Integrated Three-Tier Architecture for
High-Performance Commodity
Metacomputing

by
EROL AKARSU

B.S., Ege University, Turkey, 1991
M.S., Syracuse University, 1996

DISSERTATION

Submitted in partial fulfillment of requirements for the degree
of Doctor of Philosophy in Computer Science
in the Graduate School of Syracuse University

December, 1999

Approved

Professor Geoffrey C. Fox

Date
Abstract

Programming tools that are simultaneously sustainable, highly functional, robust and easy to use have been hard to come by in the HPCC arena. Thus we have developed a new strategy, HPcc: High Performance Commodity Computing, which 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. We add high performance to commodity systems using a multi-tiered architecture with the Globus meta-computing toolkit as the backend of a middle tier of commodity web and object servers.

More specifically, we design and implement a three-tier system. Web browser-based graphical user interface that assists the researcher in the selection of suitable applications, the generation of input data sets, specification of resources, and the post-processing of computational results, comprises tier 1. The distributed, object-oriented middle tier maps the user-task specification onto back-end resources, which form the third tier. In this way we hide the underlying complexities of a heterogeneous
computational environment, and replace it with a graphical interface through which a user can understand, define, and analyze scientific problems.
Table of Contents

LIST OF TABLES .......................................................................................................... IX

LIST OF FIGURES ........................................................................................................ XI

CHAPTER 1 – INTRODUCTION .................................................................................. 1

1.1 RELATED WORK ................................................................................................. 6

CHAPTER 2 – HIGH-PERFORMANCE COMMODITY COMPUTING ......................... 14

2.1 INTRODUCTION .................................................................................................. 14

2.2 COMMODITY TECHNOLOGIES AND THEIR USE IN MULTI-TIER SYSTEMS .... 15

2.3 THREE TIER HIGH PERFORMANCE COMMODITY COMPUTING ................. 17

CHAPTER 3 – DARP SYSTEM (DATA ANALYSIS AND RAPID PROTOTYPING) ........................................ 24

3.1 INTRODUCTION ................................................................................................ 24

3.3 DARP SERVER: INTERACTIVE CONTROL OVER AN APPLICATION ............ 26

3.4 INSTRUMENTATION OF THE CODE ................................................................. 29

3.5 DARP FRONT END ............................................................................................. 30

3.6 INTEGRATED ENVIRONMENT FOR HPF COMPILER AND INTERPRETER .......... 36

3.7 RUNTIME VISUALIZATIONS .............................................................................. 37

3.8 ADDING A DIFFERENT VISUALIZATION ENGINE TO DARP SYSTEM ........... 39

3.9 SUMMARY ......................................................................................................... 40

4.1 INTRODUCTION ................................................................................................ 42
4.3 THREE-TIERED ARCHITECTURE OF THE WebFlow ................................. 47
4.3.1 The Front End .................................................................................... 47
4.3.2 The Middle-Tier ............................................................................... 50
4.3.3 Session Manager .............................................................................. 51
4.3.4 Module Manager ............................................................................... 51
4.3.5 Connection Manager ....................................................................... 52
4.4 THE BACK END .................................................................................. 52
4.5.1 Logging into the WebFlow system ................................................... 56
4.5.2 Creating A New Module .................................................................. 56
4.5.3 Making a connection between modules ........................................... 58
4.5.4. Run/stop/destroy modules ............................................................... 61
4.6 LIMITATIONS OF ORIGINAL WEBFLOW AND OTHER ALTERNATIVE SOLUTIONS 62
5.1 IMPACT OF GATEWAY ON SEAMLESS ACCESS TO HPCC RESOURCES ........................................................................................................................................... 67
5.2 GATEWAY MIDDLE TIER .................................................................... 70
5.2.1 Motivation ......................................................................................... 70
5.2.2 GateWay Middle Tier ....................................................................... 72
5.2.3 Lifecycle Service ............................................................................... 74
5.2.4 Proxy Objects .................................................................................. 75
5.2.5 Interactions of user modules ............................................................ 75
5.3 GATEWAY BACK-END ........................................................................ 79
5.4 THE FRONT END ................................................................................ 83
5.5 COMPARISON OF GATEWAY WITH EJB ........................................ 84
8.1 A POSSIBLE FRAMEWORK OF A CLIENT-SIDE COLLABORATIVE ENVIRONMENT .......................................................... 152

8.2 HOW DO USER MODULES WRITTEN IN COM INTERACT WITH THE ONES IN CORBA? .......................................................... 153

8.3 GATEWAY CAN ACT AS FIREWALL TO REMOTE OBJECTS ............................................. 153

8.4 COMPARISONS WITH OTHER COMPONENT MODELS ............................................. 154

REFERENCES ................................................................................................................................. 159

VITAE ............................................................................................................................................... 170

GLOSSARY ......................................................................................................................................... 172
List of Tables

TABLE 1. DATA ACCESS COMMANDS OF HPF SERVER .......................................................... 33
TABLE 2. PROTOTYPING COMMANDS OF HPF SERVER .................................................... 34
TABLE 3. CONTROL COMMANDS OF HPF SERVER .......................................................... 35
TABLE 6.1. BEANCONTEXTCHILD INTERFACE ................................................................. 87
TABLE 6.2. ITERATOR AND COLLECTION INTERFACES .................................................. 90
TABLE 6.3. GATEWAYCONTEXT INTERFACE EXTENDING BEANCONTEXT AND DARP
INTERFACES ...................................................................................................................... 92
TABLE 6.4. A SIMPLE REPRESENTATIVE DARP METHOD ................................................. 102
TABLE 6.5. HOW THE DARP SERVER GETS THE NEXT COMMAND FROM THE MIDDLE TIER
........................................................................................................................................ 103
TABLE 6.6. HOW THE DARP SERVER PUTS THE RESULT OF A PREVIOUS CLIENT COMMAND
INTO THE MIDDLE TIER ..................................................................................................... 104
TABLE 6.7. DARP INTERFACE ......................................................................................... 106
TABLE 6.8. BASE ATTRIBUTE AND THREE DIFFERENT ATTRIBUTES EXTENDING FROM THE
BASE ..................................................................................................................................... 109
TABLE 6.9. AN IDL DEFINITION FILE OF AN EXAMPLE USER MODULE: ....................... 111
TABLE 6.10. XML DESCRIPTION OF USER IDL FILE IN TABLE 6.9 ................................. 112
TABLE 6.11. THE DEFINITION OF TWO METHODS FOR THE HELPER CLASS OF USER
ATTRIBUTES ................................................................................................................... 115
TABLE 6.12. THESE GATEWAY API CALLS CONSTRUCT THE CONFIGURATION IN FIGURE 5.6
............................................................................................................................................ 116
TABLE 6.13. THIS XML DOCUMENT IS SAVED IF THE USER ISSUES A \texttt{SAVESTATEINXML} REQUEST TO THE MASTER SERVER. IN ADDITION, THE USER CAN CONSTRUCT THE CONFIGURATION IN FIGURE 5.6 WITH THIS DOCUMENT

TABLE 6.17. SOLVING MULTIPLE DECLARATIONS OF MODULES WITH THE XML \texttt{ENTITY} ELEMENT

TABLE 7.1. ABSTRACT JOB SPECIFICATION IN XML DOCUMENT FOR CONFIGURATION IN FIGURE 7.3
List of Figures

FIGURE 2.1. INDUSTRY THREE-TIERED VIEW OF ENTERPRISE COMPUTING ....................... 18

FIGURE 2.2. TODAY'S HETEROGENEOUS INTEROPERATING HYBRID SERVER ARCHITECTURE
................................................................................................................................................. 20

FIGURE 3.1. DARP IMPLEMENTATION WITHIN HPcc FRAMEWORK ............................... 26

FIGURE 3.2. THE ARCHITECTURE OF DARP ........................................................................ 27

FIGURE 3.3. MIDDLE-TIER DARP MANAGER CONTROLS HPF BACK END AND
COMMUNICATES WITH OTHER SERVERS .................................................................................. 29

FIGURE 3.4. HPF INTERPRETER......................................................................................... 36

FIGURE 3.5. A SCREEN DUMP OF A DARP SESSION ....................................................... 38

FIGURE 3.6. ADDING A NEW VISUALIZATION SERVER TO THE DARP SYSTEM ............. 40

FIGURE 4.1. TOP-LEVEL VIEW OF THE WebFlow ENVIRONMENT .................................... 43

FIGURE 4.2 WebFlow FRONT END APPLET ...................................................................... 48

FIGURE 4.3. BRIDGE BETWEEN WebFlow AND GLOBUS RESOURCES (THE GRID) ........ 54

FIGURE 4.4. STARTING A USER SESSION IN THE WebFlow SYSTEM ............................. 56

FIGURE 4.5. CREATING A NEW MODULE ......................................................................... 58

4.7. RUN/STOP/DESTROY MODULES ............................................................................... 61

FIGURE 5.1. GATEWAY ARCHITECTURE............................................................................ 70

FIGURE 5.2. A HYPOTHETICAL DISTRIBUTED APPLET. EACH PANEL (A CONTAINER) OF THIS
APPLET IS PLACED ON A DIFFERENT HOST. ..................................................................... 71

FIGURE 5.3. A DISTRIBUTED GATEWAY APPLICATION.................................................... 72

FIGURE 5.4. A SIMPLIFIED REPRESENTATION OF THE GATEWAY MIDDLE TIER ............ 73
FIGURE 5.5. NAMING OF GATEWAY CONTEXTS AND USER MODULES ........................................... 74

