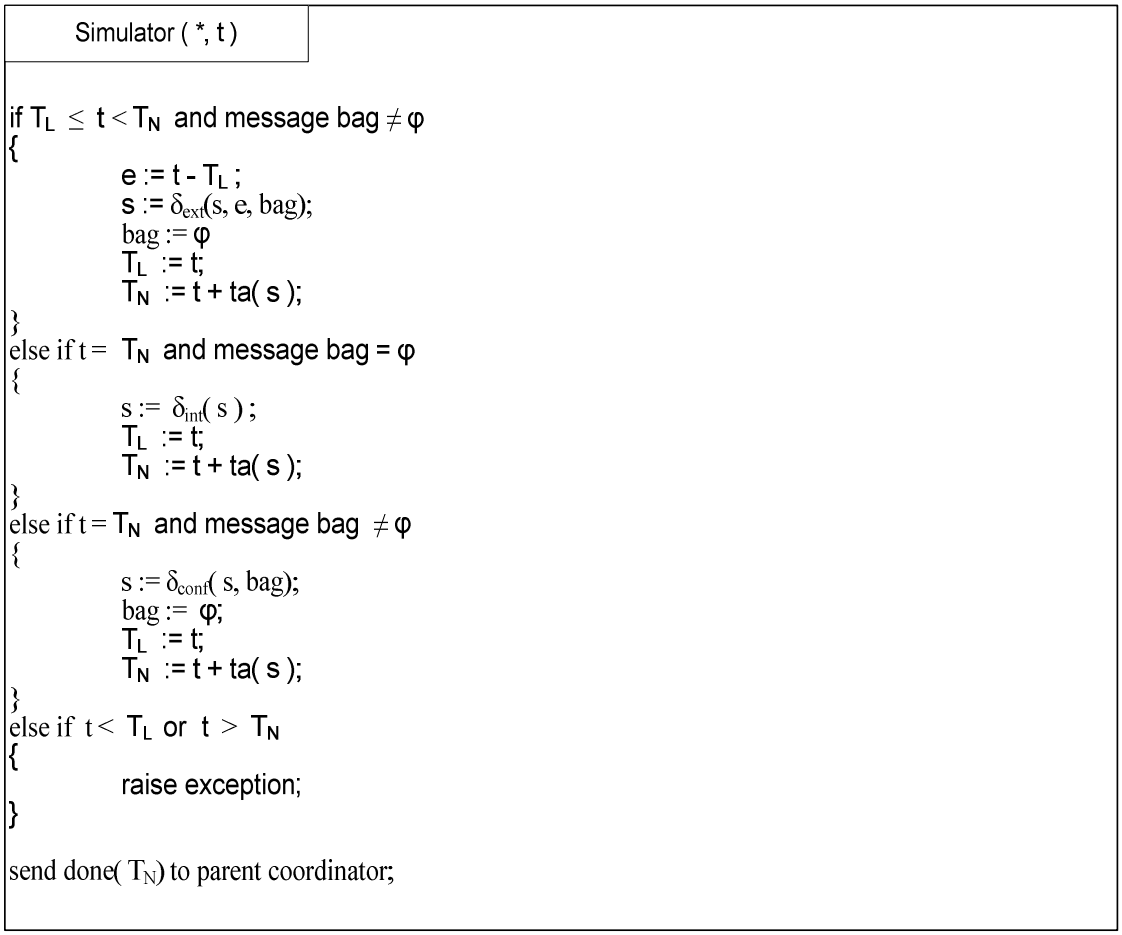

Figure 67: Simulator's reaction to an *internal message (*)*

The coordinator receives the same type of messages received by the simulator; however, it reacts in different ways to those messages. When a coordinator receives an *external message*, it simply adds it to its external message bag. When a coordinator receives an *internal message* from its parent coordinator at time *t*, the messages in the bag are forwarded to their destinations, and an *internal message* is sent to all the *processors* scheduled for state change (in the *synchronize set*); this behaviour is depicted in Figure 68:

```
Coordinator ( *, t )

if T_L ≤ t ≤ T_N
{
        for all influenced models (j), and all messages (m) in the bag
                q := z_self, j(m);
                send (q, j);
                cache j in synchronize set;
        end for;
        bag := φ

        for all processors f in synchronize set
                send (*, t) to f;
                doneCount := doneCount + 1;
        end for;

        synchronize set := φ;

        T_L := t;
}
else if t < T_L or t > T_N
{
        raise exception;
}
```

