

IMPLEMENTING PARALLEL AND DISTRIBUTED DEVS AND CELL-DEVS SIMULATION IN A WINDOWS PLATFORM

By

Bo Feng, B.Eng

A thesis submitted to
The Faculty of Graduate Studies and Research

In partial fulfillment of the requirements for the degree of
Master of Science in Information and Systems Science

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

© Copyright 2009, Bo Feng

The undersigned recommend to the
Faculty of Graduate Studies and Research

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

**IMPLEMENTING PARALLEL AND DISTRIBUTED DEVS
AND CELL-DEVS SIMULATION IN A WINDOWS
PLATFORM**

submitted by

Bo Feng, B.Eng.

in partial fulfillment of the requirements for

the degree of

Master of Science in Information and Systems Science

Professor Gabriel Wainer, Thesis Supervisor

Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

January 2009

ABSTRACT

Research advances in modeling and simulation emphasize the need for parallel and distributed methodologies and environments. The growing demand for executing complex models by general users has directed researchers to implement parallel simulation with commodity PC machines. This research presents an effective approach to executing a parallel and distributed Discrete Event System Specification (DEVS) and Cell-DEVS application in Windows cluster environments.

DEVS is a modular and hierarchical formalism for modeling and analyzing general systems that can be described by discrete events. Cell-DEVS is a DEVS-based formalism used to model complex physical systems as cell spaces. Parallel DEVS provides a way to handle simultaneously scheduled events, while keeping all the major properties of the original DEVS formalism. A logical process (LP), as a basic entity in parallel DEVS environments, receives and generates timestamped events to communicate with other LPs. The communication mechanism of the LP can be implemented with a distributed paradigm.

In this research, PCD++Win and PCD++/.NET simulation systems are presented. Both of them follow a conservative approach and use a set of commodity Windows PC machines to execute parallel DEVS and Cell-DEVS simulations. PCD++Win is based on Windows MPI and allows users to setup a Windows cluster and execute parallel DEVS and Cell-DEVS simulations with a GUI. PCD++Win can be exposed as a Web service, while another application can consume PCD++Win over the Internet. PCD++/.NET is based on Microsoft.NET. It presents an approach to combining .NET Remoting objects with a simulation engine to execute parallel and distributed DEVS and Cell-DEVS simulations. PCD++/.NET supports several communication protocols and runs on the Common Language Runtime. The performance analysis shows that the speedup of PCD++/.NET can be achieved for the simulation, which has a modest inter-LP communication load.

TABLE OF CONTENTS

ABSTRACT.....	iii
TABLE OF CONTENTS.....	iv
LIST OF TABLES.....	x
CHAPTER 1 INTRODUCTION.....	1
1.1. Motivation and Goals	2
1.2. Contribution.....	3
1.3. Thesis organization	4
CHAPTER 2 BACKGROUND	9
2.1. <i>P-DEVS and Cell-DEVS formalism</i>	9
2.2. The CD++ Toolkit.....	17
2.3. Windows MPI.....	19
2.4. Web services	20
2.5. .NET Remoting.....	20
CHAPTER 3 REVIEW DEVS-BASED SIMULATION TOOL.....	22
3.1. The overview of DEVS-based simulation tool.....	23
CHAPTER 4 PCD++WIN	27
4.1. Software architecture.....	27
4.2. Parallel DEVS abstract simulator in PCD++Win	29
4.3. The NoTime kernel in PCD++Win	39
4.4. Running PCD++Win with DeinoMPI interface	41
4.5. Performance Metrics	43
4.6. Experimental results of PCD++Win	44
CHAPTER 5 EXPOSING PCD++WIN AS WEB SERVICE.....	52
5.1. SOAP and WSDL	52
5.2. Building a PCD++Win Web Service	53
5.3. Consuming PCD++Win Web Service.....	58
CHAPTER 6 PCD++/.NET	61
6.1. An overview of distributed paradigms.....	61
6.2. .NET Remoting versus Web Services	63
6.3. The benefits of .NET	64

6.4.	<i>PCD++/.NET Remoting System</i>	65
CHAPTER 7 PERFORMANCE ANALYSIS FOR PCD++/.NET		76
7.1.	<i>Remote message of PCD++Win and PCD++/.NET</i>	76
7.2.	<i>Correctness and verification</i>	77
7.3.	<i>Experimental results and analysis</i>	78
7.3.1.	Watershed model	78
7.3.2.	Life model	83
CHAPTER 8 CONCLUSIONS AND FUTURE WORK		90
8.1.	<i>Future work</i>	92
REFERENCES		94

LIST OF FIGURES

Figure 1. OSI model.....	27
Figure 2. Architecture of PCD++ and PCD++Win	28
Figure 3. PCD++Win and OSI layer	29
Figure 4. PCD++Win major class diagram.....	31
Figure 5. Master and Slave Coordinator function	33
Figure 6. Simulator algorithm [Ch094b][Tro03]	35
Figure 7. Master coordinator algorithm [Cho94b][Tro03]	36
Figure 8. Slave coordinator algorithm [Cho94b][Tro03]	37
Figure 9. Root coordinator algorithm [Cho94b][Tro03].....	38
Figure 10. Abstract simulator in PCD++Win	39
Figure 11. NoTime kernel and PCD++Win.....	40
Figure 12. The GUI of DeinoMPI.....	41
Figure 13. Main windows running PCD++Win	42
Figure 14. Job verification tool.....	42
Figure 15. Fire model definition [Wai08]	45
Figure 16. Partition strategy.....	46
Figure 17. The experiment result of Fire model.....	46
Figure 18. State of collision avoidance model.....	47
Figure 19. The first part of collision avoidance model [Wai08]	48
Figure 20. The second part of collision avoidance model [Wei08]	49
Figure 21. One of UAV's rules	49
Figure 22. The result of the collision avoidance model	50
Figure 23. PCD++Win JAX-RPC Web service	54
Figure 24. PCD++Win Web service server-side runtime	55
Figure 25. WSDL service interface and implementation	57
Figure 26. WSDL file of the PCD++Win web service	58
Figure 27. PCD++Win Web service client.....	59
Figure 28. Web service client invokes a remote method	59
Figure 29. PCD++Win Web service client GUI.....	60
Figure 30. PCD++/.NET architecture.....	65
Figure 31. The interaction of objects in .NET Remoting [REM08]	66
Figure 32. PCD++/.NET components.....	68
Figure 33. PCD++/.NET Remoting method call.....	70
Figure 34. Message passing.....	71
Figure 35. PCD++/.NET Remoting simulators.....	72
Figure 36. msgExchange component	72
Figure 37. Flowchart of PCD++/.NET Remoting.....	73
Figure 38. Sequence diagram of PCD++/.NET Remoting.....	75
Figure 39. Partitioning a couple model into 4 atomic models	76
Figure 40. PCD++Win passing messages	77
Figure 41. PCD++/.NET passing messages.....	77
Figure 42. Hydrology Model [Moo96]	78
Figure 43. Watershed Model[wai08]	80
Figure 44. Partition of watershed model	81
Figure 45. Execution result of watershed model in PCD++Win	81
Figure 46. Execution time of watershed with PCD++/.NET Remoting.....	82
Figure 47. The first part of Life model file [wai08]	84
Figure 48. The second part of Life model file[wai08].....	84
Figure 49. Partition strategy of the Life model	85
Figure 50. Simulation results of the Life model.	85
Figure 51. Binary serialization in .NET [Her03]	88
Figure 52. Comparing two systems	89

Figure 53. Three layers of a DEVS simulation tool.....	90
Figure 54. Implementing various parallel and distributed mechanisms for PCD++.....	91

LIST OF TABLES

Table 1. DEVS-based simulation tools 23
Table 2. The message of watershed model 82
Table 3. The message of Life model..... 86

LIST OF ACRONYMS

DEVS	Discrete Event System Specification
LP	Logical Process
M&S	Modeling and Simulation
MPI	Message Passing Interface
P-DEVS	Parallel Discrete Event System Specification
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
SOAP	Simple Object Access Protocol
SOA	Service Oriented Architecture
WSDL	Web Services Description Language
RMI	Remote Method Invocation
XML	Extensible Markup Language
ECMA	European Computer Manufacturers Association
JAX-RPC	Java API for XML-Based Remote Procedure Calls
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol

CHAPTER 1 INTRODUCTION

Modeling and simulation (M&S) is a methodology used in a wide variety of fields, ranging from aerospace engineering to digital circuit design, from economics to environment studies, from weather forecast to national defense. Scientists and engineers use M&S to study and analyze complex problems. The **Discrete Event System Specification** (DEVS) [Zei76][Zei00] presents a means for the construction of hierarchical models in a modular manner and provides a discrete-event M&S mechanism, which allows developers to reuse components and reduce development and testing time. The **Timed Cell-DEVS formalism** [Wai01] combines DEVS theory with cellular automata [Neu66], allowing n-dimensional cell spaces as a basic DEVS model and according to a specified timing.

Complex system simulation usually requires massive amounts of computing time. Therefore, it is difficult to obtain results through sequential simulation. The **Parallel DEVS formalism** [Cho94a], as an extension to the DEVS, provides a way to deal with simultaneously scheduled events: It eliminates the serialization constraints existing in the original DEVS definition and enables the efficient execution of models in parallel and distributed environments.

For parallel and distributed simulations, two general categories of synchronization algorithms have been proposed. The one is a conservative approach, which tries to ascertain the distribution order of the messages, processing them to avoid causality errors. Another is the optimistic approach, which is a synchronization mechanism that allows for a higher degree of parallelism by ignoring possible causality errors, and presumes that the messages will arrive in the correct temporal order. When a causality error does occur, the protocol rolls the simulation back to the state before the time of the most recently arrived message.

Based on parallel DEVS and Cell-DEVS formalisms, parallel CD++ (**PCD++**) [Tro03] was developed. PCD++ implements a conservative synchronization algorithm and executes parallel DEVS and Cell-DEVS simulations in a clustered Linux environment.

As an extension to PCD++ [Tro03], PCD++Win and PCD++/.NET are proposed in this research. First, PCD++Win is developed by porting PCD++ to a Windows environment and taking advantage of the multi-purpose GUI of the Windows MPI middleware for the construction of a Windows cluster and the configuration of a simulation environment. Second, PCD++Win is exposed as a Web service, which can be consumed by another application on the Internet. Finally, the PCD++/.NET is created by combining a PCD++ simulation engine with Microsoft's .NET [Ram05], which is an execution environment and provides built-in Remoting services for the .NET application. Therefore, the PCD++/.NET can execute parallel and distributed DEVS and Cell-DEVS simulations in the .NET Framework.

1.1. Motivation and Goals

The motivation behind this work comes from the need to run complex simulations with commodity hardware. On one hand, traditional parallel discrete event simulation systems are typically run on dedicated hardware, such as clusters and supercomputers. Although these platforms offer the highest performance for parallel discrete event simulation applications, the availability of these resources is highly limited, and often restricted. On the other hand, PCs are increasingly popular and cheap because of the development of new semiconductor techniques. As of June 2008, the number of personal computers in use worldwide hit one billion, while another billion is expected to be reached by 2014 [wik08]. As the dominant operating system for personal computers, Windows is everywhere. Therefore, using the Windows platform for the design of parallel and distributed simulations would allow more users to know and use these powerful techniques. Using a familiar Windows-based graphical user interface to setup, configure and execute simulations provides an easy-to-use tool for general users and reduces the learning curve. Finally, the Microsoft .NET framework, which includes a large library of pre-coded solutions to common programming problems and a virtual machine to manage the execution of programs, provides network communications and allows us to write parallel and distributed applications. Based on the above ideas, PCD++Win and PCD++/.NET are proposed. PCD++Win takes advantages of Windows

MPI and presents a way of executing parallel DEVS and Cell-DEVS simulations in a Windows environment. PCD++/.NET integrates .NET Remoting with PCD++ and runs parallel and distributed DEVS and Cell-DEVS simulations in Common Language Runtime (CLR) environments, which provides a virtual machine for memory management and exception handling. The goal of developing both PCD++Win and PCD++/.NET is to combine Windows techniques with a DEVS simulation engine to greatly reduce simulation costs.

1.2. Contribution

In this thesis, a new parallel and distributed DEVS and Cell-DEVS simulation framework called PCD++/.NET is proposed. PCD++/.NET is an extension of parallel CD++ in a windows cluster environment. It combines Microsoft.NET Remoting technology with a parallel CD++ conservative engine, and allows users to execute parallel and distributed DEVS and Cell-DEVS simulations in Windows platforms. Specifically, the following efforts have been made:

- Building PCD++Win, which ports PCD++ **conservative simulators** [Tro03] to a windows environment by replacing MPICH with DeinoMPI [Dei08]. PCD++Win is a parallel simulation engine that takes advantage of the multi-purpose GUI of the DeinoMPI for the construction of PC clusters and the configuration of simulation environments. PCD++Win allows users to execute parallel DEVS and Cell-DEVS simulations with commodity Windows PC machines. With PCD++Win, it is possible for a user to execute parallel simulations in the lab, the office and home. This reduces the simulation cost and makes more users familiar with the value of parallel simulations.
- Exposing PCD++Win as a Web service that can be consumed by another application on the internet. The Web service adopts JAX-RPC (Java API for XML-Based Remote Procedure Calls), which provides a generic mechanism that enables developers to create Web services by using XML-based Remote Procedure Calls. A web service is essentially a function or method that is available to other machines on a network. Because web services use

standardized interfaces such as the Web Services Description Language (WSDL), SOAP, XML and HTTP, it is therefore independent from programming language and the platform. This means that a non-Windows user can invoke PCD++Win to execute parallel DEVS and Cell-DEVS simulations with a PCD++Win Web service.

- Designing and coding PCD++/.NET. PCD++/.NET integrates a PCD++ conservative simulation engine [Tro03] with .NET Remoting, which provides built in network services and supports various protocols such as HTTP, TCP and SMTP. .NET Remoting runs on Common Language Runtime, which is the implementation of open standard ISO 23271 and ECMA-335 (European Computer Manufacturers Association). With .NET Remoting, a speedup of parallel DEVS and Cell-DEVS simulations is achieved for some models, especially those that have modest inter-LP communication loads.
- Research results for PCD++Win and PCD++/.NET were published in [Fen08a] and [Fen08b] respectively.

1.3. Thesis organization

This thesis is organized as follows: Chapter 2 introduces the DEVS and Cell-DEVS formalisms, techniques related to the work such as Windows MPI, Web service and .NET. Chapter 3 presents a survey of DEVS-based simulation tools. Chapter 4 presents PCD++Win, a parallel conservative simulation engine that takes advantage of the multi-purpose GUI of the DeinoMPI for the construction of Windows cluster environments. PCD++Win has been developed using a modular approach that promotes code reuse and allows for easy switching to other middleware technologies. Chapter 5 presents the PCD++Win Web service, which allow users to invoke PCD++Win from any platform. Chapter 6 presents a distributed simulation framework, called PCD++/.NET, which integrates a Microsoft .NET Remoting mechanism with a PCD++ conservative simulation engine to execute a distributed DEVS and Cell-DEVS simulation. Chapter 7 covers the experimental results for PCD++/.NET. Chapter 8 presents the main conclusions of the thesis and outlines possible future research and development.

CHAPTER 2 BACKGROUND

P-DEVS (Parallel-Discrete Event System Specification) is a mathematical formalism with well-defined concepts of coupling of components, hierarchical, modular model construction for high-performance parallel discrete-event simulation. The **Timed Cell-DEVS** formalism uses DEVS to define a cell space where each cell is represented as a DEVS atomic model. In this chapter, a brief introduction of P-DEVS and Cell-DEVS will be covered in first section. Then, several parallel and distributed techniques such as Windows MPI, Web service and .NET Remoting will be discussed in the following sections.

2.1. P-DEVS and Cell-DEVS formalism

Based on dynamic systems theory, the DEVS formalism [Zei76] provides a framework for defining hierarchical models in a modular way. A system is described in DEVS as a composition of behavioral (atomic) and structural (coupled) components. The P-DEVS formalism [Cho94b] eliminates the sequential execution constraints imposed by the original DEVS definition, and provides a theoretical foundation for high-performance parallel and distributed discrete-event simulation. A P-DEVS atomic model is defined as:

$$\mathbf{M} = \langle \mathbf{X}, \mathbf{S}, \mathbf{Y}, \delta_{\text{int}}, \delta_{\text{ext}}, \delta_{\text{con}}, \lambda, \text{ta} \rangle$$

At any given time, an atomic model is in some state $s \in S$. Without the occurrence of external events, it remains in state s for a period $\text{ta}(s)$, which is referred to as the lifetime of state s . When the lifetime expires, the atomic model outputs value $\lambda(s) \in Y$, and changes to a new state given by the internal transition function $\delta_{\text{int}}(s)$. A P-DEVS model employs a bag of inputs (X^b) to support the execution of multiple concurrent events. If one or more external events $x \in X$ occur before the expiration of $\text{ta}(s)$, the model transfers to a state that is determined by the external transition function $\delta_{\text{ext}}(s, e, X^b)$, combining the functionality of multiple external transitions into a single one. A confluent transition function (δ_{con}) is defined to determine the next state in the case of collisions when a component receives external events at the same time of its internal transition.

The P-DEVS formalism has a well-defined concept of system modularity and component coupling to form composite models. A P-DEVS coupled model is defined as:

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

The sets of input and output events are defined by X and Y respectively. D is a set of indices for the components of a coupled model and, for each $d \in D$; M_d is a basic P-DEVS model (atomic or coupled). The external input coupling (EIC) specifies the connections between external and component inputs, while the external output coupling (EOC) describes the connections between the components themselves are defined by the internal coupling (IC).

The Timed Cell-DEVS formalism [Wai01] was proposed to define n-dimensional cell spaces as discrete-event DEVS coupled models, where each cell is represented as a DEVS atomic model. It defines timing constructions for each cell, allowing explicit timing specification, asynchronous model execution, and integration with other types of models. A Cell-DEVS atomic model is defined as:

$$TDC = \langle X, Y, I, S, \theta, N, \text{delay}, d, \delta_{\text{int}}, \delta_{\text{ext}}, \tau, \lambda, D \rangle$$

A cell has a modular interface (I) that is composed of a fixed number of ports; each is connected to a neighboring cell. It can input and output data (X and Y) with its neighbors as well as other models outside of the cell space. The future state of a cell is computed by the local transition function (τ) based on the cell's current state and input values. State changes in a cell are transmitted only after a delay given by the delay function (d). Each cell also has the computing apparatus (δ_{int} , δ_{ext} , and λ) as defined in DEVS atomic models. Cells are coupled by the neighborhood relationship to form a cell space, which can then be integrated with other DEVS and Cell-DEVS models. A cell space is defined as a Cell-DEVS coupled model:

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

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

2.2. The CD++ Toolkit

The stand-alone CD++ [Rod99][Tro03] is a M&S toolkit that implements DEVS and Cell-DEVS formalisms. Following the M&S framework [Zei76][Zei00], which separates the model and simulator, the abstract simulation mechanism in CD++ enables the modeler to focus on the definition of the models. The only relationship between the models and the simulation engine is defined by the manipulation of a variable containing the time of the next scheduled event, called sigma. This variable is used to implement the time advance function: it stores the time remaining until the next scheduled event. The internal transition function is activated when $\sigma = 0$, and sigma must be recomputed every time a model is activated, as each state has an associated lifetime. Every model also includes a ‘phase’ variable (whose basic states are active and passive), which can be used to verify the correctness of the functions defined. For instance, a model in the passive phase cannot have an internally scheduled event. Likewise, an active model cannot have an infinite value for sigma.

A DEVS atomic model is created by including a new class derived from the CD++ built-in class called “Atomic”. The following methods may be overloaded:

- `initFunction`. This method is invoked when the simulation starts. It allows one to define initial values and to execute setup functions for the model.
- `externalFunction`. This method is invoked when an external event arrives from an input port.
- `internalFunction`. This method is started when an internal event occurs (that is, the value of sigma is zero).
- `outputFunction`. This method is executed before the internal function in order to generate outputs for the model.

The above functions are equivalent to those defined in the formal specifications for atomic models. After defining the atomic models for a given application, they can be combined into a multi component model. Coupled models are defined using a specification language that was built following the formal definitions for DEVS coupled

models. Therefore, each of the components defined formally for DEVS coupled models can be included. Optionally, configuration values for the atomic models can be included. The [top] model always defines the coupled model at the top level. The following properties must be configured:

- Components. This describes the models integrating a coupled model.
- Out. This defines the names of output ports.
- In. This defines the names of input ports.
- Link. This describes the internal and external coupling scheme.

For the definition of Cell-DEVS model, the modeler does not need to create any new classes since CD++ already has a set of built-in classes for it. CD++ also includes a specification language allowing the description of Cell-DEVS models. These definitions are based on the formal specifications. The following parameters, such as size, influences, neighborhood and borders, are specified. These are used to generate the complete cell space. The behavior of the local computing function is defined using a set of rules of the form `VALUE DELAY {CONDITION}`. These indicate that when the `CONDITION` is satisfied, the state of the cell changes to the designated value, and it is delayed for the specified time. If the condition is false, the next rule in the list is evaluated until a rule is satisfied or there are no more rules.

In parallel CD++ (PCD++) [Tro03], the model can be partitioned into several components. Each component is executed by a CPU and communicates with other component by means of MPICH [MPI08]. The simulation is 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 PCD++ has two types of processor: a simulator that executes simulations of an atomic DEVS model by invoking the atomic model's transition and external event functions, and a coordinator that translates its children's output events.

2.3. Windows MPI

MPI (Message Passing Interface) was first discussed at the Supercomputing 92 conference (1992). The attendants agreed to develop a common standard for message passing. The first MPI standard (MPI-1) [MPI08] was completed in 1994. MPICH [MPI08] is the implementations of MPI-1. Currently, most Windows MPIs are implemented based on MPICH. Some of them are commercial products and others are in the public domain. DeinoMPI [Dei08] is an implementation of MPI-2 [MPI08] for Microsoft Windows, and provides the libraries for developers to write parallel applications and a process manager to start processes remotely on multiple machines.

DeinoMPI is a derived work from MPICH2 [MPI08] provided by Argonne National Lab. However, DeinoMPI extends MPICH2 with the following support [Dei08]:

- UNICODE support.
- Automatically allows for copying a directory with all data files out to the worker nodes and then start this job from the new directory.
- The collective operations have been optimized to minimize network traffic when multiple processes reside on each node.
- Uses public and private keys to establish secure connections between machines in the cluster; also all traffic between the process managers is encrypted.
- The process manager can launch processes that have been compiled for either DeinoMPI or MPICH2.

2.4. Web services

Web services are a set of related application functions that can be programmatically invoked over the Internet. Businesses can dynamically mix and match Web services to perform complex transactions with minimal programming. Since communicating Web services can be deployed on different locations using different implementation platforms, agreeing on a set of standards for data transmission and service descriptions is clearly very important. Web services interact with each other using XML messages. XML is not only a data format, but also a formal set of information items. By offering a standard, XML significantly reduces the burden of deploying the many technologies needed to ensure the success of Web services. Simple Object Access Protocol (SOAP) [SOA08]

provides a standard framework for packaging and exchanging XML messages [W3C08]. A SOAP message represents a method invocation on a remote object, and the serialization of in the argument list of that method that must be moved from the local to the remote environment. WSDL [WSD08] describes Web services starting with the messages that are exchanged between the requester and provider agents. WSDL contains information about ports, message types, port types, and other related information for binding two interactions. Therefore, a Web service is essentially a client server framework, wherein the Web service is published at a specific URL and clients request and consume the “service”. Both the client and the server encapsulate their message in SOAP for machine-to-machine interaction via HTTP in XML format.

