Landscape Management System

A WebFlow Application

Tomasz Haupt[1], Erol Akarsu, Geoffrey C. Fox
NPAC at Syracuse University

The work presented in this report has been done in collaboration with

Jeffrey Holland, Billy Johnson, and Clay LaHatte
CEWES, Vicksburg, MS

Fred L. Ogden
University of Connecticut

W. Michael Childress
Shepherd Miller, Inc.

Abstract

This paper describes a pilot Web-based implementation of the Landscape Management System (LMS). The Web-based implementation extends the Watershed Modeling System by adding the capability to download the input data directly from the Internet, and execute the simulation codes on a remote high-performance host. This makes it possible to run LMS anywhere from a networked laptop. Furthermore, our system allows for constructing complex simulations by coupling several independently developed codes into a single, distributed application. We demonstrate this feature by integrating the Casc2d and Edys simulations. The Web-based LMS is implemented as a WebFlow application. WebFlow is a modern three-tier commodity standards-based HPDC system that integrates a high-level graphical user interface (Tier 1), distributed scalable object broker middleware (Tier 2) , and a high-performance back end(Tier 3).

1 Introduction

1.1 WebFlow Mission

Programming tools that are simultaneously sustainable, highly functional, robust and easy to use have been hard to come by in the HPCC arena. This is partially due to the difficulty in developing sophisticated customized systems for what is a relatively small part of the worldwide computing enterprise. Thus we have developed a new strategy - termed HPcc High Performance Commodity Computing[1] - that builds HPCC programming tools on top of the remarkable new software infrastructure being built for the commercial web and distributed object areas. This leverage of a huge industry investment naturally delivers tools with the desired properties with the one (albeit critical) exception that high performance is not guaranteed. Our approach automatically gives the user access to the full range of commercial capabilities (e.g., databases and compute servers), pervasive access from all platforms and natural incremental enhancement as the industry software juggernaut continues to deliver software systems of rapidly increasing power.

Our research addresses the need for high-level programming environments and tools to support distance computing on heterogeneous distributed commodity platforms and high-speed networks spanning across

[1] Contact person, haupt@npac.syr.edu

labs and facilities. More specifically, we are developing a portable system based on industry standards and commodity software components that provide a seamless access to remote resources through Web-based user interfaces or customized application GUIs.

We have successfully employed preliminary versions of WebFlow for two classes of applications: Quantum Monte Carlo simulations[2] (QS) developed within NCSA Alliance Team B (described is Section 3.1), and the Landscape Management System (LMS) - the primary subject of this report. In addition, WebFlow is used for implementation of the Gateway system at ASC MSRC, as described in Section 3.2.

1.2 Seamless Access to Remote Resources

As described above, one of the most important goals of the WebFlow system is to provide a seamless access to remote resources. That is, our goal is to create an illusion that all resources needed to complete the user tasks are available locally (an analogy: an NFS mounted disk or a network printer). In particular, an authorized user can allocate the needed resources without explicit login to the host controlling the resources. And by "resources" we mean all hardware and software components needed to complete the task - including but not limited to - compute engines from workstations to supercomputers, storage, databases, instruments, codes, libraries, and licenses. A critical issue in providing access to remote resources is security of access (user authentication and authorization), which is a primary concern of the Gateway system implementation and it is discussed in Section 3.2.

1.3 High-Level Graphical User Interfaces

Another of our goals is to provide user-friendly, high-level graphical user interfaces (WebFlow front end) for the user applications. We simultaneously follow several approaches, as the requirements for different applications vary greatly. For Quantum Simulations we provide a Web-accessible front end (a Java applet) that allows the user to visually compose an application from pre-existing modules following a dataflow paradigm. For the Gateway projects we will use a Web-based PSE[3] (an expert system) that allows the user to interactively define a problem, identify resources (both software and hardware), submit the task for execution and, finally, analyze the results.

For LMS we face different requirements. LMS applications are composed of independent modules selected by the application developers, and the end user only selects the application that best suites the problem at hand. We refer to this as the "navigate and choose" paradigm. Another unique feature of the LMS front end is that it automates the retrieval of necessary input data from Internet repositories such as the USGS Web site.

1.4 Three-Tier Architecture

To satisfy the requirements of supporting multiple front ends and a variety of back end platforms, we implement WebFlow as a three-tier system, as shown in Fig. 1.

The heart of the system is the middle tier that provides a mapping between the application-independent Abstract Task Specification (ATS) and the platform-independent Resource Specification (RS). In addition, the middle tier offers a number of services that simplify the development of applications, and it provides support for interactions between usually distributed components of the application.

WebFlow applications are composed of independently developed modules. Module developers do not need to concern themselves with issues such as allocating and running modules on various machines, creating connections among modules, sending and receiving data across these connections, or running several modules concurrently on one machine. The WebFlow's middle tier hides these management and coordination functions from the developers who have only a limited knowledge of the system on which the modules will run.

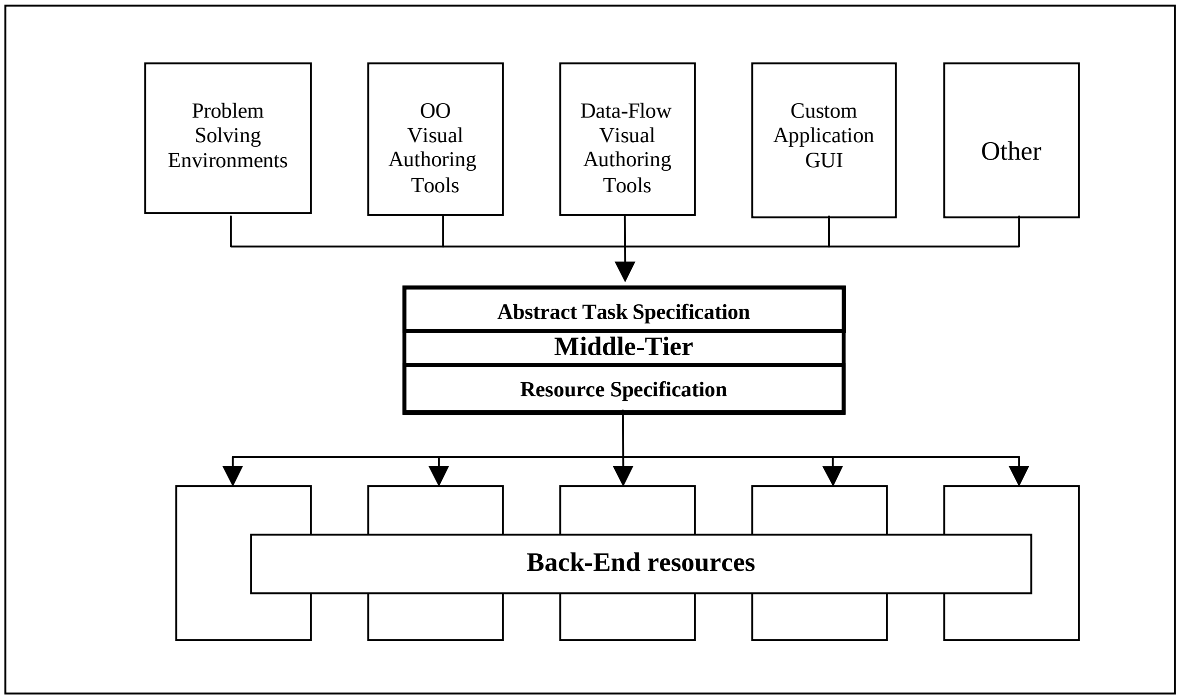

Problem Solving Environments
OO Visual Authoring Tools
Data-Flow Visual Authoring Tools
Custom Application GUI
Other
Abstract Task Specification
Middle-Tier
Resource Specification
Back-End resources

Fig. 1: Three-Tier architecture of the WebFlow system: visual authoring tools comprise Tier 1 (front end), distributed object-based, scalable, and reusable Web server and Object broker Middleware form Tier 2 (middle tier), while computational and data resources constitute Tier 3 (back end).

2 WebFlow Design

2.1 Implementation Strategy

Following our HPcc model, we build WebFlow in strict conformance to industry standards. We avoid custom protocols and follow the general trends, and whenever feasible we use commodity software components. We believe that such an approach will simplify maintenance of our system and will let us take advantage of the new developments introduced by industry. However, our solution is not unique. Web technologies are being developed at unprecedented fast pace, and competition between vendors often leads to multiple standards. We had to make choices between the competing standards (for example, we chose CORBA and not Microsoft's DCOM or Sun's Java RMI to implement the distributed objects), and it remains to be seen whether we made the right choice. Nevertheless, some of our choices, such as https (http over SSL) and XML, seem to be non-controversial. When there are no standards available (for example, AST or RS), we participate in creation of such standards (DATORR[4], NCSA Alliance).