Figure 68: Coordinator's reaction to an *internal message (*)*

When a coordinator receives an *output message (y)* from child *i* at time *t*, it checks the influencees of the message. If there are local influencees, the *output message* is translated into *external messages* that are sent to the influencees; otherwise, the *output message* is forwarded to the parent coordinator:

```
Coordinator ( y, t )

For all influencees, j of child i
        x = z_i,j(y)
        send (x, t) to child j ;
        cache j in synchronize set;
End for;

If (y to be transmitted to parent)
        send (y, t) to parent coordinator ;
```

Figure 69: Coordinator's reaction to an *output message (y)*

When a coordinator receives a *done message* at time *t*, the *doneCount* variable (variable used to record the number of *processors* that received *internal* or *collect* messages) is decremented. If *doneCount* equals zero, the minimum time for the next state transition of the child *processors* is evaluated and reported to the parent coordinator:

```
Coordinator ( D, t )

if T_L ≤ t ≤ T_N
{
        if (donCount > 0)
        {
                doneCount := doneCount – 1;

                if (doneCount = 0 )
                {
                        T_N = minimum( T_N of all child processors);
                        send done(T_N) to parent coordinator;
                }
        }
        else
        {
                raise exception;
        }
}
else  if t < T_L or t > T_N
{
        raise exception;
}
```

Figure 70: Coordinator's reaction to a *done message (D)*

When a coordinator receives a *collect message* at time *t*, it forwards it to all the imminent child *processors*, as shown in Figure 71:

```
Coordinator ( @, t )

if  t = T_N
{
        T_L = t ;
        for all imminent child processors f with T_N = t
                send (@, t) to f;
                doneCount := doneCount + 1;
                cache f in synchronize set;
        end for;
}
else
{
        raise exception;
}
```

Figure 71: Coordinator's reaction to a *collect message (@)*

The *Root coordinator* is considered as a special coordinator that is responsible for driving the simulation as a whole. It starts the simulation by sending *initialization messages (I)* to its child *processors* and responds to the *done messages* it receive by sending either a

collect message or an *internal message* depending on the sequence of messages sent to the child *processors*.

When receiving a *done message*, the *Root coordinator* checks to see if the *done message* followed a *collect* or an *internal message*. If it was a response to a *collect message*, an *internal message* is sent to the *top* coordinator to complete the simulation cycle by triggering the state transition in the *simulators*. If the *done message* was sent as a response to an *internal message*, the current simulation cycle is considered over and the *Root* coordinator initiates a new cycle by performing the following steps:

i) It checks if the simulation clock has reached the maximum execution time; if so, it stops the simulation.

ii) If the external event list is not empty, the first event in the list (with minimum timestamp) is picked, and its time stamp is compared to the time of the next change as reported by the *done message*. The minimum of the two is considered as the value of the *nextTime* variable.

iii) If *nextTime* is larger than the maximum execution time as provided by the user, the simulation is stopped.

iv) If the external event list is not empty, all the events with a timestamp=*nextTime* are sent to the *top* model coordinator.

v) If *nextTime* equals the time of the next change (T_N), an *internal message* is sent to the *top* coordinator; otherwise, a *collect message* is sent instead.