2.5. .NET Remoting

Microsoft’s .NET framework is an execution environment for Windows programs. It includes two main components: the common language runtime (CLR) and the base class library. The common language runtime is based on the ECMA-335 (European Computer Manufacturers Association) and the ISO 23271 standard [Ram05]. It is the foundation of the .NET framework and provides core services such as memory management, thread management and network Remoting. The base class library is a collection of reusable type that tightly integrates with the common language runtime. In .NET, the programming language can be C#, VB.net, C++/CLI or J#, and the code is first compiled to intermediate language from which is then compiled into machine code at runtime.

Remoting is the process of programs interacting across different processes or machines. Remoting in the .NET framework consists of numerous services that are able to invoke objects anywhere on the network. These objects could be in the same machine, on the same network, or located on a WAN. An application domain is a mechanism, used to isolate executed software applications from one another so that they do not affect each other. When a CLR process is created, it creates an application domain to host the assembly that will be used. The CLR enforces isolation by preventing direct calls between objects residing in different application domains. In order to access these objects, the .NET Remoting mechanism is used.

In .NET Remoting, the client is term denoting a component that needs to communicate with a remote object. The Server receives the request from the client object and responds. The Proxy contains a list of all classes, as well as interface methods of the remote object. It examines if the call made by the client object is valid and whether the remote object resides in the same application domain as the proxy [Ram05]. If this is the case, a simple method call is routed to the remote object. If the object is in a different application domain, the call is forwarded to a RealProxy class by calling its Invoke method. This class is then responsible for forwarding messages to the remote object. The message will pass a serialization layer: the formatter, which then converts it into a specific transfer format such as SOAP or binary. The serialized message later reaches a transport channel, which transfers it to a remote process via a specific protocol like HTTP or TCP. The HTTP channel transports messages to and from remote objects using the SOAP protocol. All messages are passed through the SOAP formatter, where the message is changed into XML and serialized, and the required SOAP headers are added to the data stream, which is then transported to the target URI using HTTP protocol. Conversely, the TCP channel uses a binary formatter to serialize all messages to a binary stream and transports the stream to the target URI using the TCP protocol. It is also possible to configure the TCP channel to the SOAP formatter. On the server side, the message also passes a formatting layer, which converts the serialized format back into the original message and forwards it to the dispatcher. Finally, the dispatcher calls the target object's method and passes back the response values through all tiers.

CHAPTER 3 REVIEW DEVS-BASED SIMULATION TOOL

3.1. The overview of DEVS-based simulation tool

Based on M&S theory and DEVS formalism, various simulation tools have been developed by researchers. Table 1 lists some.

Table 1. DEVS-based simulation tools

Name and Reference	Year	Applied Technique	Explanation
SOADEVS [Mit07]	2007	Service Oriented Architecture	Uses SOA to generate and simulate DEVS models
DEVS/RMI [Zha06]	2006	JAVA RMI	Uses Java RMI to achieve the synchronization of simulators
DEVS/P2P [Che06]	2004	Peer to Peer Network	Uses P2P paradigm to introduce distributed simulations
DEVS/Grid[Seo04]	2004	Grid Computing	Allows DEVS M&S over grid computing infrastructure
DEVSCluster[Kim04]	2004	CORBA	A CORBA-based simulator for DEVS models
DEVS/HLA[Zei99]	1999	High Level Architecture	A HLA-based large-scale distributed M&S
DEVSSJAVA [Sar98]	1998	JAVA	DEVS-based simulation environment written in Java
DEVS-C++ [Zei96]	1996	C++	DEVS-based M&S environment written in C++
CD++ [Rod99] [Wai02][Tro03] [Liu07][Mad07]	1999- 2007	C++ MPICH Web service	Supports parallel conservative, optimistic and web service DEVS and Cell-DEVS simulations

In the development of DEVS-based simulation tool, middleware systems are heavily used for the implementation of parallel and distributed DEVS simulation applications. Middleware is computer software that connects software components or applications. More specifically, middleware consists of a set of enabling services that allow multiple processes to run on one or more machine and interact across a network. CORBA [COR08] is a well-known standard middleware that provides architecture for object-based systems. CORBA-based applications are built with distributed objects that can transparently interact with each other, even if they reside on different nodes in a distributed environment. CORBA objects can be implemented in different programming languages. Their interface has to be defined in a single, language-independent interface description language (IDL). Problem-specific extensions allow additional features to be added to the underlying base architecture.

The DEVSCluster is a CORBA-based distributed DEVS simulation methodology. It is applied to the distributed object technologies as an underlying communication mechanism and transforms a hierarchical DEVS model into a non-hierarchical one, and then applies the simplified non-hierarchical simulation mechanism to the transformed model. As we know, DEVS can consist of two types of models, coupled and atomic. The DEVSCluster translates the information of coupled models into a flat-structured model information class and removes the coupled models from the DEVS model. By this translation, the hierarchical structure of the DEVS model can be flattened and a central scheduler then handle the events generated from all atomic models. A CORBA servant can invoke threads for incoming external messages to access models and simulators.

Though CORBA played an important role in distributed computing history, it had serious technical shortcomings. These include [Hen06]:

- The most obvious technical problem is the complexity of CORBA's API. Many of CORBA's APIs are far larger than necessary.
- C++ language mapping is difficult to use and contains many pitfalls that lead to bugs.
- Design flaws in CORBA's interoperability protocol make it impossible to build a high-performance event distribution service.

- The encoding of CORBA contains a large amount of redundancy, and the protocol does not support compression. This leads to poor performance over wide-area networks.
- The specification ignores threading almost completely, so threaded applications are inherently non-portable.
- CORBA does not support asynchronous server-side dispatch.
- No language mappings exist for C# and Visual Basic, and CORBA has completely ignored .NET.

In recent years, a transformation has been occurring in the architecture of computer-based applications. A new paradigm, which is called “software as a service”, has had a profound impact on the design and development of software. Envisioning software as a service requires a major change in the underlying platform that supports the interoperability of software applications. Both Sun Microsystems and Microsoft have introduced frameworks to support this new component-based, service-oriented architecture. These platforms provide support for software development, deployment, execution and management that facilitate interoperability across servers, development languages and applications. Both J2EE and .NET provide the capabilities to achieve this goal.

Microsoft .NET is a collection of resources that includes development tools and languages, server software and protocols. With the Common Language Runtime and the class library, .NET provides a standard set of data types to perform common functions. .NET has the following features [Ram05]:

- .NET provides means to access other functions, which execute outside the .NET environment.
- All .NET programs are executed under the CLR, which is the virtual machine component of the .NET framework and provides memory management, security, and exception handling properties.
- The .NET framework provides a base class library, which encapsulates a number of common functions, including file reading and writing, graphic rendering, database interaction and XML document manipulation.

- .NET provides a common security model for all applications.
- Microsoft submits the specifications for the Common Language Infrastructure, C# and C++/CLI to both ECMA (European Computer Manufacturer's Association) and the ISO, making them available as open standards. This makes it possible for third parties to create compatible implementations of the framework and its languages on other platforms.

CHAPTER 4 PCD++WIN

PCD++Win is a parallel simulation engine, which takes advantage of the multi-purpose GUI of the Windows MPI middleware for the construction of ad-hoc PC clusters and the configuration of simulation environments.

4.1. Software architecture

Computers in a network use well-defined protocols to communicate. A protocol is a set of rules and conventions between the communicating participants. Implementing these protocols over a network is quite complicated. However, this complex task could be solved by breaking up the task into pieces and solving each piece individually. In the context of networking, this approach of breaking down the task into simpler subtasks is called layering. Therefore, the communication problem is divided into pieces (layers) and each layer concentrates on providing well-defined interfaces.

7	application	message
6	presentation	message
5	session	message
4	transport	message
3	network	packets
2	data link	frames
1	physical	bits

Figure 1. OSI model

The starting point for describing the layers in a network is the International Standards Organization's (ISO) Open Systems Interconnection (OSI) model for computer

communication. This model, shown in Figure 1, was developed between 1977 and 1984 and is intended to serve as a guide, not a specification. It provides a framework in which standards can be developed for the services and protocols at each layer. One advantage of layering is to provide well-defined interfaces between the layers, so that a change in one layer does not affect an adjacent layer.

Following ISO/OSI model, both PCD++ [Tro03] and PCD++Win have a layered architecture, as shown in Figure 2.

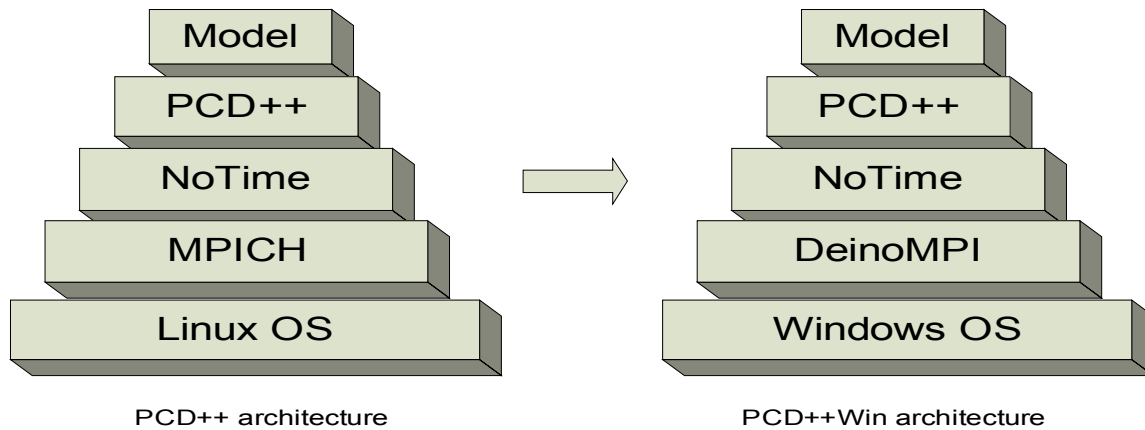


Figure 2. Architecture of PCD++ and PCD++Win

For PCD++Win, which ports PCD++ into Windows cluster environments, the operating system is at the base, serving as the platform for high performance parallel computing. Above the operating system, DeinoMPI provides the communication infrastructure for the workstation clusters with a standard message-passing library. The NoTime kernel [War95], as the part of Time warp protocol, implements NoTime protocol for conservative simulators and organizes the simulation objects. On top of the NoTime kernel, the PCD++ provides a hierarchy of classes that implement the simulation mechanisms. Although DeinoMPI belongs to layer 5 (the session layer), implementations may cover most layers of the reference model with socket and TCP being used in the transport layer. NoTime kernel, as middleware, is part of the presentation and application layer, used to manage logical processes and invoke MPI method call (shown in Figure 3).

PCD++Win	
NoTime	Application Layer
	Presentation Layer
DeinoMPI	Session Layer

Figure 3. PCD++Win and OSI layer

4.2. Parallel DEVS abstract simulator in PCD++Win

PCD++Win, which ports PCD++ into a Windows platform, includes the modeling and simulation frameworks. The modeling framework allows users to define the behavior of atomic and coupled models using C++ programming language and a built-in specification language respectively. In creating DEVS models, modelers need to provide new C++ classes inherited from the abstract atomic model defined in the modeling framework of PCD++Win. Next, modelers need to specify the properties of the cell space with specification language. The simulation framework, creating an executive entity for each component in the model hierarchy, is a set of abstract simulators executed simulation according to the DEVS or Cell-DEVS formalism. Figure 4 shows the PCD++Win major classes from PCD++. The processor class is the parent of all the classes in charge of executing the model including the simulator, coordinator, 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 a class, called MsgAdmin class. The processor would send the message to the MsgAdmin through the send method, which queues the messages until they are 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 methods for the execution of the model, as follows:

- lastChange() reports the time of the last state change;
- nextChange() reports the time of the next state change;

- `absoluteNext()` reports the absolute time of the next change (`lastChange() + nextChange()`);

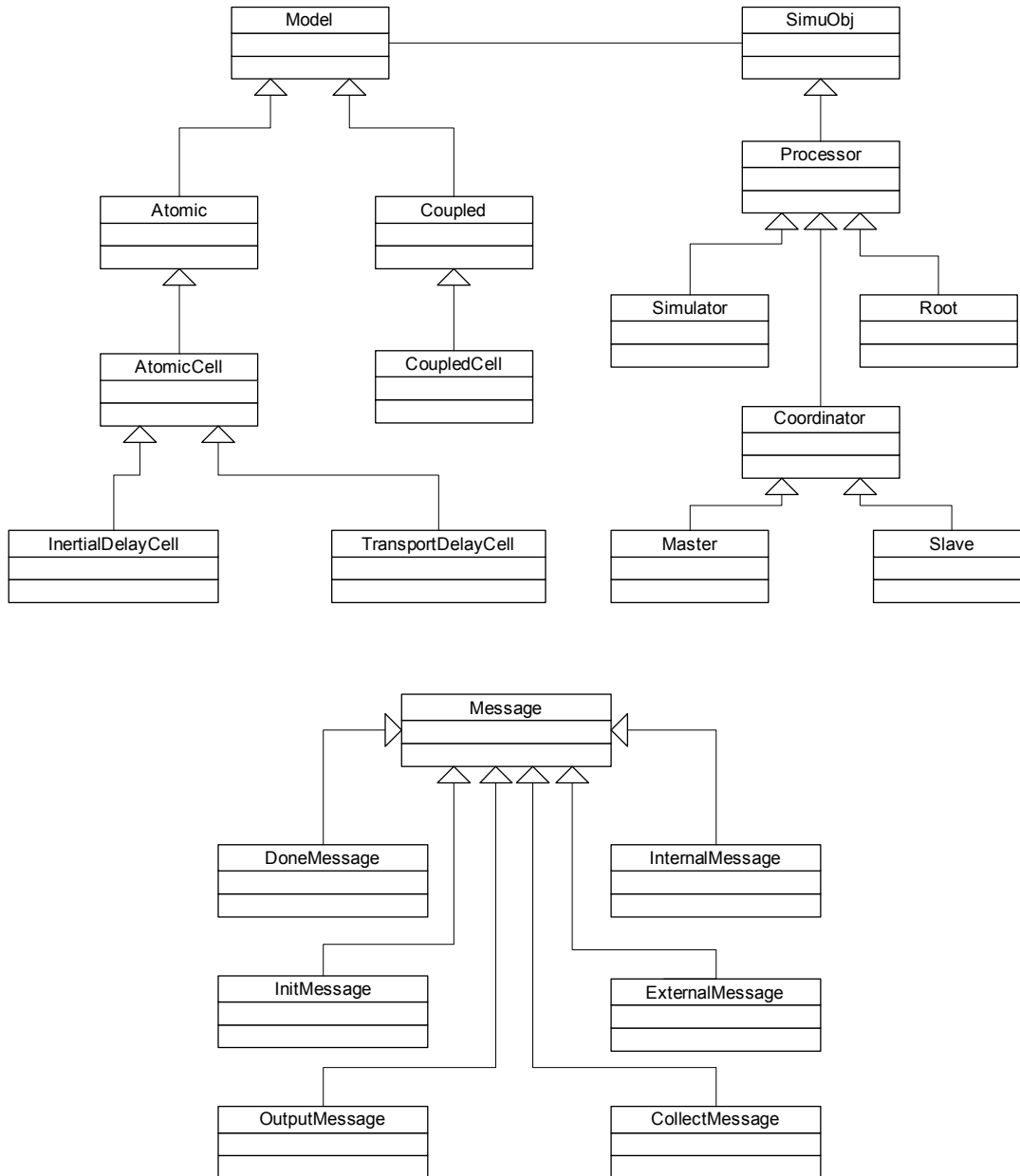


Figure 4. PCD++Win major class diagram

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 message received. 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 itself receives only specific types of messages, but not done or output messages.

Processor classes have a hierarchical structure. The coordinator class may manage several simulators or other coordinators. It 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 behavior of the method is different from that in the simulator class. There are two types of processor: a simulator that executes the simulation of an atomic DEVS model by invoking the atomic model's transition and external event function, and a coordinator that takes the responsibility of translating its children's output events. A coordinator communicates with its child processors through intra-process messages if they reside on the same logical process and through inter-process message if they are sitting on different logical processes. To reduce the communication overhead, a master/slave coordinator structure is used. As a result, when a coupled model is partitioned onto multiple nodes, a coordinator is created on each of them to execute the portion mapped on that specific node. The coordinator on the first node is the master, while all the other coordinators are slaves. The master coordinator is considered the immediate parent of the slaves residing on the other nodes.

Figure 5 shows the 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 methods. The former 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.

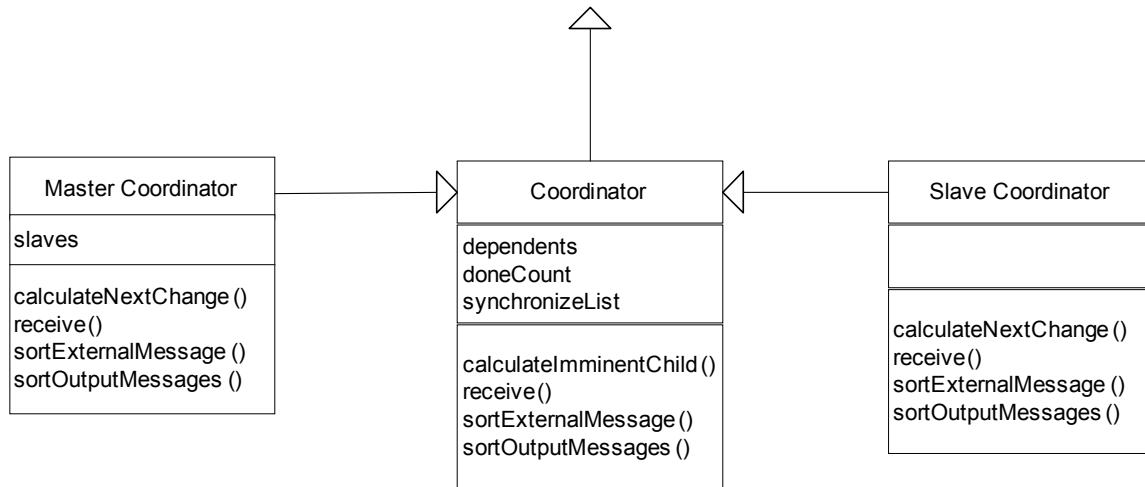


Figure 5. Master and Slave Coordinator function

According to [Cho94a], the internal transition of an atomic model is executed at the next event time for all imminent components receiving no external events. External events generated by these imminent trigger external transitions at receptive non-imminent. Therefore, the coordinator executes the `calculateNextChange` method to evaluate the imminent child processors. In the case of the master coordinator, it considers the local child processors in addition to the remote slave coordinators; in the case of the slave coordinator, it only considers the local child processors. The root coordinator is the main processor in the simulation and it is in charge of:

- starting the simulation through the `simulate` method;
- stopping the simulation through the `stop` method;
- interacting with the environment in terms of inputting external events; and
- advancing the simulation clock

Messages are implemented as separate classes, each representing a message type with all the classes inheriting the `Message` class. The message type includes initialization messages (I), collect messages (@), internal messages (*), external messages (X), done messages (D), and output messages (Y).

The simulation is carried out in a message-driven fashion. Each message has a timestamp that indicates the virtual time of the event. These messages fall into two categories: content messages include the external message (X, t) and output message (Y,

t) that encode the actual data transmitted between the models, while control messages include the initialization message (I, t), collect message (@, t), internal message (*, t) and done message (D, t), which are used to synchronize the simulation.

In [Cho94b], the abstract simulator for the parallel DEVS formalism was introduced. The main additions are the message bags and the confluent transition function. Message bags are used to hold multiple input messages arriving at the model and multiple output messages generated by the model. The confluent function allows the modeler to define the behavior of the model when it receives an external message while being scheduled for internal transition. In such a case, the confluent transition function is executed in place of the internal and external transition functions. According to [Cho94b], “Both δ_{con} and δ_{ext} depend on the events in the bag, x^b . An event in the bag is a result from an output function and all the translations on the event path. An output function depends on a state prior to a transition at the same instance.”

Figure 6 shows the algorithms for simulators. When a simulator receives a (@, t) message from its parent coordinator, it executes the output function defined in the associated atomic model and sends a (Y, t) and a (D, t) to its parent. If a (X, t) is received, the message is cached in the simulator’s message bag. On the other hand, the arrival of (*, t) triggers state transitions in the atomic model based on the simulation time and status of the message bag.

<pre> when a (@, t) message is received if $t = t_N$ then $y := \lambda(s)$ send (y, t) to the parent coordinator send (D, t) to the parent coordinator end if else raise error end when when a (X, t) message is received lock the bag add event X to the bag unlock the bag end when </pre>	<pre> when a (*, t) message is received case $t_L \leq t \leq t_N$ $e := t - t_N$ $s := \delta_{\text{ext}}(s, e, \text{bag})$ empty bag end case case $t = t_N$ and bag is empty $s := \delta_{\text{int}}(s)$ end case case $t = t_N$ and bag is not empty $s := \delta_{\text{con}}(s, \text{bag})$ empty bag end case case $t < t_L$ or $t > t_N$ raise error $t_L := t$ $t_N := t_a(s)$ send (D, t_N) to parent coordinator end when </pre>
---	---

Figure 6. Simulator algorithm [Ch094b][Tro03]

The message-processing algorithms for master coordinators are illustrated in Figure 7. A master coordinator may have three different types of child processors, including the slave coordinators on remote nodes, the local child simulators, and other lower-level master coordinators on the same node. When $(@, t)$ arrives, the master coordinator forwards the message to all imminent child processors and caches the receivers for later state transitions. Then, it waits for (D, t) from each of these receivers. Afterwards, it sends (D, t) with the updated simulation time to its parent coordinator.

