# On the Creation of Distributed Simulation Web-Services in CD++

Rami Madhoun, Bo Feng, Gabriel Wainer,

*Abstract*— **CD++ is a toolkit developed to execute discrete event simulations following the DEVS and Cell-DEVS formalisms. Three different versions of CD++ were developed to support different platforms: the stand-alone version runs on workstations, a real-time version runs on embedded platforms, and the parallel version is capable of running on clustered computers. Web service technologies have emerged in recent years to establish new platform for application deployment. We present a way of developing a web-service wrapper to expose the functionality of the CD++ toolkit as a web-service, allowing clients to interact with the toolkit through SOAP messages. In addition, we present a proposed architectural view of how web services can be used to integrate different versions of the CD++ toolkit to be able to run distributed simulations.**

***Terms*— DEVS, Cell-DEVS, CD++, Web Services.**

## I. Introduction

DEVS is a mathematically sound formalism for modeling and simulating discrete event systems where the system of interest has discrete state at any point of time [1]. It depends on splitting the model into components, called atomic DEVS models. Coupled DEVS models can be realized by integrating group of DEVS (atomic and coupled) models. The coupling scheme determines the interconnectivity between the models, as well as the relation with the external world. Cell-DEVS [2] is a formalism used to model systems that can be split into cells. It is an extension to the traditional cellular automata allowing for executing the cells asynchronously with different time delays.

CD++ [3] is a modeling and simulation toolkit built to execute DEVS and Cell-DEVS models. Different versions of CD++ were developed to support different platforms. The standalone version works on Unix/Linux and Windows platforms. The real-time version [4] works on specialized hardware and interacts with the environment to perform real time simulations. The parallel version of CD++ [5] works on distributed memory clusters to execute complex models that would take long time to run on the stand alone version.

This work stems from the need to provide a robust platform to integrate CD++ with other systems (visualization, remote session management, network resources, etc) in a collaborative environment in order to provide a user-friendly interface.

Rami Madhoun, Bo Feng and Gabriel Wainer are in the Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, CANADA, K1S-5B6 (email: gwainer@sce.carleton.ca).

Web service technologies have emerged in recent years to establish new platform for application deployment. It depends on using standard specifications and languages in order to provide machine-consumable services, as opposed to applications consumed by humans, i.e. web applications. In this paper, we present a way of developing a web-service wrapper to expose the functionality of the CD++ toolkit as web-service, allowing clients to interact with the toolkit through SOAP messages. In addition, we present a proposed architectural view of how web services can be used to integrate different versions of the CD++ toolkit to be able to run distributed simulations.

## II. Web Service-Based CD++

Web service technologies rely on standard languages and specifications, which has led to a wide acceptance among programmers and IT professionals. A web service differs from traditional web applications in that it can be accessed "consumed" by another web service/application, dispensing with the need for human interaction required with traditional web pages/applications. In addition, it provides platform independence. The main tools and technologies that have enabled web services can be listed as follows:

- XML [6]: a flexible language used to represent data in machine understandable form. Its semantics is flexible enough to allow the programmer to define the document structure according to the application requirements.
- WSDL [7]: an XML-based language to define and describe the public interface of the service. It contains information for the client to consume the web service.
- WSDD: an XML-based language to define different deployment parameters necessary to deploy the web service.
- UDDI [8]: an XML-based language used to register and query the web service (using UDDI registries).
- XML-Schema [9]: an XML-based language used to define custom data structures within an XML document.
- X-Path [10]: it is an XML-based language used to find different elements within an XML document.
- SOAP [11]: a messaging protocol designed to carry information between different web services. A SOAP message consists of an envelope which has an optional header and a mandatory body.

In the context of our modeling and simulation environment, web services are used to achieve two main objectives:

- To expose the functionality of the CD++ toolkit as a web service, allowing for executing simulations and retrieving the results through web service technologies.

- To use web service technologies, such as SOAP messing, to introduce a mechanism for integrating the different versions of CD++ in a distributed environment.