Root Coordinator (D, t)

```
T_L := t ;
T_N := D (T_N);
nextTime = T_N ;
nextMsgType ∈ { collectMsg, internalMsg };

if nextMsgType = internalMsg
{
        nextMsgType = collectMsg;
        send (*,t) to top coordinator;
}
Else
{
        If ( T_N = ∞ and externalEventList = φ )
                stop;
        else
        {
                if ( externalEventList ≠ φ )
                {
                        event = front element in externalEventList;
                        If (  event.Time < nextTime )
                                nextTime = eventTime ;

                        if (stopTime < nextTime )
                                stop;
                        else
                        {
                                For all events (ev) with Time = nextTime
                                        send ev to top coordinator;
                                end for

                                if ( nextTime = T_N )
                                {
                                        nextMsgType = internalMsg;
                                        send (@, T_N) to top coordinator;
                                }
                               Else
                                {
                                        T_L = nextTime ;
                                        T_N = 0 ;
                                        nextMsgType = collectMsg;
                                        send ( *, T_N ) to top coordinator ;
                                }
                        }
                }
        }
}
```

Figure 72: The *Root* coordinator behaviour when receiving a *done message (D)*

Implementing DCD++ required extending the *Coordinator* functionality into a *Master Coordinator* and *a Slave Coordinator*. When a *master coordinator* receives an *external message*, it adds it to the external message bag. When it receives an *internal message* at time *t*, it sorts the *external messages* stored in its bag. This includes sending *external*

messages to the local receiving *processors* and/or sending *external messages* to remote *slave coordinators*. Then an *internal message* is sent for each *processor* in the *synchronize set* and the *doneCount* variable is incremented once for each sent message. The *doneCount* variable is used to track the number of *processors* scheduled for *internal* and/or *external* transitions. After sending the *external* and *internal messages*, the message bag and *synchronize set* are emptied:

Master Coordinator (*, t)

```
if TL ≤ t ≤ TN
{
        for all messages x in the bag
                for all influencees, j in machine n of child i
                        if j is local processor
                        {
                                send (x, t) to j ;
                                cache j in synchronize set ;
                        }
                        else
                        {
                                sc := slave coordinator in machine n ;
                                if sc ∉ slaveSet
                                {
                                        send(x, i , t ) to sc ;
                                        cache sc in synchronize set ;
                                        add sc to slaveSet;
                                }
                        }
                end for
        end for

        bag := φ ;

        for all processors f in synchronize set
                send (*, t) to f ;
                doneCount := doneCount + 1;
        end for

        synchronize set := φ ;

        TL := t ;
}
else if t < TL or t > TN
{
        raise exception ;
}
```

Figure 73: The *Master* coordinator's behaviour when receiving an *internal message (*)*

When a *master coordinator* receives an *output message* at time *t*, it checks the destinations of the message. If the receiving *processor* is local, the message is translated

into an *external message* that is sent to the *processor*. On the other hand, if the receiving *processor* is a remote one, the message is sent to the *slave coordinator* of the receiving *processor* running on the destination machine:

```
Master Coordinator ( y, t)
for all influencees, j in machine n of child i
        if j is local processor
        {
                m := z_{i,j} (y) ;
                send (m, t) to child j ;
                cache j in synchronize set ;
        }
        else
        {
                sc := slave coordinator in machine n
                if sc ∉ slaveSet ;
                {
                        send(y, i, t) to slave coordinator;
                        cache sc in synchronize set ;
                        add sc to slaveSet ;
                }
        }
end for

If y to be sent to parent
        send (y, t) to parent coordinator ;
```

Figure 74: The *Master* coordinator's behaviour when receiving an *output message (y)*

When a *master coordinator* receives a *collect message* at time t, it sends it to the imminent local simulators/coordinators and to all the *slave coordinators* with $T_N = t$:

```
Master Coordinator ( @, t)
if t = T_N
{
        T_L = t ;
        for all imminent child simulators (f) with T_N = t
                send (@, t ) to f ;
                doneCount := doneCount + 1 ;
                cache f in synchronize set ;
        end for

        for all imminent slave coordinators (s) with T_N = t
                send ( @, t) to s ;
                doneCount := doneCount + 1 ;
                cache s in synchronize set ;
        end for
}
else
{
        raise exception ;
}
```