<pre> when a (@, t) is received from parent if t = t_N then t_L := t for all imminent child i with min t_N send (@, t) to child i cache i in the synchronize set end for wait until (D, t) received from all child send (D, t) to parent coordinator end if else raise error end when when a (Y, t) is received from child i for all influences, j of child i if j is a local processor X := Z_{i,j}(Y) send (X, t) to child j cache j in the synchronize set else s := coordinator(self, j) if s is not in slave-sync set then send (Y, I, t) to s cache s in the slave-sync set cache s in the synchronize set end if end if end for if y is to be transmitted upward then Y := Z_{i,self}(Y) send (Y, t) to parent coordinator end if clear slave-sync set end when </pre>	<pre> when a (X, t) is received from parent lock the bag add event X to the bag unlock the bag end when when a (*, t) is received from parent if t_L ≤ t ≤ t_N for all X in bag for all receivers of X, j in I_{self} if j is a local processor X := Z_{self,j}(X) send (X, t) to j cache j in the synchronize set else s := coordinator(self, j) if s not in slave-sync set then send (X, t) to s cache s in the slave-sync set cache s in the synchronize set end if end if end for end for clear slave-sync set end for empty bag for all I in the synchronize set send (*, t) to i end for wait until all (D, t_N) received t_L := t t_N := min of components' t_N clear the synchronize set send (D, t_N) to parent else raise error end when </pre>
--	--

Figure 7. Master coordinator algorithm [Cho94b][Tro03]

Three different cases may occur upon the arrival of (Y, t) : if the message is sent to a local receiver, the master coordinator translates it into (X, t) and forwards it to the destination. If the message targets remote receivers, the master coordinator figures out the corresponding slave coordinators, and relays the message to each of them. Otherwise, the message is forwarded to the higher-level parent coordinator. The processing of (X, t) is the same as in the simulators. The master coordinator flushes all external messages in its message bag to their destinations upon the arrival of $(*, t)$. It also sends $(*, t)$ to each child that has a scheduled internal or external state transition. After the state transitions, the master coordinator calculates the next simulation time and sends the information to its parent coordinator in (D, t) .

The slave coordinator handles $(@, t)$, (X, t) , and $(*, t)$ messages in the same manner as the master coordinator. However, they differ in one aspect: whenever (Y, t) has to be sent to a remote receiver, the slave coordinator will forward it to its parent master coordinator. In this master/slave structure, a slave coordinator can have only two types of child processors, namely the local child simulators and lower-level master coordinators (i.e., it will not have other slave coordinators as descendants). Figure 8 shows the slave coordinator algorithm for (Y, t) .

```

when a  $(Y, t)$  is received from child  $i$ 
  sent_to_master := false
  for all influences,  $j$  of child  $i$ 
    if  $j$  is a local processor
       $X := Z_{i,j}(Y)$ 
      send  $(X, t)$  to child  $j$ 
      cache  $j$  in the synchronize set
    else
      if not sent_to_master
        send  $(Y, t)$  to parent coordinator
        sent_to_master := true
      end if
    end if
  end for
  if  $Y$  is to be transmitted upward then
    if not sent_to_master
      send  $(Y, t)$  to parent coordinator
    end if
  end if
end when

```

Figure 8. Slave coordinator algorithm [Cho94b][Tro03]

When (Y, t) arrives, the slave coordinator transforms the message into (X, t) and sends the resulting (X, t) to the local child receivers. If the (Y, t) targets remote receivers on other nodes, the slave coordinator simply forwards the message to its parent master coordinator, which in turn will send the message to other slave coordinators if necessary. Notice that only one (Y, t) is forwarded to the master coordinator, as guaranteed by the `sendToMaster` flag.

```

while  $t \neq \infty$ 
  if  $t = t_N$  of queue
    for all  $q$  in queue with time  $t$ 
      send  $(X, t)$  to topmost coordinator
    end for
  end if

  if  $t = t_N$  of topmost coordinator
    send  $(@, t)$  to topmost coordinator
    wait until  $(D, t)$  is received
  end if

  send  $(*, t)$  to topmost coordinator
  wait until  $(D, t)$  is received

end while
raise simulation completed

```

Figure 9. Root coordinator algorithm [Cho94b][Tro03]

The root coordinator is a special processor that controls the whole simulation and handles events exchanged between the simulated model and the environment. Figure 9 shows the root coordinator algorithm for controlling the simulation, where it sends $(*, t)$ and $(@, t)$ alternatively with potential external events as (X, t) messages to the top-level master coordinator to drive the simulation forward.

In PCD++Win, the above algorithms (or protocols) are implemented with Microsoft Visual Studio 2005. Figure 10 illustrates the abstract simulators of a two-node PCD++Win cluster, which includes a master coordinator and a slave coordinator with connections using Windows MPI. The coordinators and simulators obey the protocols, which allow them to work together in a transparent way. The protocols have to be seen as

abstract schemes without considering implementation details and performance requirements.

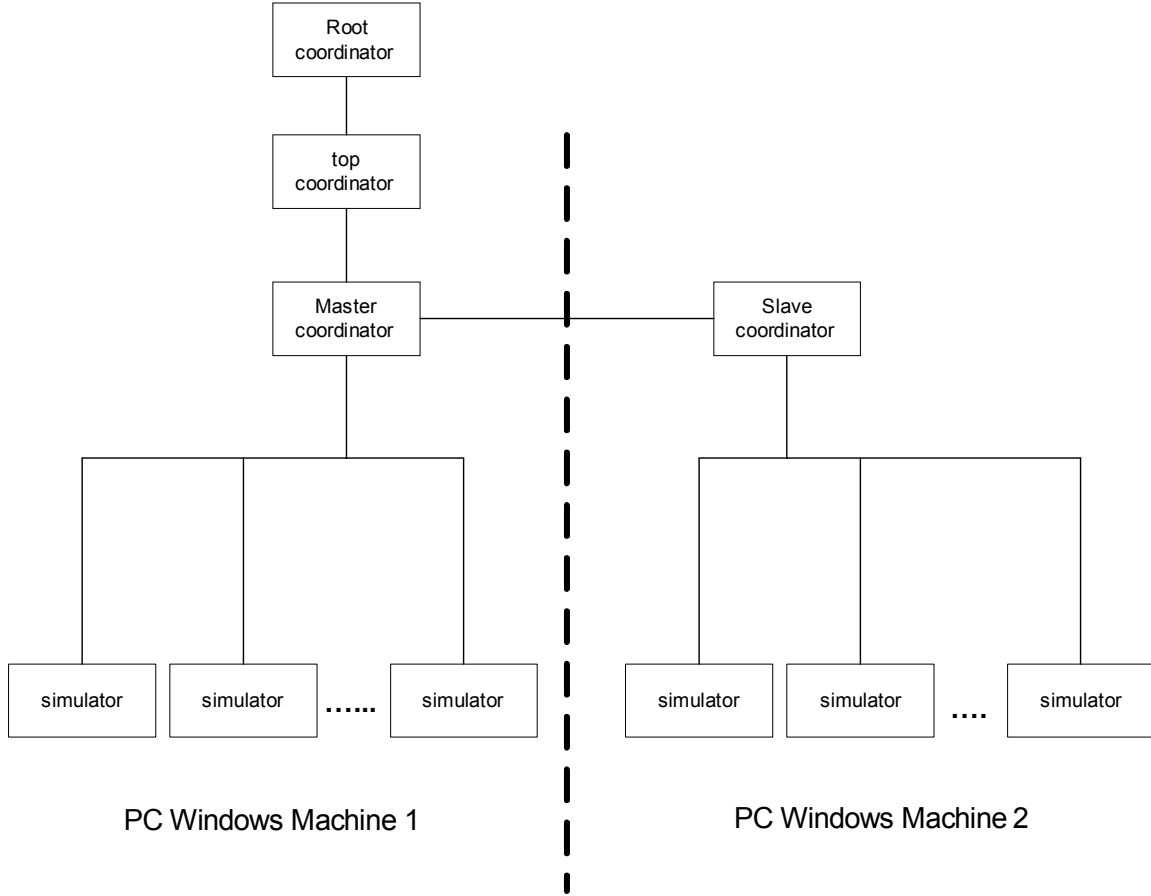


Figure 10. Abstract simulator in PCD++Win

4.3. The NoTime kernel in PCD++Win

The TimeWarp [War95] simulation kernel is a parallel synchronization protocol, which was developed at the University of Cincinnati. As part of TimeWarp, the NoTime kernel is used for parallel and stand-alone simulations that use no synchronization at all. In PCD++Win, the NoTime kernel is compiled by Microsoft Visual Studio 2005. The major functionalities of NoTime for PCD++Win are as follows:

- The interface of PCD++Win. The NoTime presents some interfaces for events, states and simulation objects. Several classes of PCD++Win are derived from these interfaces. PCD++Win is responsible for initializing the simulation objects and defining the activities of each simulation object, while NoTime provides the

basic functions for sending and receiving events between simulation objects. Control is passed between PCD++Win and NoTime through function calls.

- **Event Management.** PCD++Win defines different types of events by deriving from the BasicEvent, which is the class of NoTime. Events are organized in the input and output queues in NoTime. While an output queue is created for each simulation object, a single input queue is shared by all the simulation objects mapped on a logical process.
- **Communication Management.** There are two types of communications in the PCD++Win: message passing between simulation objects residing on different processors (remote communications), and message-passing between simulation objects on the same processor (local communications). Remote communications are controlled by a communication manager and local communications are done via direct function calls.

Figure 11 illustrates the relationship between NoTime and PCD++Win.

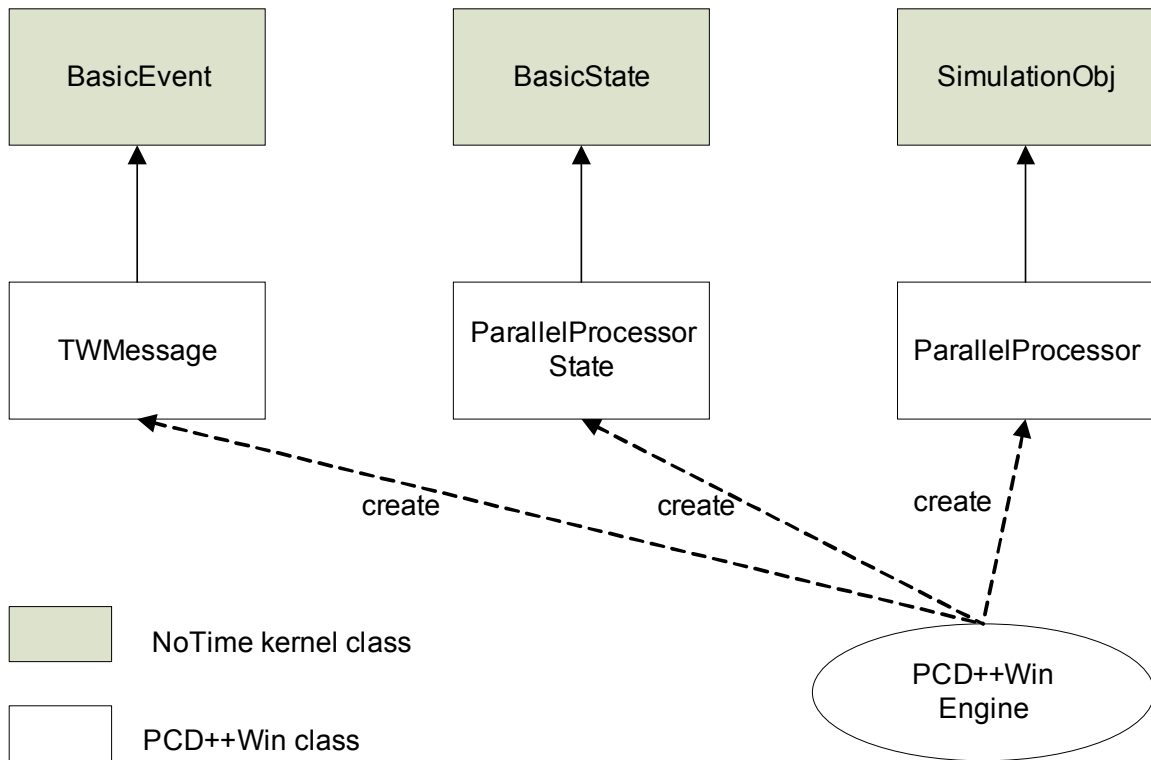


Figure 11. NoTime kernel and PCD++Win

4.4. Running PCD++Win with DeinoMPI interface

By using the DeinoMPI GUI tool, users can easily configure a PC cluster to carry out parallel simulations with PCD++Win. Figure 12 shows the GUI for cluster configuration. The user can add computers on a network to the panel either by scanning the whole network or by specifying their host names. The tool can automatically check the machines and present their status. For instance, Figure 12 shows that both ARS-14 and ARS-7 do not install DeinoMPI package and hence cannot be involved in the cluster. Other machines are available nodes that have all necessary software to carry out parallel simulations. Detailed information about each node, including CPU speed, memory size, disk space and network connectivity, is featured in the panel. With this information, the user can then select the appropriate nodes to form a cluster.

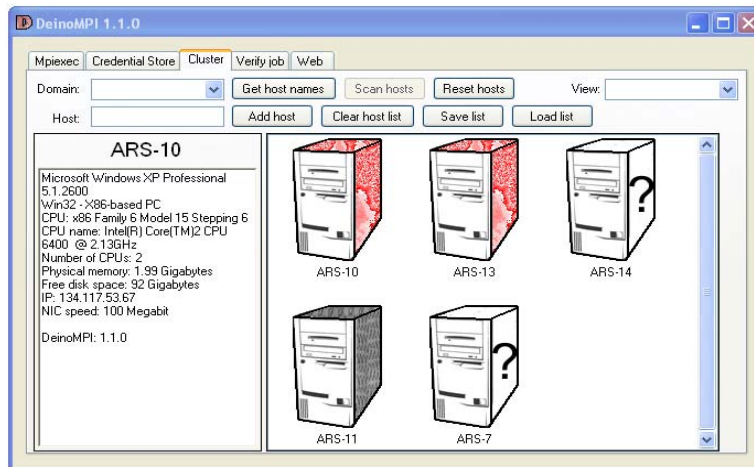


Figure 12. The GUI of DeinoMPI

The main execution window is illustrated in Figure 13. Users can specify the simulation parameters using the GUI tool and then dynamically change the nodes involved in the cluster. The simulation-related information is shown in the window underneath.

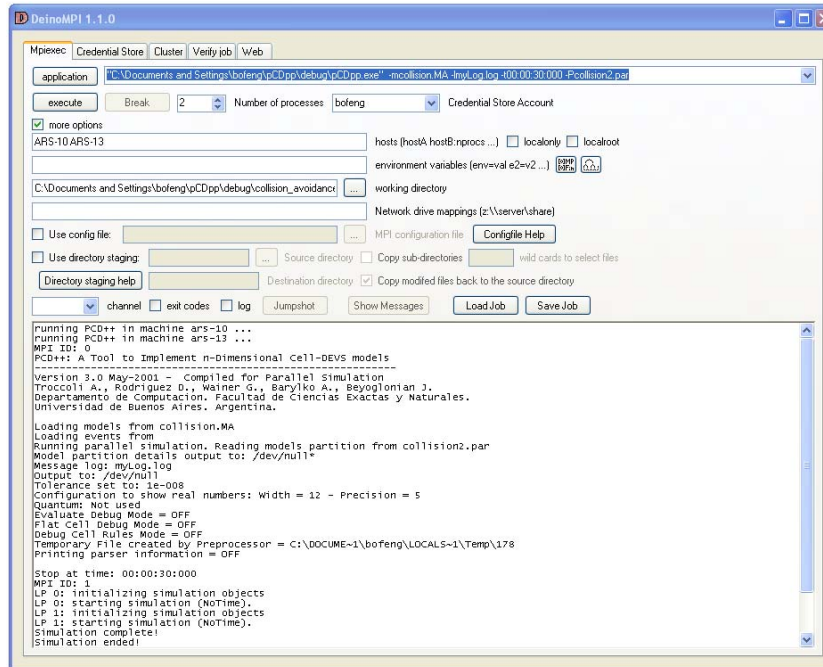


Figure 13. Main windows running PCD++Win

If errors happen in the simulation, users can diagnose the error condition by using the job verification tab. As shown in Figure 14, this tool gives a detailed description of each job and the possible causes of the failure.

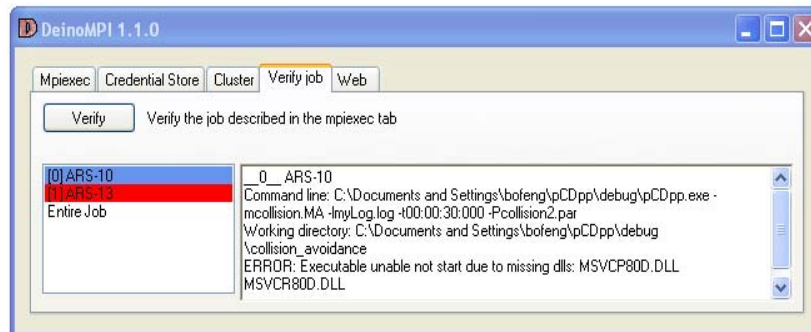


Figure 14. Job verification tool

As we can see, PCD++Win provides a user-friendly environment for conducting parallel simulations by leveraging the easy-to-use GUI tool of DeinoMPI. Windows-based PCs interconnected via a LAN can be used to form a cluster platform for parallel simulations. Therefore, PCD++Win makes advanced simulation technologies available not only to users in a traditional office environment equipped with wired desktop PCs, but also to practitioners on the move working on laptops connected by wireless networks.

4.5. Performance Metrics

The most commonly used performance metric for parallel computing is speedup, which signifies the performance gain of parallel processing versus sequential processing. However, with different emphases, speedup has been defined differently. One definition focuses on how much faster a problem can be solved with N processors. Thus, it compares the best sequential algorithm with the parallel algorithm under consideration. This definition is referred to as absolute speedup. Another speedup, called relative speedup, deals with the inherent parallelism of the parallel algorithm under consideration. It is defined as the ratio of elapsed time of the parallel algorithm on one processor to elapsed time of the parallel algorithm on N processors. The reason for using relative speedup is that the performance of parallel algorithms varies with the number of available processors. It gives information on the variations of parallelism to compare the algorithm itself with different numbers of processors. Two well-known speedup formulations have been proposed based on relative speedup. One is Amdahl's law [Amd67] and another is Gustafson's scaled speedup [Gus88]. Amdahl's law has a fixed problem size and is interested in how fast the response time could be. It suggests that massively parallel processing may not gain high speedup. Under the influence of Amdahl's law, many parallel computers have been built with a small number of processors. Gustafson approaches the problem from another point of view. He is interested in how large a problem could be solved within this time. In this work, we use the following definition for speedup:

$$\text{Overall Speedup} = T(1)/T(N)$$

$T(N)$ represents the total execution time taken by the simulation running on N nodes, and $T(1)$ represents the best possible execution time measured on one node with the same algorithm used in $T(N)$. In [Amd67], the author states that a small portion of the program which cannot be parallelized will limit the overall speedup, and any program will typically consist of several parallelizable parts and several non-parallelizable (sequential) parts. This means that if a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule, regardless of how many

processors are added. Therefore, no program can run more quickly than the longest chain of dependent calculations, since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

4.6. Experimental results of PCD++Win

This section presents a performance analysis of PCD++Win. PCD++Win was executed on a group of desktop workstations, which are Intel Core 2 Duo Processor E6400 @ 2.13 GHz, 2GB DDR2-Synch DRAM machines, running Microsoft Windows XP Professional connected through a LAN and communicate with DeinoMPI 1.1.0. The log data generated during the simulation were stored to the local file system on each workstation.

Two models were used during the performance analysis. One of the models was a fire spreading in a forest, and it was implemented as a 20×20 coupled Cell-DEVS model [Ame01]. The other was a collision avoidance model [Wai08], consisting of robots encountering obstacles as they traversed a specific area.

The fire model was composed of 20×20 cell space, with each cell representing a square area of the forest. The cell was considered to be burned if its temperature exceeded a specific value. The delay time was 100 ms. Figure 15 shows the model definition.

```
[top]
components : Fire

[Fire]
type : cell
dim : (20,20)
delay : inertial
defaultDelayTime : 100
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

[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 15. Fire model definition [Wai08]

The cell space uses inertial delay. The neighborhood of the cell is defined by the neighbors construct; the cell is neighborhood by 8 cells from all sides. Fire(-1,-1) represents the cell in the North West side, Fire(0,-1) represents the cell in the west, etc. The rules that define the state of the cells in each simulation cycle are defined using the local transition construct. There are rules defining the time it takes for the cell to be burned if one of its neighbors is burned. For example, the first rule dictates that if the cell in the southwest side of the cell is burned, the cell will take $(21.552615/17.967136) \times 60000$ milliseconds to burn. 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 by the speed of the fire, the time it takes for fire to spread is evaluated in minutes and by multiplying it by 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) once the delay elapses.

A simple partition strategy was used in the Fire model testing (as shown in Figure 16). It evenly divides the cell space into horizontal rectangles and each partition is run by one PC workstation. Figure 17 shows the execution time and overall speedup for the Fire model with PCD++Win.

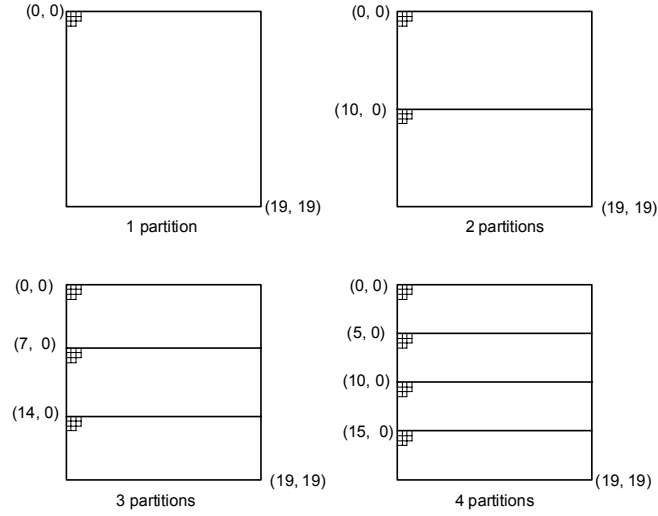


Figure 16. Partition strategy

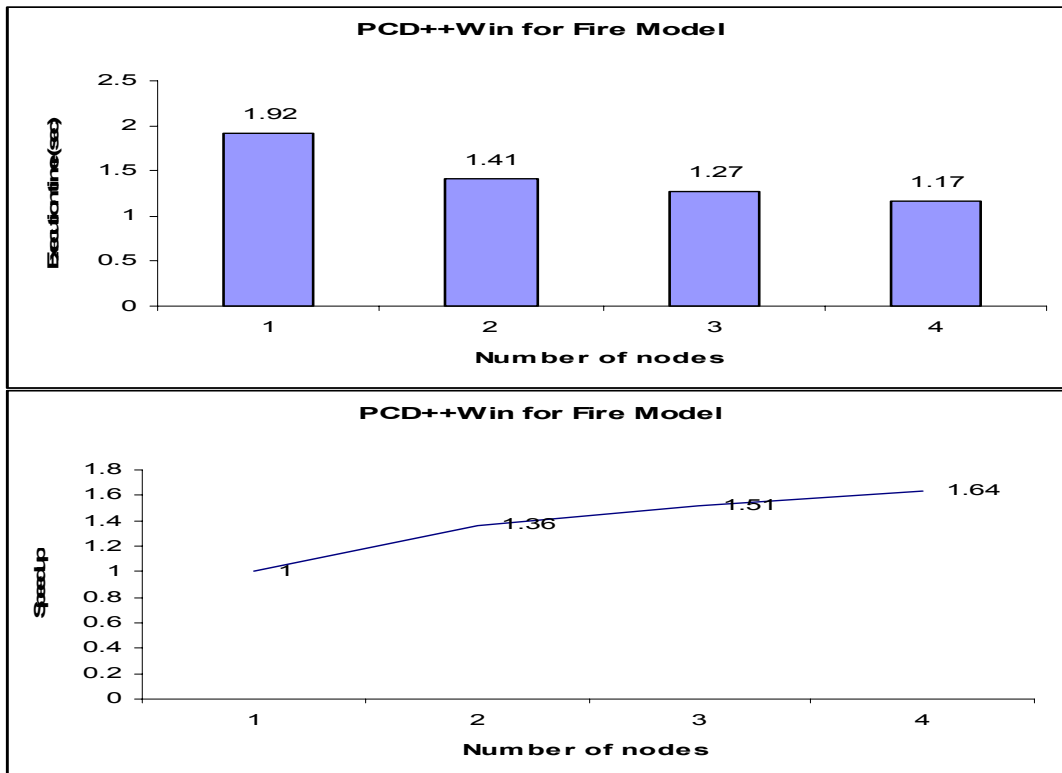


Figure 17. The experiment result of Fire model

Figure 17 illustrates the speedups obtained by the PCD++Win simulator using 1 to 4 processors for the fire model. It can be seen that the execution time decreases with increasing computing nodes. For instance, the execution time decreases from 1.92 to 1.17

seconds when the number of nodes climbs from 1 to 4. The shortest execution time is achieved on 4 nodes, which is the most we used. This result, of course, comes first from the enough parallelism of the model. In increasing the nodes, each partition of the model has enough workload to compensate for the cost of increased communication incurred by Windows MPI calls across LAN.

The second model describes that Unmanned Aerial Vehicles (UAV) encountering static and dynamic obstacles when they traverse a specific area. It is necessary for the UAV to be able to avoid such obstacles and continue its mission. Figure 18 represents the state of the model. Here, state 1 represents a UAV (Robot), state 9 represents various static obstacles and state 5 represents dynamic obstacles. The UAV's mission is to cross the area from top to bottom while avoiding all encountered obstacles.

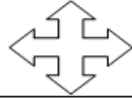

	Empty Cell	Robot	Dynamic Obstacle	Static Obstacle
Movement	None			None
State	0	1	5	9

Figure 18. State of collision avoidance model

The model can be tested for multiple UAVs running at the same time, and the results were verified for a specific set of scenarios based on the condition that there is an appropriate separation between the UAV, so that they don't interfere with one another. The model is initialized in the model file, as shown in Figure 19, and the starting position of the entities can be changed by reinitializing the model.