## III. WEB SERVICE-BASED CD++

The CD++ toolkit is capable of executing two kinds of models, DEVS and Cell-DEVS. To execute DEVS models, the modeler needs to define each atomic DEVS model as a C++ class that is to be integrated in the class hierarchy of CD++. For coupled DEVS models, and Cell-DEVS models, the modeler needs to provide a configuration file in a text format. The configuration file includes (among other things) the coupling scheme for the coupled model, initial values for the cells, rule definition to calculate the state of the cells, etc. In regular invocation of CD++, the user submits the model definition and configuration files to the simulator as arguments. Once the simulation is over, results are written in an output file (if the top level coupled model has output port(s), and log file(s) that is/are used to show and animate the progress of the simulation (using the tools included in the CD++ toolkit). In addition, there are some additional files used for debugging, generated only upon request. In order to integrate the web service technologies in the CD++ toolkit, a web service wrapper was developed to interact with the CD++ toolkit and wrap its functionality to be accessed by web service clients.

Java Native Interface (JNI) [12] was used to establish the interface between the two parts of the wrapper. JNI is a collection of APIs and is part of the Java Virtual Machine (JVM) developed by Sun. It allows Java programs to access functions written in native C/C++ code. In addition, it allows programs written in C/C++ to execute and access Java objects.

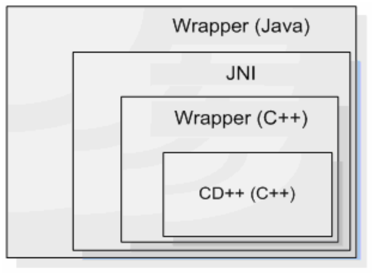


Fig. 1. Components of the web service wrapper

The wrapper functions as a web service interface for the CD++ toolkit that allows the client to:

- Receive the required files to define the model and execute the simulation. These files include: C++ (for DEVS atomic models), coupled model and external inputs.
- Execute the simulation providing the client with the option to check the progress of and terminate the simulation.
- Send the simulation results to the client, including simulation log files or debug information.

We used Apache Axis [13], an open source SOAP engine with HTTP server functionality, which can run as a web application within an application server (in our case, Tomcat application server [14]). The following figure shows a typical interaction between a WS client and CD++ through SOAP.

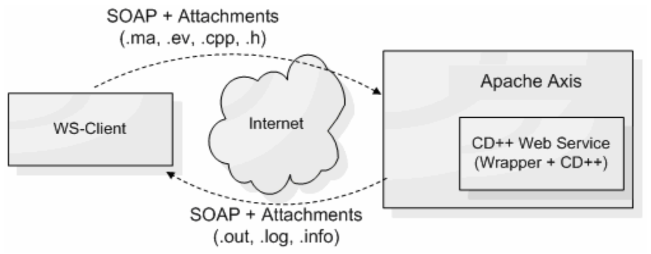


Fig. 2. A typical invocation of the CD++ web service

In order to use the CD++ web service, the client needs to have access to the WSDL document defining the service interface. Then, the client can choose either to generate client-side stubs (in which case he/she can deal with the functions offered by as local methods), or dynamically invoke the service methods through dynamic creation of SOAP messages. The WSDL document consists of five main elements; some of them are used to define an *abstract service/process*, while others define a *concrete service/process*. The abstract service description involves describing the interface of the service in terms of the parameters used and return values. On the other hand, the concrete web service description describes the actual binding of the web service to an actual network protocol (SOAP messages are usually transported over HTTP, because it is one of the most widely used protocols on the Internet). The following figure shows an excerpt of each of the main elements of the WSDL document defining the CD++ web service interface.

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

Fig. 3. Excerpt of the *message* element definition

WSDL *type* elements define non-standard attributes of the messages exchanged between the web service and client. The *message* element defines request and response SOAP messages. In Fig. 3, ***setDEVSModel*** takes four arguments (through the message ***setDEVSModelRequest***): the name of the header file for the class, a *DataHandler* representing the file (sent as a SOAP attachment), a C++ file including the DEVS class implementation, and a *DataHandler* object representing the C++ file. *DataHandler* is "a class that provides a consistent interface to data available in many different sources and formats" [15], in our case, the *DataHandler* represents a file serialized by the Axis server into a SOAP attachment, and deserialized to a file on the other end of the transport process. ***setDEVSModelResponse*** represents the return type of ***setDEVSModel*** (a string stating whether the operation was successful or not).

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

Fig. 4. Excerpt of the *portType* element definition

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

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

Fig. 5. Excerpt of the *binding* element definition

*Binding* defines the binding of the web service messages to an actual protocol (HTTP or SMTP). In addition, it defines the encoding style (RPC/message) and type (encoded/literal).