Figure 75: The *Master* coordinator's behaviour when receiving a *collect message (@)*

Receiving a *done message* by a *master coordinator* at time *t* causes the *doneCount* variable to be decremented. If *doneCount* equals zero, it indicates that all the child *processors* scheduled for *internal* or *external* transitions are done. Then the *master coordinator* evaluates the minimum time of the next state transition of the local child *processors* and remote *slave coordinators* and reports the obtained value to its parent coordinator through a *done message*, as shown in Figure 76:

```
Master Coordinator ( D, t )

if TL ≤ t ≤ TN
{
        if (donCount > 0)
        {
                doneCount := doneCount – 1;

                if (doneCount = 0 )
                {
                        TN1 = minimum ( TN of all child processors) ;
                        TN2 = minimum ( TN of all slave coordinators) ;
                        TN  = minimum (TN1 and TN2) ;
                        send done(TN) to parent coordinator ;
                }
        }
        else
        {
                raise exception;
        }
else
{
        raise exception;
}
```

Figure 76: The *Master* coordinator's behaviour when receiving a *done message (D)*

When a *slave coordinator* receives an *external message*, it adds it to the external message bag. The behaviour of the *slave coordinator* when receiving a *collect message* is identical to the behaviour of the coordinator implementing the P-DEVS algorithms:

```
Slave Coordinator ( @, t )

if  t = TN
{
        TL = t ;
        for all imminent child processors f with TN = t
                send (@, t) to f;
                doneCount := doneCount + 1;
                cache f in synchronize set;
        end for;
}
else
{
        raise exception;
}
```

Figure 77: The *Slave* coordinator's behaviour when receiving a *collect message (@)*

When a *slave coordinator* receives an *output message* at time *t*, it translates it into an *external message* that is sent to the local receiving *processors* and/or it forwards it to its parent *master coordinator* to either be sent to a remote *processor*, or upper coordinator, or both.

Slave Coordinator (y, t)

```
For all influencees, j in machine n of child i
        if j is local processor
        {
                m := z_{i,j} (y) ;
                send (m, t) to child j ;
                cache j in synchronize set ;
        }
        else
        {
                if sentToMaster = false
                {
                        send (y , t ) to parent coordinator ;
                        sentToMaster := true ;
                }
        }

If y to be sent to parent
        if sentToMaster = false
                send (y, t) to parent coordinator ;
```

Figure 78: The *Slave* coordinator's behaviour when receiving an *output message (y)*

The behaviour of the *slave coordinator* when receiving a *done message* at time *t* is identical to the behaviour of the coordinator implementing the P-DEVS algorithms:

Slave Coordinator (D, t)

```
if T_L ≤ t ≤ T_N
{
        if (donCount > 0)
        {
                doneCount := doneCount – 1;

                if (doneCount = 0 )
                {
                        T_N = minimum( T_N of all child processors);
                        send done(T_N) to parent coordinator;
                }
        }
        else
        {
                raise exception;
        }
}
else if t < T_L or t > T_N
{
        raise exception;
}
```

Figure 79: The *Slave* coordinator's behaviour when receiving a *done message (D)*

When receiving an *internal message (*)*, the *slave coordinator* forwards the *external messages* in its bag to their local receiving *processors* and sends *internal messages* to all the *processors* cached in the *synchronize set*. At the end of this process, the message bag and *synchronize set* are emptied:

Slave Coordinator (*, t)

```
if TL ≤ t ≤ TN
{
        for all messages x in the bag
                for all influencees, j in machine n of child i
                        if j is local processor
                        {
                                send (x, t) to j ;
                                cache j in synchronize set ;
                        }
                        else
                        {
                                do nothing ;
                        }
                end for
        end for

        bag := φ ;

        for all processors f in synchronize set
                send (*, t) to f ;
                doneCount := doneCount + 1;
        end for

        synchronize set := φ ;

        TL := t ;
}
else
{
        raise exception ;
}
```