FIGURE 5.6. GATEWAY EVENT MODEL .................................................................................... 78

FIGURE 5.7. GATEWAY SERVICES AT BACK END...................................................................... 81

FIGURE 5.8. BASICS OF AN EJB ENVIRONMENT .................................................................... 86

FIGURE 6.1. THE DETAILS OF HOW \texttt{ADDNEWMODULE} IS EXECUTED IN THE GATEWAY SYSTEM ....................................................................................................................... 94

FIGURE 6.2. INTERACTION OF OBJECTS DURING THE REMOVAL OF MODULE M2............. 95

FIGURE 6.3. MAKING AN ASSOCIATION BETWEEN MODULES M1 AND M2 .................... 97

FIGURE 6.4. MAKING A DISSOCIATION BETWEEN MODULES M1 AND M2 ...................... 98

FIGURE 6.5. HOW THE DARP SERVER, MIDDLE TIER, AND CLIENT INTERACT WITH EACH OTHER TO FULLY CONTROL THE DISTRIBUTED EXECUTION ........................................... 105

FIGURE 6.6. GATEWAY PERSISTENCY MODEL ......................................................................... 108

FIGURE 6.7. RECOVERING PROXY MODULE IN ROOT GATEWAY SERVER ......................... 120

FIGURE 6.8. RECOVERING A REMOTE USER MODULE ........................................................ 122

FIGURE 6.9. GATEWAY SECURITY ARCHITECTURE .................................................................. 123

FIGURE 6.10. GATEWAY SECURITY MODEL ............................................................................. 124

FIGURE 7.1. LOGICAL STRUCTURE OF THE LMS SIMULATIONS IMPLEMENTED BY THIS PROJECT ........................................................................................................................................... 131

FIGURE 7.2. EXCHANGE OF DATA BETWEEN \texttt{CASC2D} (LEFT-HAND SIDE) AND \texttt{EDYS} (RIGHT-HAND SIDE). IT IS IMPORTANT TO NOTE THAT \texttt{CASC2D} IS RUN ONLY ONCE. IT PAUSES WHILE WAITING FOR THE NEW DATA AND QUITS ONLY AFTER ALL EVENTS ARE PROCESSED. IN CONTRAST, \texttt{EDYS} IS LAUNCHED EACH TIME THE DATA ARE NEEDED.. 133

FIGURE 7.3. WEBFLOW IMPLEMENTATION OF LMS ............................................................. 134
FIGURE 7.4. WebFlow API calls to construct the configuration in the Figure 7.3

.................................................................................................................................. 136

FIGURE 7.5. LMS front-end main panel........................................................................... 139

FIGURE 7.6. A screen-dump of a WMS session: in the central window just downloaded DEM data are displayed. The raw data must be pre-processed now, including selecting a watershed region, smoothing, and format conversion. .......................................................... 142

FIGURE 7.7. LMS front-end simulation panel. ................................................................. 143

FIGURE 7.8. Logical structure of the quantum simulation application ................. 145

FIGURE 7.9. WebFlow implementation of the quantum simulations problem... 146

FIGURE 7.10. Example WebFlow session of QS......................................................... 147
Acknowledgements

I would like to express my appreciation to my advisor, Professor Geoffrey C. Fox, for his guidance throughout my research and for his wise and acute observations on how to improve my work.

I gratefully acknowledge Dr. Tomasz Haupt, Dr. Wojtek Furmanski, Dr. David Bernholdt, Dr. Edward Lipson, and Dr. Ehat Ercanli for serving on my defense committee.

I appreciate Dr. Tomasz Haupt for giving shape to my thesis and for preparing many demos on DARP, WebFlow and Gateway. I cannot forget the long discussions we had in order to find the best solutions for several problems. I thank him for applying DARP and Gateway to Quantum Simulation and the Landscape Modeling System.

I would like to thank Dr. Furmanski for preparing several demos and papers. I worked with his group to integrate DARP with WebFlow, and I was greatly motivated by the many ideas he gave me.

Also, I want to thank my wife for her enormous patience during the course of my work toward a Ph.D. degree. I could not have finished it without her help.

I am especially grateful to Mrs. Elaine Weinman for her help in English and for her patience in proofreading endless revisions of this manuscript.

I would like to express my appreciation to many friends who contributed to my work with their encouragement. Among them, Dr. Kivanc Dincer, Dr. Haluk Topcuoglu, Mr. Erdogan F. Sevilgen, Mr. Mehmet Sen, Mr. Ozgur Balsoy, Mrs. Zeynep Ozdemir,
and Mr. Hasan Timucin Ozdemir deserve special thanks. Their moral support and
suggestions were of great value.
CHAPTER 1 – INTRODUCTION

Developing large applications is a complex process and the assistance of adequate programming tools is always welcome. Not surprisingly, there are numerous commercially available tools for this purpose. Visual debuggers, profilers, data analysis, and visualization packages are integral parts of the workstation environments of scientists and engineers. The situation is different for high-performance, parallel, or distributed architectures. Performance tuning, debugging, and data analysis are more difficult, and yet tools that are simultaneously sustainable, highly functional, robust, and easy to use for these purposes are not widely available to the HPCC arena. This is partially due to the difficulty of developing sophisticated, customized systems for what is a relatively small part of the worldwide computing enterprise. If we consider the entire computing market, the user base of different types of computing can be illustrated as a pyramid with a narrow top and much wider base. HPCC technologies have been developed at the top of the computing pyramid, mostly by federally funded organizations where moving down into a broader user base was encouraged [Fox96]. However, the results have never been completely satisfactory due to the lack of common open interfaces among personal computers with Windows-based user interfaces and Unix workstations.

Even if we had HPCC programming tools, we would still have problems with some types of applications, specifically metacomputing applications (meta-applications). There are many definitions of “metacomputing” whose origin is believed to have been the CASA project. Larry Smarr, the NCSA director, defined it as “the use of powerful computing resources transparently available to the user via a networked environment,” and he is generally credited with popularizing the term. We define
metacomputing as “a means of integrating legacy codes into distributed environments and providing the user with seamless access to remote resources.” We consider the metacomputing environment to be a “computational grid” [HPccGridBook] that gives dependable, consistent, and seamless access to computational or remote resources. A computational grid is a dynamic environment that any type of computational resource can join or leave. “Meta-application” is usually an interdisciplinary application that works on a computational grid and needs heterogeneous machines, scientific instruments, archival storage, visualization, and multiple users working together.

Real-world systems bring many computational complexities to software developers to model them. A metacomputing application needs not only a substantial number of cycles, but also the use of heterogeneous models and various hardware and software resources with which to implement the various parts of the solution. However, there is little software support available for incorporating such heterogeneous resources into a virtual program. The resulting model is one in which an application consists of distributed data and individual programs (scientific algorithms or database and visualization servers), where files are used to transfer data from one component to another. There is also a configuration problem where the user has to manually start components separately on various machines and create connections (data and control) between components. There is little software in the HPCC community for configuring such a collection of components into a single virtual application (meta-application). The HPCC and metacomputing research communities have produced remarkable software for various types of parallel machines as well as software infrastructures for building meta-applications [FK96globus, LG96legion]. Unfortunately, there is no high-
level visual development environment in which we can use both different parallel machines and commodity machines (WIN/NT PCc) and clusters (PC or workstation clusters) as one virtual machine to solve meta-applications.

For example, various DoD and DoE-funded projects have produced multiple ecosystem management/modeling tools such as Terrain Modeling and Soil Erosion Simulation, Ecological Modeling for Military Land-Use Decision Support, Watershed Modeling System, etc., which are really legacy codes written in different languages and possibly working on different machines. Managing land and water resources is a challenging task that needs an integrated modeling/decision support environment capable of simulating atmospheric-surface/water-groundwater connectivity, cleaning and rehabilitating contaminated sites, managing coastal zone, watershed, and riverine resources, etc. Even though these modeling tools help land and water resource managers, they currently are disconnected codes that must be united into an integrated framework to achieve their highest productivity. We emphasize that this integrated framework is the computational grid we must establish. Therefore, a new initiative, the Land Management System (LMS), has begun to design, develop, support, and apply an integrated capability for the modeling and decision support technologies needed for applications relevant to the management of DoD land, water, and airspace. The most important decision to be made here is whether or not we are going to provide an environment that brings relevant science and technology to DoD land managers in a complete and responsive manner. We have to use existing diverse investments in science and technology (commodity software) to design an evolutionary and scalable computational grid environment that gives a uniform point of access to both scientists
and managers so that we can maximize synergism between them. Our grid environment allows us to develop protocols for model-to-model and model-to-data connectivity so that new technology investments in modeling and simulation, basic science, and information technology will seamlessly integrate with new data collection, assimilation, and management activities.

This dissertation proposes and develops **Gateway**, an integrated environment for High Performance Commodity Metacomputing (HPCM). We consider the contributions of this work to be the following:

- **Gateway** provides an integrating layer such that user-defined front ends and high-performance computing or commodity back-end elements (database, visualization servers and instrumentation server, and directory server, etc.) can be plugged dynamically into **Gateway**.

- **Gateway** makes it possible to configure and create the hierarchical meta-application across heterogeneous machines through the use of Gateway API or by preparing and introducing one’s own abstract job specification in XML to the Gateway. After starting the meta-application, it gives the user full control in running the application and saves and restores its distributed state with industry-standard XML.

- **Gateway** allows source-level debugging and monitoring of each component of the application if a malfunctioning component is found. There are currently a few tools that aid in debugging the distributed application.
• **Gateway** permits legal program statements to be interpreted in the same language as the component and thus facilitates the dynamic prototyping of complex software systems.