2.2 WebFlow Design

WebFlow consists of objects that interact with each other through event notifications. The basic object is a WebFlow server, which is a CORBA container object (also referred to as a context). The server contains other containers and modules. The hierarchy of the containers and modules is composed to fit the application at hand. An example is shown in Fig. 2. In this particular configuration we have three WebFlow

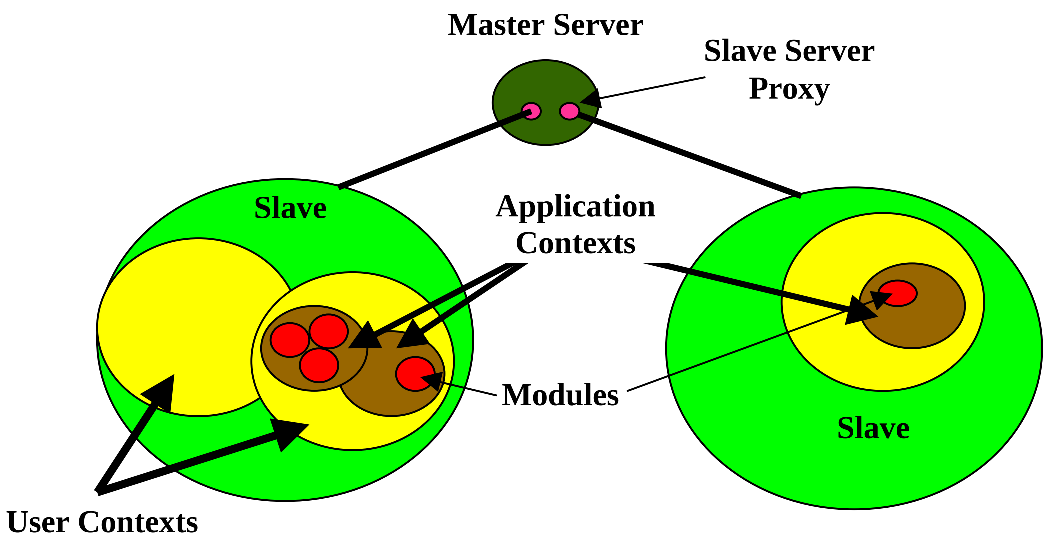
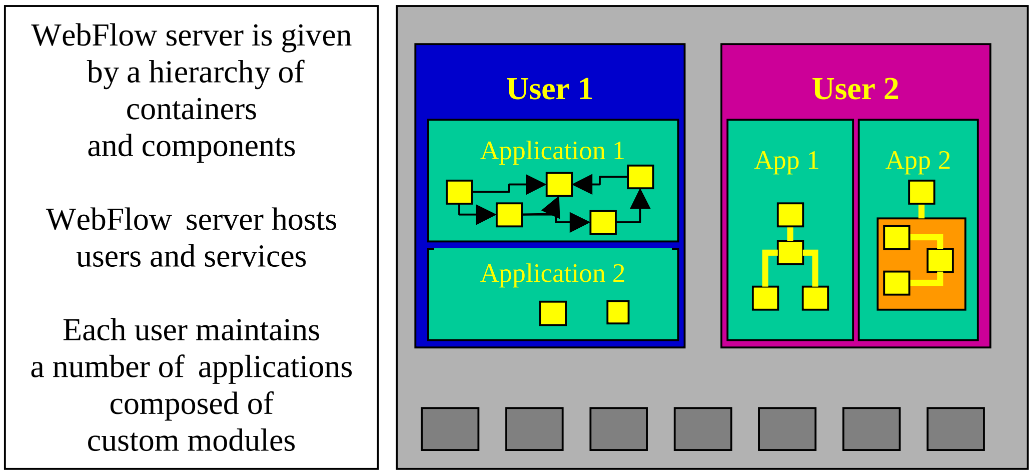


Figure 2: Example of WebFlow object hierarchy. See text for explanations.

servers, each running as a separate process (typically on three different hosts), one of which serves as a master, the two others as its slaves. The master server publishes its reference (IOR – Interoperable Object Reference) so that clients can connect to it and request its services. In addition, the master keeps references to its slaves (slave server proxies), which are returned to the client upon request. Within the slave WebFlow context one or more users creates his or her own contexts, which in turn contain the user applications. The application consists of modules. In this way a number of users can create several applications, each made of independent (possibly hierarchically composed) modules and WebFlow middle-tier services (cf. Fig. 3). Modules and services are executable written in Java. There is no difference between the module and the service, other than that the module is provided by the user (or the application developer) and placed within the user's application context, while the service is provided by the WebFlow system itself and implements a standard task such as job submission or access to mass storage or file manipulation. Applications differ in how the modules interact with each other. In Fig. 3 we illustrate some


WebFlow server is given by a hierarchy of containers and components

WebFlow server hosts users and services

Each user maintains a number of applications composed of custom modules

Figure 3: WebFlow server

of the possibilities. In application 1 of user 1, modules interact in the most general way: each module can invoke any public method of the other modules. In application 1 of user 2 a data-flow model is used: the modules exchange data through their input and output ports. Application 2 of user 1 consists of two modules that do not interact. They can run concurrently, and they can exchange messages “privately”, that is, without WebFlow help say, using MPI. Actually, a single module may represent a data parallel application implemented in HPF, or C++ with MPI. In LMS we use yet another way of communicating between the modules, as described in Section 4 below.

The modules (and services) are technically CORBA objects implemented in Java. However, that does not mean that the actual functionality of the module must be implemented in Java. Legacy applications can be easily encapsulated as CORBA objects and thus used as WebFlow modules (as we did in the case of LMS, see Section 4). But typically a module serves merely as proxy for services rendered by the back end. As an example, the middle tier provides the service of submitting a job using Globus[5]. To submit a job the service acts as a client to the GRAM (Globus Resource Allocation Manager) server. More specifically, it sends a request expressed in the Globus RSL (Resource Specification Language) that defines the target machine, location of executable and input files, as well as instructions what for dealing with the standard output and standard error streams. Optionally, through GASS (Global Access to Secondary Storage) both executable and input data sets can be staged prior to the execution of the job, and output files can be uploaded to a specified location after the job is completed. The Java code of the WebFlow proxy module generates the RSL command, and the WebFlow module developer never needs to see the actual application, let alone make an attempt to rewrite the application in Java.

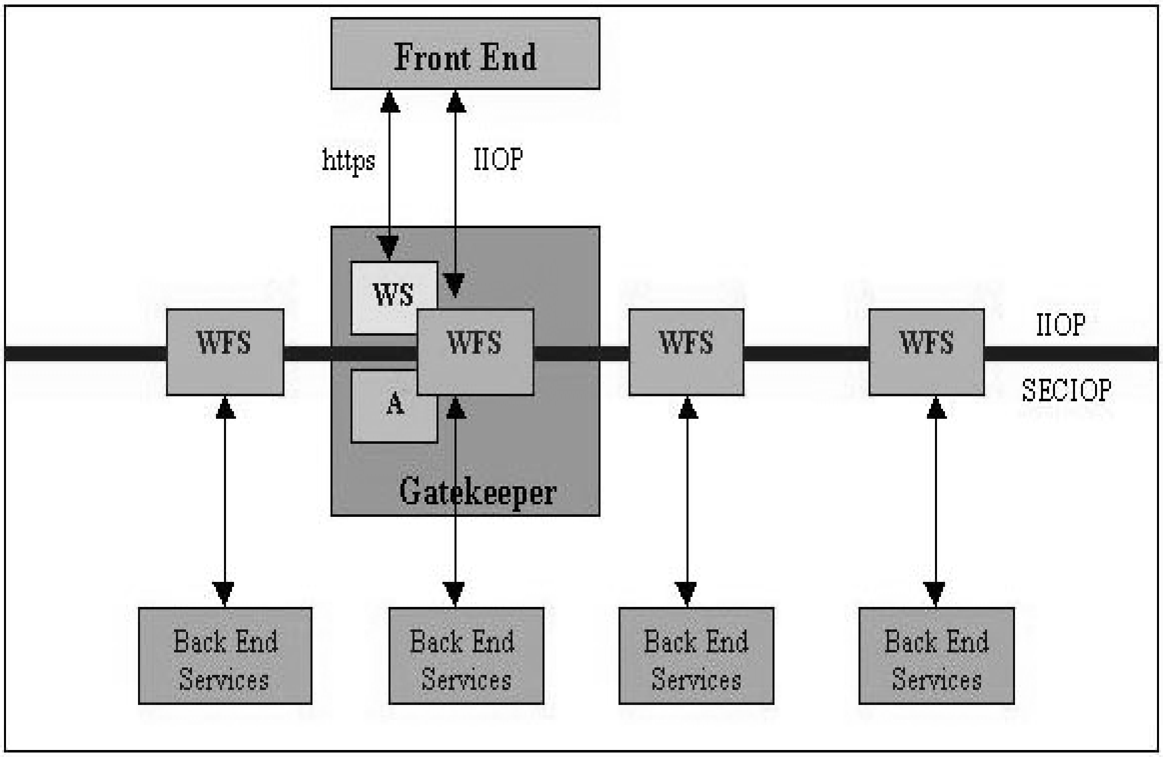
WebFlow servers are started by the WebFlow administrator using the command *java WebFlow.Server file.conf,* where file.conf is the server configuration file. Objects within WebFlow server contexts (such as Application context or module) are created using methods of the WebFlow server addContext*(contextName)* and *addModule(moduleName)*. Interactions between modules can be defined using WebFlow server method *attachEvent(ORES,EventName,ORT,MethodName)*, where ORES is an the object reference of the event source, Event Name is the name of the event fired by the module that serves as the event source, ORT is the object reference of the event target, and MethodName is the name of the method of the target module that is to be invoked as the response to the event. This information is stored dynamically by the event adapter, which allows us not to have to implement the event target as the Java or JavaBeans event listener, which would violate our design goal for all modules to be developed independently of each other and be reused by different applications. Instead, we use dynamical invocation of methods of remote objects through both DII (Dynamic Interface Invocation) and DSI (Dynamic Stub Invocation). Events (which, as with everything else in WebFlow are CORBA objects) encapsulate data to be sent from one module to the other.

2.3 Middle Tier

The middle tier is given by a mesh of WebFlow servers. If the client is implemented as a Java applet, one of the WebFlow servers, namely, the master server, plays the role of gatekeeper. It is accompanied by a (secure, i.e., SSL based) Web server, as shown in Fig. 4.

The user downloads the applet from the Web server (and since https protocol is used she is authenticated in this process and, optionally through the Akenti server receives authorization to use the WebFlow resources – see Section 3.2 for more details on WebFlow security). The applet reads the master server IOR, which is posted as an html document, and establishes communication with the WebFlow. Because of the Java sandbox security mechanism, the applet can communicate exclusively with the master WebFlow server, which runs on the same host as the Web server. This is the reason we introduced the proxy servers maintained by the master. The client (i.e., the applet) communicates with the remote slave WebFlow server through these proxies and constructs the WebFlow context hierarchy as needed using WebFlow server methods. In particular, it builds an application within a selected context.

Simple applications can be run on a single WebFlow server. We have introduced a distributed middle tier, a mesh of WebFlow servers, because, in general, different hosts may have access to different resources (hardware, operating system, software, access control, etc.). As shown in Fig. 5, each WebFlow server manages a particular set of user modules and services. The communication between modules on different hosts is managed transparently by the middle-tier for the user.