Figure 80: The *Slave* coordinator's behaviour when receiving an *internal message (*)*

Appendix-B: Web Service Components

The web service components of the simulation services were implemented using Java. They communicate with the simulation components through the *WrapperProxy*, which is implemented in C/C++ and loaded as a shared library by the Axis server. Figure 81 shows a UML diagram of the main classes of the web service components:

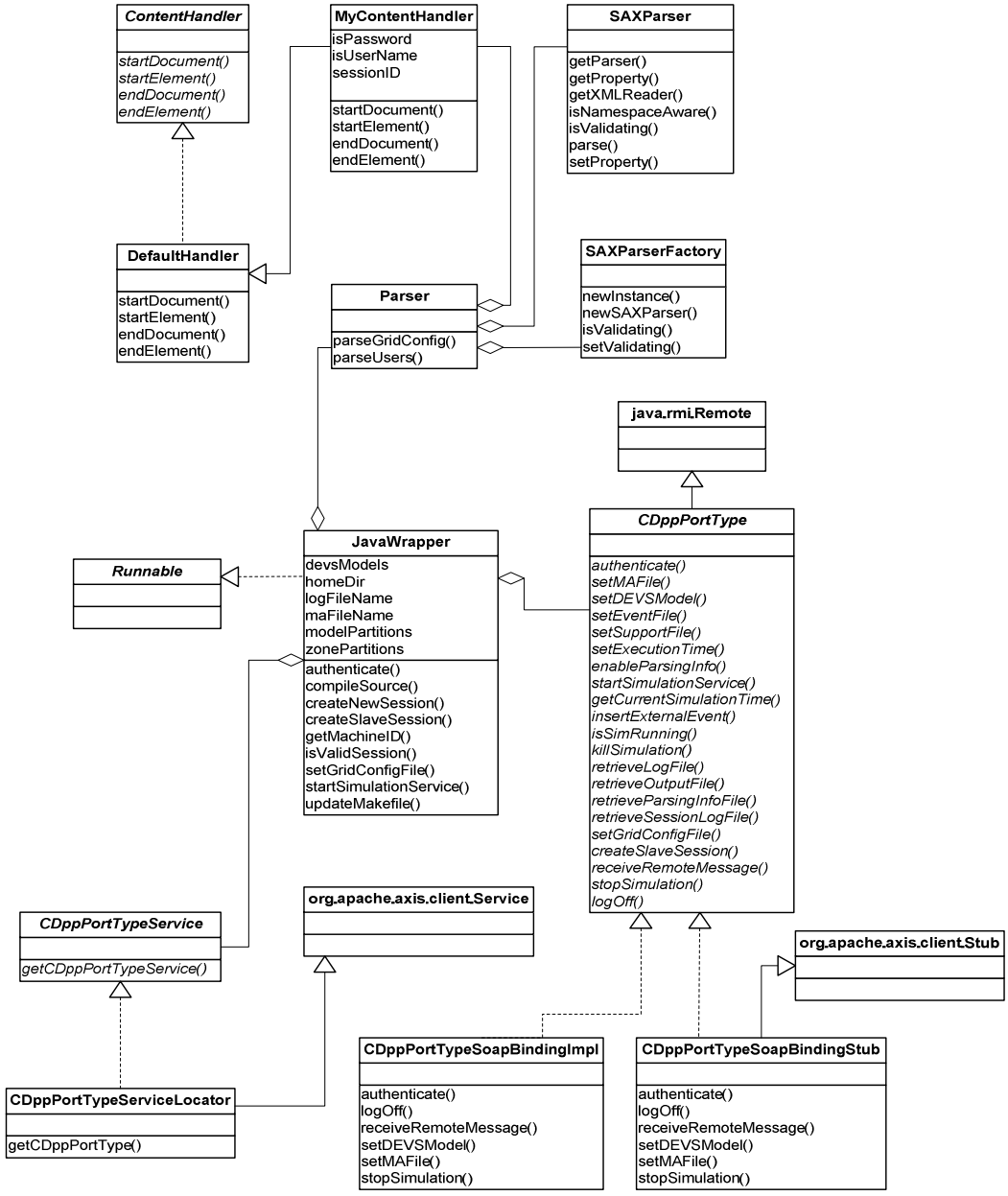

Figure 81: Web service components

The main class in the diagram is the *JavaWrapper* class, which constitutes the backbone of the web service components. In this appendix, a detailed description of the methods defined in the classes is presented. Some of the classes used are standard classes in Java 2 or part of the Axis libraries; those won't be covered here since their functionality is described in the official Java/Axis documentations. Those include:

- *ContentHandler (org.xml.sax.ContentHandler)*: it is an interface that receives notifications while parsing an XML document depending on the logical contents of the document.
- *DefaultHandler (org.xml.sax.DefaultHandler)*: it is the base class for *SAX2* event handlers.
- *Remote (Java.rmi.Remote)*: It is an interface used to identify objects whose methods can be executed on non-local virtual machines.
- *Runnable (java.lang.Runnable)*: it is an interface that should be implemented by any class whose instance to be executed as a thread.
- *SAXParser (javax.xml.parsers.SAXParser)*: it is an abstract class that wraps the functionality of an *XMLReader* implementation class; *XMLReader* is an interface for reading XML documents based on notifications.
- *SAXParserFactory (javax.xml.parsers.SAXParserFactory)*: it is a factory class that enables applications to obtain SAX-based parsers to parse XML documents.
- *Service (org.apache.axis.client.Service)*: it is Axis' JAXRPC implementation of the *javax.xml.rpc* interface. The *Service* class is the starting point for accessing SOAP web services.