```
[top]
components : uav

[uav]
type : cell
dim : (20, 20)
delay : transport
defaultDelayTime : 100
border : wrapped

neighbors : uav(-2,-2) uav(-1,-2) uav(0,-2) uav(1,-2) uav(2,-2)
```

```

neighbors : uav(-2,-1) uav(-1,-1) uav(0,-1) uav(1,-1) uav(2,-1)
neighbors : uav(-2, 0) uav(-1, 0) uav(0, 0) uav(1, 0) uav(2, 0)
neighbors : uav(-2, 1) uav(-1, 1) uav(0, 1) uav(1, 1) uav(2, 1)
neighbors : uav(-2, 2) uav(-1, 2) uav(0, 2) uav(1, 2) uav(2, 2)
neighbors : uav(3, -2) uav( 3,-1) uav(3, 0) uav(3, 1) uav(3, 2)

initialvalue : 0
initialrowvalue : 0      0000010000000000000000
initialrowvalue : 5      0000000009990000000000
initialrowvalue : 9      0000909000000000090900
initialrowvalue : 10     0009999900000000099900
initialrowvalue : 15     0000000090000000000000
initialrowvalue : 17     0099900000009990000000

```

Figure 19. The first part of collision avoidance model [Wai08]

The model execution conforms to a set of rules that specify how the model behaves under certain conditions. Figure 20 shows a set of possible rules that the UAV obeys:

```

localtransition : uav-rule

[uav-rule]
rule : 1 100 { (0,0)=0 and (0,-1)=0 and (0,1)=0 and (-1,0)=1 }
rule : 0 100 { (1,0)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,-1)=0 and (0,1)!=0 and (-1,0)=1 }
rule : 0 100 { (1,0)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,-1)!=0 and (1,-2)=0 and (1,0)=0 and
               (0,-1)=1 and (2,-2)!=0 and (2,0)=0 }
rule : 0 100 { (0,1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,1)!=0 and (1,2)=0 and (1,0)=0 and
               (0,1)=1 and (2,2)!=0 and (2,0)=0 }
rule : 0 100 { (0,1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,1)!=0 and (1,2)=0 and (1,0)=0 and
               (0,1)=1 and (2,2)=0 and (2,0)=0 }
rule : 0 100 { (0,1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,1)!=0 and (1,2)=9 and (0,1)=1 and
               (1,0)=0 }
rule : 0 100 { (0,-1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,-1)!=0 and (0,1)=0 and (-1,0)=1 }
rule : 0 100 { (1,0)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,1)=1 and (1,1)=0 and (2,1)!=0 and
               (1,0)!=0 and (1,2)!=0 and (2,0)=0 and (2,2)=0 }
rule : 0 100 { (0,-1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,1)!=0 and (0,-1)!=0 and (1,0)=0 and
               (1,1)!=0 and (1,-1)=0 and (-1,0)=1 }
rule : 0 100 { (1,0)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,1)=1 and (1,1)=0 and (2,1)=9 and
               (1,0)!=0 and (1,2)!=0 and (2,0)=0 and (2,2)!=0 }
rule : 0 100 { (0,-1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (0,1)!=0 and (0,-1)!=0 and (1,0)=0 and
               (1,1)=0 and (1,-1)!=0 and (-1,0)=1 }
rule : 0 100 { (1,0)=1 and (0,0)=1 }

```

```

rule : 1 100 { (0,0)=0 and (0,1)=1 and (1,1)=0 and (2,1)!=0 and
               (1,0)!=0 and (1,2)!=0 and (2,0)!=0 and (2,2)!=0}
rule : 0 100 { (0,-1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,-1)!=0 and (1,-2)!=0 and (1,0)=0 and
               (0,-1)=1}
rule : 0 100 { (0,1)=1 and (0,0)=1 }
rule : 1 100 { (0,0)=0 and (1,-1)!=0 and (1,-2)!=0 and (1,0)!=0 and
               (0,-1)=1}
rule : 0 100 { (0,1)=1 and (0,0)=1 }

```

Figure 20. The second part of collision avoidance model [Wei08]

In Figure 20, there are a set of rules for the UAV. For example, Figure 21 shows the precondition and result for “*rule : 1 100 { (0,0)=0 and (0,-1) and (0,1)=0 and (-1,0)=1}*”, which makes the UAV move one step from left to right.

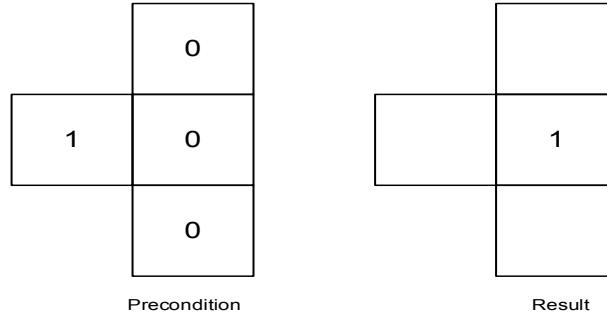


Figure 21. One of UAV's rules

The model is again 20*20 cell spaces. Therefore, the partition strategy shown in Figure 16 is also used for execution the simulation of collision avoidance mode. Figure 22 shows the results.

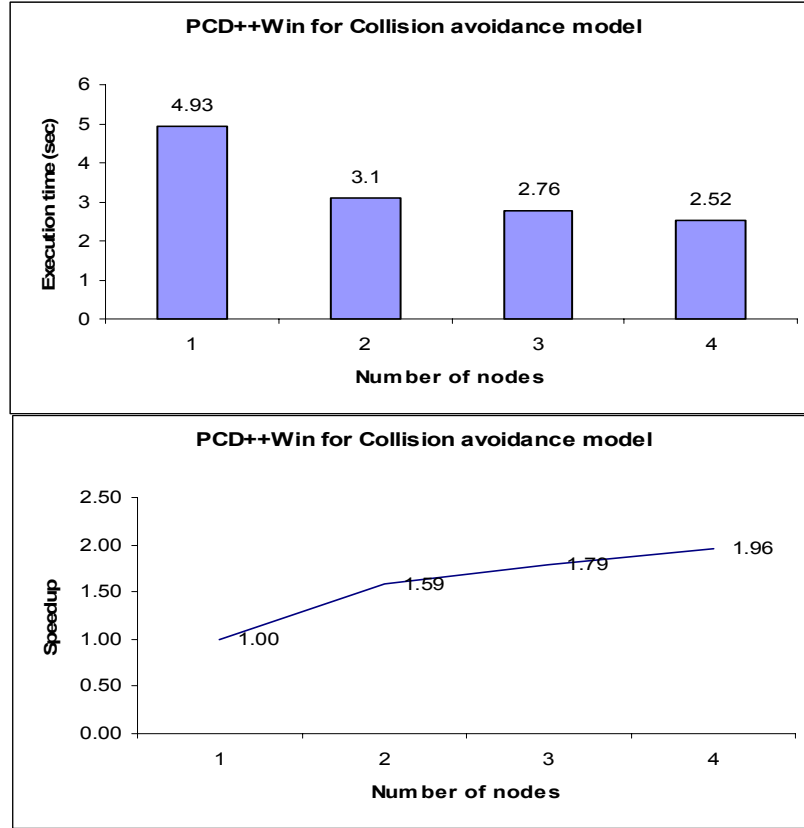


Figure 22. The result of the collision avoidance model

From Figure 22, it can be seen that the speedup of the collision avoidance model is achieved by increasing the number computing nodes. For example, the simulation time decreases by 59% when the node increased from 1 to 2. However, when the number of nodes increases further, the downward trend of execution time is less steep. This means that the overhead involved in inter-LP communication increases when the model is partitioned onto more nodes. In fact, the downward trend of execution time is even reversed if a certain number nodes are used. Therefore, choosing the appropriate number of nodes to execute a given model is a trade-off between the benefits of a higher degree of parallelism and the overhead cost.

Besides the speedup of execution time, the PCD++Win provides a user-friendly environment for conducting parallel simulations by leveraging the easy-to-use GUI tool of DeinoMPI. Also, several techniques are used to enhance the modularity, portability and capability. The following is a summary of the major efforts:

- Modular software using dynamic link libraries (DLL): a DLL contains the implementation of a shared library that allows for modular software development and promotes code reuse. We compiled the NoTime kernel as a separate DLL module, which is dynamically linked to the PCD++Win at runtime. This approach reduces the memory footprint of PCD++Win and makes it possible to switch to other middleware technologies in future developments.
- Multi-platform compilation with preprocessor macros: although PCD++Win is intended for Windows-based PC clusters, the code was written to be compiled on Linux OS with slight revision. This is achieved by defining preprocessor macros in the source code and using conditional compilation to generate appropriate versions for different platforms, increasing the portability of the toolkit.
- Porting code from GCC (GUN Compiler Collection) to Microsoft Visual Studio: PCD++Win was developed based on a PCD++ conservative simulation engine [Tro03], which uses GCC on Linux systems. The source code was ported to Microsoft Visual Studio, which supports the .NET framework. As a result, it is more convenient for further extension.

CHAPTER 5 EXPOSING PCD++WIN AS WEB SERVICE

Service Oriented Architecture (SOA) is a framework consisting of various W3C standards in which various computational components are available as “services”. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. In this chapter, we present a way of exposing PCD++Win as a Web service, which can be consumed by another application on the Internet.

5.1. SOAP and WSDL

In Chapter 2, we mentioned both Web service and SOAP (Simple Object Access Protocol), which forms the foundation of the Web service. A SOAP message represents a method invocation on a remote object, and the serialization of in the argument list of the method, which must be moved from the local to the remote environment. SOAP is a protocol for exchanging XML-based messages over networks, normally using HTTP/HTTPS and SMTP. There are several different types of messaging patterns in SOAP, but by far the most common is the Remote Procedure Call (RPC) pattern, in which the client sends a request message to the server and the server immediately sends a response message to the client. SOAP is platform independent and language independent. Therefore, SOAP removes the requirement that two systems must run on the same platform or be written in the same programming language.

WSDL (Web Service Definition Language) is an XML-based language that provides a model for describing Web service. WSDL describes Web services starting with messages that are exchanged between the requester and provider agents. WSDL contains information about ports, message types and other relating information for binding two interactions. WSDL is often used in combination with SOAP and XML to provide web services over the internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. The client can then use SOAP to call one of the functions listed in the WSDL.

5.2. Building a PCD++Win Web Service

We can mix Web services to perform complex function with minimal programming. There are many ways that a requester entity might engage and use a Web service. In this work, the following steps are required:

1. The requester and provider entities of PCD++Win become known to each other;
2. The requester and provider entities somehow agree on the service description and semantics that will govern the interaction between the requester and provider agents;
3. The service description and semantics of PCD++Win are realized by the requester and provider agents;
4. The requester and provider agent of PCD++Win exchange messages, thus performing some task on behalf of the requester and provider entities.

We use the NetBeans tool [Net07], which is an open-source integrated development environment written entirely in Java, to build the PCD++Win Web service. NetBeans supports the development of all Java application types (J2SE, Web application, Web service, Enterprise JavaBean and mobile applications), and includes a Sun Java System Application Server that provides the underlying core functionality necessary for the development and deployment of application. The Sun Java System Application Server is a fully featured Java platform application server providing the foundation for building scalable and manageable applications and allows us to develop the Web service with JAX-RPC (Java API for XML-Based Remote Procedure Calls). Therefore, we use the JAX-RPC programming model to develop a SOAP-based PCD++Win Web service.

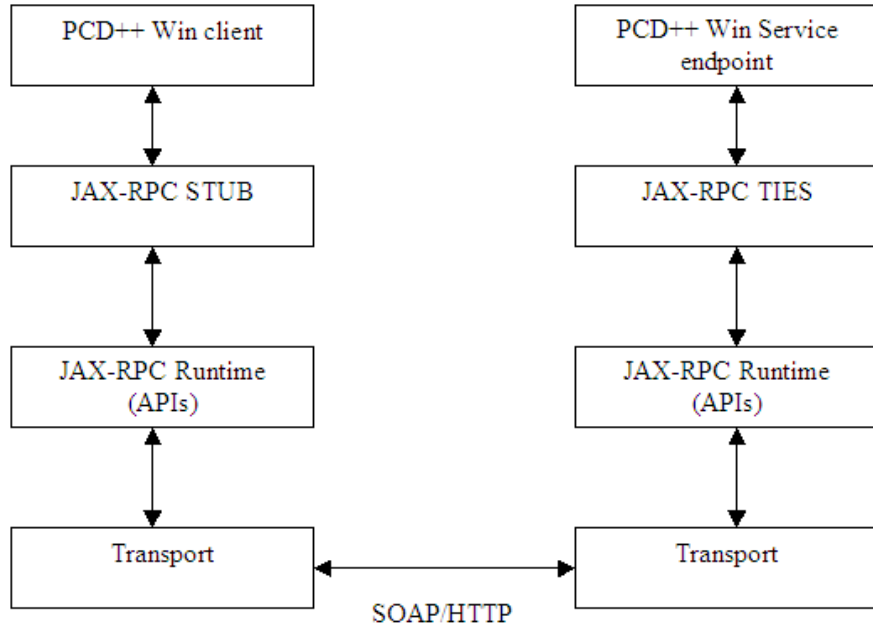


Figure 23. PCD++Win JAX-RPC Web service

Figure 23 shows the architecture of the PCD++Win Web service with JAX-RPC, which provides a generic mechanism that enables developers to create and access Web services by using XML-based Remote Procedure Calls. The current release of the JAX-RPC Reference Implementation uses SOAP as the application protocol and HTTP as the underlying transport protocol. We published the PCD++Win Web services by providing the WSDL definitions for these services. The JAX-RPC specification defines a mapping of Java types to WSDL definitions. When a client locates a service in an XML registry, the client retrieves the WSDL definition to get the service interface definition.

In Figure 23, the PCD++Win service side contains a JAX-RPC service runtime environment and a service endpoint. The client side contains a JAX-RPC client runtime environment and a client application. The JAX-RPC client and service runtime systems are responsible for sending and processing the remote method call and response, respectively. The PCD++Win client creates a SOAP message to invoke the remote method and the PCD++Win service runtime transforms the SOAP message to a Java method call and dispatches the method call to the service endpoint. JAX-RPC hides the details of SOAP from the developer because the JAX-RPC service/client runtime environments perform the mapping between remote method calls and SOAP messages.

STUBS and TIES are classes that enable communication between the PCD++Win service endpoint and the client. The STUB class sits on the client side, between the service client and the JAX-RPC client runtime system. The STUB class is responsible for converting a request from a JAX-RPC service client to a SOAP message and sending it to the service endpoint using the HTTP protocol. It also converts the response from the service endpoint to the format required by the client. Similarly, the TIES class resides on the server side, between the service endpoint and JAX-RPC runtime system. The TIES class handles the marshalling and unmarshalling the data between the service endpoint class and the SOAP format.

Using JAX-RPC, PCD++Win clients and web services have a big advantage: the platform independence of the Java programming language. In addition, JAX-RPC is not restrictive a JAX-RPC client can access a web service that is not running on the Java platform, and vice versa. This flexibility is possible because JAX-RPC uses technologies defined by HTTP, SOAP and WSDL.

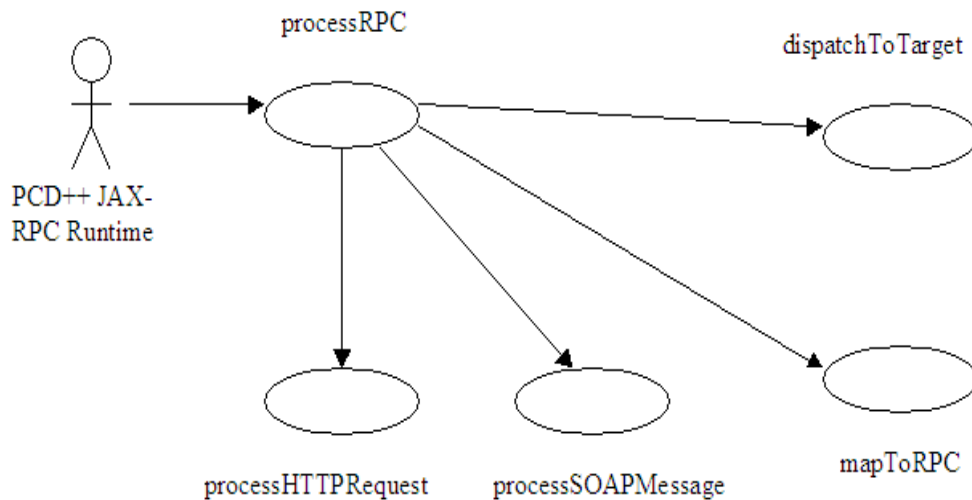


Figure 24. PCD++Win Web service server-side runtime

Processing of the remote method call on the PCD++Win Web Service server side includes the following steps (as shown in Figure 24):

- Processing of the HTTP request: PCD++Win Server-side JAX-RPC runtime system receives and processes the HTTP request.

- Processing of the SOAP message: PCD++Win JAX-RPC runtime system extracts the SOAP message from the received HTTP request and processes the SOAP message to get access to the SOAP envelope, header, body and any attachments. The processing of the SOAP message uses streaming parser XML processing.
- Mapping of the SOAP message to a remote method call: the PCD++Win JAX-RPC runtime system maps the received SOAP message to a method invocation on the target service endpoint. SOAP body elements carry parameters and return a value for this remote call.
- Dispatch to the target JAX-RPC service endpoint: the PCD++Win service JAX-RPC runtime system invokes a method on the target service endpoint based on the mapping of the received remote method invocation. Return values, out parameters and exceptions are carried in the SOAP body and fault elements respectively and are processed as part of the HTTP response.

In creating a PCD++Win Web service with JAX-RPC, we have to define a service interface which will be converted to a WSDL file later. A service interface definition is an abstract or reusable service definition that may be instantiated and referenced by multiple service implementation definitions. In WSDL, the service interface contains elements that comprise the reusable portion of the service description: binding, portType, message and type elements as depicted in Figure 25. In the portType element, the operations of the PCD++Win web service are defined. The operations define what XML messages can appear in the input and output. The message element specifies which XML data types constitute various parts of a message. The message element is used to define the abstract content of messages that comprise an operation. The use of complex data types within the message is described in the types element. The binding element describes the protocol, data format, security and other attributes for a particular service interface.

The service implementation definition describes how a PCD++Win service interface is implemented by the service provider. It also describes its location so that a requester can interact with it. In WSDL, a Web service is modeled as a service element.

A service element contains a collection of port elements. A port associates an endpoint (URL) with a binding element from a service interface definition.

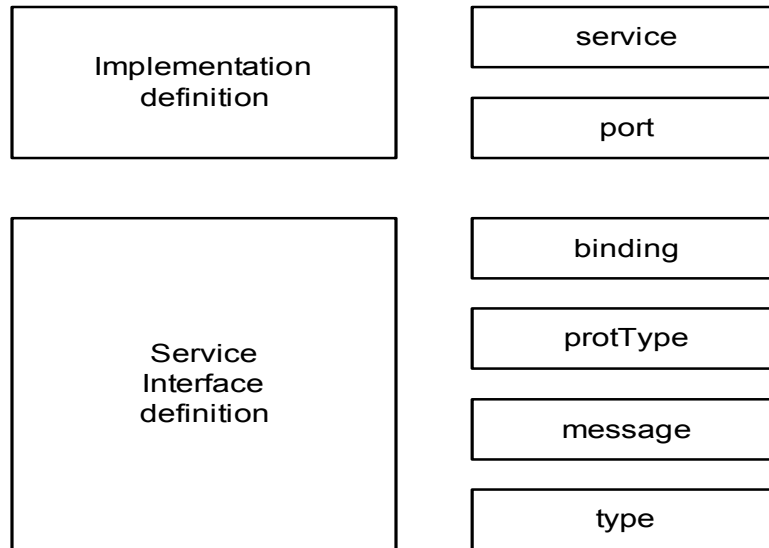


Figure 25. WSDL service interface and implementation

The service interface definition together with the service implementation definition makes up a complete WSDL definition of the service. This pair contains sufficient information to describe to the service requestor how to invoke and interact with the web service. The WSDL file of the PCD++Win Web Service is shown in Figure 26.