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

Fig. 6. Excerpt of the *service* element definition

The *service* element groups a number of ports together. Each port links a binding type to a specific Uniform Resource Identifier (URI) used to access the service. The operations offered by the CD++ web service are:

- ***authenticate***: it is responsible for authenticating users and initializing a new session for each successful login.
- ***setMAFile***: it is used to set the model definition file.
- ***setDEVSModel***: it is used to set a DEVS model by C++ header and implementation files.
- ***setEventFile***: set the external input events file.
- ***setSupportFile***: set support files that need to be available to the simulator, such as a file containing the initial values of the cells (in case of Cell-DEVS models).
- ***setExecutionTime***: set the simulation end time.
- ***enableParsingInfo***: informs the simulator to generate information for debugging the Cell-DEVS model.
- ***startSimulationService***: it starts the simulation.
- ***isSimRunning***: check whether the simulation is running.
- ***getCurrentSimulationTime***: checks the current simulation time.
- ***insertExternalEvent***: it is used to insert external events to the model while the simulation is running.
- ***killSimulation***: it is used to terminate the simulation.
- ***retreiveLogFile***: it is used to retrieve the log file generated by the simulator.
- ***retreiveOutputFile***: it is used to retrieve the output file generated by the simulator.
- ***retrieveParsingInfoFile***: retrieves the generated information file that can be used to debug Cell-DEVS models.
- ***retrieveSessionLogFile***: retrieves the session log file which includes the output messages generated by the simulator while running.
- ***logoff***: logs the current user off and terminate the session.

## IV. Implementation Details

JVM cannot load the same native shared library more than once during the lifetime of the class loader used to load the library. In addition, the same library cannot be loaded by two different class loaders. This restriction was imposed on the JVM as of Java 1.2 to avoid class name conflicts [12]. Considering these issues, the wrapper designed to avoid the limitations of the JVM and provide a robust environment for running different simulation sessions concurrently and independently. The wrapper was designed to overcome these limitations with two goals in mind:

- Providing a separate running workspace for each simulation session.
- Making the part of the wrapper responsible for accessing the internal data structures of the simulator a shared library, which should be able to handle all active sessions.

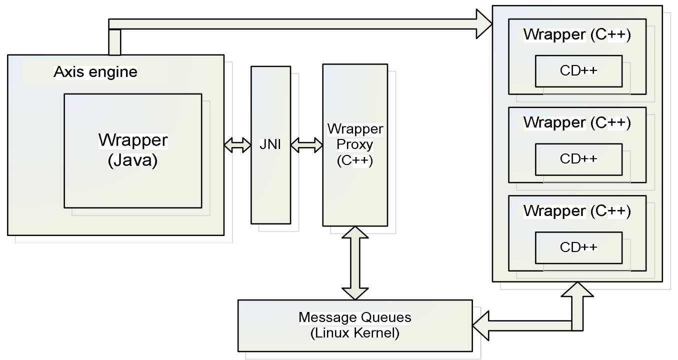


Fig. 7. CD++ wrapper using JNI and message queues

Once the Axis server is started, the CD++ wrapper is loaded and is ready to receive new user connections. Once the user invokes the static method *authenticate*, his/her credentials are verified against a password file stored locally on the server. If the authentication is successful, a new instance of the wrapper is created, and the following steps are performed:

- A new session ID is created and assigned to the user.
- A new directory is created on the server to provide working space for the new session. The source files of the simulator are copied to the new session folder.
- The wrapper invokes a method in the proxy to initialize a new session. This proxy is responsible for the communications between the wrapper instance and the wrapper of the corresponding CD++ session. The proxy is implemented as a shared library and is loaded only once during the lifetime of the Axis server, avoiding the constraint of JVM.
- The proxy creates two message queues, one to send messages from the wrapper to the corresponding CD++ sessions, and the other to receive CD++ messages.
- After initialization at the point of authentication, the user can set the different files and parameters for the model.