- *Stub (org.apache.axis.client.Stub)*: it is an abstract base class for all stub classes.

JavaWrapper:

The methods implemented in the *JavaWrapper* class are:

* public void **addMachine**(Integer machineId, String uri) Used to add a machine id and address by the parser.

* public void **addModelPartition**(String model, Integer machineId) Used to add model partition information by the parser.
* public void **addRemoteModelPartition**(String model, Integer machineId) Used to add remote model partitions by the parser (used when integrating DCD++ and PCD++).
* public static void **addUser**(String userName ,String password): Used by the *parser* to add a user credentials.
* public static void **addUserRole**(String userName, String role): Used by the parser to add a user role.
* public void **addZonePartition**(String zone, Integer machineId) Used by the parser to add Cell-DEVS model partitions.
* private void **addZonePartitions**() Used to send the Cell-DEVS zone partitioning information to the simulator.
* private void **archiveLogFiles**() Used to archive the log files into a (.tar) file to be retrieved by the user.
* public static int **authenticate**(String username, String password, boolean isPCDpp) Used to authenticate the users.
* private boolean **compileSource**() Used to compile the source code of the simulator with the code of the added DEVS models.
* private void **copyDirs**(java.io.File srcDir, java.io.File destDir) Used to copy directories during the creation of a new session.
* private void **copyFiles**(java.io.File srcFile, java.io.File destFile) Used to copy files during the creation of a new session.
* private static int **createNewSession**(String userName, boolean isPCDpp) Used by the *authenticate* method to create new sessions.
* public static boolean **createSlaveSession**(int sessionID, String userName): Used by the master node to initialize slave sessions (when running distributed simulation).
* public String **enableParsingInfo**()