WFS: WebFlow Server **WS**: secure Web Server **A**: AKENTI server

Fig. 4: Architecture of the WebFlow middle tier

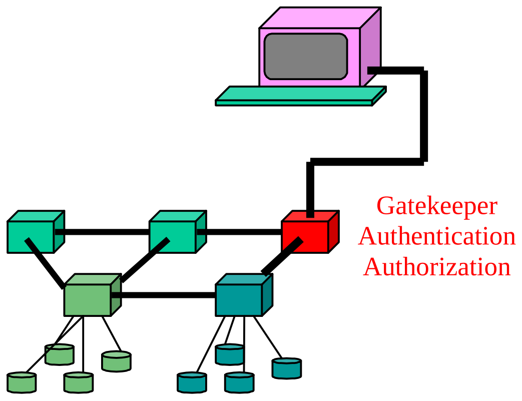


Fig. 5: Mesh of WebFlow Servers implemented as CORBA objects that manage and coordinates distributed computation.

3 WebFlow applications

3.1 Quantum Monte Carlo Simulations (QS)

The QS project is a part of Alliance Team B, and its primary purpose is to demonstrate the feasibility of layering WebFlow on top of the Globus metacomputing toolkit. WebFlow thus serves as a job broker for Globus, while Globus (or more precisely, GRAM – Globus Resource Allocation Manager) takes responsibility of actual resource allocation, which includes authentication and authorization of the WebFlow user to use computational resources under Globus control (Fig. 6).

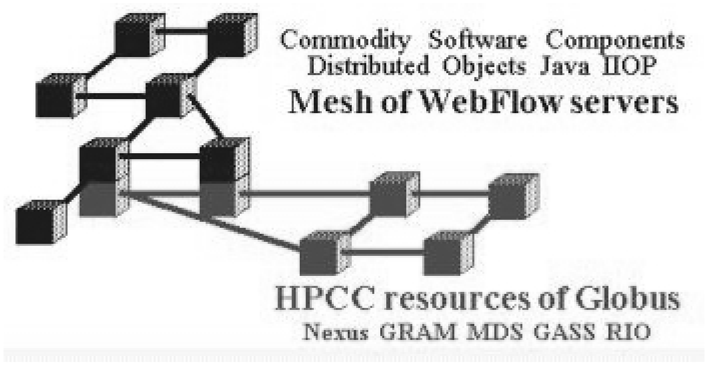


Fig.6: WebFlow over Globus

The system is given by two networks: one controlled by the WebFlow (workstations) and the other controlled by Globus (HPCC resources). There must be at least one node with both WebFlow server and GRAM client installed.

This application can be characterized as follows. A chain of high-performance applications (both commercial packages such as GAUSSIAN or GAMESS or custom developed) is run repeatedly for different data sets. Each application can be run on several different (multiprocessor) platforms and, consequently, input and output files must be moved between machines. Output files are visually inspected by the researcher; if necessary, applications are rerun with modified input data sets. The output file of one application in the chain is the input of the next one, after a suitable format conversion.

GAUSSIAN and GAMES are run as Globus jobs on Origin2000 or Convex Exemplar at NCSA, while all file editing and format conversion is performed on the user's desktop.

For QS we are using the WebFlow editor applet as the front end (Fig. 7). The WebFlow editor provides an intuitive environment to visually compose (click-drag-and-drop) a chain of data-flow computations from preexisting modules. In the edit mode, modules can be added to or removed from the existing network, and connections between the modules can be updated. Once created, the network can be saved (on the server side) to be restored at a later time. The workload can be distributed among several WebFlow nodes (WebFlow servers) with interprocessor communications being taken care of by the middle-tier services. Thanks to the interface to the Globus system in the back end, execution of particular modules can be delegated to powerful HPCC systems.

The visual representation of the application is translated into the Abstract Task Specification and sent to the middle tier as an XML document. The AST itself is expressed in terms of DTD (Document Type Definition), and it is listed in Fig. 8. XML documents conforming to this specification provide all necessary information to create the WebFlow context hierarchy (see Fig. 9 for a simple example); the <taskspec> tag provide references for the user context, and the <task> tag describes the application context. The computational graph visually created by the user is represented by the sequence of the <module> and <connection> tags.

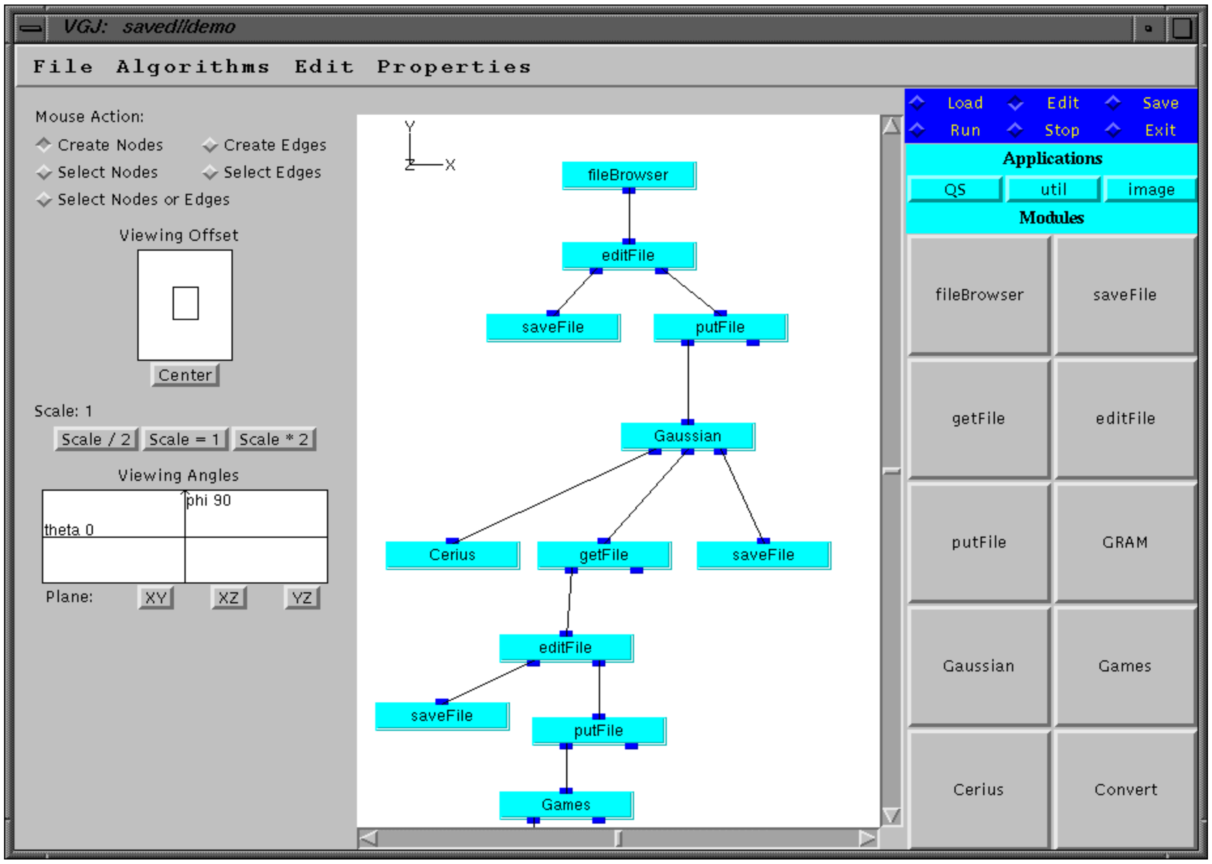


Fig.7: WebFlow data-flow editor applet used as the front end for Quantum Monte Carlo simulations. It allows the user to perform interactive, visual composing of the application from pre-existing modules.

The XML document is parsed by a middle-tier service and is used to generate a Java code on-the-fly that serves as internal client for the WebFlow servers (Fig. 10). This means that once the application is committed by pressing the run button nothing can be changed until the application is completed or aborted. However, since each module comes with its own front-end control panel, the parameters that the module developer made visible to the user can be modified, even at runtime.

```
<!ELEMENT taskspec (task)+>
 <!ATTLIST taskspec
          UserContextRef CDATA #REQUIRED
          UserName       CDATA #REQUIRED
 >
<!ELEMENT task  ((task | module)*,connection*) >
 <!ATTLIST task
         AppName  CDATA #REQUIRED
>
 <!ELEMENT module (#PCDATA) >
 <!ATTLIST module
          modulename CDATA #REQUIRED
          host       CDATA #REQUIRED
 >
 <!ELEMENT connection (out,in)>
 <!ELEMENT in EMPTY>
 <!ELEMENT out EMPTY>
 <!ATTLIST out
          modulename CDATA #REQUIRED
          eventname  CDATA #REQUIRED
 >
 <!ATTLIST in
          modulename CDATA #REQUIRED
          method     CDATA #REQUIRED
  >
```

Fig.8 XML Document Type Definition used as the Abstract Task Specification.

```
<taskspec
UserContextRef="IOR:000000000000001649444c3a576562466c6f772f50726f78793a312e3000000000000
0010000000000000003000010000000000f3132382e3233302e32312e3231320000047400000000000100000
000036f66c3800088f6800000002" UserName="haupt">
<task AppName="SimpleTest">
<module modulename="FileBrowser.1" host="localhost">
</module>
<module modulename="SaveAs.2" host="localhost">
</module>
<connection>
<out modulename="FileBrowser.1"  eventname="Event101"/>
<in modulename="SaveAs.2" method="Method1"/>
</connection>
</task>
</taskspec>
```

Fig.9 A simple XML document conforming to the DTD specification of Fig. 8.