```

1  <binding name="PCDWinPortBinding" type="tns:PCDWin">
2    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
3      style="document"/>
4    <operation name="execSimu">
5      <soap:operation soapAction=""/>
6      <input>
7        <soap:body use="literal"/>
8      </input>
9      <output>
10       <soap:body use="literal"/>
11     </output>
12   </operation>
13 </binding>
14 <service name="PCDWinService">
15   <port name="PCDWinPort" binding="tns:PCDWinPortBinding">
16     <soap:address location="http://mygreenland.cunet.carleton.ca:8080/
17       PCDWinService/PCDWin"
18       xmlns:wSDL="http://schemas.xmlsoap.org/wsdl/"
19       xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/">
20   </port>
21 </service>

```

Figure 26. WSDL file of the PCD++Win web service

The WSDL file defines a single network endpoint and a set of ports. In this case, we define a Web service, called “*PCDWin*”, which includes a function, namely “*execSimu*”. Whenever *execSimu* is invoked remotely by a client, the following java code will be executed in the server side:

```
Runtime.getRuntime().exec( "cmd /c " + command);
```

Here, “command” is a batch file, which tells PCD++Win to execute a parallel DEVS or Cell-DEVS simulation.

5.3. Consuming PCD++Win Web Service

To consume a PCD++Win Web service, we need to create a PCD++Win Web service client. For this, NetBeans provides a client creation facility - a Web service client wizard that generates code for looking up a web service and developing a Web service client.

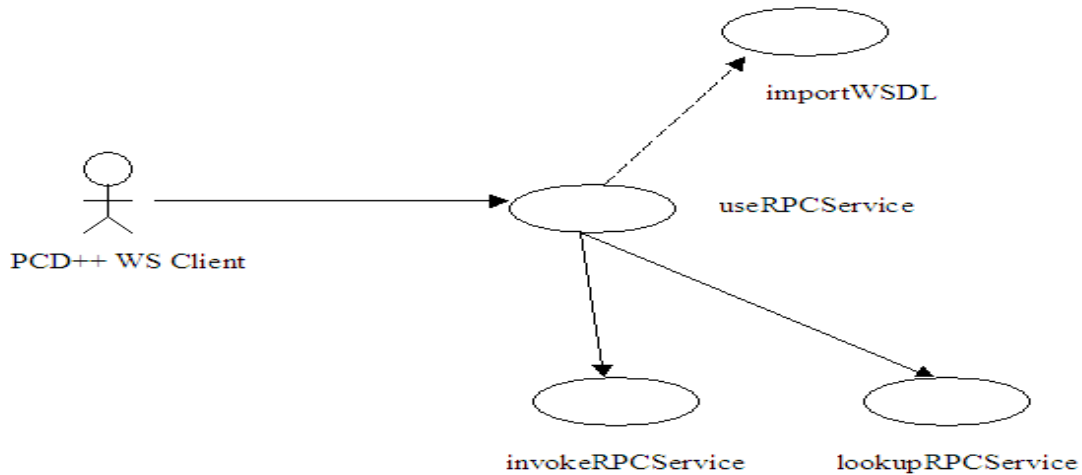


Figure 27. PCD++Win Web service client

In Figure 27, the PCD++Win Web service client is shown. The client uses a WSDL document to import the PCD++Win Web service. A WSDL-to-Java mapping tool generates client-side artifacts (including stub class, service endpoint interface and additional classes) for the PCD++Win service and its ports. Then, client lookups and

invokes the PCD++ Win Web Service. Figure 28 shows the piece of code that illustrates how a Web service client invokes a remote method.