Enables the parsing debug option in CD++ (used for Cell-DEVS models).
* public String **getCurrentSimTime**() Returns the current simulation time by checking the *nextChange* variable in the *Root* coordinator (through the *WrapperProxy* and *CPPWrapper*).
* public static int **getMachineID**(int sessionID) Returns the machine id by examining the address of the simulation service, and comparing it with the addresses in the *grid configuration file*.
* public int **getSessionID**() Used to get the session id of the *JavaWrapper* instance.
* public static JavaWrapper **getWrapperInstance**(int sessionID) Used by the server-side stubs to retrieve a *JavaWrapper* instance corresponding to the session id (sessionID).
* private boolean **initialize**() Used to initialize the message queues to communicate with CD++.
* private boolean **initializeSlaveSessions**() Used to initialize slave sessions by sending the model and *grid configuration* files.
* public void **insertExternalEvent**(String time_, String port_, double value_) Used to insert external events while the simulation is running.
* public static boolean **isLoggedIn**(String userName) Checks to see if the current user has a running session.
* public boolean **isSimRunning**() Used to check if the simulation process is still running.
* public static boolean **isValidSession**(int sessionID) Checks to see if the sessionID matches a valid session.
* private static boolean **isValidUser**(String userName, String password) Checks if the user is a valid one.
* public **JavaWrapper**(boolean isPCDpp) The constructor is used to distinguish between DCD++ and PCD++ services.
* public void **killSimulation**() Used to kill the simulation process.

* public static boolean **logOff**(int sessionID) Used to log off a user and to invalidate his session.
* public static int **machineForModel**(int sessionID, String modelName) Returns the machine id that is executing the model (modelName).
* public void **receiveRemoteMessage**(int msgType, String msgTime, int srcProcId, String nextChange, int PortId, double value, int senderModelId, boolean isFromSlave, int destProcId) Used to receive a remote message sent as SOAP (when running distributed simulation).
* private boolean **registerDEVS**() Used to modify the *register.cpp* file (part of CD++) to add a DEVS model(s).
* public String **retrieveLogArchiveName**() Used by the server-side stubs to retrieve the name of the log archive to be sent to the user.
* public String **retrieveOutputFileName**() Used by the server-side stubs to retrieve the name of the output file to be sent to the user.
* public String **retrieveParsingInfoFileName**() Used by the server-side stubs to retrieve the name of the parsing information file to be sent to the user.
* public String **retrieveSessionLogFileName**() Used by the server-side stubs to retrieve the name of the session log file to be sent to the user.
* private boolean **retrieveSlaveLogFiles**() Used to retrieve the slave log files at the end of a distributed simulation session.
* public void **run**() This method is required by the *Runnable* interface; it is responsible for streaming the simulator output into the session log file and for starting the message monitor (used to monitor the message queues).
* public static void **sendRemoteMessage**(int sessionID, int msgType, String msgTime, int srcProcId, String nextChange, int portId, double value, int senderModelId, boolean isFromSlave, int machineId, int destProcId) Used to send a remote message using SOAP (when running distributed simulation).

* public String **setDEVSModel**(String cppFileName,DataHandler dhCPPFile,String hFileName, DataHandler dhHFile) Used to set DEVS header and implementation files.
* public String **setEventFile** (String eventFileName,DataHandler dhEventFile) Used to set the external events file by the user.
* private void **setFilePerms**() Used to set the file permissions of the CD++ executable during the creation of a new session.
* public String **setGridConfigFile**(String gcFileName, DataHandler dhGCFile) Used to set the *grid configuration file* by the user.
* public String **setMAFile**(String maFileName, DataHandler dhMAFile) Used to set the model definition file (.ma) by the user.
* public String **setNumberOfNodes**(int noNodes) Sets the number of nodes used by the cluster (when integrating DCD++ and PCD++).
* public String **setPartitionFile**(String partitionFileName, DataHandler dhPartitionFile) Used to set the partition file by the user (used for the PCD++ service).
* public String **setSimulationTime**(String simTime) Sets the execution time.
* public String **setSupportFile** (String supportFileName, DataHandler dhSupportFile)) Used to set the initial values file (for Cell-DEVS models).
* public String **startSimulationService**() Used to start the simulator.
* public void **stopSimulation**() Used to stop the simulation in the slave nodes (when running distributed simulation).
* private boolean **stopSlaveSessions**() Used to stop the slave sessions at the end of a distributed simulation session.
* public boolean **updateMakeFile**() Used to update the *make* file to incorporate the added DEVS model(s).