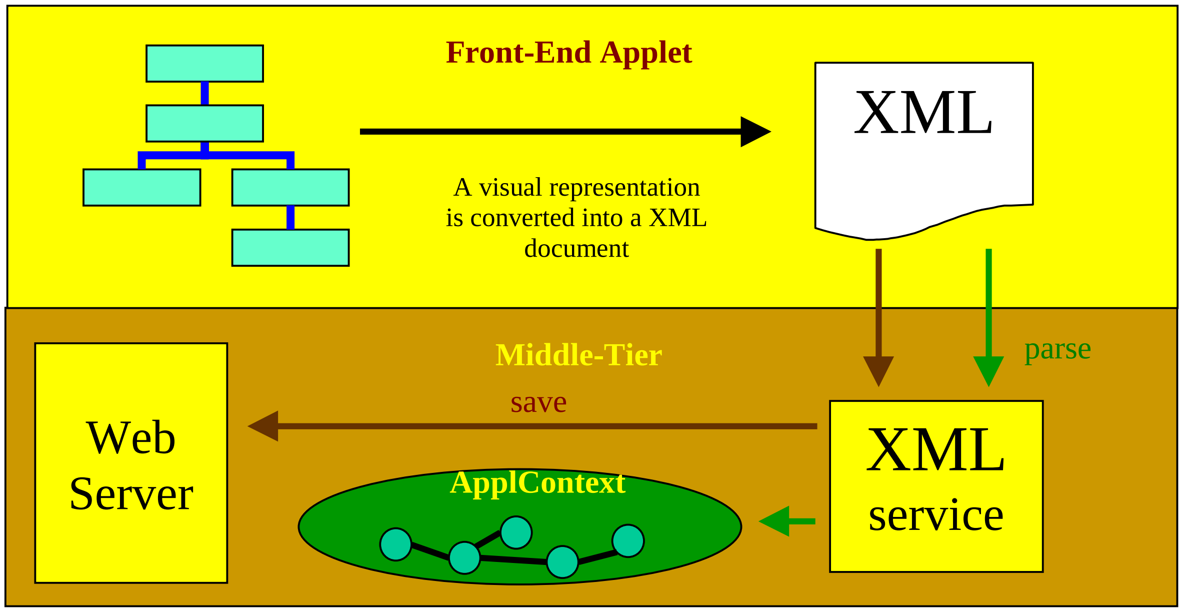


Fig.10: A visual representation of the application is translated into an XML document and sent to the middle tier. An XML parser, implemented as a service, processes the document and generates the Java code that serves as a client for the WebFlow server to create the context hierarchy, instantiate modules, and bind events.

3.2 Gateway Project

This project performed in collaboration with ASC MSRC and the Ohio Supercomputing Center is in the very early stage (it started Jan 1, 1999). Our goal is to combine the CCM Problem Solving Environment under development at OSC with secure, seamless access to resources provided by the WebFlow system. Security is the most important new aspect of WebFlow to be addressed in this project. It is required that Gateway will operate in an environment protected by Keberos5 in conjunction with SecurID technology.

Our suggested strategy is to run the client (an applet) in a keberized environment that will allow the user to generate the Keberos ticket locally without sending cleartext passwords through the networks. Then the user connects the Web Server to download the front-end applet. The Web server creates the user context in a new process on a selected host that runs with the user’s credentials. The applet will communicate with the WebFlow servers using CORBA security services implemented on top of the Keberos5. Access to the back-end services is provided by Globus GRAM-keeper linked against the Keberos5 libraries rather than the default SSL used by Globus.

4 LMS

4.1 Description of the Project

This project is sponsored by CEWES MSRC at Vicksburg, under the DoD HPC Modernization Program, Programming Environment and Training (PET). The pilot phase of the project can be described as follows. A decision-maker (the end user of the system) wants to evaluate changes in vegetation in some geographical region over a long time period caused by some short-term disturbances such as a fire or human activity. One of the critical parameters of the vegetation model is the soil condition at the time of the disturbance. This, in turn, is dominated by rainfalls that possibly occur at that time. Consequently, the implementation of this project requires (cf. Fig.11):

- Data retrieval from many different remote sources (web sites, databases)
- Data preprocessing to prune and convert the raw data to a format expected by the simulation software.
- Execution of two simulation programs: EDYS for vegetation simulation including the disturbances and CASC2D for watershed simulations during rainfalls. The latter results in generating maps of the soil condition after the rainfall. The initial conditions for CASC2D are set by EDYS just before the rainfall, and the output of CASC2D after the event is used to update parameters of EDYS.
- Visualizations of the results.

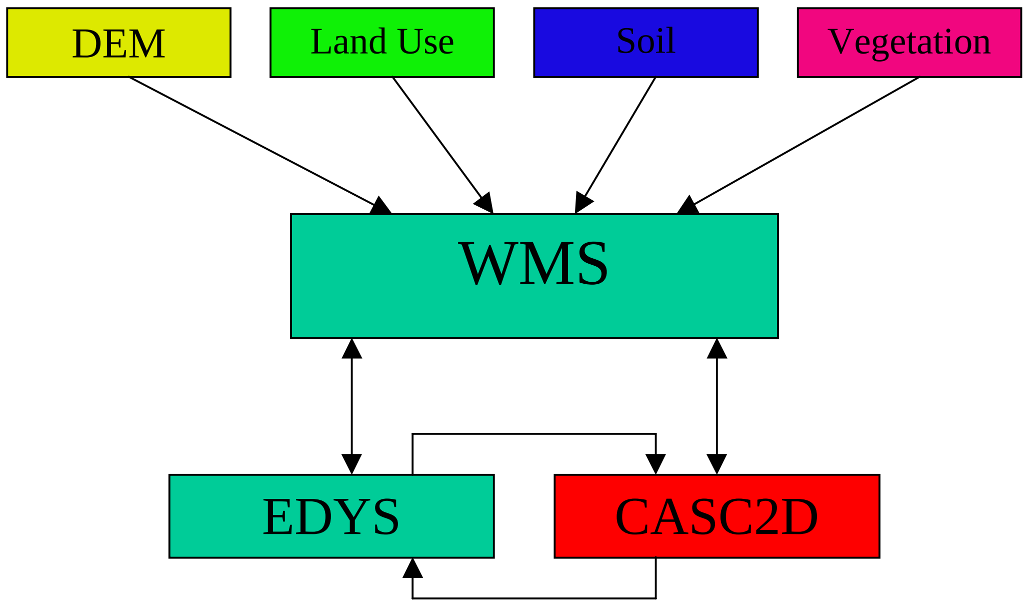


Fig. 11: Logical structure of the LMS simulations implemented by this project

The purpose of this project was to demonstrate the feasibility of implementing a system that would allow launching and controlling the complete simulation from a networked laptop. We successfully implemented it using WebFlow with WMS and EDYS encapsulated as WebFlow modules running locally on the laptop and CASC2D executed by WebFlow on remote hosts.

4.2 Interaction between Casc2d and Edys simulations

The casc2d[6] and Edys[7] codes were developed independently of each other. We cannot provide many details on these codes as that goes beyond our expertise. Please contact the authors of the codes directly for further information. The discussion presented here is rudimentary and is concentrated on issues directly relevant to WebFlow-based implementation**.**

Casc2d simulates watersheds. It runs in a loop over rainfall events, and in each iteration of the loop the program simulates water flow in the area of interest. Once the simulation of the rainfall event is completed (according to some predefined criteria) the simulation switches to a “dry” mode in which the program simulates the condition of soil in the absence of precipitation. A new rainfall starts a new iteration.

Edys simulates the evolution of vegetation taking into account the soil condition at the beginning of simulation. During the simulation, it uses averaged precipitation data rather than data describing actual rainfall, event after event. Edys runs for a specified time period, and before it exits it saves its state on a disk. This means that the simulation can be resumed later.

The accuracy of the Edys simulation can be improved by coupling it with casc2d, that is, by feeding Edys with accurate data on soil condition after each rainfall. We implemented the coupling in the following way (cf. Fig. 12). We start the simulation with casc2d (on a host running Unix). It reads its input files and determines the time of the first rainfall. It writes the data to the disk and starts the loop over events. However, in each iteration, before proceeding with the simulations, casc2d waits until new data on the soil condition generated by Edys are available. Technically, every ten seconds it checks the modification time of its input files[*]. In the meantime, the data written by casc2d are sent to the host running Edys (a laptop running Windows NT). The received data include the date of the next rainfall. The Edys simulation is launched and continued until that date. The simulation program exits, and its output files are sent to the host of casc2d. Casc2d detects the arrival of the new data and resumes the simulations. As soon as the current is completed, casc2d saves the results to a file and begins a new iteration. The results are sent to the laptop, Edys is run until the next event, and its output is sent to the Unix host to let casc2d continue. This pattern is repeated until all rainfall events are processed. Then casc2d exits, and the final run of Edys is performed. The run terminates at a predefined date, typically 20 years after the first rainfall.

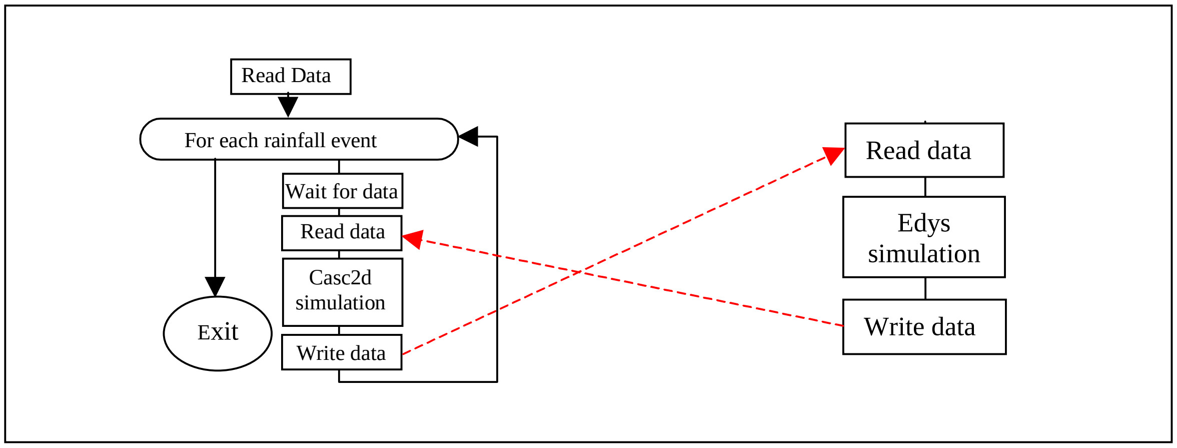


Fig.12: Exchange of data between casc2d (left-hand side) and Edys (right-hand side). It is important to note that casc2d is run only once. It pauses while waiting for the new data and quits only after all events are processed. In contrast, Edys is launched each time the data are needed.