- If the user chooses to set DEVS models by sending C++ files, the wrapper will update the makefile (used to compile the simulator and the models) to incorporate the newly added models. In addition, the source code of the simulator is updated to register the new DEVS models.
- When the user starts the simulation, if there is at least one DEVS model, the wrapper will compile the source code of the simulator with the newly added models.
- On CD++, two additional parameters are provided: the full path of the session directory, and the session ID.
- Once the user invokes the *startSimulationService,* CD++ will invoke a method to initialize the session through the wrapper. CD++ will use the full path of the session to query the kernel for the message queues created by the wrapper proxy. These queues are used to communicate with the wrapper instance associated with the current simulation session, as follows:

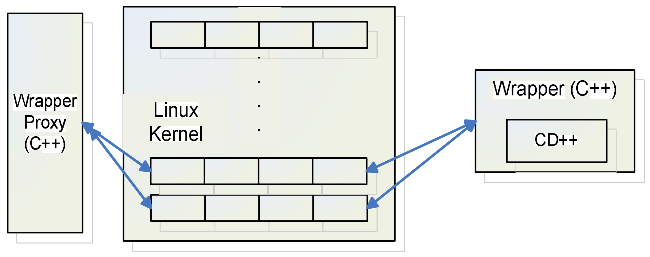


Fig. 8. Message queues for CD++ wrapper proxy

For each session there are two Java and two Linux-POSIX threads. One Java thread executes CD++ and streams its output into the sessions log file; the other is responsible for responding to the client requests while the simulation is running such as getting the current simulation time, inserting external events, etc. On the CD++ side, one thread runs the main simulator, and the other monitors the message queues for an incoming message from the wrapper.

- Once the simulation is over, the user can get the log and output files generated by the simulator.

## V. DEFINING CD++ WEB-SERVICE WITH J2EE

We have followed a similar approach using the J2EE platform to provide the server side and client side support for developing the web services. J2EE consists of technologies that support the development of distributed enterprise applications and services. These technologies fall into three broad categories: components, service and communication [16].

- Component Technologies: used by to create the essential parts of a web service. *Client* components provide support for different types of clients to interact with components on the server side. *Web* components provide a response to requests received via HTTP. *Enterprise JavaBeans* components are designed for business logic. Using these component technologies ensures standardization of the application or service, enabling reusability and portability.
- Service Technologies: support the J2EE container to function properly. Among the required services, we can include naming, deployment, transaction and security.
- Communication Technologies: J2EE requires a set of standard communication mechanisms to bring the components and services together, including Internet protocols (TCP/IP, HTTP, SSL), Remote Method Invocation (RMI) protocols, Messaging and web service technologies.

These technologies offer different benefits: they simplify architecture and development, ensure support for emerging web service standards, allows integration with existing information systems, and it is scalable. Fig. 9 shows the definition of the CD++ web service architecture in J2EE platform.

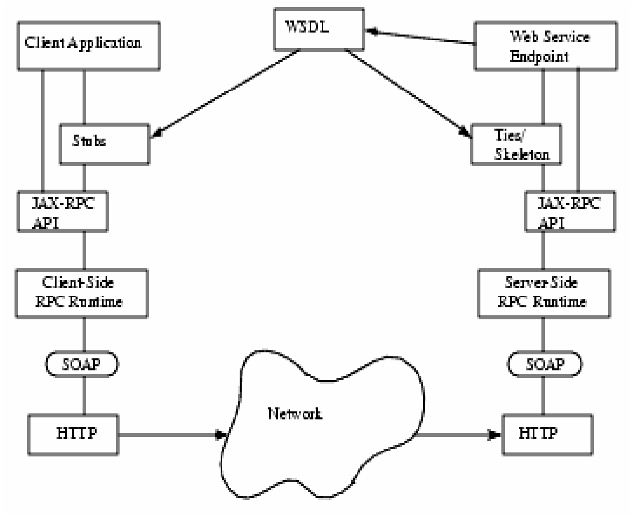


Fig. 9. CD++ web service Architecture

CD++ (*Client application*) invokes methods on generated stubs based on the contents of a WSDL description of a service. These stubs are configured with information about the CD++ web service and its endpoint. The stubs invoke remote methods available in the CD++ web service endpoint. In J2EE, the CD++ web service is described as a set of communication endpoints capable of exchanging messages. An endpoint is made of two parts: the abstract definition of operations (and messages); and the concrete binding of those abstract definitions to a concrete protocol (SOAP over HTTP with a message format). The idea of this separation is to allow the reuse of abstract definitions regardless of the present or future network protocols. The compiler generates the stubs for the client and the skeleton code for the server side. JAX-RPC (Java API for XML-based RPC) is a runtime library that provides services for JAX-RPC mechanisms and APIs. Besides invoking a web service, WSDL can be used to discover a web service through service registries. If a web service is described by its WSDL, a potential client can read the description, decide whether the service meets his needs, and invoke the service automatically.