• The user can easily dynamically attach his metacomputing, visualization, and database or batch job submitter services to Gateway system, and individual components in the system can use it immediately.

• One unique feature of the Gateway system combines high-performance services with commodity computing tools without sacrificing high performance.

• Security and transaction protocol can be easily added into the Gateway system simply by putting these policies into proxy objects for each individual remote component. We already have the facility for generating necessary JDBC calls for component attributes and automatically storing and restoring the component state from/to the database.

• The Gateway system provides a server-side, object-collaboration framework and also permits TANGO [Tango97SIAM], a client-side interactive collaboration system, to be plugged into Gateway.

We followed our new strategy of High-Performance Commodity Computing (HPcc) [HPCCEuroPar98Gf] to construct the **Gateway** framework. HPcc 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 delivers tools naturally with the desired properties, with the one (albeit
critical) exception that high performance is not guaranteed. The user can build his metacomputing environment as a multi-tier architecture that has a Web-based visual front-end and a middle-tier consisting of multiple Gateway servers and back-end modules. We add high performance to commodity systems with a Globus metacomputing toolkit as the backend. This three-tiered system architecture is very similar to Enterprise JavaBeans (EJB) architecture.

We performed two experiments to create Gateway. First, we built the DARP (Data Analysis and Rapid Prototyping system) system, an integrated program-development environment. DARP helps the user build a fine-grained meta-application and gives full control on remote applications. Second, we participated in developing and extending WebFlow, the primitive version of Gateway with other NPAC project members. Finally, we constructed Gateway based on these experiments.

We performed three applications of Gateway: Quantum Simulation (QS) and Land Management System (LMS) described above, which are coarse-grained meta-applications. The last one is “seamless access” infrastructure to HPCC resources, which has a three-tiered architecture whose middle tier is Gateway.

The middle-tier was based on the commercial distributed-objects technology, CORBA, and works on the CORBA product of any ORB vendor. We especially benefited from JavaBeans, Enterprise JavaBeans (EJB), the CORBA component and multiple interfaces specifications, and adapter and delegation design pattern.

1.1 Related Work
**UNICORE** provides seamless and secure access to distributed computing resources using the World Wide Web. It has a three-tier architecture similar to that of Gateway. In tier one of the applet, Job Preparation Agent (APA), the user prepares an Abstract Job Object (AJO) from some group of Java abstract classes and sends it to the middle tier, the UNICORE site, (which operates a gateway running the https server). The middle tier includes a security servlet where the user’s certificate is authenticated and mapped to the local Unix userid. The Network Job Supervisor (NJS) interprets AJO, possibly using Java reflection mechanisms, and creates a job for the third tier, the local Resource Management System (RMS) or forwards sub-jobs to other UNICORE sites. We provide the facility both to prepare more flexible XML documents to construct abstract jobs and to interactively construct user jobs through the Gateway API.

**WebSubmit** [WebSubmit], a product of the National Institute for Standards Technology (NIST), is similar to UNICORE in that it is a mechanism for obtaining a seamless interface to computational platforms using the web. Instead of being Java-based, it uses more traditional CGI scripts, mainly based on Tcl. Tasks to be submitted are generated in application- and platform-specific format at the time of creation. Job consignment and result delivery are handled as a single (possibly long) transaction. Currently, the implementation is specific to the LoadLeveler batch subsystem.

**Legion** [LegionACM97] is an object-based project for developing software in support of a “world wide virtual computer.” The project envisions a system in which a user sits at a Legion workstation and has the illusion of a single, very powerful computer. When the user invokes an application on a data set, it is Legion's
responsibility to schedule application components transparently to processors, manage data transfer and coercion, and provide the necessary communication and synchronization. System boundaries, data location, and faults are to be made invisible. Since the user should not be aware of specific physical resources comprising the virtual metacomputer, this requires Legion itself to locate, schedule, and synchronize the necessary resources at run time. Legion proposes to establish its own model for security, allowing each user to establish security restrictions on an object basis. It would appear that Legion is a very ambitious project at the forefront of computer science.

**Globus** [GlobusIntlJ97] is a large U.S.-based project that is developing the fundamental technology needed to build computational grids-execution environments that enable an application to integrate geographically distributed instruments, displays, and computational and information resources. As with the Legion project, the primary focus of Globus is to combine distributed resources into a virtual metacomputer in support of grand challenge (possibly parallel) problems. This implies a need to synchronize access to resources at run time and to support, in the interest of bandwidth availability, message passing among components of the application over a network connection under the control of Globus components.

Our main intent in this thesis is to build a user-friendly, Web-based metacomputing environment. We used the Globus toolkit whenever we needed high performance in the back-end (third tier). But as front-ends for different problem domains can be plugged into the Gateway system, our Gateway middle tier is generic enough that any low-level (Globus), object-based (Legion), or some other type of
metacomputing toolkit can be interchanged with each other. Because we follow a commodity-computing model (HPcc), we strongly believe in using whatever software is available for solving any problem.

One of the distributed systems most similar to Gateway is Sun Microsystems’ JINI [JINIWeb] where main goal is to create a new distributed computing architecture. It is an object-oriented framework that embodies a model of how devices and software inter-operate as well as how distributed systems function. The infrastructure consists of two main components: “Discovery and Join” and “Lookup”. Discovery and Join is a two-phased mechanism whereby a device or application identifies itself to the network. First, the entity broadcasts a discovery package that contains sufficient information to enable the network to start a dialogue with the entity that has just joined. Second, once acknowledged, the entity can join by sending a message containing details about its own characteristics. Lookup is a component that stores information about Jini-registered devices and applications. Clients using Jini use Lookup to find the services that they wish to access. The distributed programming model used by Jini promotes three technologies: leasing, distributed transactions, and distributed events. Leasing is when an object negotiates the usage of a service for a period of time. Communication within Jini is based on Remote Method Invocation (RMI). The Jini-distributed programming model is based on the JavaSpaces model, which is itself based on Linda.

Jini supports the distributed model based on the flow of objects among processes, as opposed to method calls. Gateway adopts the publish-subscriber model for services, but Jini uses the template-matching model to find an entry, E, in JavaSpace through a lookup operation with an appropriate template, T. The template T can match
an entry E in the space only if the field values of T are matched exactly by the value in
the same field of E. Gateway identifies services with just a simple name that the user
can choose. Whenever a service is attached to or detached from the Gateway system, all
of the registered objects in the Gateway system are notified.

The event model of Jini is similar to Gateway’s event model. Whenever a Jini
client writes an entry matching a particular template written into JavaSpace, registered
listeners are automatically notified. Jini’s event model supports the interposing of a
third-party object between the event producer and the consumer. This intermediary
object can behave as mailbox, store-and-forward, or notification-filter agent. Our
Gateway supports these types of agents in exactly the same way. All events fired by
any object in the Gateway system are captured by the parent context of this object and
forwarded (“push” event type) or kept (“pull” event type) until the event consumer
explicitly picks it up.

Another emerging technology is the server component model, Enterprise
JavaBeans EJB [EJBWeb]. The Gateway system has some similarities to EJB.
Gateway is a generic middle tier consisting of a tree of two types of components: user
modules and context (a container that keeps other containers and user modules).
Gateway, like EJB, supports many sessions or users simultaneously, and both Gateway
and EJB (last version 1.1) chose XML for a persistency model. As the user puts his
own EJB objects into the EJB container, Gateway users put user modules into Gateway
contexts. Both EJB and Gateway generate an automatic code to perform serialization in
XML. The EJB object and general proxy intercept client requests of EJB and Gateway,
respectively. Currently, unlike EJB, we do not have a transaction model. The EJB
container and the Gateway contexts for EJB and Gateway systems, respectively, handle security. Gateway has a distributed event model, while in the current EJB 1.1 specification EJB does not.

DISCWorld [DiscWHPCN97] develops a middleware infrastructure for distributed high-performance computing applications. The Distributed Information Systems Control World (DISCWorld) is a smart middleware system designed to integrate processing and storage resources across wide-area heterogeneous networks, exploiting broadband communications where available.

NetSolve [Netsolve97IJSP] is a client-server application that enables users to solve complex scientific problems remotely. This system allows users to access both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available and, using “retry” for fault-tolerance, solves a problem and returns the answers to the user. Although Netsolve tries to solve simple problems, Gateway solves both simple and complex problems that may need applications consisting of many user modules or other sub-applications.

We have another related project, PAWS [PAWSWeb] (Parallel Application WorkSpace), that provides a framework for coupling parallel applications. The coupled applications can be running on different machines, and the data structures in each coupled component can have different parallel distributions. PAWS can help us to connect different Gateway applications. As stated earlier, Gateway can support the simultaneous running of many applications.
We have other NPAC research that has culminated into the Virtual Programming Laboratory (VPL) [VPL97ConcJ], that is a Web-based virtual programming environment based on a client-server architecture. It can be accessed from any platform (Unix, PC, or Mac) using a standard Java-enabled browser. Software delivery over the Web imposes a novel set of constraints on design. It outlines the tradeoffs in a design space, motivates the choices necessary to deliver an application, and details the lessons learned in the process. VPL facilitates the development and execution of parallel programs. The initial prototype supports high-level parallel programming based on Fortran 90 and High-Performance Fortran (HPF), as well as explicit low-level programming with the MPI message-passing interface. Supplementary Java-based platform-independent tools for data and performance visualization are integral parts of VPL. Pablo SDDF trace files generated by the Pablo performance instrumentation system are used for postmortem performance visualization.