[*] 10 second in a negligible short time as compared to an average time to complete one casc2d iteration, which is about 15 minutes on SGI O2 workstation.

4.3 LMS Middle Tier

This application requires two computational modules: one encapsulating EDYS to be run on WindowsNT box and the other encapsulating CASC2D to be run on a Unix workstation. Consequently, we need two application contexts, one on each machine. As usual, we also need the master server, which we place on the WindowsNT box, as shown in Fig. 13, because we run the client application there. In addition, we run Web servers on both machines. They are needed primarily to exchange the data between the modules. We also publish the master IOR on the Web server on the WindowsNT side.

The servers are started manually using the *java WebFlow.Server file.conf* command, with a different configuration file for each servers. For the master server, we specify the full path of the file where the IOR

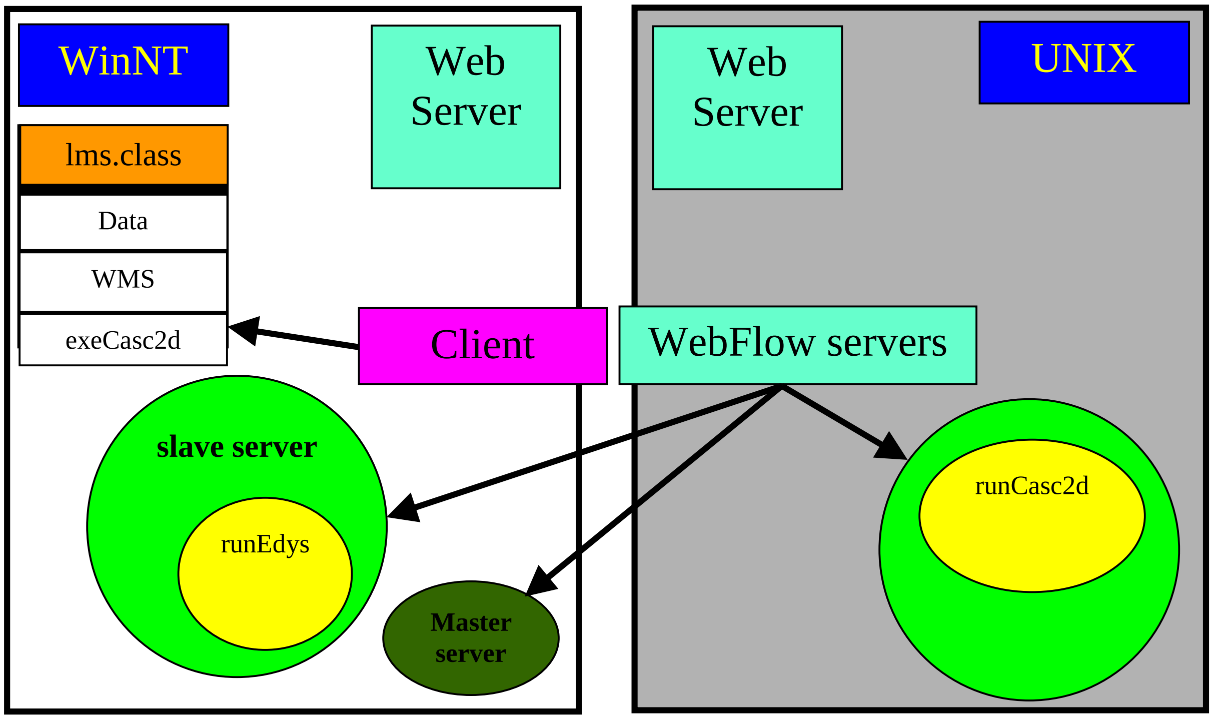


Fig. 13: WebFlow implementation of LMS

is to be stored. For the slave server, we specify the URL at which the IOR is available, and define the modules under the server control (see Fig. 14). At this time the configuration files must be generated manually. We believe that in the future releases of the WebFlow system we will be able to automatically generate the configuration files from the IDL specifications.

The servers are accessed by the client code that is a part of the LMS front end. Fig. 15 shows the relevant part of the code. First, the client initializes the ORB object (line 1), and then reads an IOR of the master server from the URL (line 3). For the IOR it creates a CORBA object obj (line 4) and casts it to the correct type: WebFlowContex (line 5). Now it can call methods of this object. In lines 9 and 10 it uses the method getWFServer(serverName) to retrieve references to both slave servers. It adds module "runEdys" to the ntserver context (line 11) and module "runCasc2d" to osprey4 contexts (line 12). In line 13 it casts obj p2 to type runCasc2d, as it needs to invoke one of its methods (in line 16). Then it connects the modules, event "EdysDone", fired by module runEdys, will invoke method "runAgain()" of module runCasc2d (line 14), and event "casc2dDone", fired by runCasc2d, will invoke method "receiveData()" of runEdys. Now everything is ready to start the execution. It is triggered by invoking method run() of module runCasc2d.

```
A)
Server name = master
File=D:\Jigsaw\Jigsaw\WWW\Gateway\IOR\master.ref
URL=none
Modules:==================================

B)
Server name = ntserver
File=none
URL= http://maine.npac.syr.edu:8001/Gateway/IOR/master.txt
Modules:==================================
runEdys     lms.idl   WebFlow.lms.runEdysImpl

C)
Server name = osprey4
File=none
URL= http://maine.npac.syr.edu:8001/Gateway/IOR/master.txt
Modules:==================================
runCasc2d  lms.idl WebFlow.lms.runCasc2dImpl
```

Fig .14: Configuration files for A) master, B) WindowsNT slave C) Unix slave WebFlow servers
Each module is described by its name, IDL file, and Java class name that implements it.

```
 ORB orb = ORB.init(args, new java.util.Properties());
String masterURL = args[0];
String ref=getIORFromURL(masterURL);
org.omg.CORBA.Object obj=orb.string_to_object(ref);
WebFlowContext master=WebFlowContextHelper.narrow(obj);
WebFlowContext slave;
 try {
      org.omg.CORBA.Object p1,p2;
     slave1=WebFlowContextHelper.narrow(master.getWFServer(“ntserver”));
     slave2=WebFlowContextHelper.narrow(master.getWFServer(“osprey4”));
      p1 = slave.addNewModule(“runEdys”);
      p2 = slave.addNewModule(“runCasc2d”);
      runCasc2 rc=runCasc2dHelper(p2);
     master.attachEvent(p1,”EdysDone”,p2,”runAgain”);
     master.attachEvent(p2,”Casc2dDone”,p1,”receiveData”);
     rc.run();
 } catch(Exception e) {};
```

Fig.15: Fragment of the client code

Adding new modules to the client is simply a matter of adding a few lines of code (addNewModule and narrow). If the methods of these modules are to be invoked directly from the client, add attaching events, if any.

In the next releases we plan to reuse the XML parser that we developed for the Quantum Simulations project, after which there will be no need to provide a client code in Java at all. Instead, the application will be defined by a static XML document, available from a Web server, and instantiated dynamically at runtime. More, the modification of the application will be possible just by editing the XML file without introducing any changes to the code.

Figs. 16 and 17 show the pseudocode of both modules, runCasc2d and runEdys, respectively. The simulation starts by invoking the method run() by the client.