The following WSDL segment is used:

```
<message name="runMakeFileResponse">
<part name="parameters" element="tns:runMakeFileResponse"/>
</message>
<portType name="GenerateMakeFile">
  <operation name="runMakeFile">
  <input message="tns:runMakeFile"/>
  <output message="tns:runMakeFileResponse"/>
  </operation>
</portType>
<binding name="GenerateMakeFilePortBinding"
       type="tns:GenerateMakeFile">
<soap:binding transport=
 "http://schemas.xmlsoap.org/soap/http" style="document"/>
```

```
<operation name="runMakeFile">
    <soap:operation soapAction=""/>
    <input>
      <soap:body use="literal"/>
    </input>
     <output>
       <soap:body use="literal"/>
```

In our case, ***GenerateMakeFile*** is a web service containing the ***runMakeFile*** function. The WSDL document specifies the binding of a port type (which is a collection of abstract operations), to a concrete transport protocol and data format. Therefore, our port type is bound to SOAP/HTTP as the communication protocol. SOAP can support either document style or RPC style (we used the document style). The ***runMakeFile*** is for DEVS model and it generates Dependencies Files, creates new Makefiles, run them and generates a simulator.

Fig. 10 illustrates this process. We use FTP to upload DEVS models to a CD++ server; then, we invoke the ***GenerateMakeFile*** web service. Once the service is running, ***runMakeFile*** will execute the above procedures step by step and finally generate a simulator including the new model.

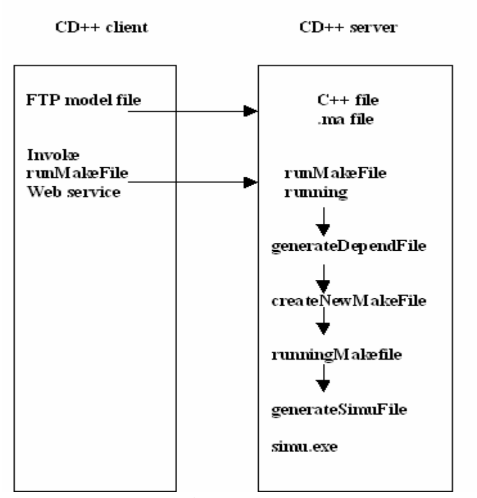


Fig. 10. The *runMakeFile* function

A different web service we created is ***SimuModel***, which is used to execute the model simulators as follows:

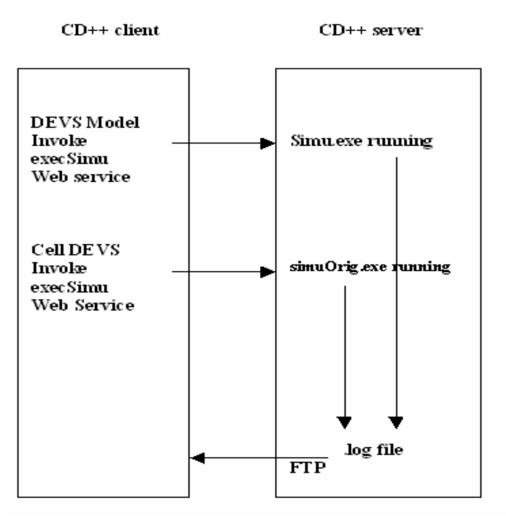

CD++ client
CD++ server
DEVS Model Invoke execSimu Web service
Cell DEVS Invoke execSimu Web Service
Simu.exe running
simuOrig.exe running
.log file
FTP

Fig. 11. The *execSimu* function

J2EE also provides API to create web service clients. For example, we can create a Java application with NetBeans, and then add a web service by specifying WSDL URL, finally consumes the web service. Client applications follow certain steps to use a web service. They must first lookup or locate the service, make a call to the service, and process any returned data. Client applications that run in J2EE environments use the JNDI *InitialContext.lookup* method to locate a service, and to use the *javax.xml.rpc* Service interface API to access the web service. Fig. 12 shows the generics of a Java application named ***JavaCdppClient.*** The application has two web service references (***GenerateMakeFileService*** and ***SimuModelService***) that bind each port to an associated function. For this case, ***runMakeFile*** and ***execSimu*** functions are used. The CD++ web client can invoke these functions as if they were local functions. Web, EJB, and J2EE application client module deployment descriptors use service references to locate the JAX-RPC mapping files as well as the service's WSDL file.