VPL is excellent work that was based on emerging Web technologies a couple of years ago, and it is still a very good Web-based system that anybody can assume as a starting point. We derived much stimulating motivation from VPL.

The SUN client-side Java component model, **JavaBeans** [JavaBeansWeb], also influenced our Gateway project. We adopted the JavaBeans model at the beginning, but later removed its introspection mechanism, detecting attributes from its related get/set methods and events from its corresponding add/removeXListener methods for event X. We designed an introspection model on an XML document that is generated automatically for user modules with the help of IR (Interface Repository). We were
also guided by the JavaBeans containment model (JavaBeans Glasgow specification), but in Gateway, we have Gateway context, distributed across machines, as opposed to JavaBeans context.

Another related work is SCIRun [SciRun97Press], a scientific programming environment that permits interactive construction, debugging, and steering of large-scale scientific computations. SCIRun supports only data-flow models and allows scientists to modify the simulation parameters interactively, but our Gateway supports both event-driven and data-flow models. The computational steering aspect of SCIRun includes only user-defined parameters, but the DARP functionality of Gateway gives more detailed steering for user-defined variables and all of the program variables defined in any program segment. SCIRun uses only local computing resources, while Gateway can also use distributed objects.

In Chapter 2 we explain what we mean by HPcc (High Performance Commodity Computing) and HPCM (High Performance Commodity MetaComputing). Chapter 3 discusses the TCP/IP client-server-based DARP (Data Analysis and Rapid Prototyping) architecture. Chapter 4 explains the architecture of WebFlow, early stage of the Gateway, and Chapter 5 discusses a new distributed-object-based Gateway middle tier that fully orchestrates a distributed meta-application consisting of various components. Chapter 6 explains two applications of Gateway such as Quantum simulations (QS) and LMS with which we evaluate our Gateway system. Chapter 7 provides conclusions derived from our experiments and an outline of our plans for future work.
CHAPTER 2 – High-Performance Commodity Computing

2.1 Introduction

We believe that industry and the loosely organized worldwide collection of commercial, academic, and freeware programmers are developing a remarkable new software environment of unprecedented quality and functionality. We believe this environment can benefit HPCC in several ways by allowing the development of both more powerful parallel programming environments (DARP) and new distributed metacomputing systems (WebFlow and Gateway) [HpccGridBook, HPCCEuroPar98Gf]. We abstract these to a three-tier model with largely independent clients connected to a distributed network of servers that host various services, including object and relational databases and, of course, parallel and sequential computing. High performance can be obtained by combining concurrency at the middle server tier with optimized parallel back-end services. The resultant system, HPcc-High-
Performance Commodity Computing, combines the needed performance of large-scale HPCC applications with the rich functionality of commodity systems. In the second section we define “commodity technologies” and explain the ways they can be used in HPCC. In the third section, we define an emerging HPcc architecture in terms of a conventional three-tier commercial computing model.

2.2 Commodity Technologies and Their Use in Multi-Tier Systems

The Web is not just a document-access system supported by the somewhat limited HTTP protocol. Rather, it is a distributed-object technology that can build general multi-tiered enterprise intranet and internet applications.

There are many driving forces and many aspects to HPcc architecture, but we suggest that the three critical technology areas are the Web, distributed objects, and databases. These are being linked, and we see them subsumed in the next generation of "object-web" technologies, illustrated by the recent Netscape and Microsoft browsers. Databases are older technologies, but their linkage to the web and distributed objects is transforming their use and making them more widely applicable.

In each commodity technology area we have impressive and rapidly improving software artifacts. As examples, we have at the lower level a collection of standards and tools such as HTML, HTTP, MIME, IIOP, CGI, Java, JavaScript, Javabeans, CORBA, COM, ActiveX, VRML, dynamic Java servers, and clients that include applets and servlets. The new W3C base technologies include XML, DOM, and RDF. At a higher level of collaboration, security, electronic commerce, multimedia, and other applications/services are rapidly developing using standard interfaces or frameworks and facilities. This emphasizes that we have a set of open interfaces enabling
distributed modular software development. These interfaces are at both low and high levels, and the ones at high levels generate a very powerful software environment in which large preexisting components can be quickly integrated into new applications. In our work, we designed and built an integrated environment called **Gateway** that facilitates the construction of large applications from built-in user modules.

We believe that there are significant incentives for building HPCC environments in a way that naturally inherits all the commodity capabilities so that HPCC applications can benefit from the impressive productivity of commodity systems. NPAC's HPcc activity is designed to demonstrate that this is possible and useful so that one can simultaneously achieve both high performance and the functionality of commodity systems. We demonstrated this with a couple of applications: seamless access to HPCC resources, Landscape Modeling Simulation (LMS) and Quantum Simulation (QS) which will be explained in Chapter 7.

Note that commodity technologies can be used in several ways. XML and Web servers will help customization and installations of distributed objects and servers across the Internet. Enterprise JavaBeans, RMI, COM, and CORBA will accelerate the usage of distributed-object technology.

However, our main target is not such pointed solutions but rather adapting the architecture of commodity systems for high-performance parallel and distributed computing. Even though we have seen over the last 30 years many other major broad-based hardware and software developments, they have not had a profound impact on HPCC software. Based on our many experiments, we strongly believe that **HPcc** architecture gives us a world-wide/enterprise-wide distributing computing environment.
Previous software revolutions could help individual components of an HPCC software system, but HPcc architecture is the backbone of a complete HPCC software system. To achieve our goal, we added high performance to this architecture and in this way inherited a multi-billion-dollar investment and what is, in many respects, the most powerful and productive software environment ever built.

2.3 Three Tier High Performance Commodity Computing

We start with a common modern-industry view of commodity computing with the three tiers shown in Figure 2.1. Here we have customizable client and middle tier systems accessing "traditional" back-end services such as a relational database. A set of standard interfaces allows a rich set of custom applications to be built with appropriate client and middleware software. As indicated in Figure 2.1, the client sitting in the first tier can access to the middle tier via URL, COM/CORBA request, low-level socket, XML-based protocol, or a CGI program. The middle tier can be a simple server waiting on a port, COM/CORBA server(s) or a group of CGI programs written usually in PERL and other scripting languages. Accessing to a database at the back end is achieved through a JDBC interface. The database keeps the attribute values of application server objects as well as other related information.
The rapidly evolving commercial architecture is exploring several co-existing approaches in today's distributed information system. The most powerful solutions involve distributed objects. There are three important commercial object systems: CORBA, COM/DCOM, and Enterprise JavaBeans. We envision the possibility of coming up with an enterprise commodity architecture that allows the use of many different technologies at the same time. Actually, we realized this with our sophisticated and extendible Gateway component model that will be explained in Chapter 5.

Enterprise JavaBeans is a "pure Java" solution-cross platform, but, unlike CORBA, not cross-language! Legion is an example of a major HPCC-focused
distributed object approach; currently, it is not built on top of one of the three major commercial standards. The HLA/RTI standard for distributed simulations in the forces-modeling community is another important domain-specific, distributed-object system. It appears to be moving toward integration with CORBA standards.

Although a distributed-object approach is attractive, most network services today are provided in a more ad hoc fashion. Originally, these services were client-server with proprietary network access protocols like TCP/IP. Later, web-linked databases and enterprise intranets naturally produced a three-tier distributed service model, with an HTTP server using a CGI program to access the database at the backend. There is a trend toward using Web servers with the servlet mechanism for the services, but today we can build databases as distributed objects with a middle-tier Enterprise JavaBean or CORBA using JDBC to access the backend database. We can summarize this evolution as follows:

<table>
<thead>
<tr>
<th>client access method</th>
<th>middle tier object or executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java sockets</td>
<td>--&gt; Java program (low-level network standard)</td>
</tr>
<tr>
<td>HTTP</td>
<td>--&gt; Java servlet or CGI script</td>
</tr>
<tr>
<td>IIOP, RMI, or COM</td>
<td>--&gt; Distributed objects (high-level network standard)</td>
</tr>
</tbody>
</table>

As shown in Figure 2.2, we see today a "Pragmatic Object Web" mixture of distributed service and distributed object architectures. CORBA, COM, JavaBean, HTTP Server + CGI, Java Server and servlets, databases with specialized network accesses, and other services co-exist in the heterogeneous environment with common themes but disparate implementations. Our NPAC's HPcc strategy involves building
this architecture (Figure 2.2) and adding high performance in the third tier to this system.