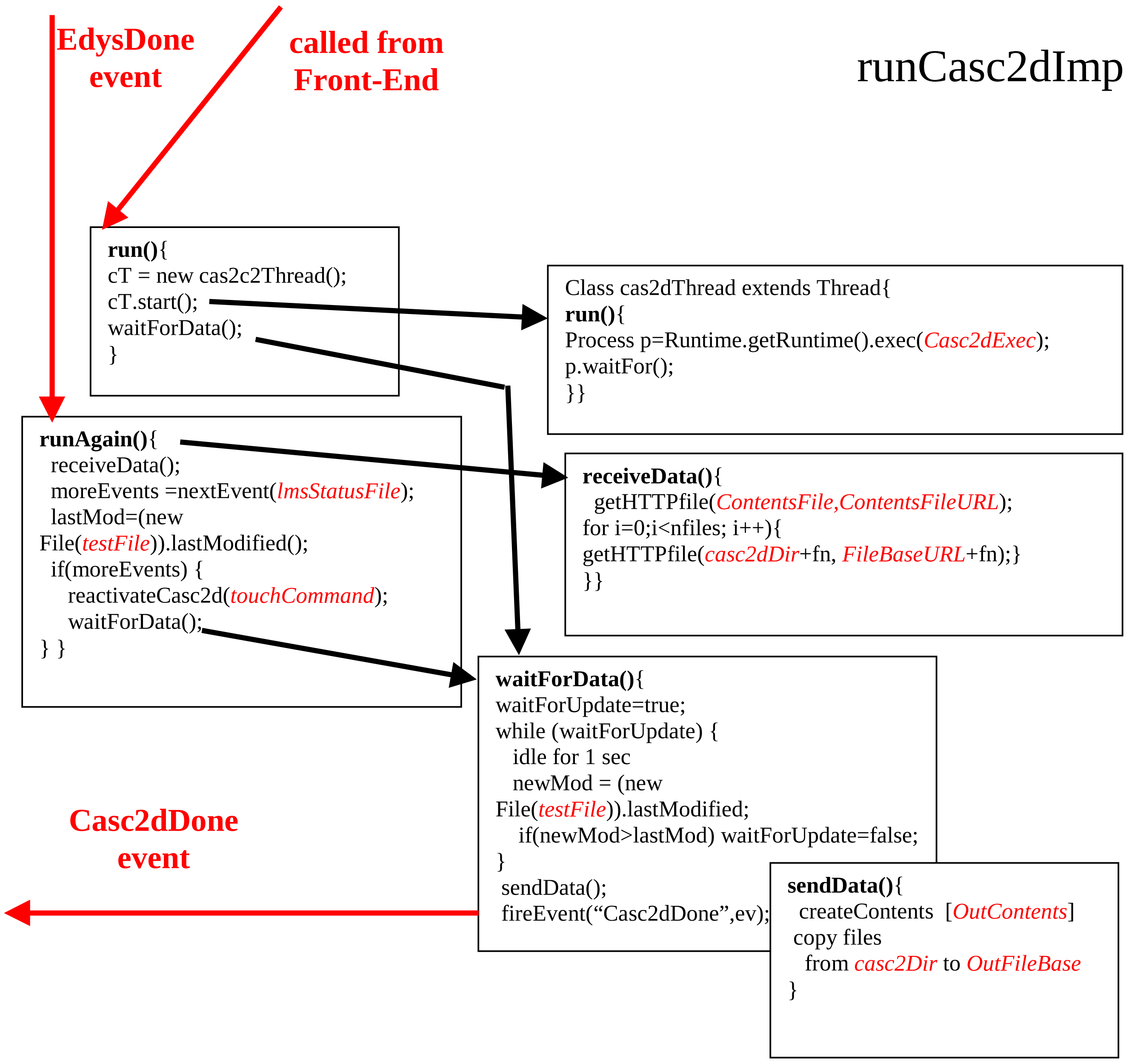
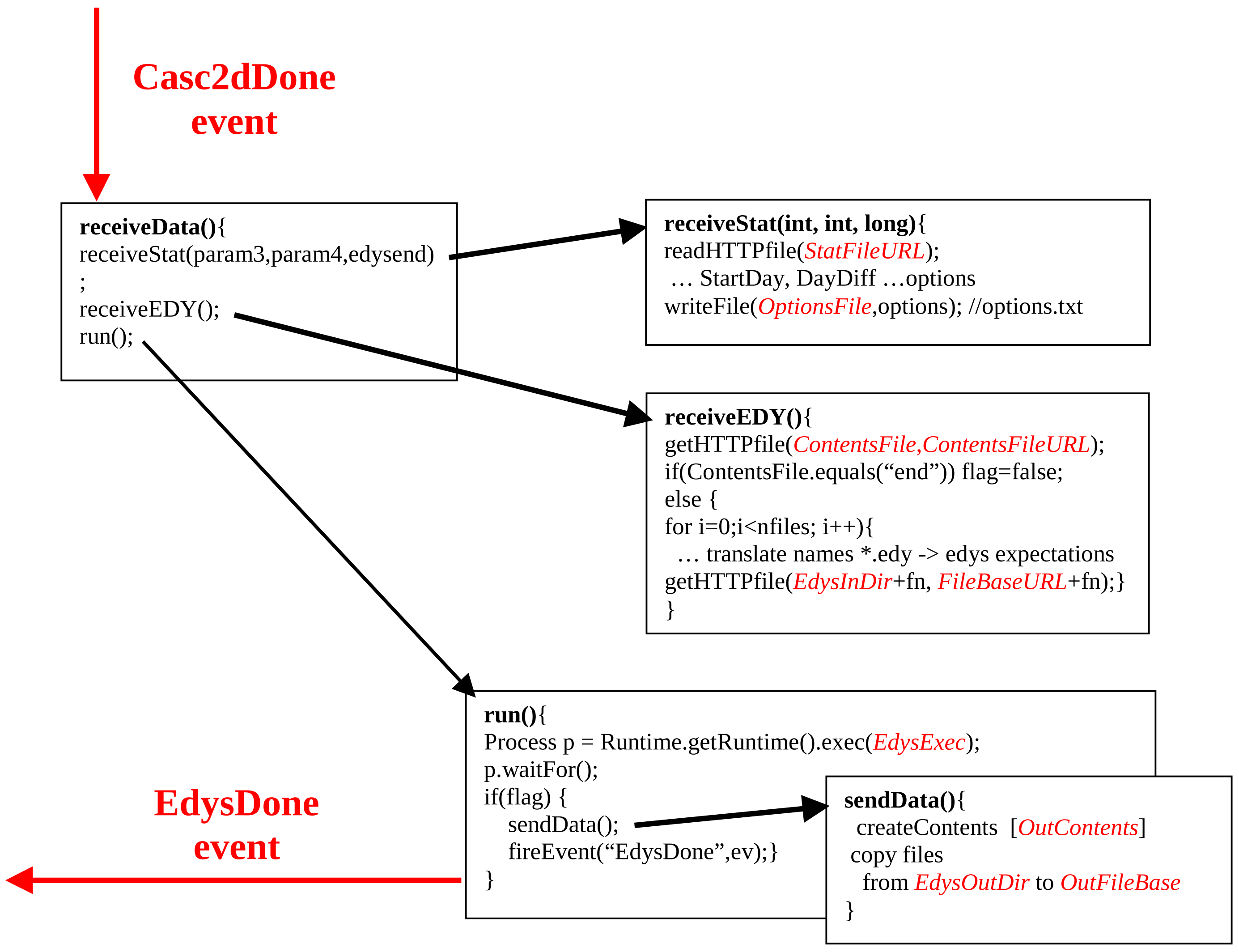

EdysDone event
called from Front-End
runCasc2dImp
run(){
cT = new cas2c2Thread();
cT.start();
waitForData();
}
Class cas2dThread extends Thread{
run(){
Process p=Runtime.getRuntime().exec(Casc2dExec);
p.waitFor();
}}
runAgain(){
receiveData();
moreEvents =nextEvent(lmsStatusFile);
lastMod=(new File(testFile)).lastModified();
if(moreEvents) {
reactivateCasc2d(touchCommand);
waitForData();
} }
receiveData(){
getHTTPfile(ContentsFile,ContentsFileURL);
for i=0;i<nfiles; i++){
getHTTPfile(casc2dDir+fn, FileBaseURL+fn);}
}}
waitForData(){
waitForUpdate=true;
while (waitForUpdate) {
idle for 1 sec
newMod = (new File(testFile)).lastModified;
if(newMod>lastMod) waitForUpdate=false;
}
sendData();
fireEvent("Casc2dDone",ev);
sendData(){
createContents [OutContents]
copy files
from casc2Dir to OutFileBase
}
Casc2dDone event

Fig. 16: Pseudocode of the runCasc2d module

Module runCasc2d is started from the front end by invoking its run() method, which creates a new Java thread that runs casc2d code in a separate process. It then invokes the waitForData() method. This method waits until casc2d generates the first data set for Edys, copies the files to a location seen by the Web server, and fires event "Casc2dDone" that invokes the run() method of run Edys.

When Edys fires the "EdysDone" event, the runAgain() method of runCasc2d is invoked. This method receives data from Edys (using the Java URLconnection class to access files from the Web server on the Edys host), and executes the UNIX touch command on a selected control file. By "touching" this file we change the 'last modified' property of that file, and this triggers casc2d to resume its operations. The controls then go to the waitForData() method, described above.



```
receiveData(){
receiveStat(param3,param4,edysend)
;
receiveEDY();
run();
```

```
receiveStat(int, int, long){
readHTTPfile(StatFileURL);
 … StartDay, DayDiff …options
writeFile(OptionsFile,options); //options.txt
```

```
receiveEDY(){
getHTTPfile(ContentsFile,ContentsFileURL);
if(ContentsFile.equals("end")) flag=false;
else {
for i=0;i<nfiles; i++){
  … translate names *.edy -> edys expectations
getHTTPfile(EdysInDir+fn, FileBaseURL+fn);}
}
```

```
run(){
Process p = Runtime.getRuntime().exec(EdysExec);
p.waitFor();
if(flag) {
   sendData();
   fireEvent("EdysDone",ev);}
}
```

```
sendData(){
 createContents  [OutContents]
copy files
  from EdysOutDir to OutFileBase
}
```

Fig. 17: Pseudocode of the runEdys module

The RunEdys module is triggered by the Casc2dDone event that invokes the receiveData() method. First, a file options.txt is generated. This files defines input parameters: start date of the simulation (StartDay), number of days to be run (DayDiff), parameter 3, which is a toggle to switch the Edys visualizations on and off, and parameter 4 which defines disturbances, if any. Then the data from the Web server of the casc2d host are downloaded. Finally, the Edys code is launched. After it is completed, its output is copied to a location seen by the Web server, and the "EdysDone" event is fired.

The sendData() method (which is identical in both modules) actually does not send any data. Instead, it copies data from the Edys (or Casc2d) working directory to a document directory of the Web server (Fig.18). This step could be avoided letting the codes write and read their input and output files directly from the WebServer. This would require slight modifications of these codes, and we had no access to their sources.