```
try{  
    cdpp_client.PCDWinService myService = new cdpp_client.PCDWinService_Impl();  
    cdpp_client.PCDWin myPort = myService.getPCDWinPort();  
    myPort.execSimu() ;  
}
```

Figure 28. Web service client invokes a remote method

The processing of a remote method call includes the following steps:

- Mapping of a remote method call to the SOAP message representation: this includes mapping of the parameters, return values and exceptions to the corresponding SOAP message. The serialization and deserialization are based on the mapping between Java types and XML data types.
- Processing of the SOAP message: this includes the processing of the SOAP message based on the mapping of the remote method call to the SOAP representation; the encoding style defines how parameters, return values and types in a remote method call are represented in the SOAP message.
- Processing of the HTTP request: this includes the transmission of the SOAP request and response message as part of an HTTP request; a SOAP response message is transmitted as an HTTP response.

Figure 29 shows the GUI of the PCD++Win Web service client

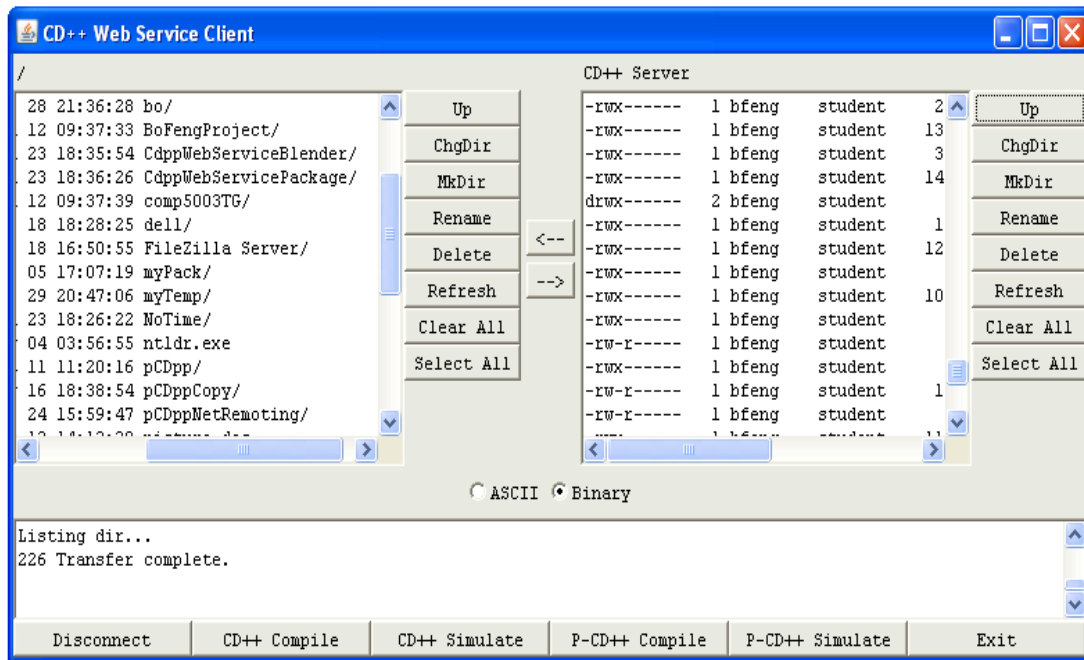


Figure 29. PCD++Win Web service client GUI

In Figure 29, the application includes a FTP function that can upload the model file and download simulation log file. CD++ Compile and P-CD++ Compile are two functions, that can invoke GCC 2.7 and Microsoft Visual Studio 2005 to compile the C++ file respectively. CD++ Simulate and P-CD++ Simulate can invoke CD++ and PCD++Win respectively. The users can also build their own client application by using WSDL. Such applications must first look up the service, and then makes a call to the service and process any return data. Here, SOAP provides a standard framework for packaging and exchanging XML messages. The PCD++Win Web service is published at a specific URL; clients can then request and consume the service. Both the client and the server encapsulate their message in SOAP for machine-to-machine interaction via HTTP in XML format.

CHAPTER 6 PCD++/.NET

In this chapter, we present PCD++/.NET, which is a new approach to implementing distributed simulations based on Microsoft's .NET Remoting technique. We first give an overview of various distributed paradigms and then introduce PCD++/.NET architecture and its main components. Some performance results are discussed in Chapter 7.

6.1. An overview of distributed paradigms

For the past twenty years, distributed paradigms have experienced rapid growth with the development of the Internet, the demand for high performance distributed computing and the boost of enterprise applications. A brief overview of distributed paradigms is as follows:

DCE/RPC Distributed Computing Environment (DCE) [DCE08], designed by the Open Software Foundation (OSF) during the early 1990s, was created to provide a collection of tools and services that would allow easier development and administration of distributed applications. The DCE framework provides several base services such as Remote Procedure Calls (DCE/RPC), Security Services, Time Services, and so on. DCE/RPC is the foundation for many current higher-level protocols.

CORBA This stands for Common Object Request Broker Architecture [COR08], which was designed by the Object Management Group (OMG). CORBA's aim is to be the middleware of choice for heterogeneous systems. CORBA is only a collection of standards that enables software components written in multiple computer languages to run on multiple computers. CORBA uses an interface description language (IDL) to specify the interfaces that objects will present to the outside world. CORBA then specifies a mapping from IDL to a specific implementation language. The benefits of CORBA are language and Operating System independence.

DCOM Distributed Component Object Model (DCOM) [DCO08] is an extension that fits in the Component Object Model (COM) architecture. DCOM allows for component-oriented application development and uses a pinging process to manage the object's lifetimes; all clients that use a certain object will send messages after certain intervals. When a server receives these messages, it knows that the client is still alive; otherwise, it will destroy the object.

Java RMI Java Remote Method Invocation (Java RMI) [RMI08] is a Java interface for performing the object equivalent of remote procedure calls. Java RMI uses a proxy/stub compilation cycle. In contrast to DCE/RPC and DCOM, the interfaces are not written in an abstract IDL (Interface Description Language) but in Java.

Java EJB Enterprise Java Beans (EJB) [EJB08] was developed by Sun Microsystems. Unlike CORBA, which is only a standard, EJB comes with a reference implementation. EJB is the server-side component architecture for Java Platform and enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology. EJB specification intends to provide a standard way to implement the back-end business code typically found in enterprise applications.

Web Services/SOAP Web services provided a solution to cross-platform and cross-language interoperability. Web services invoke a remote method via HTTP with XML format. Two different XML encodings are currently in use: XML-RPC and SOAP. XML-RPC defines a lightweight protocol and SOAP, or Simple Object Access Protocol, defines a much richer set of services; the specification covers not only remote procedure calls, but also the Web Services Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI). WSDL is SOAP's interface definition language and UDDI serves as a directory service for the discovery of Web services.

.NET Remoting .NET Remoting [REM08] was designed by Microsoft and gives a flexible and extendible framework that allows for different transfer mechanisms, HTTP

or TCP, and encodings in SOAP or binary. With these options, we can choose between HTTP-based transport for the internet and a faster TCP-based one for LAN applications.

6.2. .NET Remoting versus Web Services

Both .NET Remoting and Web Services are powerful technologies that provide a suitable framework for developing distributed application. The following are their different factors:

- **Performance:** .NET Remoting can use any protocol and formatter. Therefore, .NET Remoting provides faster communication between the nodes by using TCP protocol and binary formatter.
- **State Management:** Web service is a stateless programming model, which means that each incoming request is handled independently. Each time a client invokes a Web service, a new object is created to service the request. The object is destroyed after the method call is completed. .NET Remoting supports a range of state management and can correlate multiple calls from the same client and support callbacks.
- **Interoperability:** Web services support interoperability across platforms and are good for heterogeneous environments. .NET Remoting only support Windows platforms.

6.3. The benefits of .NET

Microsoft's .NET Framework is an execution environment for Windows programs. We can briefly sum up the benefits of .NET with following points:

- .NET supports C++/CLI (Common Language Infrastructure) language, which can interoperate with native C++ language. That allows for the reuse of the PCD++ code.
- .NET Common Language Running is the implementation of the Common Language Infrastructure, which is an open standard published by ISO (ISO 23271) and European Computer Manufacturers Association (ECMA-335).

- .NET provides built-in services for Remoting, proxies, marshalling, distributed services, and other important network programming tools.
- .NET provides a large library of pre-coded solutions to common programming problems, which greatly reduces developing time.
- .NET supports multi-server and multi-client. This means that a client can ask one server to pass its call to another server.
- .NET supports various protocols and formatting.
- .NET supports object serialization.
- .NET provides automatic memory management, therefore developers don't need to implement freeing routines or worry about the sequence in which cleanup is performed or be concerned about whether or not an object is still referenced.

6.4. PCD++/.NET Remoting System

PCD++/.NET Remoting is a distributed simulation system based on parallel CD++. Its aim is to combine the .NET Remoting technique with parallel DEVS and Cell-DEVS to support distributed simulations across network nodes. Compared to PCD++Win, PCD++/.NET swaps MPI for .NET Remoting, which allows for the execution of distributed computing across LANs and WANs. Figure 30 shows the PCD++/.NET architecture.

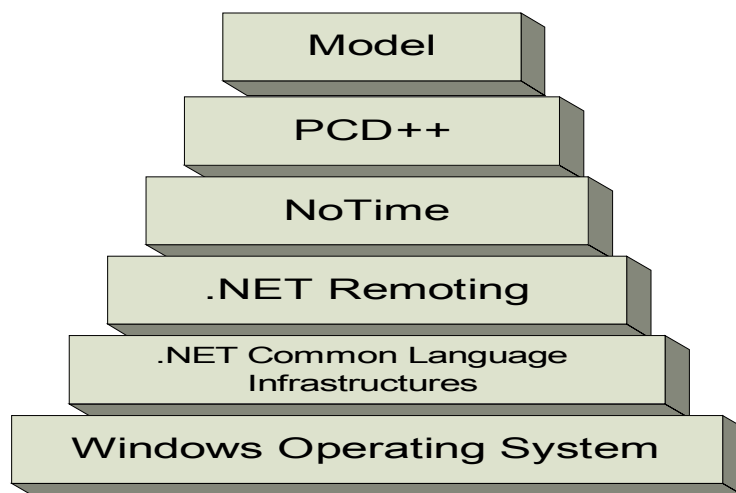


Figure 30. PCD++/.NET architecture

Here, DEVS or Cell-DEVS models are built on top of the system. PCD++ classes interact with the NoTime Kernel, which manage Logical Process (LP) and messages. The .NET Remoting API, instead of MPI, is implemented to send and receive remote messages. The Common Language Infrastructure is an open specification that describes the executable code and runtime environment. All the code in PCD++/.NET is first compiled to the Intermediate Language (IL) at compile time, and then IL is compiled to the machine code at runtime and then executed on the platform.

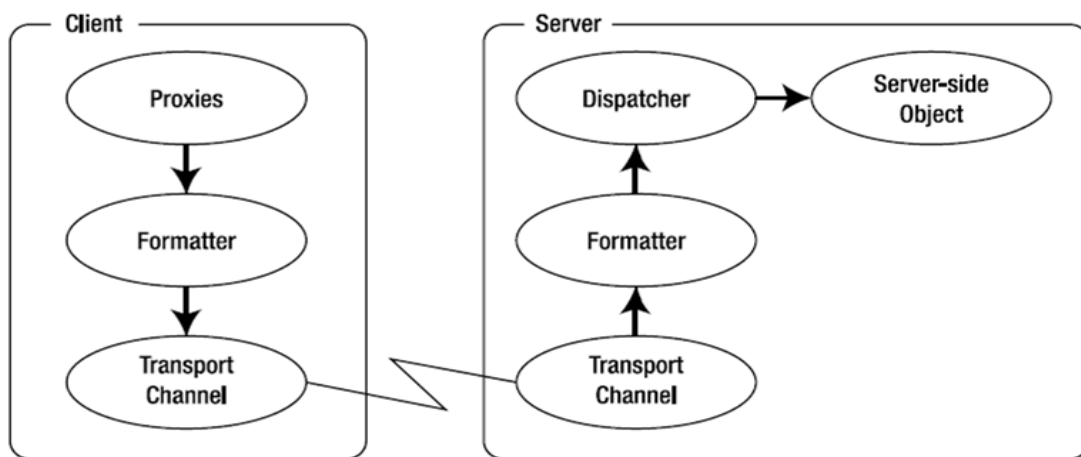


Figure 31. The interaction of objects in .NET Remoting [REM08]

In .NET Remoting (as shown in Figure 31), the client is the component that needs to communicate with a remote object. The server receives the request from the client object and responds. The Proxy contains a list of all classes, as well as interface methods of the remote object. It examines whether the call made by the client object is a valid method of the remote object and if an instance of the remote object resides in the same application domain as the proxy. If true, a simple method call is routed to the remote object. If the object is in a different application domain, the call is forwarded to a RealProxy class by calling its Invoke method. This class is then responsible for forwarding messages to the remote object. The message will pass a serialization layer: the formatter, which converts it into a specific transfer format such as SOAP or binary. The serialized message later reaches a transport channel, which transfers it to a remote process via a specific protocol like HTTP or TCP, as following:

- **The HTTP Channel** transports messages to and from remote objects using the SOAP protocol. All messages are passed through the SOAP formatter, where the message is changed into XML and serialized, and the required SOAP headers are added to the stream. The resulting data stream is then transported to the target URI using the HTTP protocol.
- **The TCP Channel** uses a binary formatter to serialize all messages to a binary stream and transport the stream to the target URI using the TCP protocol.

On the server side, the message also passes through a formatting layer, which converts the serialized format back into the original message and forwards it to the dispatcher. Finally, the dispatcher calls the target object's method and passes back the response values through all tiers.

There are two very different types of remote interaction between components in .NET Remoting. One uses serialized objects that are passed as a copy to the remote process. The second uses server-side (remote) objects that allow the client to call their methods. The first object is called *ByValue Object*. This ability to serialize objects is provided by the .NET Framework when we set the attribute [Serializable] for a class or implement ISerializable interface. The second object is called *MarshalByRef Object* and runs on the server and accepts method calls from the client. Its data are stored in the server's memory and its methods executed in the server's Application Domain. Instead of passing around a variable that points to an object of this type, only a pointer -- called *ObjRef* -- is passed around.

In PCD++/.NET design, remote objects are used to implement methods, which can send and receive messages, while serialized objects are used to form the messages, which are passed between the client objects and remote objects. To implement a PCD++/.NET system, the following procedures are followed:

- **Define shared interfaces:** When creating a distributed application, we can define the base classes or interfaces for remote objects. This assembly is used on both the client and server. The real implementation is placed only on the server and is a class that extends the base class or implements the interface.
- **Define a server assembly:** The server implements the method that is defined in the interface.

- **Define a client assembly:** The client consumes the service that server provided.
- **Serialization of Data:** In .NET, the encoding/decoding of objects is natively supported. We just need to mark such objects with the [Serializable] attribute and the rest will be taken care of by the framework. This assembly is used to define a mobile message, which can be passed between nodes.

Following the above procedure, a set of components is designed (as shown in Figure 32).

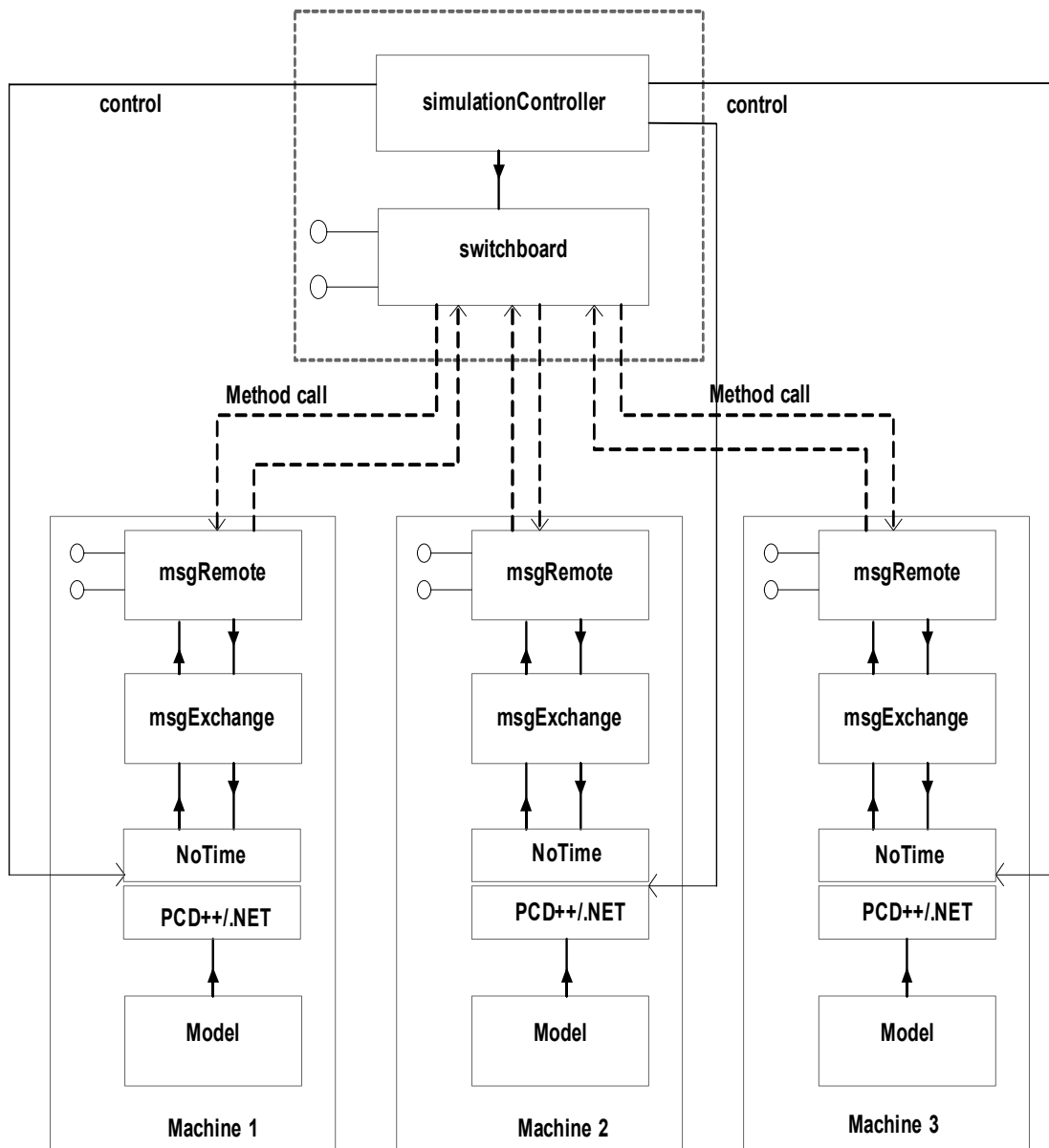


Figure 32. PCD++/.NET components

These components include:

- *simulationController* is used to control PCD++/.NET residing in each node.
- *switchboard* is used to conduct a coming method call to the destination node.
- *msgRemote* is used to send or receive a method call to or from the switchboard.
- *msgExchange* is used to translate a C++/CLI message into a native C++ message and vice versa.

The above components interact according to following procedure:

1. Whenever the *simulationController* sends a start signal, the PCD++/.NET in each node will load the partition model and enter the simulation loop.
2. If an inter-process message needs to be sent to another node, the C++ message is translated to C++/CLI message by *msgExchange* component. Then, the message is passed to *msgRemote*.
3. The *msgRemote* invokes the sever object of the *switchboard* and sends a message to the destination node.
4. The *msgRemote* of the destination node receives the incoming message, and passes it to *msgExchange*.
5. The *msgExchange* translates the incoming C++/CLI message to a C++ message, and then passes it to PCD++/.NET.

Figure 33 shows the detail of how the *switchboard* and the *msgRemote* interact in a system that has three nodes. The *switchboard* has one server that is Server_SB, and three clients that are Client1, Client2 and Client3. Each node has one server and one client. Thus, node1 has server1 and Client_SB1, node2 has server2 and Client_SB2, and, node3 has server3 and Client_SB3. Each client in the node (such as Client_SB1, Client_SB2 and Client_SB3) is associated with *switchboard*'s server Server_SB. Client1, Client2 and Client3 are associated with server1, server2 and server3 respectively. If Machine1 needs to send a call to Machine3, Client_SB1 first registers a TCP channel and connects to Server_SB. Then, Server_SB passes the call to Client3. Finally, Client3 sends the call to server3. The path of the method call is:
Machine1 → switchboard → Machine3.

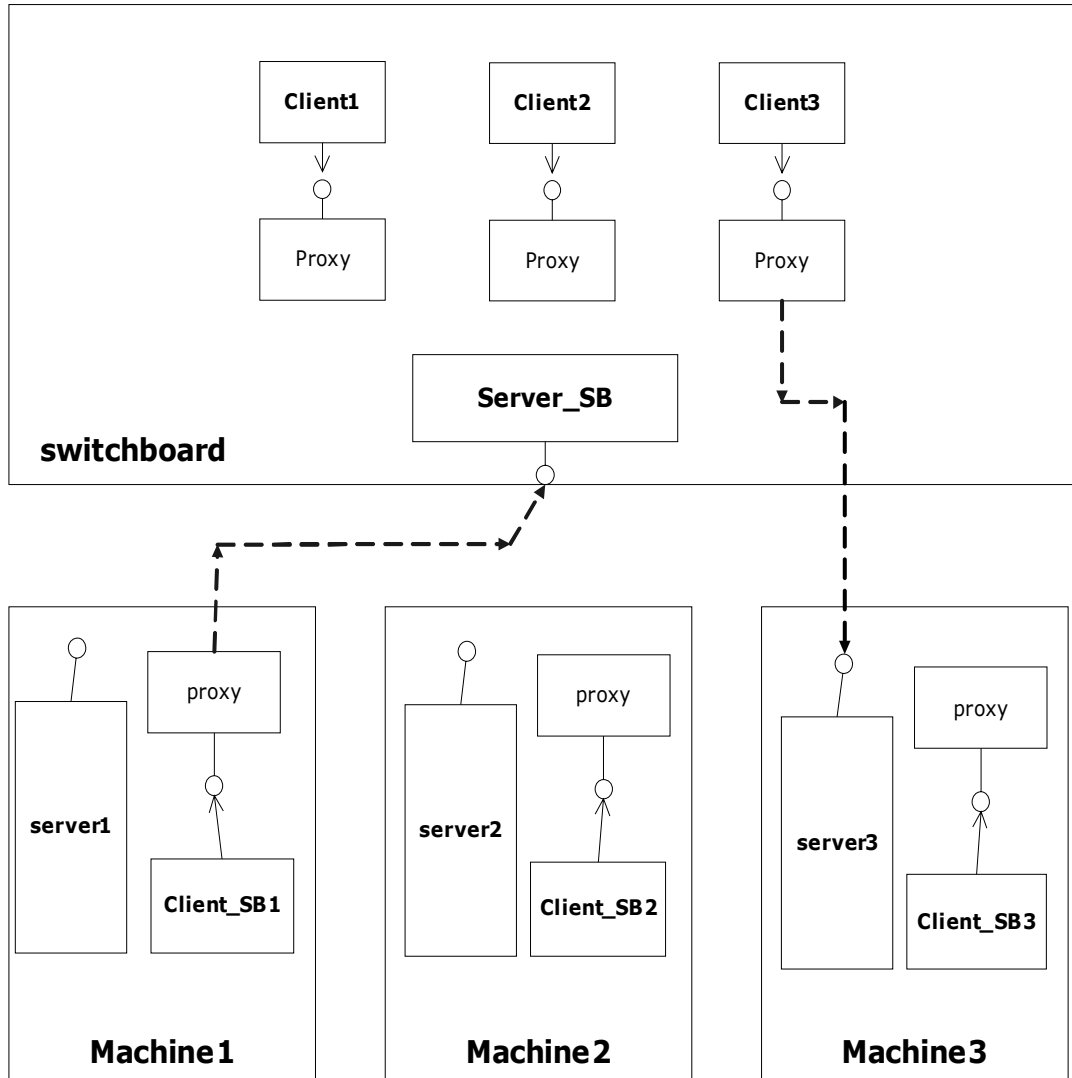


Figure 33. PCD++/.NET Remoting method call

It is worthy of note that a copy of the message in one node is sent to another node directly, it does not pass through the switchboard. This is illustrated in Figure 33. In Figure 33, we have node A, node B and the *switchboard*. A circle represents a client and a rectangle represents a server. In node B, *SeverObject2* has a method *sendMsg(Message msg)*, which just adds a message to the local message queue. In the *switchboard*, *ClientObject2* associated with *SeverObject2* and can invoke the remote method *sendMsg(Message msg)*. If the node A needs to send a message to node B, *ClientObject1* will invoke its remote method *sendMsgToNode(Message msg)* in *switchboard*. As shown in Figure 34, *sendMsgToNode* includes a method *sendMsg*, which sends the message to node B. Therefore, the path of message is $A \rightarrow B$.

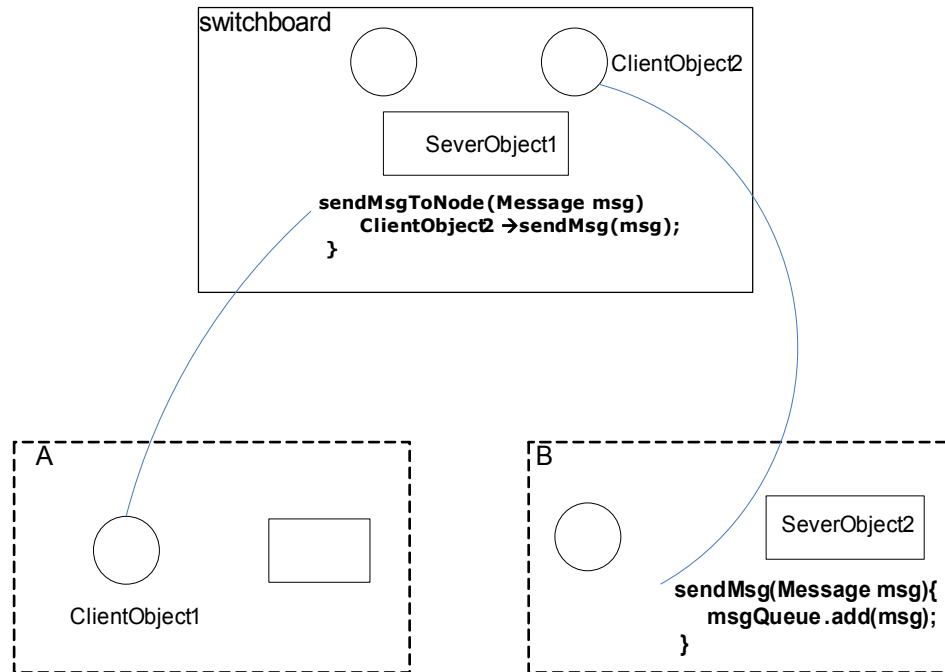


Figure 34. Message passing

It can be seen that, instead of MPI, the PCD++/.NET Remoting system uses the *switchboard* component to pass messages. The *switchboard* can be put in any machine. For example, Figure 35 shows that *switchboard* and PCD++/.NET Master coordinator are put in same machine. In Figure 35, the wide line represents a connection with Remoting routine and the thin line represents the direct connection in the same Application Domain. *Switchboard* and Master coordinator are put together in machine0, but they belong to different Application Domains, so they connect with the Remoting routine. Whenever *switchboard* starts, it reads a text file *machineList.txt*, which just lists the names of the machines. Then, *switchboard* will create a set of N client objects for sending messages and one sever object for receiving messages as we explained earlier.

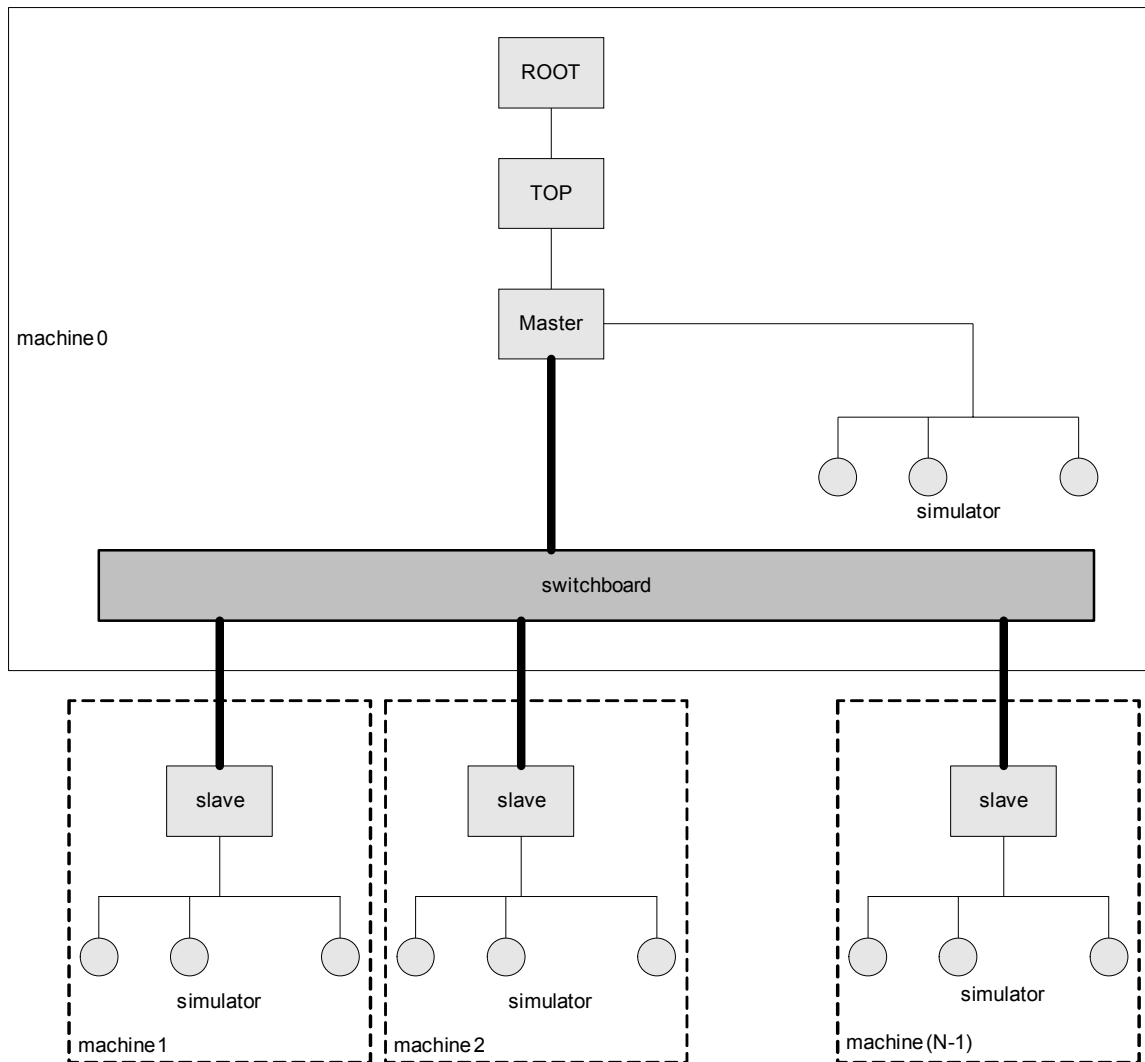


Figure 35. PCD++/.NET Remoting simulators

Another important component in the system is *msgExchange* (Figure 36). In PCD++/.NET, any message is derived from the *BasicEvent* object or *EventMsg* object. In order to pass these message objects among distributed simulators in .NET platform, a C++/CLI class *managedBasicEvent* is created and marks its serialized attribute so that it can be transferred between nodes.

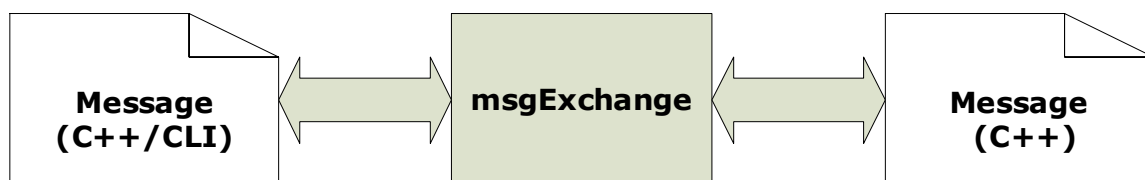


Figure 36. msgExchange component

The component *msgExchange* includes two functions: *changeToManagedMsg* and *changeToUnmanagedMsg*. The first function passes a *BasicEvent* or *EventMsg* object and returns a *managedBasicEvent* object. The second function passes a *managedBasicEvent* object and returns a *BasicEvent* or *EventMsg* object. A flowchart (Figure 37) represents how these components work.

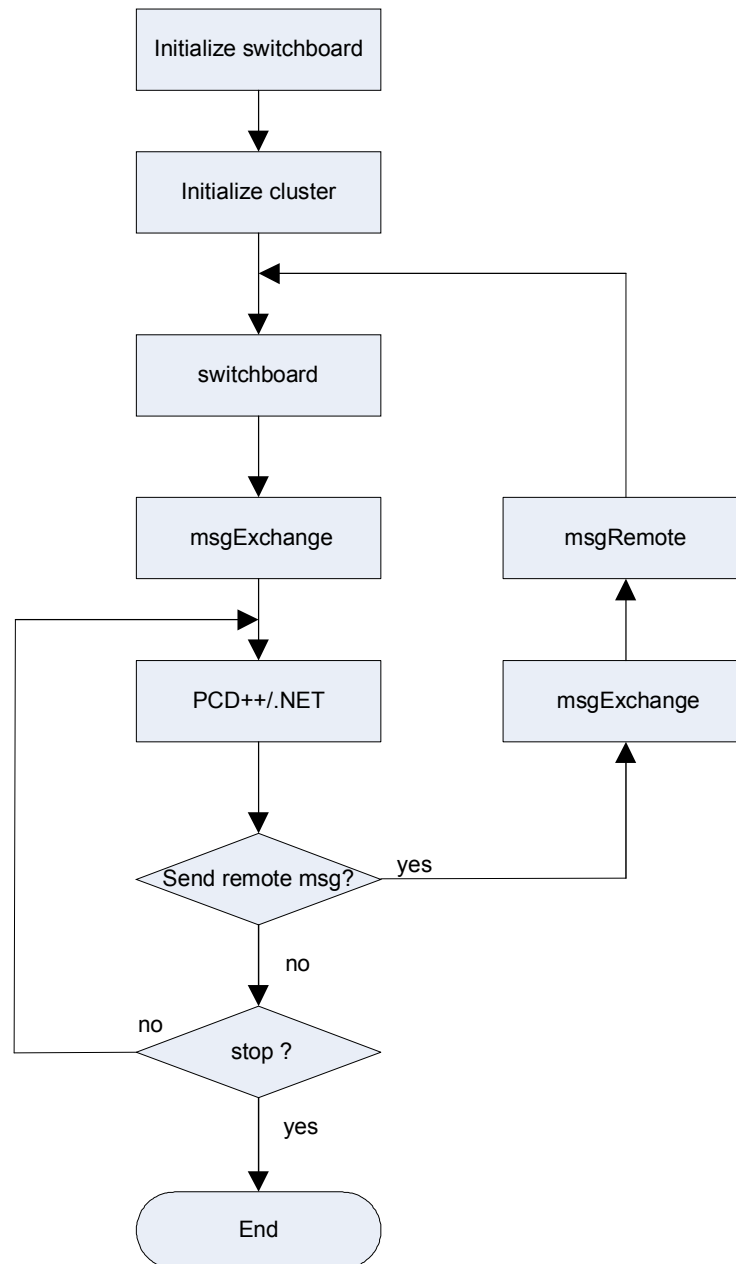


Figure 37. Flowchart of PCD++/.NET Remoting

In Figure 37, the following operation is executed as follows:

- *switchboard* is initialized by starting the switchboard program and read a *machineList.txt* file and create all client and server objects for receiving and sending messages.
- cluster is initialized by starting the cluster program in each PC machine. Each node of cluster will create one client object and one server object. PCD++/.NET is waiting for the starting signal that comes from *switchboard*.
- *switchboard* sends the starting signal to each machine in the cluster, PCD++/.NET in the machine reads a partition file and loads the model to execute the simulation.
- If PCD++/.NET needs to send a remote message, *msgExchange* will translate the message from a native C++ object to a C++/CLI object, and *msgRemote* sends a method call to *switchboard*.
- If *switchboard* receives a method call, it will conduct the call to the right destination in the cluster. At this point, *msgRemote* at the destination receives a copy of the message.
- If a machine receives a message, it puts the message into a message queue and then *msgExchange* translates it from a C++/CLI object to a native C++ object and passes it to PCD++/.NET.
- Once the cluster finishes the simulation, it exits the simulation loop and stops.

From the above analysis, it can be seen that the *switchboard* component plays a unique role in the system. A sequence diagram (Figure 38) shows how *switchboard* interacts with nodes in specified function. Here, the system has two nodes.

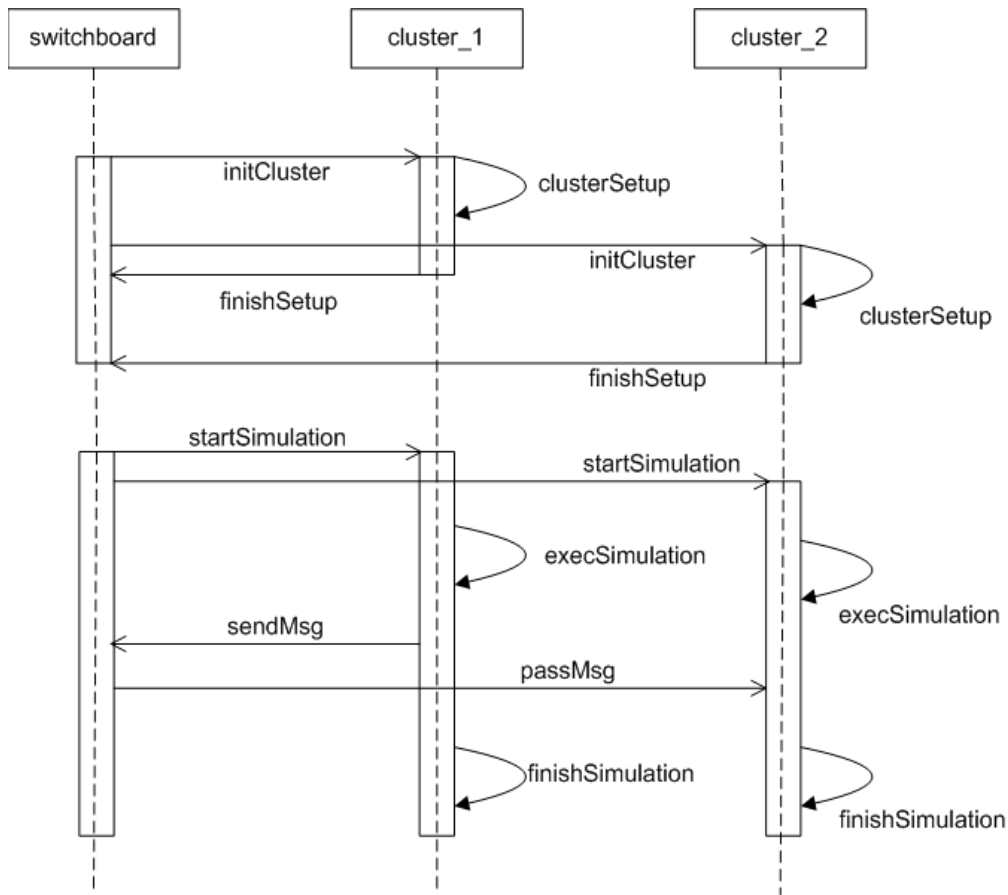


Figure 38. Sequence diagram of PCD++/.NET Remoting

First, *switchboard* uses the *initCluster* method to initialize all nodes, and then each node uses *clusterSetup* to create the server and client object. Once *switchboard* receives *finishSetup* from all nodes, it will start the simulation with a method called *startSimulation*. Each node will execute the *execSimulation* method to enter the PCD++/.NET simulation loop, if *cluster_1* needs to send a message to *cluster_2*, a *sendMsg* call will be forwarded to the *switchboard*, and the *switchboard* invokes a *passMsg* method to send a message copy to *cluster_2*.

CHAPTER 7 PERFORMANCE ANALYSIS FOR PCD++/.NET

In this chapter, a performance analysis is presented for PCD++/.NET. The experiments were carried out on a group of desktop workstations (Intel Core 2 Duo Processor E6400@ 2.13 GHz, 2GB DDR2-Synch DRAM) running Microsoft Windows XP Professional connected through a local area network (LAN). Microsoft .NET Framework 2.0 was installed in each workstation.

7.1. Remote message of PCD++Win and PCD++/.NET

Since both PCD++Win and PCD++/.NET use same simulation engine, they have same quantity of remote messages for executing the same model with the same partition strategy. We can explain this by using an example. Suppose we have a Cell-DEVS model that has a row of 4 atomic models, we can have 4 partitions and each includes an atomic cell, and each cell will send a message to its neighbor (see Figure 39 below).

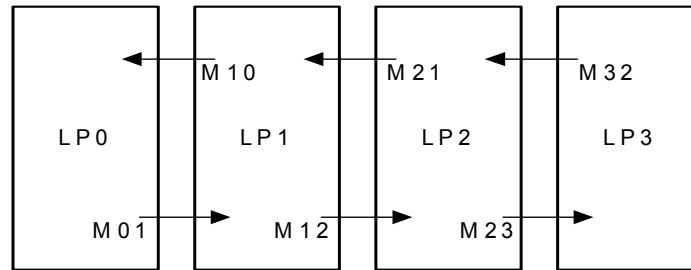


Figure 39. Partitioning a couple model into 4 atomic models

In PCD++Win, message M01 is the message between master coordinator (machine 0) and slave coordinator (machine 1). It is directly sent from machine 0 to machine 1 and needs a remote message. In the same way, message M10 needs a remote message. M12, M23, M32 and M21 are all messages between two slave coordinators. Each of them needs two remote messages: one slave coordinator sends a message to the master coordinator, and then the master coordinator sends the message to another slave coordinator. Figure 40 shows this case.

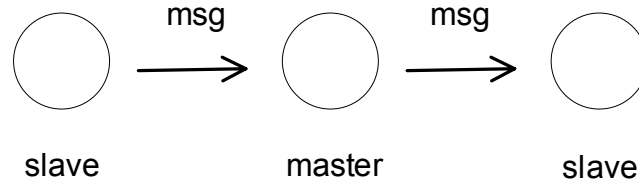


Figure 40. PCD++Win passing messages

So, PCD++Win system needs 12 remote messages to complete the message sending.

In PCD++/.NET, a message-passing component “*switchboard*” is added. The *switchboard* is used to forward method call. According to the analysis in Chapter 6 (see Figures 33 and 34), one message is directly sent to the destination node by means of two method calls (as shown in Figure 41).

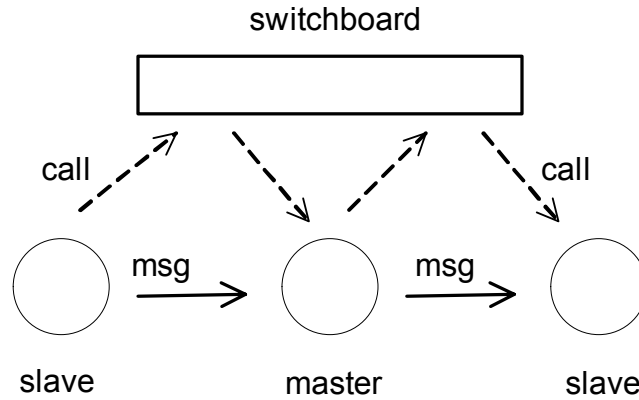


Figure 41. PCD++/.NET passing messages

Therefore, PCD++/.NET has the same quantity of remote messages as the PCD++Win system for executing the same model with same partition strategy.

7.2. Correctness and verification

In [Fre02], the authors present the specifications of major time-warp components along with verification criteria used to assure correctness. According to [Fre02], a distributed simulation is correct when it produces simulation results that are the legal result for a traditional, single process simulator. In the experiments, we first run a stand-alone CD++ toolkit, and the generated simulation results are used as reference outputs for verification purposes. Then, these models are executed with PCD++/.NET (or PCD++Win) on

multiple nodes. After each run, the simulation results are compared with the reference outputs to ensure that the same results are generated with PCD++/.NET (or PCD++Win) as those produced with the standalone version.

7.3. Experimental results and analysis

Two models are used during the performance analysis. The one is the watershed [Ame01] 15*15*2 coupled Cell-DEVS model. The other is the Life [Gar70] 12*12 coupled Cell-DEVS model.

7.3.1. Watershed model

The first model tested was the watershed model, which represents water flow and accumulations depending on characteristics of different vertical layers: air, vegetation, surface waters, soil, ground water, and bedrock [Moo96].

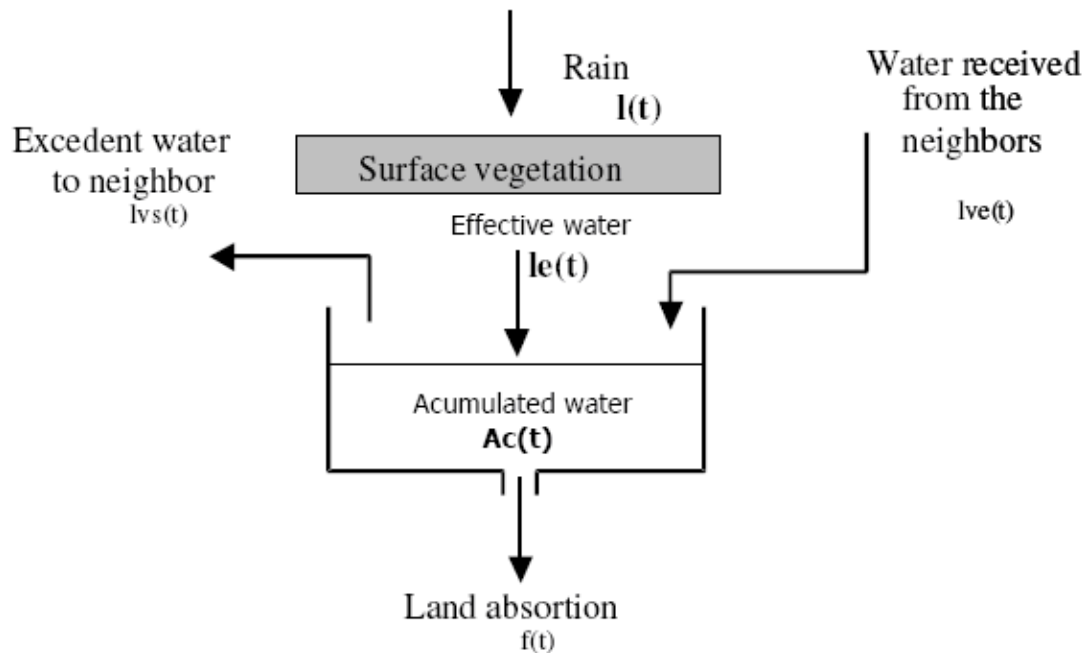


Figure 42. Hydrology Model [Moo96]

In [Moo96], the authors defined a hydrology model in which they identified several vertical layers composing a watershed. Figure 42 shows a description of this model. When the rain is absorbed by the vegetation, the rest is received by the surface. Depending on the topology, the cells can also receive and send water from and to the neighbors. Part of the water received is lost due to the filtration over the land and stones. The accumulated water in a period depends on: the quantity of effective water (rain), the quantity of water dumped from the neighboring cells (effective rain plus the water received from the neighbors minus water sent to the neighbors) minus the water filtered by stones and soil. In [Ame01], the watershed model was created as a three-dimensional Cell-DEVS model to simulate the accumulation of water under the presence of constant rain (7.62mm/hr). Figure 43 shows the model definition, which is a 15*15*2 three dimensional cell space. The model has two surfaces, one to represent the height of the water retained (surface 0) and one to represent the topography of the terrain (surface 1).