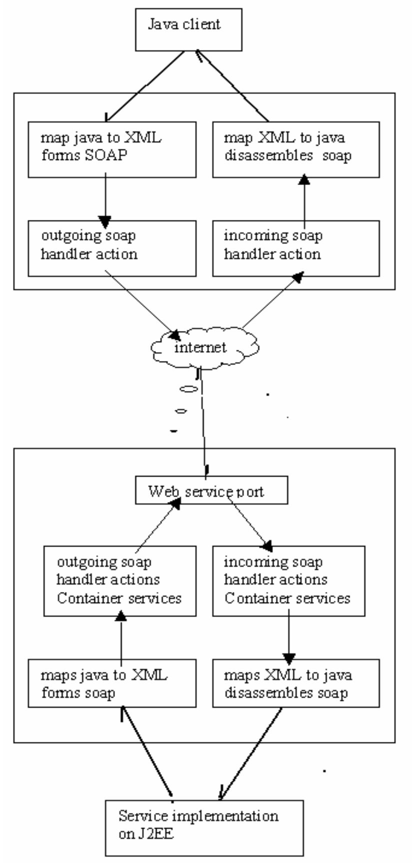
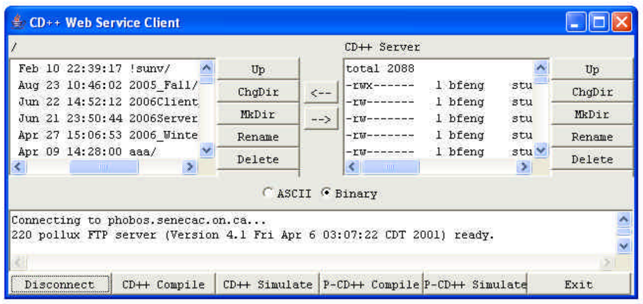

Java client
map java to XML forms SOAP
map XML to java disassembles soap
outgoing soap handler action
incoming soap handler action
internet
Web service port
outgoing soap handler actions Container services
incoming soap handler actions Container services
maps java to XML forms soap
maps XML to java disassembles soap
Service implementation on J2EE

Fig. 12. The CD++ web service on J2EE platform

The service reference element maps a service to a JNDI resource name and also specifies the service endpoint interface for those clients using stubs and dynamic proxies. It also specifies the WSDL file for the service and the qualified name for the service in the WSDL file. The JAX-RPC mapping file specifies the package name containing the generated runtime classes and defines the namespace URI for the service.

Figure 13 shows the interface of CD++ web service client. The application can connect to the CD++ web server, download or upload files, compile and simulate DEVS and Cell-DEVS models.



```
/                                      CD++ Server
Feb 10 22:39:17 !sunv/                 total 2088
Aug 23 10:46:02 2005_Fall/             -rwx------   1 bfeng   stu
Jun 22 14:52:12 2006Client             -rw-------   1 bfeng   stu
Jun 21 23:50:44 2006Server             -rw-------   1 bfeng   stu
Apr 27 15:06:53 2006_Winte             -rw-------   1 bfeng   stu
Apr 09 14:28:00 aaa/                   -rw-------   1 bfeng   stu

Connecting to phobos.senecac.on.ca...
220 pollux FTP server (Version 4.1 Fri Apr 6 03:07:22 CDT 2001) ready.
```

Fig. 13. CD++ client application

## VI. CONCLUSION

We have presented a mechanism for web-service wrappers to expose the functionality of the CD++ toolkit as a web service, allowing clients to interact with the toolkit through SOAP messages. CD++ is a toolkit developed to execute discrete event simulations following the DEVS (Discrete Event System Specification) and Cell-DEVS formalisms.

In order to introduce the web service capabilities to the CD++ toolkit, two approaches were described. The first one depends on developing the simulation service as two main components (C++ and Java) interfaced together through the *WrapperProxy*. The advantage of this approach is that C++ and Java were used were they fit the best. C++ is fast and well-integrated with the original CD++ code, and Java is well-supported by web service middleware. The other approach depends on using J2EE platform in order to expose the functionality using web services, as shown in the following figure.