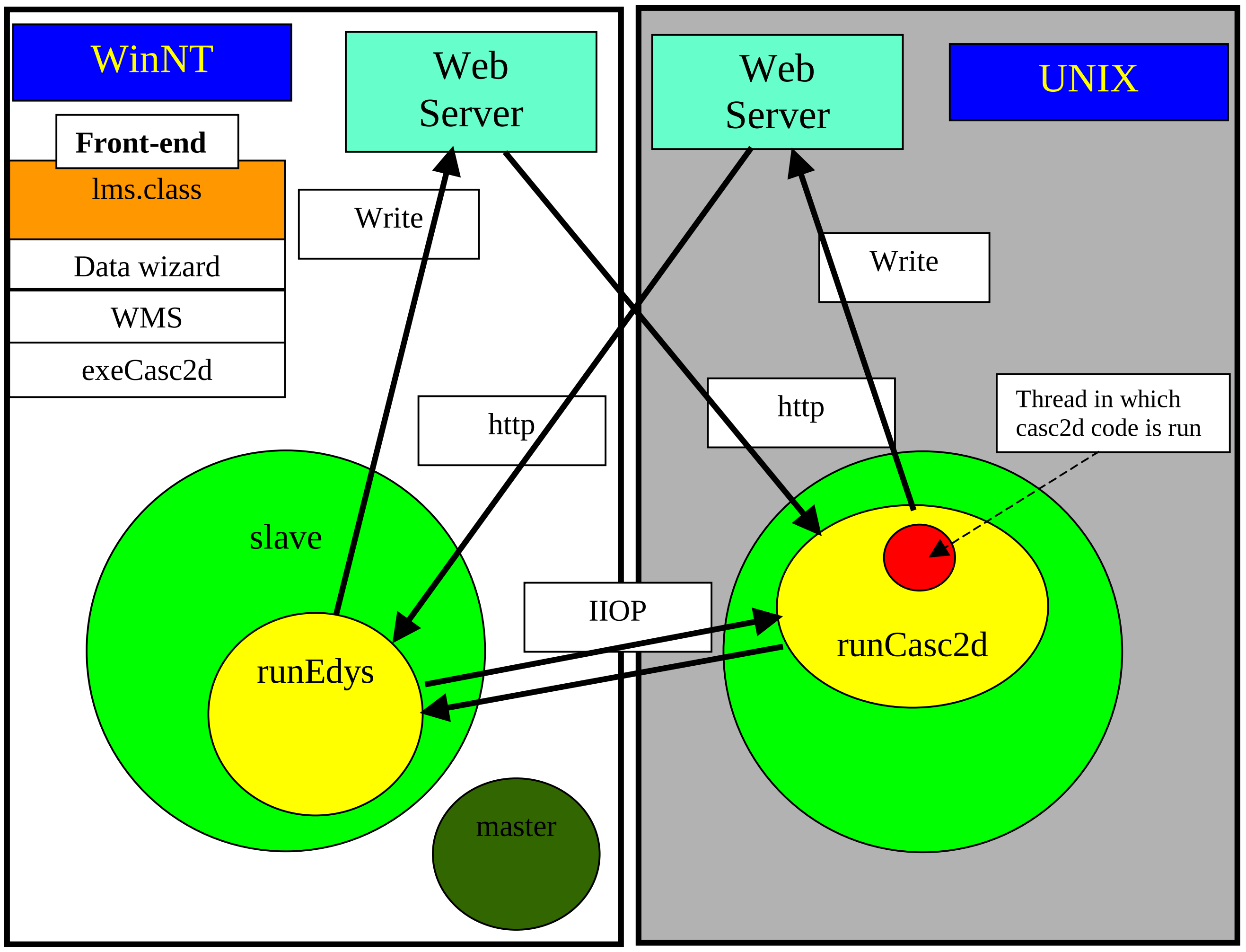


Fig. 18: Data exchange between runEdys (left-hand side) and runCasc2d (right-hand side) modules. Events are exchanged through IIOP, data sets are downloaded from Web servers.

4.4 LMS Back-End

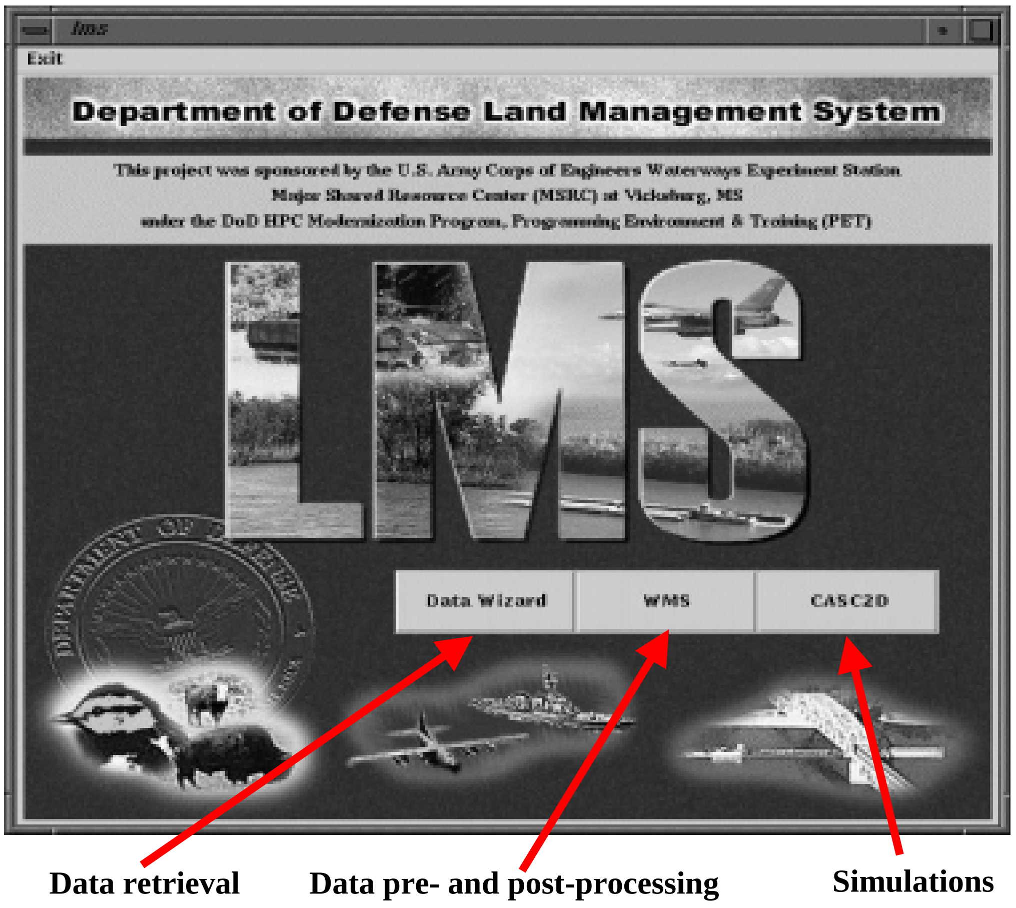
This pilot implementation of the Web-based LMS does not require any powerful computational resources, and consequently we provided only a limited support for back-end services. In particular, we simply used Java Runtime class to run the WebFlow modules on the same host on which the WebFlow server runs. As discussed is Section 3 above, we are prepared to provide a secure access to remote, high-performance resources when it is needed.

4.5 LMS Front End

For this project we developed a custom front end implemented as a Java application (as opposed to Web-accessible Java applets used in projects described in Section 3). There are several reasons for that. One is that we were explicitly asked to do so. Second, we did it for performance reasons. Finally, the front end is an extension to the WMS system that needs to be installed on the client side, anyway. Therefore, it does not really matter if the extensions to WMS are to be downloaded as an applet each time the LMS is run, or downloaded once and stored permanently on the client machine.

The WMS program (Watershed Modeling System) is a rich collection of tools for data pre- and post-processing. Furthermore, it allows us to run the simulation locally and visualize the results. WMS is available on many platforms, including Windows 95/98/NT and numerous flavors of Unix.

We made WMS the centerpiece of our front end. We enhanced it by providing the capability to import raw data sets directly from the Internet, submitting the simulations to remote hosts and, as described above, making different simulations interact with each other. Consequently, the LMS front end consists of three parts: data wizard, WMS, and job submission. Each part is accessible by pressing the corresponding button on the LMS main panel (Fig.19).



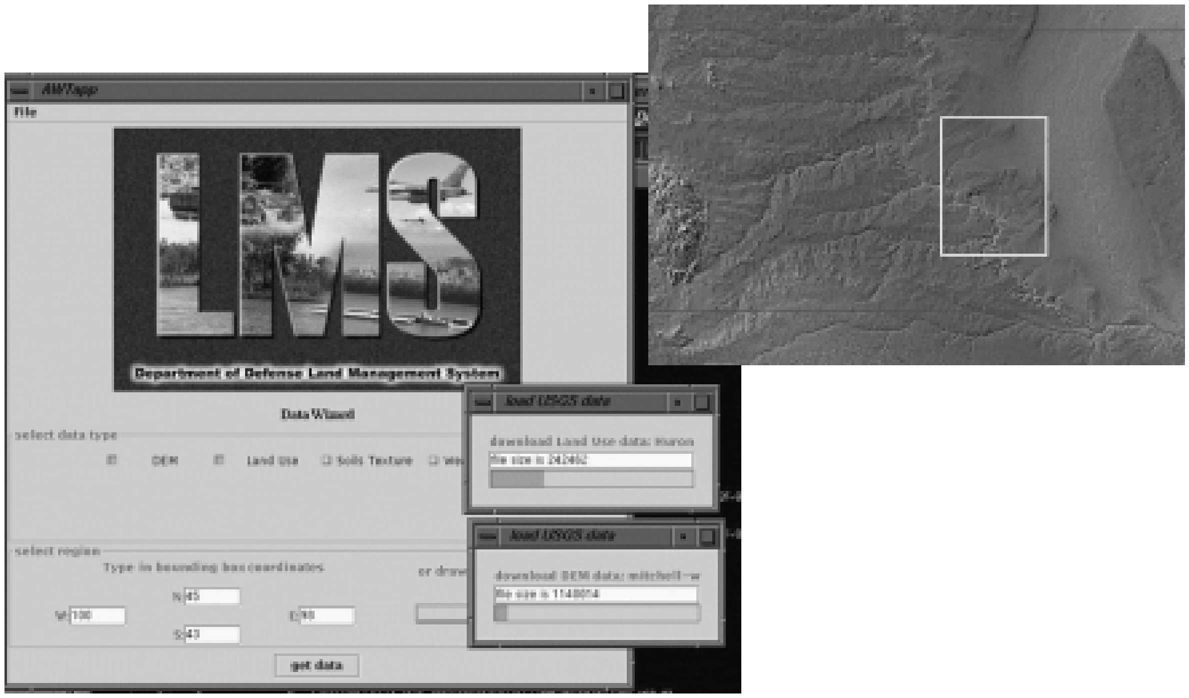
Department of Defense Land Management System

This project was sponsored by the U.S. Army Corps of Engineers Waterways Experiment Station
Major Shared Resource Center (MSRC) at Vicksburg, MS
under the DoD HPC Modernization Program, Programming Environment & Training (PET)

Fig. 19: LMS front end main panel

4.5.1 Data Wizard

The data wizard panel shown in fig. 20 allows us to select the data type to be retrieved (currently we support DEM and Land Use maps) and to define the region of interest. This can be done either by directly typing coordinates of the bounding box into the provided text fields, or by drawing boundaries of the region on a map. In the latter case, the position of the rectangle is automatically translated into coordinates.