```
[top]
components : watershed

[watershed]
type : cell
dim : (15,15,2)
delay : inertial
defaultDelayTime : 1000
border : nowrapped
neighbors : watershed(-1,0,0)
neighbors : watershed(0,-1,0) watershed(0,0,0) watershed(0,1,0)
neighbors : watershed(1,0,0)

neighbors : watershed(-1,0,1)
neighbors : watershed(0,-1,1) watershed(0,0,1) watershed(0,1,1)
neighbors : watershed(1,0,1)

initialValue : 0
initialCellsValue : init.val
zone : grass { (0,0,0)..(14,6,0) }
zone : stones { (0,13,0)..(14,14,0) }
localtransition : hydrology

[grass]
rule : {0.07 + (0,0,0) - if(((((-1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((-1,0,1) + (-1,0,0))))),((((((0,0,0) + (0,0,1) - (-1,0,0) - (-
1,0,1))/1000) * (0,0,0))/1000),0) - if((((1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((1,0,1) + (1,0,0))))),((((((0,0,0) + (0,0,1) - (1,0,0) -
(1,0,1))/1000) * (0,0,0))/1000),0) - if((((0,-1,0) != ?) and (((0,0,1)
+ (0,0,0))>((0,-1,1)+(0,-1,0))))),((((((0,0,0) + (0,0,1) - (0,-1,0) -
(0,-1,1))/1000) * (0,0,0))/1000),0) - if((((0,1,0) != ?) and (((0,0,1)
```

```

+ (0,0,0))>((0,1,1) + (0,1,0))),(((0,0,0) + (0,0,1) - (0,1,0) -
(0,1,1))/1000) * (0,0,0))/1000),0) + if(((((-1,0,0) != ?) and (((-1,0,1)
+ (-1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (-1,0,1) - (0,0,0) -
(0,0,1)) * (-1,0,0))/1000),0) + if((((1,0,0) != ?) and (((1,0,1) +
(1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (1,0,1) - (0,0,0) -
(0,0,1)) * (1,0,0))/1000),0) + if((((0,-1,0) != ?) and (((0,-1,1) +
(0,-1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,-1,1) - (0,0,0) -
(0,0,1)) * (0,-1,0))/1000),0) + if((((0,1,0) != ?) and (((0,1,1) +
(0,1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,1,1) - (0,0,0) -
(0,0,1)) * (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }

```

```

[stones]
rule : { 0.09 + (0,0,0) - if(((((-1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((-1,0,1) + (-1,0,0)))),(((0,0,0) + (0,0,1) - (-1,0,0) - (-
1,0,1))/1000) * (0,0,0))/1000),0) - if((((1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((1,0,1) + (1,0,0)))),(((0,0,0) + (0,0,1) - (1,0,0) -
(1,0,1))/1000) * (0,0,0))/1000),0) - if((((0,-1,0) != ?) and (((0,0,1)
+ (0,0,0))>((0,-1,1)+(0,-1,0)))),(((0,0,0) + (0,0,1) - (0,-1,0) -
(0,-1,1))/1000) * (0,0,0))/1000),0) - if((((0,1,0) != ?) and (((0,0,1)
+ (0,0,0))>((0,1,1) + (0,1,0)))),(((0,0,0) + (0,0,1) - (0,1,0) -
(0,1,1))/1000) * (0,0,0))/1000),0) + if(((((-1,0,0) != ?) and (((-1,0,1)
+ (-1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (-1,0,1) - (0,0,0) -
(0,0,1)) * (-1,0,0))/1000),0) + if((((1,0,0) != ?) and (((1,0,1) +
(1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (1,0,1) - (0,0,0) -
(0,0,1)) * (1,0,0))/1000),0) + if((((0,-1,0) != ?) and (((0,-1,1) +
(0,-1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,-1,1) - (0,0,0) -
(0,0,1)) * (0,-1,0))/1000),0) + if((((0,1,0) != ?) and (((0,1,1) +
(0,1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,1,1) - (0,0,0) -
(0,0,1)) * (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }

```

```

[hydrology]
rule : { 0.12 + (0,0,0) - if(((((-1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((-1,0,1) + (-1,0,0)))),(((0,0,0) + (0,0,1) - (-1,0,0) - (-
1,0,1))/1000) * (0,0,0))/1000),0) - if((((1,0,0) != ?) and (((0,0,1) +
(0,0,0))>((1,0,1) + (1,0,0)))),(((0,0,0) + (0,0,1) - (1,0,0) -
(1,0,1))/1000) * (0,0,0))/1000),0) - if((((0,-1,0) != ?) and (((0,0,1)
+ (0,0,0))>((0,-1,1)+(0,-1,0)))),(((0,0,0) + (0,0,1) - (0,-1,0) -
(0,-1,1))/1000) * (0,0,0))/1000),0) - if((((0,1,0) != ?) and (((0,0,1)
+ (0,0,0))>((0,1,1) + (0,1,0)))),(((0,0,0) + (0,0,1) - (0,1,0) -
(0,1,1))/1000) * (0,0,0))/1000),0) + if(((((-1,0,0) != ?) and (((-1,0,1)
+ (-1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (-1,0,1) - (0,0,0) -
(0,0,1)) * (-1,0,0))/1000),0) + if((((1,0,0) != ?) and (((1,0,1) +
(1,0,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (1,0,1) - (0,0,0) -
(0,0,1)) * (1,0,0))/1000),0) + if((((0,-1,0) != ?) and (((0,-1,1) +
(0,-1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,-1,1) - (0,0,0) -
(0,0,1)) * (0,-1,0))/1000),0) + if((((0,1,0) != ?) and (((0,1,1) +
(0,1,0))>((0,0,1) + (0,0,0)))),(((0,0,0) + (0,1,1) - (0,0,0) -
(0,0,1)) * (0,1,0))/1000),0) } 1000 { cellpos(2)=0 }
rule : { (0,0,0) } 1000 { t }

```

Figure 43. Watershed Model[wai08]

In Figure 43, the model defines areas of different soil type: grass and rocks. Then, hydrology model is used to calculate the water accumulation. A simple partition strategy is used in the watershed model testing (shown in Figure 44).

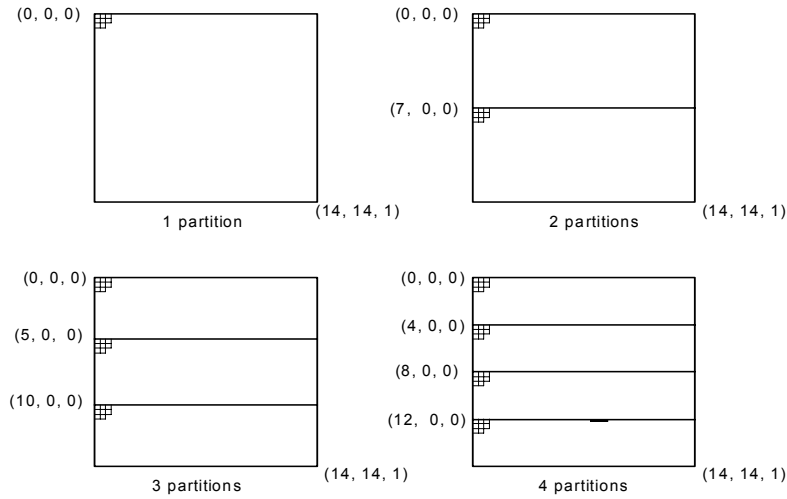


Figure 44. Partition of watershed model

In Figure 44, the cell space is evenly divides into horizontal rectangles and each partition is run by one PC workstation. Figure 45 shows the execution time and overall speedup for the model with the PCD++Win system.

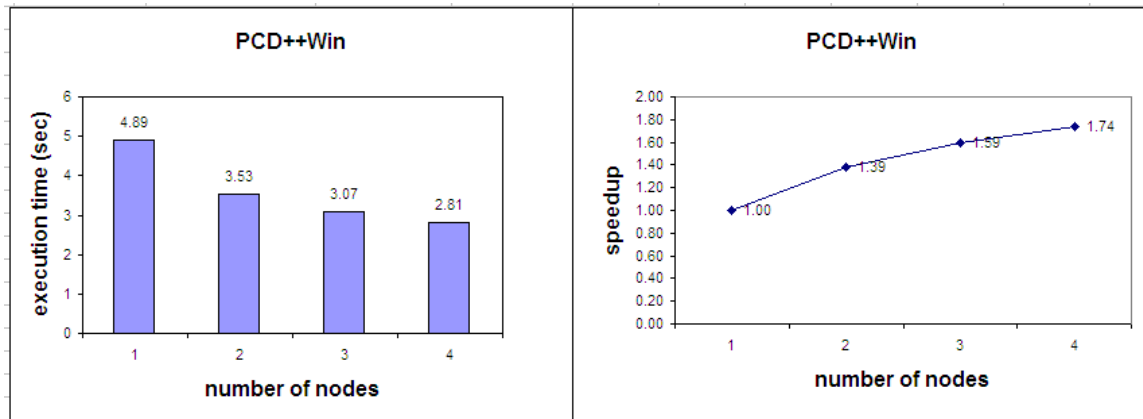


Figure 45. Execution result of watershed model in PCD++Win

The watershed model is a three-dimensional space model and each cell has nine neighbors. Therefore, the watershed model has an extensive inter-LP communication load. In Figure 45, it can be seen that the speedup of the watershed model for PCD++Win is achieved by increasing computing nodes. The execution time of the model decreases from 4.89 to 2.81 seconds when the number of nodes climbs from 1 to 4.

In contrast with PCD++Win, PCD++/.NET has different situation for the watershed model. In Figure 46, the upward trend of execution time is presented when the number of nodes increases.

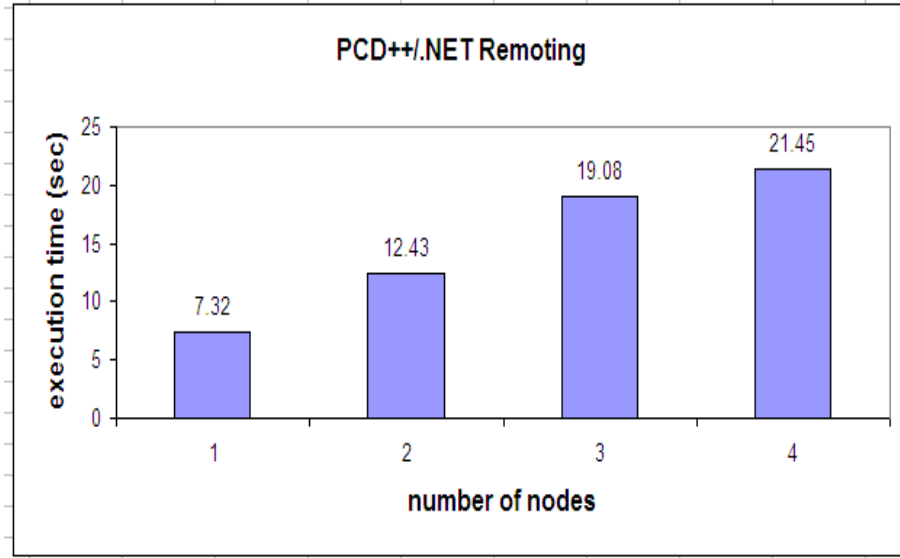


Figure 46. Execution time of watershed with PCD++/.NET Remoting

Table 2 shows the ratio of local message and remote message.

Table 2. The message of watershed model

Watershed model				
	1 node	2 nodes	3 nodes	4 nodes
Local Message (%)	100	78.87	72.76	64.79
Remote Message (%)	0	21.13	27.24	35.21
Total Messages	65876	67102	68901	71245

From above experiment results, the following analysis can be done:

- In the watershed model, PCD++/.NET took 7.32 seconds to execute the model with one node, while PCD++Win has only took 4.89 seconds. This means that PCD++/.NET needs more time for system initialization.
- As we mentioned early, PCD++Win has the same remote message numbers with the PCD++/.NET for the same model. Therefore, the experimental results show that PCD++/.NET has a lower speed for inter-LP communication than does PCD++Win.
- The watershed model has an amount of remote messages between nodes (for example, it has 35.21% remote messages when 4 nodes are used). The bandwidth

of the PCD++/.NET system cannot offer such extensive communication loads and leads to performance degradation.

From the above experiment results, it can be inferred that for the watershed model, the upward trend of simulation execution time is due to the increasing of communication workloads among the nodes. It would be worthwhile to examine what may happen if the workload is increased on distributed cells without increasing the communication workload.

7.3.2. Life model

The second model tested is the “Life” Cell-DEVS model, created by John Conway, which uses a two-dimensional grid where cells can be either alive or dead. The rule is: a new cell is born when it has exactly 3 neighbors; an existing cell survives if it has 2 or 3 neighbors; otherwise the cell dies. The model file is shown in Figures 47 and 48.

```
[top]
components : life
in : in
out : out
link : out@life out
link : in in@life
[life]
type : cell
width : 12
height : 12
delay : inertial
defaultDelayTime : 100
border : wrapped
neighbors : life(-1,-1) life(-1,0) life(-1,1)
neighbors : life(0,-1) life(0,0) life(0,1)
neighbors : life(1,-1) life(1,0) life(1,1)
initialvalue : 0
initialrowvalue : 0      00000001110000000000
initialrowvalue : 1      00000100100100000000
initialrowvalue : 2      00000101110100000000
initialrowvalue : 3      00000100100100000000
initialrowvalue : 4      00000001110000000000
initialrowvalue : 5      00000001110000000000
initialrowvalue : 6      00000100100100000000
initialrowvalue : 7      00000101110100000000
initialrowvalue : 8      00000100100100000000
initialrowvalue : 9      00000001110000000000
initialrowvalue : 10     00000001110000000000
initialrowvalue : 11     00000100100100000000
```

Figure 47. The first part of Life model file [wai08]

As demonstrated in Figure 47, the input and output port are added to the model. This is because the Root coordinator advances the clock of the simulation and the simulation continues until at least one of the following conditions holds: there are no more events/messages scheduled by any of the processors, or the simulation clock reaches the maximum execution time as provided by the user. Therefore, input events (represented by an event file) keep the simulation running until the maximum execution time is reached. As Figure 48 shows, a set of input ports are linked to the model.

```

localtransition : conrad-rule
in : in
out : out
link : in in@life(4,2)
link : in in@life(4,3)
link : in in@life(4,4)
link : in in@life(3,3)
link : in in@life(5,3)
link : out@life(4,6) out
link : out@life(2,6) out
link : out@life(3,4) out
link : out@life(5,6) out
link : out@life(6,4) out
link : out@life(1,7) out
[conrad-rule]
rule : 1 1000 { (0,0) = 1 and (truecount = 3 or truecount = 4 ) }
rule : 1 1000 { (0,0) = 0 and truecount = 3 }
rule : 0 1000 { t }

```

Figure 48. The second part of Life model file[wai08]

An event file is used to specify the input event at a time points, such as

```

00:00:00:100 in 1
00:00:01:400 in 1
...
00:00:10:200 in 1

```

The following partition strategy (Figure 49) is used to execute the simulation for the “Life” model.

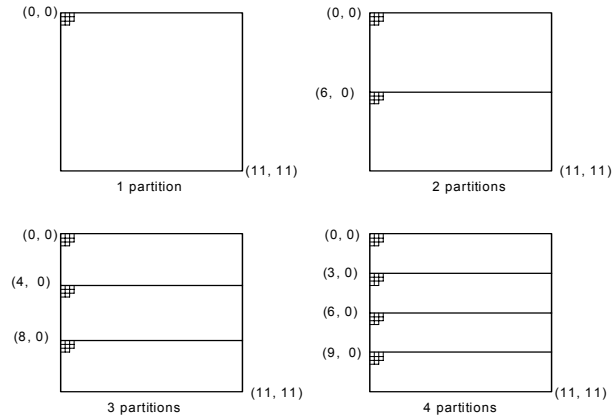


Figure 49. Partition strategy of the Life model

Figure 50 shows the simulation results for both the PCD++Win and PCD++/.NET.

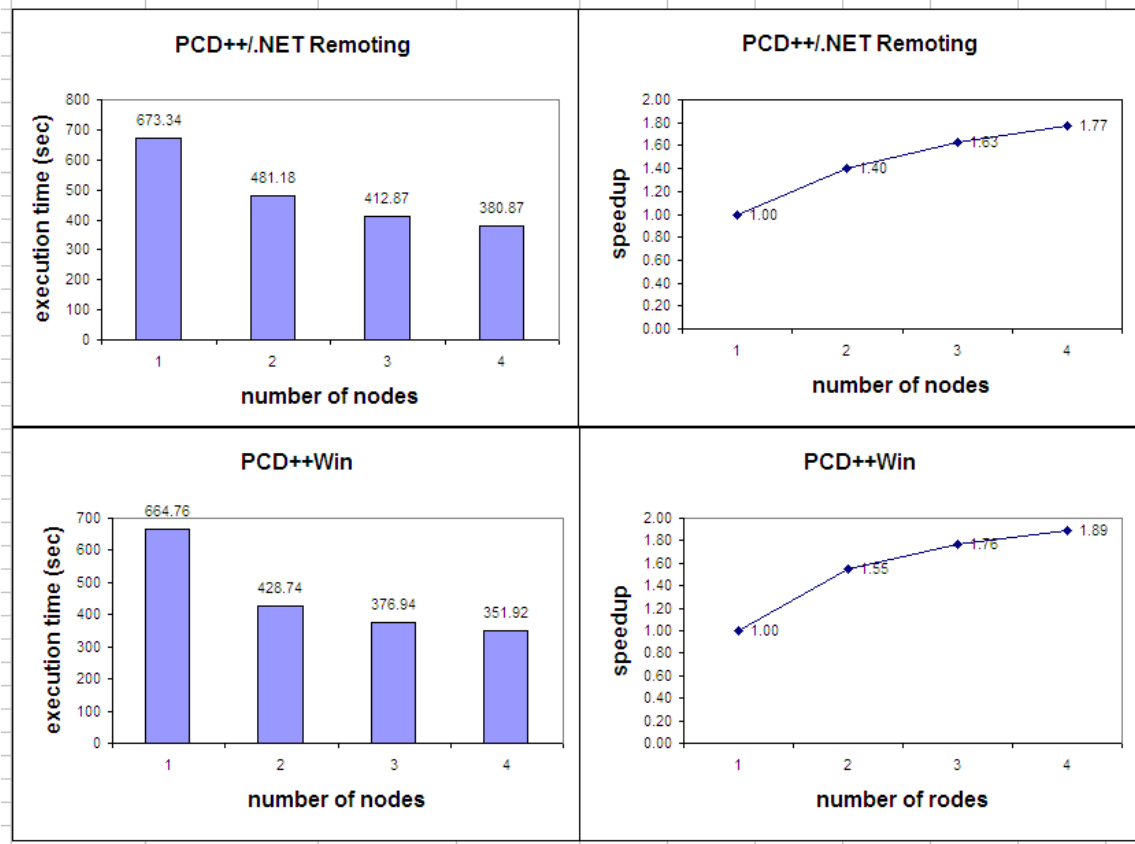


Figure 50. Simulation results of the Life model.

Table 3. The message of Life model

Life model				
	1 node	2 nodes	3 nodes	4 nodes
Local Message (%)	100	97.58	94.37	92.09
Remote Message (%)	0	2.42	5.63	7.91
Total Messages	8895321	8896763	8897981	8898813

Table 3 shows the ratio of local and remote messages to total messages of the Life model. It can be seen that the ratio of remote messages to total messages of the Life model is less than that of the watershed model. For example, the Life model has 7.91% remote messages and the watershed model has 35.21% remote messages when 4 nodes are used (see Table 2). Therefore, we have the following conclusion:

- Remote messages have a great effect on simulation performance for PCD++/.NET.

To explain the above conclusion, it is necessary to know how both PCD++Win and PCD++/.NET transfer a remote message. In PCD++Win, the following MPI call is used to send a message:

```
MPI_Bsend(&buffer, count, type, dest, tag, comm)
```

The contents of the message are stored in a block of memory referenced by the argument buffer. Dest refers to the destination of the message. Here, parameter count, type, tag and comm are attributes of the sending message. This command sends a message to the destination receiver, which is located in a local network. In PCD++/.NET, sending a remote message must go through a serialization and deserialization process. The message class is defined as follows in C++/CLI:

[Serializable]

```
public ref class basicMessage {  
    public:  
        int msgType;  
        int senderLP;  
        int destLP;  
        long sequence;  
        int objId;  
        int lpId;  
        int tokenNum;  
        ...  
}
```

The key word “Serializable”, which allows for object serialization and deserialization, is added in the class definition. Serialization has three important steps [Her03]:

- Obtain the object state information.
- Marshal the state information to a stream, where the in-memory representation has to be changed to the representation, used for the serialized stream.
- Write the stream to memory, file, a database, or other location.

Deserialization also consists of three steps:

- Read the stream from memory, file, the database or other location.
- De-marshal (parse) the state information from the stream.
- Create the corresponding objects and set the attributes and relations.

In [Her03], the authors present a research for object serialization on .NET platforms from the performance and size perspective. They use three different types of objects and different number of objects to make a comparison, which reflects real-world circumstances. These objects are:

- TestObject 31 bytes
- Contract 155 bytes
- InsuranceInformation 4786 bytes

The experimental results are shown in Figure 51 and have the following conclusions:

- Serialization and deserialization performance depends on the size of the objects (larger size objects require more time). Therefore, “InsuranceInformation” object (4786 bytes) takes longer time than the “TestObject” and “Contract” objects for serialization and deserialization.
- Serialization and deserialization performance also depends on the number of objects (with the increased number of simultaneous objects, the time required to create those objects by the deserialization raises and becomes the most time consuming factor).

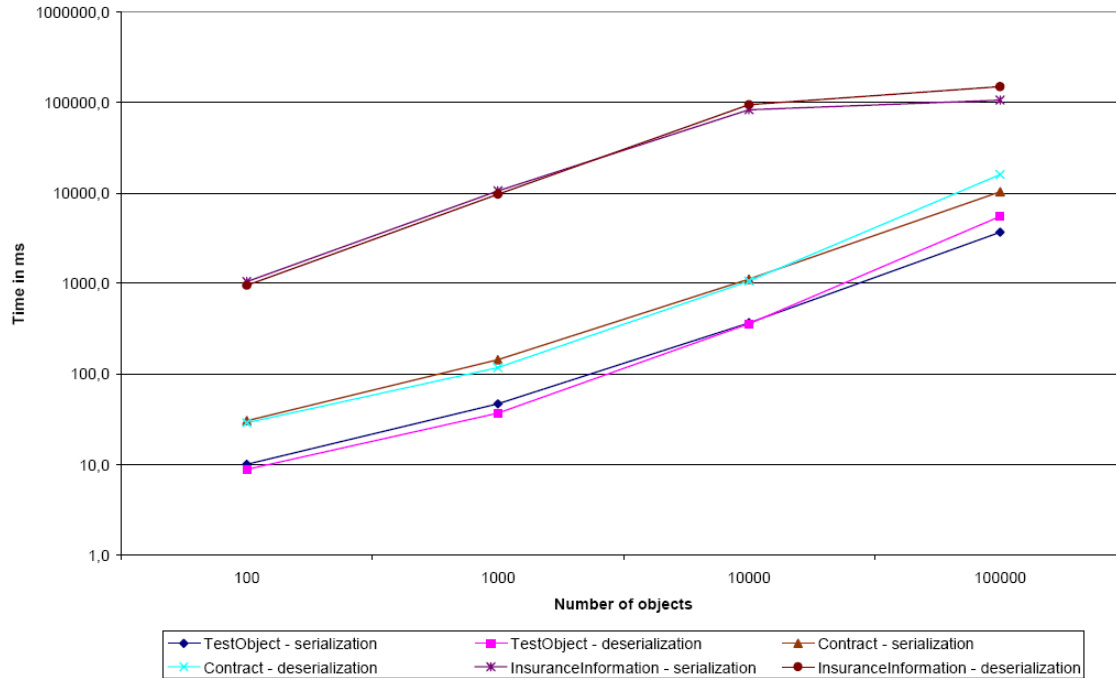


Figure 51. Binary serialization in .NET [Her03]

With the above result, the following analysis can be made:

1. Because PCD++Win has no serialization and deserialization process, it should have better performance than PCD++/.NET for the same model. The results are shown in Figure 52, which demonstrates that PCD++/.NET is slower than PCD++Win by 1.3% - 12.2% in the Life model.

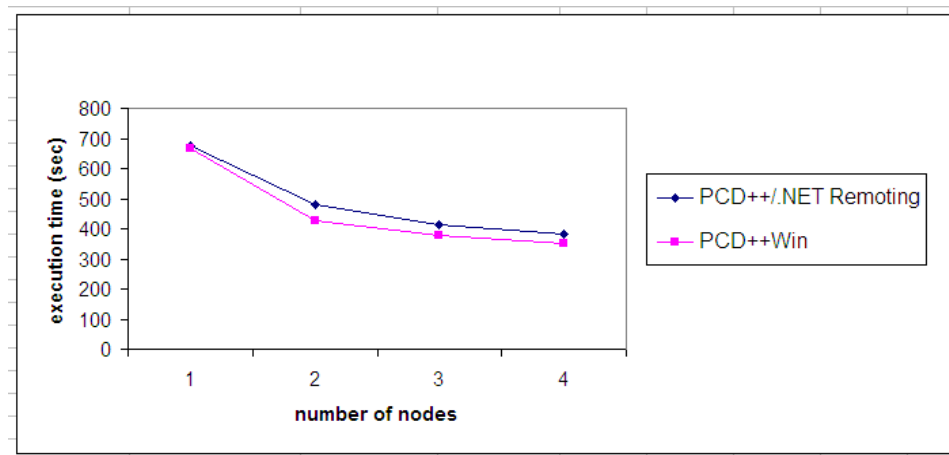


Figure 52. Comparing two systems

2. The inter-LP communication workloads in the Life model are less than those in the watershed model. This means that the Life model can be executed at a lower bandwidth.
3. The watershed model has extensive inter-LP communication workloads when multi nodes are used for simulation. That needs high bandwidth usage, which cannot be offered by PCD++/.NET. Therefore, the simulation performance degrades as the nodes increase.

In summarizing PCD++/.NET, the following features can be identified:

- PCD++/.NET can achieve a speedup for the parallel DEVS and Cell-DEVS simulation, which has modest inter-LP communication loads.
- PCD++/.NET is executed in Common Language Runtime, which is the implementation of open standard ISO 23271 and ECMA-335 (European Computer Manufacturers Association).
- PCD++/.NET presents a way to implement parallel DEVS and Cell-DEVS simulations with Microsoft .NET Remoting, which provides built-in network service and supports any protocol (e.g. TCP, HTTP, SMTP).

CHAPTER 8 CONCLUSIONS AND FUTURE WORK

A parallel and distributed DEVS simulation deals with ways to use multiple processors in a single simulation. All processors together serve to simulate an integrated set of DEVS models. According to the functionality, we can represent a DEVS simulation tool with three layers (Figure 53).

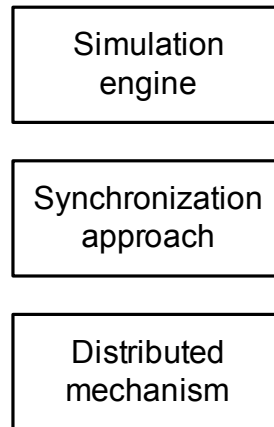


Figure 53. Three layers of a DEVS simulation tool

Here, simulation engine represents the implementation of parallel DEVS formalism. The synchronization approach represents a commonly used synchronization algorithm (e.g. conservative, optimistic and combined). Distributed mechanism represents specified parallel and distributed paradigms (e.g. MPI, CORBA, RMI, Web service, and .NET).

In this work, the simulation engine used is PCD++ [Tro03], the synchronization approach is NoTime kernel and distributed mechanisms are Windows MPI (in PCD++Win) or .NET Remoting (in PCD++/.NET). The methodology of approach is to avoid touch simulation engine layer (PCD++) and the synchronization approach layer (NoTime Kernel), and use a specified parallel and distributed paradigm executing a DEVS and Cell-DEVS simulation. The advantages of the approach are as follows:

- Reuse of code and a reduction of system implementation (only modifies the distributed mechanism layer).
- Allows for the use various distributed mechanisms for the same simulation engine and performance comparisons.

Therefore, we can use TimeWarp middleware to connect a specified distributed mechanism, which could be MPICH, Windows MPI, Windows .NET and Web service (as shown in Figure 54).

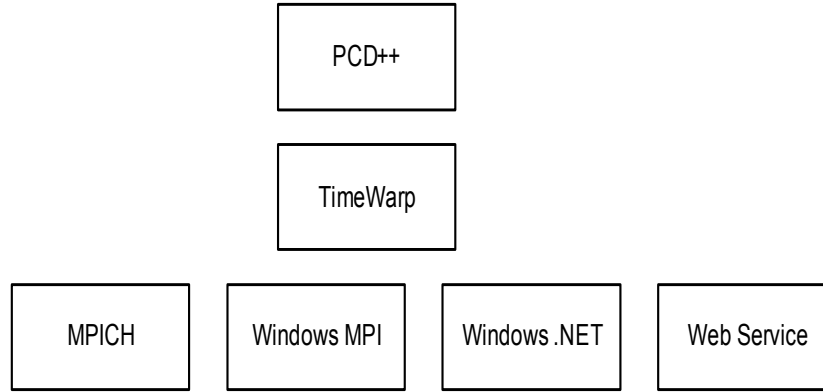


Figure 54. Implementing various parallel and distributed mechanisms for PCD++

In this work, we port PCD++ into the Windows environment and integrate Windows MPI and .NET respectively. In doing so, the following efforts are made:

- Building PCD++Win, which executes parallel DEVS and Cell-DEVS simulation by means of Windows version MPI. MPI is a portable, flexible, vendor independent and platform independent standard for messaging using high performance computing (HPC). It is the specification that specifies how communication occurs between nodes on an HPC cluster. Therefore, MPI ties nodes together, providing a powerful inter-process communication mechanism. PCD++Win takes the advantage of MPI and ports the PCD++ engine from a Linux to a Windows environment. PCD++Win allows general users to manage clusters with a GUI and execute a parallel DEVS and Cell-DEVS simulation.
- Exposing PCD++Win as a Web service that allows access PCD++Win from any platform. Web service is powerful technology that provides a suitable framework for developing distributed applications. Since Web services enable the exposure of functions and methods using HTTP, XML and SOAP, so it is independent of platform and programming languages.
- PCD++/.NET integrates Microsoft .NET Remoting with PCD++ and executes parallel and distributed DEVS and Cell-DEVS simulations in .NET platform.

.NET Remoting is a technology that allows programs and software components to interact across application domains, processes, and machine boundaries. This enables our applications to take advantage of remote resources in a networked environment. The PCD++/.NET system has the following features: 1) PCD++/.NET can achieve the speedup for a parallel DEVS and Cell-DEVS simulation, which has modest inter-LP communication loads. 2) PCD++/.NET runs in a Common Language Runtime environment and takes advantages of the .NET framework. 3) PCD++/.NET supports any protocol to execute a parallel DEVS and Cell-DEVS simulation across a network.

8.1. Future work

The following works would be worth to investigating in the future:

- A comparison between the performance of PCD++/.NET and Web service-based simulation tools: DCD++ [Mad07] is a Web service-based CD++ tool. DCD++ provides the high level of interoperability with full support for WSDL and SOAP over HTTP. PCD++/.NET is designed for .NET platform and is more extensible in terms of enabling communication between objects using different transport protocols and serialization formats. Though both DCD++ and PCD++/.NET implement distributed CD++, each is designed with a particular level of expertise and flexibility in mind to benefit different target users. A comparison between the performance of PCD++/.NET and DCD++ will allow us to identify and evaluate distributed CD++ performance issues.
- Conducting further experiments to analyze the relationship between PCD++/.NET performance and the model's behavior: In this work, it is shown that the model, which has extensive inter-LP communication workloads, may degrade the performance when multi nodes are used in the simulation. Further experiments should be conducted to analyze the relationship between the performance and the inter-LP communication. In addition, the efficient algorithm should be proposed to determine the optimal partition schedule for reducing communication overhead.

- Improving PCD++/.NET performance: One of the performance bottlenecks is object serializing. In PCD++/.NET, the remote message must be serialized into a byte stream and be marshaled from source to destination machine. Serialization and marshaling costs represent a significant proportion of PCD++/.NET communication overhead. We should marshal data efficiently and prefer primitive types. That means complex types should be considered only if these simple types are not sufficient.

REFERENCES

[Amd67] Amdahl, G. “The validity of the single processor approach to achieving large-scale computing capabilities”. In proceedings of AFIPS Spring Joint Computer Conference, Atlantic City, N.J., U.S.A., 1967.

[Ame01] Ameghino J.; Troccoli, A.; Wainer, G. “Models of complex physical systems using Cell-DEVS”. The 34th IEEE/SCS Annual Simulation Symposium. Phoenix, AZ, U.S.A., 2001.

[Che04] Cheon, S.; Seo, C.; Park, S.; Zeigler, B. “Design and implementation of distributed DEVS simulation in a peer to peer network system”. Advanced simulation Technologies Conference – Design, Analysis, and Simulation of Distributed Systems Symposium. Arlington, Virginia, U.S.A., 2004.

[Cho94a] Chow, A. C.; Zeigler, B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism”. Proceedings of the Winter Computer Simulation Conference. Orlando, Fl. U.S.A., 1994.

[Cho94b] Chow, A. C.; Zeigler, B. Kim, D. “Abstract simulators for the Parallel DEVS formalism”. Proceedings of the fifth Annual Conference on AI, Simulation and Planning in High Autonomy Systems. Gainesville, Fl, U.S.A., 1994.

[COR08] CORBA. “CORBA explained simply”. <http://www.ciaranmchale.com/corba-explained-simply> (accessed in Oct, 2008).

[DCE08] DCE home page. “DCE Portal”. <http://www.opengroup.org/dce/> (accessed in Oct, 2008).

- [DCO08] Microsoft MSDN. “DCOM Technical Overview”
<http://msdn.microsoft.com/en-us/library/ms809340.aspx> (accessed in Oct, 2008).
- [Dei08] DeinoMPI. “Documentation”. <http://mpi.deino.net/> (accessed in Oct, 2008).
- [EJB08] Sun Microsystems. “Enterprise JavaBeans Technology”.
<http://java.sun.com/products/ejb/> (accessed in Oct, 2008).
- [Fen08a] B. Feng, Q. Liu, G. Wainer. “Parallel simulation of DEVS and Cell-DEVS models on Windows-based PC cluster systems”, 2008 high performance computing symposium (HPC’08): DEVS session 2, pages 439-446, March 2008.
- [Fen08b] B. Feng, G. Wainer. “A .NET Remoting-based distributed simulation approach for DEVS and Cell-DEVS models”, Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications, pages 292-299, Oct 2008.
- [Fer02] Frey P.; Radhakrishnan, R.; Carter, H.W.; Wilsey, P. A.; Alexander, P. “A formal specification and verification framework for Time Warp-based parallel simulation”. IEEE Transactions on Software Engineering, Vol. 28(1), pages 58-78, Jan 2002.
- [Gar70] Gardner, M. “Mathematical games: The fantastic combinations of John Conway’s new solitaire game life”. Scientific American 223, pages 120-123, Oct 1970.
- [Gli06] Glinsky, E. Wainer, G. “New Parallel simulation techniques of DEVS and Cell-DEVS in CD++”. Proceedings of the 38th IEEE/SCS Annual Simulation Symposium, Huntsville, Al., U.S.A., 2006.
- [Gus88] Gustafson, J. “Reevaluating Amdahl’s law”. Communications of the ACM, Vol.31, pages 523-533, May 1988.

[Her03] Hericko, M. “Object Serialization Analysis and Comparison in Java and .NET”, ACM Sigplan Notices, Vol. 38(8), pages 44-54, August 2003.

[Hen06] Michi Henning, “The rise and fall of CORBA”, ACM Queue, 4(5): pages 28-34, June 2006.

[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.

[Liu07] Q. Liu, G. Wainer. “Parallel environment for DEVS and Cell-DEVS models”, Simulation, Vol. 83(6), pages 449-471, June 2007.

[Mad07] Madhoun, R.; G. Wainer. “Studying the Impact of Web-Services Implementation of Distributed Simulation of DEVS and Cell-DEVS Models”. Proceedings of the 2007 spring simulation multiconference, Vol.2, pages 267-278, March 2007.

[Moo96] Moon, Y.; Zeigler, B.; Ball, G; Guertin, D., “DEVS representation of spatially distributed systems: validity, complexity reduction”. IEEE Transactions on Systems, Man and Cybernetics. Pages 288-296, May 1996.

[MPI08] MPICH. “Documentation”. <http://www-unix.mcs.anl.gov/mpi/mpich1/> (accessed in Oct, 2008).

[Mit07] S. Mittal, J.L. Risco. “DEVSMML: Automating DEVS Execution over SOA towards Transparent Simulators.” In a Special Session on DEVS Collaborative Execution and Systems Modeling over SOA, DEVS integrative M&S Symposium DEVS’07, pages 287-295, March 2007.

[Net07] NetBeans, “Download NetBeans IDE”
<http://www.netbeans.org/> (accessed in April, 2007).

[Neu66] John von Neumann. Theory of Self-reproducing Automata. University of Illinois Press, Edited and completed by Arthur W. Burks. 1966.

[Ram05] Rammer, I. Szpuszta, M. “Advanced .NET Remoting”, Apress, March 2005

[REM08] Microsoft MSDN. “.NET Remoting”. <http://msdn.microsoft.com/en-us/magazine/cc188927.aspx> (accessed in April, 2008)

[RMI08] Sun Microsystems. “Remote Method Invocation Home”. <http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp> (accessed in April, 2008)

[Rod99] Rodriguez D.; Wainer, G. “New Extensions to the CD++ Tool”. Proceedings of the 32th SCS Summer Computer Simulation Conference. Vancouver, Canada. 1999.

[Sar98] Sarjoughian, H. S.; Zeigler, B. “DEVSJAVA: Basis for a DEVS-based collaborative M&S environment”. Proceedings of the International Conference on Web-based Modeling and Simulation. Vol.5, pages 29-36, San Diego, CA. U.S.A. 1998

[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. U.S.A., 2004.

[SOA08] W3C, “SOAP Version 1.2 Part 0: Primer (Second Edition)”. <http://www.w3.org/TR/soap12-part0/> (accessed in Oct, 2008)

[Tro03] Troccoli, A. Wainer, G. “Implementing Parallel Cell-DEVS” 36th Annual Simulation Symposium, pages 273-280, 2003

[W3C08] W3C. “Simple Object Access Protocol (SOAP) 1.1”. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508> (accessed in Oct, 2008)

[Wai01] WAINER, G.; GIAMBIASI, N. "Timed Cell-DEVS: modeling and simulation of cell spaces ". Discrete event modeling and simulation technologies: a tapestry of systems and AI-based theories and methodologies, **pages 187-214**, 2001.

[Wai02] WAINER, G. "CD++: a toolkit to develop DEVS models", Software – practice and Experience. Vol. 32 (13), **pages 1261-1306**, 2002

[Wai06] G. Wainer. E. Glinsky. "Advanced Parallel simulation techniques for Cell-DEVS models". EUROSIM. Special Issue on Parallel and Distributed Simulation. Vol. 16(2), **pages 25-36**, Sept. 2006

[Wai08] Wainer, G. Web page. "Model Samples".
http://www.sce.carleton.ca/faculty/wainer/wbgraf/samplesmain_1.htm (accessed in Oct, 2008)

[War95] Dale E. Martin, Timothy J. McBrayer, Computer Architecture Design Laboratory, Cincinnati. "Warped a TimeWarp Parallel Discrete Event Simulator"
<http://www.eecs.uc.edu/~paw/warped/doc/index.html> (accessed in Oct, 2007)

[Wik08] Wikipedia. "Personal computer".
http://en.wikipedia.org/wiki/Personal_computers (accessed on Nov. 2008)

[Wol86] Wolfram, S. "Theory and applications of cellular automata". Vol.1. Advances Series on Complex Systems. World Scientific. Singapore. 1986.

[WSD08] W3C. "Web Services Description Language (WSDL) 1.1"
<http://www.w3.org/TR/wsdl> (accessed in Oct, 2007)

[Zha06] Zhang, M.; Zeigler, B.; Hammonds, P. "DEVS/RMI – An auto-adaptive and reconfigurable distributed simulation environment for engineering studies", DEVS Integrative M&S Symposium (DEVS'06). Huntsville, Alabama, U.S.A., 2006

[Zei76] Zeigler, B.P., Theory of Modeling and Simulation, Wiley, N.Y., 1976

[Zei96] Zeigler, B.; Moon, Y.; Kim, D.; Kim, J. G. “DEVS-C++: A high performance modeling and simulation environment”. The 29th Hawaii International Conference on System Sciences. Hawaii , U.S.A., 1996

[Zei99] Zeigler, B.; Sarjoughian H. S. “Support for hierarchical modular component-based model construction in DEVS/HLA”. Simulation Interoperability Workshop. University of Arizona, U.S.A., 1999.

[Zei00] Zeigler, B.P.; T.G. Kim, and H. Praehofer. Theory of Modeling and Simulation. 2 ed. New York, NY: Academic Press, 2000.