Figure 2.2. Today's Heterogeneous Interoperating Hybrid Server Architecture

We also believe that the resultant architecture will be integrated with the web so that the latter will exhibit distributed-object architecture. More generally, the emergence of IIOP (Internet Inter-ORB Protocol) and the realization that CORBA is naturally synergistic with Java is starting a new wave of "Object Web" developments that could have profound importance. The resultant architecture shows a small object broker (a so-called ORBlet) in each browser as in Netscape. Most of our remarks are valid for all these approaches to a distributed set of services.
We used this service/object-evolving three-tier commodity architecture as the basis of our HPcc environment. We need to naturally incorporate (essentially) all services of the commodity web and use its protocols and standards wherever possible. We insist on adopting the architecture of commodity distribution systems, because complex HPCC problems require the rich range of services offered by the broader community systems. Porting commodity services to a custom HPCC system requires continued upkeep with each new upgrade of the commodity service. By adopting the architecture of commodity systems we make it easier to track their rapid evolution and we expect it will give high functionality to HPCC systems, which will naturally track the evolving Web/distributed object worlds. This requires us to enhance certain services to obtain higher performance and to incorporate new capabilities such as high-end visualization (e.g., CAVEs) or massively parallel systems where needed. This is the essential research challenge for HPcc, for we must not only enhance performance where needed but do it in a way that is preserved as we evolve the basic commodity systems. We certainly have demonstrated clearly with our applications of QS, LMS, and Seamless Access that this is possible through deploying our distributed object-based Gateway middle tier. In order to achieve this, we exploit the three-tier structure and keep HPCC enhancements in the third tier, which is inevitably the home of specialized services in object-web architecture. This strategy isolates HPCC issues from the control or interface issues in the middle layer. We have successfully built an HPcc environment that offers the evolving functionality of commodity systems without significant re-engineering as advances in hardware and software lead to new and better commodity products.
Returning to Figure 2.2, we see that it elaborates Figure 2.1 in two natural ways. First, the middle tier is promoted to a distributed network of servers; in the "purest" model these are CORBA/COM/Enterprise JavaBean object-web servers, but any protocol-compatible server is obviously possible. This middle-tier layer includes not only networked servers with many different capabilities (increasing functionality), but also multiple server instantiations to increase performance on a given service. The use of high functionality but modest performance communication protocols and interfaces at the middle tier limits the performance levels that can be reached in this fashion. However, this first step gives a modest performance scaling, and a parallel (implemented, if necessary, in terms of multiple servers) HPcc system that includes all commodity services such as databases, object services, transaction processing, and collaborators. The next step is applied only to those services with insufficient performance. Naively, we "just" replace an existing back end (third-tier) implementation of a commodity service by its natural HPCC high-performance version: specifically, we used **globus, MPI, and HPF**.

### 2.4 Comparison of three-tiered and two-tiered (client-server) architecture models

Even though the programming effort for a client-server is much easier, it has many disadvantages compared with three-tier systems. For applications that thousands of clients connect to, a two-tier system doesn’t scale well and may not take the benefits of multithreading. A three-tier system solves the scalability and multithreading problems by pushing them down into the middle tier. In two-tier systems, client code is mixed with user interface code and some of the actual server codes and therefore has
fatty code. Three-tier systems have thin client code and put all of the business methods of the server into the third tier. The coordination codes for these methods and the codes for allocation of resources are put into the middle tier. The security and transaction policies are inserted into the middle tier in three-tier systems, as opposed to server codes in two-tier environments. The Java-security sandbox doesn’t allow connecting to the host, other than the applet host, in two-tier systems, but introducing a middle tier will solve this problem in three-tier systems.
CHAPTER 3 – DARP SYSTEM (DATA ANALYSIS AND RAPID PROTOTYPING)

3.1. Introduction

The development of large distributed/parallel applications is a complex process. We face the problem of software integration, as different software components often follow different parallel programming paradigms. On the other hand, we witness a rapid progress of Web-based technologies that are inherently distributed, heterogeneous, and platform-independent. Of particular interest are the definition and standardization of interfaces that enable cross-platform software interoperability.

The integration of compiled and interpreted HPF gives us an opportunity to design a powerful application development environment targeted for high-performance parallel and distributed systems. This DARP environment includes a source-level debugger, data visualization and data analysis packages, and an HPF interpreter. The capability of alternating between compiled and interpreted modes provides the means for interaction with the code at real time while preserving an acceptable performance. Thanks to the interpreted interfaces typical of Web technologies we can use our system as a software integration tool.

The fundamental feature of our system is that the user can interrupt execution of the compiled code at any point and get an interactive access to the data. For visualizations, the execution is resumed as soon as the data transfer is completed. For data analysis, the interrupted code pauses and waits for the user's commands. The set of available commands closely reproduces functionality of a typical debugger (setting
breakpoints, displaying or modifying values of variables, etc.). However, a unique feature of our system is that it can issue HPF commands to modify values of distributed arrays. In this sense, our system can be thought of as an HPF interpreter. For more complex data transformations, the user can dynamically link precompiled functions written in HPF or other languages, which enables rapid prototyping. In particular, parallel libraries that do not necessarily follow the HPF computational model can in this way be integrated dynamically with the HPF code through the HPF extrinsic interface.

Implementing proxy libraries in Java further increases the functionality of our system, allowing us to design and develop the DARP system as a three-tiered system rather than a traditional client-server one. Now we can treat components of the DARP system as distributed objects to be implemented as CORBA ORB-lets or JavaBeans. We use this mechanism for dynamical embedding of calls to a visualization system (such as SciViz [Sciviz98ACMJava]) or for coupling this system with the WebFlow [WebFlow97Furm].

This section is organized as follows. In Section 3.2 we discuss the overall architecture of the system in the context of the High-Performance Commodity Computing paradigm. Sections 3.3–3.6 describe the three-tiered design of the DARP system and its components: the tier-2 DARP server, the instrumentation server, DARP front-end and HPF interpreter, respectively. Section 3.7 demonstrates the integration of the DARP system with the visualization package, using a proxy library. Finally, in Section 3.8 we give our summary and conclusions.
3.2 High-Level Architecture of DARP

The design of the DARP system follows the idea of High-Performance commodity computing (HPcc). Conceptually, the architecture of this three-tiered system can be described as follows (cf. Figure 3.1): The DARP system uses an interpreted Web client interacting dynamically with a compiled code. At this time the system uses an HPF back end, but the architecture is independent of the back-end language. The Java or JavaScript front end holds proxy objects produced by an HPF front end operating on the back-end code. These proxy objects can be manipulated with interpreted Java commands to request additional processing, visualization, and other interactive computational steering and analysis.

Figure 3.1. DARP implementation within HPcc framework

3.3 DARP Server: Interactive control over an application

As shown in Figure 3.2, the heart of the DARP system is the DARP server, which controls the execution of the application. The server accepts commands from a client...
implemented as a Java applet. Control over the execution of an application and interactive accesses to the data are achieved by a source-level instrumentation of the code.

Figure 3.2. The architecture of DARP

Since HPF follows a simple global name space, "data parallel paradigm," the DARP server can be implemented simply as an extrinsic HPF LOCAL procedure, in which case the server is part of the application and comes into existence only after the application is launched. In this scenario the application code is instrumented in such a way that the initialization of the DARP server is the first executable statement of the application. Once initialized, the server blocks the application waiting for the client to connect. From that point on, the execution is controlled by the client. Optionally, the initialization of the server may include processing a script that sets action/breakpoints and forces resuming of the execution without waiting for the user's commands.

In a general SPMD paradigm this simplistic implementation of the DARP server is not sufficient because the client loses control over the application when the code on a
single node dies. We therefore extended the server architecture. Now (Figure 3.3), the
manager, which is independent of the application, accepts requests from the client and
multicasts them to all nodes participating in the computations. The DARP server is a
part of the instrumented HPF application and is replicated over the nodes participating
in the computation. The client communicates only with the DARP manager on a
selected node (Figure 3.3). The DARP manager is a separate server that receives
requests from the front-end user applet and the instrumentation server. In order for the
server to get the user’s request even when the application is running, the server checks
any request at the beginning of the server code. Remember that the function call to the
server is inserted through instrumentation before each executable program statement.
This mechanism gives full control of the program execution to the user and allows the
user to monitor the updated sequences of any program variable during execution and
suspend, continue, or stop the execution dynamically.
The interprocessor communication required by the distributed application is not implemented using the Web-based protocols (such as CORBA IIOP), as is the case for client-manager interactions. Instead, we use the native HPF runtime support or MPI directly. For meta-computations, in our approach controlled by a network of managers, we consider replacing low-level MPI by Nexus\cite{NexusJPDC97} and other services provided by Globus as the high-performance communication layer.

3.4 Instrumentation of the code

The instrumentation of the code involves three steps:

1. Adding server functions

2. Insertion calls to the HPF server before each HPF statement

3. Identification of the types of all variables used in the application
The process is fully automated and requires no user intervention. A preprocessor that transforms a valid HPF source code into an instrumented one does the instrumentation. The instrumented code is, itself, a valid HPF code to be compiled by a generic HPF compiler.

We built the preprocessor using the HPF Front End (HPFfe) [HPFfeWeb] developed at NPAC within the PCRC consortium [PCRCWeb]. HPFfe is based on the SAGE++ system [Sage++94Gannon], which, in addition to parsing, provides the means to access and modify the abstract parse tree (AST), the symbol table, the type tables, and source-level program annotations. For our purposes we developed functions that identify attributes of all variables used in the HPF application (including the data type and runtime memory addresses) and operates on the AST to insert variables' "registration" calls (allowing the server to determine the size and location of the data to be sent) and calls to the server.

Since HPF is a superset of Fortran 90 we can apply our preprocessor to any sequential Fortran code, particularly, to a node code of any parallel application developed in Fortran that uses explicit calls to a message-passing library such as MPI or PVM. The capability of processing HPF compiler directives enhances our system in that we can preserve information on the intended data distributions and assertions on the (lack of) data dependencies.

3.5 DARP Front End

The DARP front end is implemented as a Java Applet (Figure 3.5). The user interacts with the code through an interface that closely resembles the interface provided by a typical debugger. The repertoire of commands includes setting break-
and action-points, stopping/resuming execution of the application (including stepping one instruction at a time or one iteration of a loop at a time), changing values of the application's variables, requesting data (including distributed arrays), dynamically linking and executing shared objects (including codes generated on the fly by the interpreter), and more. Our prototype implementation supports three types of commands, of which the first category is data access commands (Table 1), the second is prototyping commands (Table 2), and the last one is control commands (Table 3). Unless otherwise stated in the fourth column of these tables, the user in the front end needs to submit the chosen processor ID (MIMD case) or no ID (SIMD case), the specific command tag (in the first column), and input parameters to the DARP manager, which delegates the request to the processor specified in the request. In the opposite direction, after the processor has performed the user’s request, it sends the result (with the ACKCOMMAND command tag) back to the manager, which finally forwards the result to the applet. If we look carefully at the return values of the user requests (in the third column of the tables), we see that we send back all of the information that was sent by the user. The reason for this symmetry is that our DARP system allows many users to connect to one running program. To do that, the manager keeps the currently connected users and whenever it receives a command result from the HPF server, it multicasts it to the bound users.