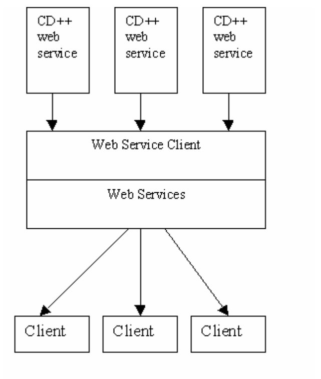


Fig. 18. CD++ Middleware

J2EE can be used to develop web service clients using EJB components, which may themselves be service endpoints as well as clients for other web services. So it is possible to create CD++ middleware with J2EE API. We can integrate various CD++ web services in the middleware and provide high performance for the CD++ clients.

The web service functionality of the CD++ service has two main applications. The first application is it allows for developing a distributed simulation engine to execute complex models in a heterogeneous environment. The other application is a collaborative system that integrates the simulation capabilities of CD++ with visualization capabilities, network management, and resource management resources, as shown in Figure 19.

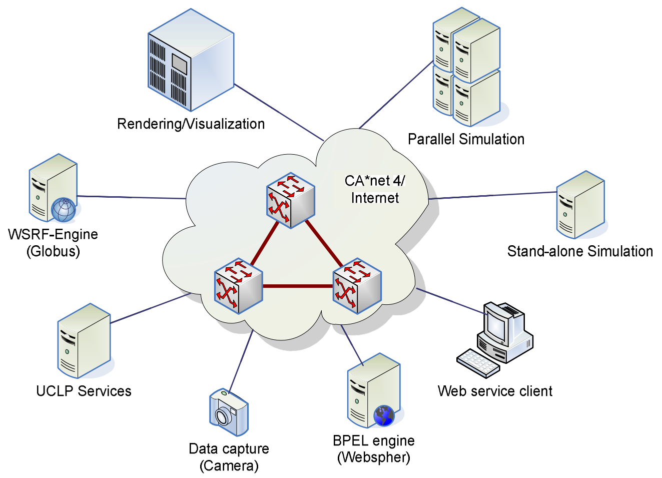


Fig. 14. Parallel simulation and visualization environment based on web-service CD++

In the latter application, the user has a friendly interface to submit the model for execution using SOAP (and its extensions) to the CD++ service and then examine the simulation progress through high-end visualization of the model components. This will dispense the user with the need to examine the details of the original log files generated by CD++.

## REFERENCES

[1] B. Zeigler; T. Kim; H. Praehofer: Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems, Academic Press, 2000.
[2] G. Wainer; N. Giambiasi: "Application of the Cell-DEVS Paradigm for Cell Spaces Modeling and Simulation", *Simulation*, Vol. 71, No. 1, pp. 22-39, January 2001.
[3] G. Wainer: "CD++: a Toolkit to Define Discrete-Event Models", *Software, Practice and Experience, Wiley*, Vol. 32, No 3. pp. 1261-1306. November 2002
[4] E. Glinsky; G. Wainer: "Performance Analysis of Real-Time DEVS models", In *Proceedings of 2002 Winter Simulation Conference*, San Diego, U.S.A.
[5] A. Troccoli; G. Wainer: "Implementing Parallel Cell-DEVS", In *Proceedings of Annual Simulation Symposium.* Orlando, FL. U.S.A. 2003.
[6] World Wide Web Consortium (W3C), Extensible Markup Language, http://www.w3.org/XML/
[7] World Wide Web Consortium (W3C), Web Service Description Language 1.1. http://www.w3.org/TR/wsdl
[8] Organization for the Advancement of Structured Information Standards (OASIS), UDDI technical committee, http://www.uddi.org/
[9] World Wide Web Consortium (W3C), XML-Schema, http://www.w3.org/XML/Schema
[10] World Wide Web Consortium (W3C), X-Path, http://www.w3.org/TR/xpath
[11] World Wide Web Consortium (W3C), http://www.w3.org/TR/soap/
[12] S. Liang: Java Native Interface (JNI), Programmer's Guide and Specification, Addison-Wesley, 1999
[13] Web Services-Axis, http://ws.apache.org/axis/
[14] Apache Tomcat, http://tomcat.apache.org/
[15] JavaBean Activation Framework API Documentation, http://java.sun.com/products/javabeans/glasgow/javadocs/javax/activation/package-summary.html
[16] http://www.netbeans.org/
[17] http://java.sun.com/javaee/