N 45
W 100
E 90
S 43

download Land Use data: Huron
file size is 242462

download DEM data: mitchell-w
file size is 1148014

Fig. 20: LMS front end data wizard panel

Next, the coordinates are translated into names of the corresponding set of maps available from the USGS web site, and the selected maps are downloaded, uncompressed, and saved in a directory accessible by the WMS package.

4.5.2 WMS

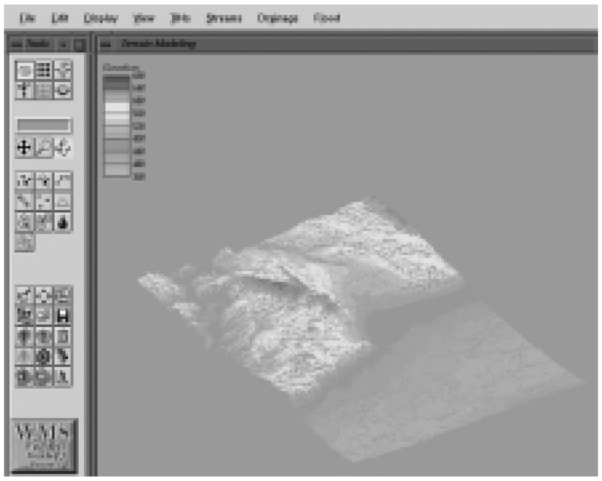
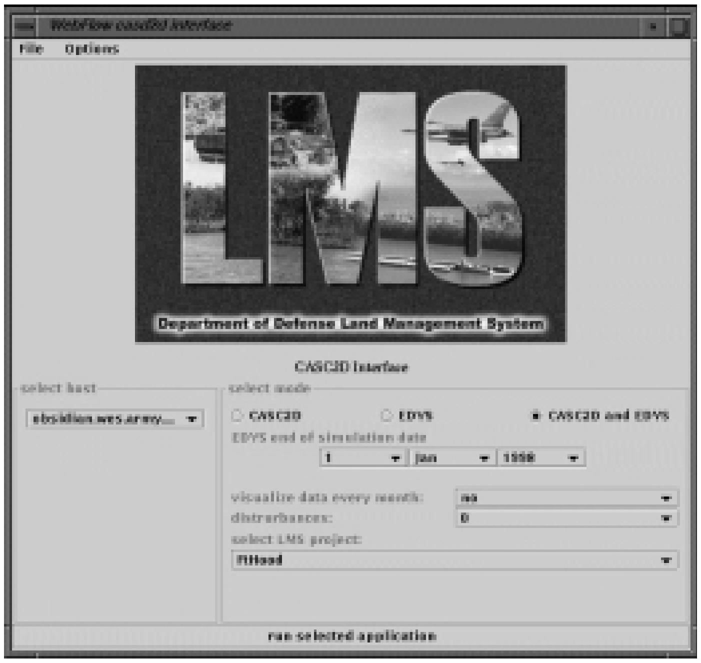


Fig. 20: A screendump of a WMS session: in the central window just downloaded DEM data are displayed. The raw date must be pre-processed now, including selecting a watershed region, smoothing, and format conversion.

The WMS button on the main panel starts the WMS program on the local host by the using Java Runtime class in a separate thread. The WMS controls are available during the entire LMS session.

4.5.3 Simulations

The functionality of this part can be deduced from Section 4.3 above. Figure 21 shows the front-end panel that is used to start the simulations.



select host: obsidian.wes.army...

select mode: ( ) CASC2D ( ) EDYS [✓ CASC2D and EDYS]

EDYS end of simulation date: 1 jan 1998

visualize data every month: no

disturbances: 0

select LMS project: FtHood

Fig. 21: LMS front-end simulation panel. The controls allow for selecting the simulation mode: Casc2d alone, Edys alone, and both simulations coupled, as described in Section 4.3. In the latter case, the user also provides the end date of the Edys run, Edys visualization toggle and disturbances, the directory that contains casc2d input data files and finally the user select the host on which the casc2d simulation is to be run.

This part of the front end acts as a client to the middle-tier services, and it communicates with the WebFlow servers using CORBA IIOP.

5 Future Work

This report describes a pilot implementation of the web-based LMS. We successfully demonstrated the feasibility of our approach. Nevertheless, the system is not mature enough to be used in a production mode, and it requires many improvements. The most important areas of the future development include:

- Secure access to resources
- Access to high-performance resources
- Resource detection
- Fault tolerance
- Simplified installation
- Extensibility; adding new application specific modules and reusable middle-tier services
- More general ways of implementing of different kinds of interactions between modules
- Customization of the front-end
- Increased functionality of the front

The obvious next steps are to take advantage of our experience by using WebFlow in other projects: Quantum Simulations and Gateway. In particular, the Gateway project is focused on providing a secure access to resources in the environment protected by Keberos5 in conjunction with SecurID technology and fault tolerance. It seems to us that solutions developed for the Gateway project will be directly applicable to LMS. Within the Quantum Simulation we have already developed methods of providing access to high-performance resources that can be used for this project.

Resource detection is an important feature that is missing in the current implementation. There are several commercial (such as Jini[8] by Sun Microsystems, Inc.) or academic (such as Ninja[9] developed at UC Berkeley) technologies available than can possibly be adopted for LMS.

Ease of installation and extensibility are other key features for turning this academic prototype into a product that can be disseminated to end users. Currently, the system is built from many freeware components coming from various vendors. This goes along with our HPcc philosophy, and we will adhere to this approach as we see it to be beneficial in the long term in terms of maintenance and future upgrades. The only area of improvement here is the simplification of the configuration procedure of these components to make them interoperate with each other. In particular, some parameters of the in-house developed components (WebFlow servers) are hardwired in the code, while they should be part of the configuration files.

Extensibility is far less trivial. We expect to add many new modules to LMS in order to support numerous applications and simulation codes to truly support the idea of "navigate and choose" resources (hardware and software) to solve the problem at hand. Currently, the middle-tier services and the front-end client are tightly coupled in the sense that adding a new middle-tier module (or just modifying its interface) requires modifications in the front end. This means that the module developer has to be capable of editing and compiling front-end Java classes. Admittedly, this is very bad. Fortunately, we can reuse the ideas we implemented in the Quantum Simulation project. As discussed in Section 3.1, the QS front end allows the composing an application from modules on the fly. The basic idea is to replace the Java client code by an XML document. The advantages of that are:

- Composing an application is reduced to generating a single XML document.
- Adding a new module does not require any modifications of the front-end code.
- The XML document that defines the application can serve a dual purpose: in the front end it can be converted to HTML – the application documentation, and in the middle tier it can be used to generate the WebFlow client code on the fly.

Further, it makes sense to convert the data wizard to a WebFlow module, too. This way we will decouple the front-end graphical interface from the functionality (such as downloading the data from the Internet) that logically belongs to the middle tier. This will allow us to implement more powerful methods to search and download data including on-the-fly format conversions, from Web sites and other data repositories, including data bases and mass storage devices. Also, when encapsulated as a WebFlow module, the data wizard will be capable of sending high-volume data directly from their source to the high-performance computer that will run the simulation.

Finally, we need to generalize the way in which the modules interact with each other. Our preference is to adhere to the object-oriented approach (even when the application are written in a procedural language such as Fortran). As an example, we suggest a different encapsulation of casc2d code as a CORBA object. Instead of a loop over rainfall events, it should implement a method to process a single event. Then we could add another object – a time manager – that would dispatch the work between Edys and casc2d as needed (driven by the rainfall events). Moreover, yet another object could resolve differences in temporal and spatial resolutions of maps used by these codes.

6 Summary

To summarize, we used WebFlow - a scalable, high-level, commodity standards-based HPDC system that integrates:

- High-level front ends for visual programming, steering, and data visualization built on top of the Web and OO commodity standards (Tier 1).
- Distributed object-based, scalable, and reusable Web server and Object broker Middleware (Tier 2)

- High-performance back end implemented using the metacomputing toolkit of GLOBUS [2] (Tier 3)

We use this system to implement the Landscape Management System extending its original WMS interface. Our extensions allow downloading data directly from the Internet and launching coupled simulations on remote hosts. The pilot implementation successfully proved the concept of the Web-based interface. We also outlined the next steps toward turning this academic prototype into a powerful "navigate and choose" tool that can boost productivity of the end users, and what is most important, the tool that provides access to necessary software and hardware anytime, anywhere, given the connection to the Internet.

Acknowledgments

This work has been funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET). Supported by Contract Number: DAHC 94-96-C-0002 with Nichols Research Corp.

7 Bibliography

1. G. Fox and W. Furmanski, "HPcc as High Performance Comodity Computing", chapter for the "Building National Grid" book by I. Foster and C. Kesselman, http://www.npac.syr.edu/uses/gcf/HPcc/HPcc.html
2. Contact person: Lubos Mitas, NSCA, http://www.ncsa.uiuc.edu/Apps/CMP/cmp-homepage.html
3. Contact person: Ken Flurchick, Ohio Supercomputing Center, http://www.osc.edu/~kenf/Gateway
4. DATORR: Desktop Access to Remote Resources, home page http://www-fp.mcs.anl.gov/~gregor/datorr/
5. Globus Metacomputing Toolkit, home page http://www.globus.org
6. casc2d has been written by Fred Ogden, University of Connecticut, ogden@eng2.uconn.edu
7. Edys has been written by Michael Childress, Shepherd Miller ,Inc., mchildress@shepmill.com
8. Sun Microsystems, Inc., JINI Connection Technology, home page http://www.sun.com/jini/index.html
9. UC Bekeley, NinjaProject, home page http://ninja.cs.berkeley.edu/