Note that since the code is instrumented on the source level, our "debugger" gives access only to the source-level data. In particular, we are unable to provide a complete state of the machine (registers, buffers, etc.) at any given time as many commercial debuggers do, and as is recommended by the High-Performance
Debugging Forum [HPDForumWeb]. Also, since at this time we exclusively address
applications in HPF, we ignore several features necessary for supporting a more general
SPMD paradigm. In particular, we assume that interprocessor communications are
facilitated by a bug-free HPF runtime system. However, the more advanced
implementation of the DARP system with the independent DARP manager (cf. Section
3.3) makes it possible to control applications that use explicit message passing.
Anyway, the DARP system is not designed to be a system-level debugger, but to
perform such actions as manipulating large distributed data objects in order to
investigate the convergence and stability of algorithms used in scientific simulations.

Typically, a client-server architecture is used to implement a portable debugger
for distributed systems (cf. [DAQV96Sth, FalconWeb, Tuchman91Vis,
CumulvsWeb, Panorama93MF, CH94p2d2]). Our approach is unique in that we use
a three-tiered architecture and can easily integrate our source-level debugger with the
HPF interpreter and a visualization tool that together comprise a powerful application
development environment. Moreover, we envision that we will have different HPCC
codes written by the HPCC community and we need to use them together to solve
highly complex and computation-intensive applications. Our DARP manager helps to
abstract a parallel application running on multiple nodes into a single application, thus
allowing the user to manipulate an application through the manager, which can talk to
the other managers and servers.
<table>
<thead>
<tr>
<th>Data commands</th>
<th>Access Parameters</th>
<th>Input Parameters</th>
<th>Return Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAYVARIABLE</td>
<td>Variable Name</td>
<td>ACKCOMMAND Received command + Current Line Number(LN) + My Processor PID + Error code + DATADESC + DATAVALUE</td>
<td>Get the value of program variable, DATAVALUE whose data description is DATADESC.</td>
<td></td>
</tr>
<tr>
<td>DISARRAYSECTION</td>
<td>Variable Name + Array Section Stmt</td>
<td>Same as above</td>
<td>Get the array section specified by F90 array section expression</td>
<td></td>
</tr>
<tr>
<td>SETVARIABLE</td>
<td>Variable Name + DATAVALUE</td>
<td>Same as above except last two items</td>
<td>Set the value of variable to DATAVALUE</td>
<td></td>
</tr>
<tr>
<td>SETPARVAR</td>
<td>Variable Name + DATAVALUE</td>
<td>Same as above except last two items</td>
<td>Set the Data parallel(SIMD) array to the value DATAVALUE</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Data Access commands of HPF server
<table>
<thead>
<tr>
<th>Prototyping Commands</th>
<th>Input Parameters</th>
<th>Return Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERPRETER</td>
<td>Interpreted legal HPF Statement</td>
<td>None</td>
<td>After receiving, it sends HPF stmt to Instrumentation server</td>
</tr>
<tr>
<td>INST_CODE</td>
<td>Created function from user’s interpreted stmts to achieve same affect of these stmts</td>
<td>ACKCOMMAND Received command + Current Line Number(LN) + My Processor PID + Error code + Interpreted HPF Stmts</td>
<td>This command is received from Instrumentation server which creates function out of user stmts.</td>
</tr>
<tr>
<td>SCIVISENV</td>
<td>The host and port of SciVis server</td>
<td>Same as INST_CODE except last one</td>
<td>Set Scivis Host and Port</td>
</tr>
<tr>
<td>SCIVIS_FUNLIST</td>
<td>None</td>
<td>Same as SCIVISENV plus SciVis function names</td>
<td>Send SciVis interface function names</td>
</tr>
<tr>
<td>SEND_LOCALS</td>
<td>None</td>
<td>Same as SCIVISENV plus local variable names</td>
<td>Send local variable names of current execution point</td>
</tr>
<tr>
<td>ADD_ACTION</td>
<td>SciVis function name + LN</td>
<td>Same as SCIVISENV</td>
<td>Register Scivis function call for this line LN</td>
</tr>
<tr>
<td>DELETE_ACTION</td>
<td>Same as just above</td>
<td>Same as SCIVISENV</td>
<td>Deregister function call</td>
</tr>
<tr>
<td>ACTION_LIST</td>
<td>None</td>
<td>Same as SCIVISENV plus line numbers of action points</td>
<td>Return line numbers of all action points.</td>
</tr>
<tr>
<td>GET_FUNCTION</td>
<td>Line number(LN) and which function</td>
<td>Same as SCIVISENV plus arguments of function at LN</td>
<td>Returns the argument names of that function</td>
</tr>
<tr>
<td>STORE_CONFIG</td>
<td>File name</td>
<td>Same as SCIVISENV</td>
<td>Stores action or break points and other parameters in specified file</td>
</tr>
<tr>
<td>GET_PROFILE</td>
<td>None</td>
<td>Same as SCIVISENV plus profile info. For each program line</td>
<td>Returns number of executions and total time for each line.</td>
</tr>
<tr>
<td>SEND_CONFIGS</td>
<td>None</td>
<td>Same as SCIVISENV plus config files</td>
<td>Returns stored config file names</td>
</tr>
<tr>
<td>READ_PARSE_TREE</td>
<td>None</td>
<td>Same as SCIVISENV plus parse tree info</td>
<td>Returns the parse tree info. Of all project files</td>
</tr>
</tbody>
</table>

Table 2. Prototyping commands of HPF Server
<table>
<thead>
<tr>
<th>Control Commands</th>
<th>Input Parameters</th>
<th>Return Value(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAUSE</td>
<td>None</td>
<td>ACKCOMMAND</td>
<td>Pause the execution at program line, LN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Received command + Current Line Number(LN) + My Processor PID + Error code</td>
<td></td>
</tr>
<tr>
<td>PUTBREAKON</td>
<td>LN</td>
<td>Same as above</td>
<td>Put break point at line, LN</td>
</tr>
<tr>
<td>PUTBREAKOFF</td>
<td>LN</td>
<td>Same as above</td>
<td>Remove break point at line, LN</td>
</tr>
<tr>
<td>STEPINTO</td>
<td></td>
<td>Same as above</td>
<td>Go to next line, LN</td>
</tr>
<tr>
<td>STEPOVER</td>
<td>None</td>
<td>Same as above</td>
<td>Go to next line, LN (entering into function calls if any)</td>
</tr>
<tr>
<td>NSTEPINTO</td>
<td>N-the number of steps</td>
<td>Same as above</td>
<td>Perform N StepInto commands at once</td>
</tr>
<tr>
<td>NSTEPOVER</td>
<td>N-the number of steps</td>
<td>Same as above</td>
<td>Perform N StepOver commands at once</td>
</tr>
<tr>
<td>NITERATION</td>
<td>N-the number of loop iteration</td>
<td>Same as above</td>
<td>If execution inside a loop, make N iteration</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>None</td>
<td>Same as above</td>
<td>Continue the execution until one break point is reached or program is stopped</td>
</tr>
<tr>
<td>STACKDOWN</td>
<td>None</td>
<td>Same as above</td>
<td>Go to one lower level in stack frame</td>
</tr>
<tr>
<td>STACKUP</td>
<td>None</td>
<td>Same as above</td>
<td>Go to one upper level in stack frame</td>
</tr>
<tr>
<td>STOP</td>
<td>None</td>
<td>Same as above</td>
<td>Stop the program execution</td>
</tr>
</tbody>
</table>

Table 3. Control commands of HPF Server
3.6 Integrated Environment for HPF Compiler and Interpreter

The architecture of this system allows for real-time interaction with an executing HPF code. At each synchronization point (when the DARP server is accepting requests), the data can be extracted and processed as if an explicit call to an HPF extrinsic procedure were made. HPF statements, in particular, can be executed in such an interactive fashion.

In this way the system achieves the functionality of an HPF interpreter. The interaction between the running application and the user's commands is based on dynamical
linking of UNIX shared objects with the application. Any precompiled stand-alone or library routine with a conforming interface can be called interactively at a break point or at selected action points. In order to execute a new code entered in the form of an HPF source it must first be converted to a shared object. To do this, the user submits any legal HPF sequence of statements to the manager, which forwards the request to the specified processor(s). The processor(s) send statements to the instrumentation server, which creates a legal HPF subroutine code by using the HPFfe system and submits the code to the processor. The processor compiles it using an HPF compiler (Figure 3.4). Since any "interpreted" code is in fact compiled, the efficiency of the resulting code is as good as that of the application itself. Nevertheless, the time needed to create the shared object is prohibitively long for attempting to run complete applications, statement after statement, in the interpreted mode. On the other hand, the capability to manipulate and visualize data at any time during the execution of the application, without recompiling and rerunning the whole application, proves to be very time effective.