CDppPortType:

The *CDppPortType* interface defines the main methods offered by the simulation service. The functionality of each method is the same as the one provided for the *JavaWrapper* class except for the log and output file retrieval methods; since they return the actual files instead of the file names. Those methods are:

public static int **authenticate**(String username, String password, Boolean isPCDpp)
public static boolean **createSlaveSession**(int sessionID, String userName)
public String **enableParsingInfo**()
public String **getCurrentSimTime**()
public void **insertExternalEvent**(String time_, String port_, double value_)
public boolean **isSimRunning**()
public void **killSimulation**()
public static boolean **logOff**(int sessionID)
public void **receiveRemoteMessage**(int msgType, String msgTime, int srcProcId, String nextChange, int PortId, double value, int senderModelId, boolean isFromSlave, int destProcId)
public String **retrieveLogArchive**()
public String **retrieveOutputFile**()
public String **retrieveParsingInfoFile**()
public String **retrieveSessionLogFile**()
public String **setDEVSModel**(String cppFileName,DataHandler dhCPPFile,String hFileName, DataHandler dhHFile)
public String **setEventFile**(String eventFileName,DataHandler dhEventFile)
public String **setGridConfigFile**(String gcFileName, DataHandler dhGCFile)
public String **setMAFile**(String maFileName, DataHandler dhMAFile)
public String **setSimulationTime**(String simTime)
public String **setSupportFile**(String supportFileName, DataHandler dhSupportFile)
public String **startSimulationService**()
public void **stopSimulation**()

CDppPortTypeSoapBindingImpl:

It is a server-side stub class that implements the *CDppPortType* interface and is deployed in Axis as part of the service deployment process. The class is generated by the Axis tools as a *skeleton* class that is filled with the implementation by the web service designer/programmer. The methods implemented in the class are exactly the same as the ones described for the *CDppPortType* interface.

CDppPortTypeSoapBindingStub:

This is the client-side stub that is used to access the simulation service. It implements the *CDppPortType* in order to create the SOAP requests and responses for the interface methods.

CDppPortTypeService:

It is an interface that defines the methods necessary to locate the web service given its URL. It defines two methods:

* public *CDppPortType* **getCDppPortType**() It returns a stub class implementing the *CDppPortType* interface using the *local host* address as the address of the web service.
* public *CDppPortType* **getCDppPortType**(java.net.URL) It returns a stub class implementing the *CDppPortType* interface using the URL provided as the address of the web service.

CDppPortTypeServiceLocator:

It is a class implementing the *CDppPortTypeService* interface and is used as the starting point to locate and access the web service.

Parser:

It is the main class used for parsing XML documents in the simulation service. Those documents include: the *users file*, and *grid configuration file*. The methods implemented in the class are:

* public static void **parseUsers**() It is used to parse the *users file*; the users file contains the usernames, passwords, and roles of all the users authorized to use the simulation service.
* public static void **parseGridConfig**(int sessionID, String fileName) It is used to parse the *grid configuration file*, which contains the addresses of the machines participating in the simulation in addition to the model partition.

MyContentHandler:

This class implements the methods defined in the *ContentHandler* interface that gets called by a *SAXParser* when parsing XML documents. The methods implemented in the class are:

* public void **registerSessionID**(int sessionID) This method is called by the *JavaWrapper* class in order to set the session id before parsing the *grid configuration file*.
* public void **startElement**(String namespaceURI, String localName, String rawName, Attributes atts) This method is called by the *SAXParser* at the beginning of each element in the XML document.
public void **endElement**(String namespace, String localName, String rawName) This method is called by the *SAXParser* at the end of each element in the XML document.
public void **characters**(char[] ch, int start, int length) * This method is called by the *SAXParser* between the start and end of each element in the XML document with a *char* array (ch) containing the element contents.