3.7 Runtime Visualizations

For visualizations we used the Scientific Visualization System, SciVis, a portable system developed at NPAC entirely in Java. With a very rich and user-extensible set of data filters, and full support for collaborative use, it is a very powerful tool for a rapid data analysis. From the user’s point of view, it consists of a stand-alone server that is typically run on the user's workstation and a client that supplies the data. The upper left panel shows the front-end applet with a fragment of an HPF code. The action points at which the SciVis proxy library is called are highlighted and a triangle on the left points to the current line (Figure 3.5).
The architecture of the SciVis system makes it particularly attractive for integration with the DARP system. The SciVis client API allows us to design a proxy library in Java with a simple and very intuitive interface. The library, on behalf of the user, causes the automatic creation of a SciVis client routine that corresponds to the data type requested by the user. The client is then dynamically linked with the running application and executed at a specified action point which results in sending the data to the SciVis server, which in turn displays them on the user's workstation screen.
The same mechanism can be used with dedicated proxy libraries to integrate the DARP system with other software packages such as computational libraries, data storage systems, or other visualization systems. By using proxy libraries the DARP system may request or provide services from other tier-2 servers, or become a module in data-flow type computations [WebFlowDARP].

3.8 Adding a different visualization engine to DARP system

Bringing a different visualization engine to DARP is nothing more than adding a new “Visualization Manager” instance with a special configuration for this new engine (Figure 5.6). The Visualization Manager is a generic module that can be configured for a visualization engine that may accept only special data formats and may have specific environmental variables. The HPF server sends a visualize command that includes the data with its descriptive information to the DARP manager, which forwards the data in order to the visualization manager. The last manager makes the necessary format conversion on the received data according to the manager’s configuration by the user at startup time and sends the data in accepted format to the visualization engine. With this method, the HPF server does not need to deal with any visualization engine-related operation, but may simply send the data with its definition to its DARP manager.
3.9 Summary

By reusing commodity components and technologies we have built a powerful tool for data analysis and rapid prototyping to be used by an HPF application developer. The most important feature of the system is interactive access to distributed data which makes it possible to select and send data to a visualization system at an arbitrary point of the application execution. The data can be modified using either native HPF commands or dynamically linked computational modules.

Consistent with our HPcc strategy, the system implements a three-tiered architecture: The Java front end holds proxy objects produced by an HPF front end operating on the back-end code. These proxy objects can be manipulated by an
interpreted Web client interacting dynamically with compiled code through a middle tier (middleware). We successfully ran DARP as a WebFlow module and demonstrated it at SC’97. We fully integrated DARP functionality into WebFlow, which is explained in Chapter 5.

Although targeted for the HPF back end, the system's architecture is independent of the back-end language and can be extended to support other high-performance languages such as HPC++[HPC++Web] or HPJava[HPJavaACM98]. Finally, since we follow a distributed objects approach, the DARP system can be easily incorporated into a collaborative environment such as Tango [TANGOWeb] or Habanero [HabaneroWeb].
CHAPTER 4 – EARLY STAGE OF GATEWAY: WEBFLOW ARCHITECTURE

4.1 Introduction

Programming tools that are simultaneously sustainable, highly functional, robust and easy to use have been hard to come by in the HPCC arena, thus we have developed a new strategy, as explained in Chapter 2. It is called HPcc: High Performance Commodity Computing [HPccGridBook], and it 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. We add high performance to commodity systems using a multi-tiered architecture with the Globus [GlobusIntlJ97] metacomputing toolkit as the back end of a middle tier of commodity web and object servers.

This approach was not possible a few years ago when enterprise computing was still mainly client-server (two-tiered) and based on expensive custom solutions such as proprietary TP Monitors. The onset of the Web and the associated intranets accelerated the development of scalable and open three-tiered standards such as CORBA [OMGWeb], DCOM [COMWeb], and Enterprise JavaBeans (EJB)
We can therefore now prototype a dedicated and advanced, yet commercial-quality HPDC system, for HPCC applications by integrating open commodity standards for distributed enterprise computing with traditional (such as MPI or HPF) and emerging (GLOBUS) HPCC infrastructures that are optimized for performance.

Figure 4.1. Top-level view of the WebFlow environment

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 that span across laboratories and facilities. More specifically, we are developing WebFlow—a scalable, high-level, commodity-standards-based HPDC system (Figure 4.1) that integrates the following:
• High-level front ends for visual programming, steering, run-time data analysis and
  visualization, and collaboration built on top of the Web and object-oriented
  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 (Tier 3).
  
  Note that this can be applied to either parallel or metacomputing applications and
  provides a uniform cross-platform, high-level computing environment. We believe that
  such an ambitious and generic framework as Webflow can be successfully built only
  when closely related to some specific large-scale application domains that can provide
  specification requirements, testbeds, and user feedback during the entire course of the
  system design, prototyping, development, testing, and deployment. We view NCSA
  Alliance and DoD modernization programs as attractive application environments for
  HPcc because of its unique, clear mission and advanced computational challenges,
  opportunities, and requirements.

  **WebFlow** is a specific programming paradigm implemented over a virtual
  Web-accessible metacomputer and provided by a dataflow-programming model (other
  models under experimentation include data parallel, collaborative, and televirtual
  paradigms). The WebFlow application is given by a computational graph visually
  edited by end-users, using Java applets. Modules are written by module developers,
  people who have only limited knowledge of the system on which the modules will run.
  They need not concern themselves with issues such as allocating and running the
modules on various machines, creating connections among the modules, sending and receiving data across these connections, or running several modules concurrently on one machine. The WebFlow system hides these management and coordination functions from the developers, allowing them to concentrate on the modules being developed.

4.2 WebFlow Overview

Another NPAC research group [WebFlow97Furm] originally developed WebFlow and we extended it to experiment with some high-performance simulation codes. The visual HPDC framework introduced by this WebFlow project offers an intuitive Web browser-based interface and a uniform point of interactive control for a variety of computational modules and applications running at various labs on different platforms and networks. New applications can be composed dynamically from reusable components just by clicking on visual module icons, dragging them into the active WebFlow editor area, and linking by drawing the required connection lines. The modules are executed using Globus [GlobusWeb] optimized components combined with the pervasive commodity services where native high-performance versions are not available. For instance, today one links Globus-controlled MPI programs to WebFlow (Java connected) Windows NT and database executables. When Globus becomes extended to full PC support, the default WebFlow implementation is replaced by the high-performance code.

Individual modules are typically represented by visualization, control, steering, or collaboration applets, and the system also offers visual monitoring, debugging and administration of all of the distributed applications and the underlying metacomputing
resources. In the following chapter we introduce more sophisticated distributed-object-based Gateway architecture, offering tools for easy conversion of existing (sequential, parallel, or distributed) applications to visual modules via suitable CORBA, COM, or JavaBeans-based [JavaBeansWeb] wrapper/proxy techniques.

New applications created within the WebFlow framework follow a natural modular design, that accumulates in the first phase of a project a comprehensive, problem-domain-specific library of modules. It is then possible to explore the computational challenges of the project in a visual interactive mode in an effort to compose the optimal solution of a problem in a sequence of on-the-fly trial applications. The scripting capabilities of WebFlow, coupled with database support for session journalizing, facilitates playback and the reconstructing of optimal designs discovered during such rapid prototyping sessions.

For parallel object and module developers, we will also provide finer-grained visual and scripted parallel software development tools using the new Uniform Modeling Language (UML) [UMLWeb] recently accepted as an OMG standard. UML offers a spectrum of diagrammatic techniques that allows us to address various stages of the software process and several hierarchy layers of a complex software system. In this way WebFlow will combine the features of UML-based visual tools such as Rational Rose with both high performance and the proven value of data-flow-based visual programming environments such as Khoros and AVS.

Nothing prohibits the user from encapsulating a data-parallel application as a single WebFlow module. In this case the user is solely responsible for interprocessor communications (we used HPF- and MPI-based codes to run WebFlow modules on a
multiprocessor system [DARPSC97]). Moreover, using the DARP system [DARP98Conc] implemented as a WebFlow module, we were able to interactively control an HPF application at runtime and dynamically extract the distributed data and send it to a visualization engine. This approach can be used for computational steering, runtime data analysis, debugging, and interprocessor communications on demand. Finally, we integrated two independently written applications that write checkpoint data [SC98Pres]. We used WebFlow to detect the existence of the new data, and transfer it to the other application.

4.3 Three-Tiered Architecture of the WebFlow

In our approach we adopted integrative methodology, i.e., we set up a multiple-standards-based framework in which the best assets of various approaches accumulate and cooperate rather than compete. We started the design from the middleware, which offers a core or a “bus” of modern three-tiered systems, and we adopted Java as the most efficient implementation language for the complex control required by the multi-server middleware. Atomic encapsulation units of WebFlow computations are called “modules,” and they communicate by sending objects along channels attached to a module. Modules can be dynamically created, connected, scheduled, run, relocated, and destroyed.

4.3.1 The Front End

The WebFlow Applet is the front end of the system. Through it users can request new modules to be initialized, their ports connected, and the whole application run and, finally, destroyed.
The WebFlow editor provides an intuitive environment for visually composing (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 as well. Once created, a 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 taken care of by the middle-tier services. With the help of the interface of the Globus system in the back end, execution of particular modules can be delegated to powerful HPCC
systems. In the run mode, the visual representation of the meta-application is passed to the middle tier by sending a series of requests (module instantiation, intermodule communications) to the Session Manager.

Control of module execution cannot be exercised just by sending relevant data through the module's input ports. The majority of modules that we have developed require some additional parameters that can be entered via “Module Controls” (in a way similar to that of systems such as AVS). These module controls are Java applets displayed in a card panel of the main WebFlow applet. The communication channels (sockets) between the back-end implementation of a module and its front-end module controls are generated automatically during the instantiation of the module.

Not all applications follow closely the data-flow paradigm. Therefore, it is necessary to define an interface so that different front-end packages can be "plugged in" to the middle tier, giving the user a chance to use the front end that best fits the application at hand. Currently, we offer visual editors based on GEF [GEFWeb] and VGJ [VGJWeb]. In the future, we will add an editor based on the UML standard, and we will provide an API for creating custom editors.

While designing the WebFlow we assumed that the most important feature of the front end should be a capability to create dynamically many different networks of modules tailored to application needs. However, working with real users and real applications we found that this assumption is not always true. WebFlow can be used just as a development tool by taking advantage of graphical authoring tools to create the application (or a suite of applications). Once created, the same application (i.e., network of modules) can be run by the end user over and over again without any changes in the
different input sets. In such a case the design of the front end should be totally different. The decided functionality is to provide an environment that allows navigating and choosing the application and data that will solve the problem at hand, with any technical nuances of the application being hidden from the end user.

The front end provides a platform-independent and web-accessible interface to a high-performance metacomputing environment. Given access to the Internet, the user can create and execute an application using adequate computational resources anywhere, anytime, and even from a laptop personal computer. It is the responsibility of the middle tier to identify and allocate resources and to provide access to the data.

4.3.2 The Middle-Tier

Our prototype WebFlow system is given by a mesh of Java-enhanced Web Servers [APACHEWeb] running servlets that manage and coordinate distributed computation. This management is currently implemented in terms of three servlets: Session Manager, Module Manager, and Connection Manager. These servlets are URL addressable and can offer dynamic information about their services and current state. They can also communicate with each other through sockets. Servlets are persistent and application-independent.

In the implementation of WebFlow we ignored security issues. Again, in line with our HPcc strategy, we closely watched the development of industry standards. At this time the SSL suite of protocols is clearly the dominant technology for authorization, mutual authentication, and encryption mechanisms. The most recent release of Globus implements SSL-based security features. In order to access Globus high-performance computational resources, a user must produce an encrypted
certificate digitally signed by the Globus certificate authority, and in return the Globus side (more precisely, the GRAM gatekeeper) presents its own certificate to the user. This mutual authentication is necessary for exchanging encrypted messages between the two parties. However, authorization to use the resources is granted by the system administration that owns the resources, and not by Globus. We are experimenting with a similar implementation for WebFlow.

### 4.3.3 Session Manager

The Session Manager is the part of the system in charge of accepting user commands from the front end and of executing them by sending requests to the rest of the system. The user requests, that the Session Manager honors, are creating a new module, connecting two ports, running the application, and destroying the application. Since the Session Manager and the front end generally reside on separate machines, the Session Manager keeps a representation of the application that the user is building, much like the representation stored in the front end. The difference between these two representations is that the Session Manager needs to worry about the machines on which each of the modules has been started, while the front end worries about the position of the representation of the module on the screen. The Session Manager acts as a server for the front end but uses the services of the Module and Connection Managers. All of the requests received from the user are satisfied by a series of requests to the Module and Connection Managers, which store the actual modules and ports.

### 4.3.4 Module Manager

The Module Manager is in charge of running modules on demand. When the creation of a module is requested, that request is sent to the Module Manager residing on the
particular machine on which the module should be run. The Module Manager creates a separate thread for the module (thus enabling concurrent execution of multiple modules), and loads the module code, making the module ready for execution. Upon receipt of a request to run a module, the Module Manager simply calls a run method, which each module is required to have. That method, written by the module developer, implements the module's functionality. Upon receipt of a request to destroy a module, the Module Manager first stops the thread of execution of the module, then calls the special “destroy” method. The destroy method is again written by the module developer, which performs all the clean-up operations deemed necessary by the developer.

4.3.5 Connection Manager

The Connection Manager is in charge of establishing connections between modules, or more precisely, between input and output ports of the modules. As the modules can be executed on different machines, the Connection Manager is capable of creating connections across the network, in which case it serves as a client to the peer Connection Manager on the remote WebFlow server. The handshaking between the Managers follows a custom protocol.

4.4 The Back End

The module API is very simple: it implements a specific WebFlow Java interface, the metamodule. In practice, the module developer has to implement three methods: initialize, run, and destroy. The initialize method registers the module itself and its ports to the Session Manager and establishes the communication between itself and its front-end applet -- the module controls. The run method implements the desired functionality of the module, while the destroy method performs clean-up after the
processing is completed. In particular, the destroy method closes all socket connections that are not destroyed by the Java garbage collector.

It follows that the development of WebFlow modules in Java is straightforward. Taking into account the availability of more and more Java APIs such as JDBC, this allows the creation of quite powerful, portable applications in Java. To convert existing applications written in languages other than Java, the Java native interface can be used. Moreover, the execution of the module can be delegated to an external system capable of resource allocation such as Globus. Indeed, at Supercomputing’97 in San Jose, California, we demonstrated an HPF application run under the control of WebFlow [DARPSC97]. The WebFlow front end gave us the control to be able to launch the application on a remote parallel computer, extract data at runtime, and process the data by WebFlow modules written in Java running on the local machine. The runtime data extraction was facilitated by the DARP system converted to a WebFlow module.

For more complex meta-applications, a more sophisticated back-end solution is needed. As usual, we go for a commodity solution. Since commercial solutions are practically nonexistent, in this case we use technology that comes from the academic environment, the metacomputing toolkit of Globus. The Globus toolkit provides all the functionality we need. The underlying technology is a high-performance communication library, Nexus. MDS (Metacomputing Directory Services) allows resource identification, while GRAM (Globus Resource Allocation Manager) provides secure mechanisms to allocate and schedule the resources. The GASS package (Global Access to the Secondary Storage) implements a high-performance, secure data transfer
that is augmented with an RIO (Remote Input/Output) library that provides access to parallel data file systems.

In order to run WebFlow over Globus there must be at least one WebFlow node capable of executing Globus commands, such as “globusrun.” In other words, there must be at least one host on which both Globus and the WebFlow server are installed. This host serves as a "bridge" between two domains (cf. Figure 4.3), a network of WebFlow servers and a network of resources controlled by Globus called the Grid. The jobs that require the computational power of massively parallel computers are directed to the Globus domain, while others can be launched on much more modest platforms, such as the user’s desktop or even a laptop running Windows NT.

![Figure 4.3. Bridge between WebFlow and Globus resources (the Grid)](image)

Both Globus and WebFlow gain from this symbiotic coexistence. From the WebFlow perspective, Globus is an optional, high-performance (and secure) back end, while WebFlow serves as a high-level, web-accessible visual interface and job broker for Globus. Together they cover a much wider application domain: Globus adds the HPCC world to WebFlow, and WebFlow adds commodity software, particularly the software that is available only on Microsoft Windows95/98/NT.
We are aware that providing a remote access to Globus resources (either via front-end applets or WebFlow server-to-server connections) we may introduce a security hole into the system. We made several experiments to upgrade WebFlow security standards in order to match those of Globus. However, at this time we have postponed incorporating any security mechanism into WebFlow until we rebuild the middle tier using CORBA and until some widely accepted standards emerge – preferably defined by DATORR [JGrandeWeb].

WebFlow interacts with Globus via the GRAM gatekeeper. A dedicated WebFlow module serves as a proxy of the gatekeeper client, which in turn sends requests to GRAM. Currently, the proxy is implemented using Java native interface. However, in collaboration with the Globus development team, we are working on a pure Java implementation of the gatekeeper client.

At this time GASS supports only the Globus native x-gass protocol, which restricts its use to systems in which Globus is installed. We expect support for other protocols, notably ftp, soon. This will allow us to use a GASS secure mechanism for data transfer from and to systems outside the Globus domain that are under control of the WebFlow. We also collaborate with the Globus development team to build support for other protocols, namely, HTTP and LDAP. In particular, support for HTTP will allow us to implement data filtering on the fly, as the URL given to GASS may point not to the data directly, but to a servlet or CGI script instead.
4.5.1 Logging into the WebFlow system

The user starts a session by sending the "start-session" command to the Session Manager (SM) that returns the host, **SMHost** (on which SM runs), and the port, **SMport** (SM port), to the user. It also returns the user-unique client identifier, **ClientID**, and the URL file, **moduleListURL**, that includes descriptions of user modules. This step is done automatically when the user downloads the WebFlow Front End applet from the HTTP server.

![Diagram showing the login process into the WebFlow system](image)

Figure 4.4. Starting a user session in the WebFlow system

4.5.2 Creating A New Module

While describing the creation of a new module, we will also give necessary details concerning the Session Manager (SM) and the module manager. The user
always initiates the commands for the SM by visual authoring tools inside the applet with click and drag-drop mouse operations, and these SM commands are sent automatically on behalf of the user. We will refer to each entry of “X table” or “X list” as “X Object” in the following discussion.

SM holds the **WFlowSession** list (indexed by **ClientID**). Each entry of this list includes the **ModuleRepresentation** table (indexed by **ModuleID**) and the **ViewerList**-html Strings (indexed by **htmlKey**) for user modules and the module description list. The **ModuleRepresentation** object for each module keeps the port **CMPort** (its Connection manager Port), the port **MMPort** (its ModuleManager Port), and the host in which it lives. The MM (Module Manager) holds the **ModuleWrapper** table that, indexed by **ModuleID**, holds a **ModuleWrapper** object for the user-defined module in each entry. MM sends the commands to **ModuleWrapper** (running as a separate thread), which forwards them to the Module itself. The advantage to this is that **ModuleWrapper** can return control to the MM instantly, especially when running the Module, because **ModuleWrapper** runs the actual module **run** method as a separate thread.

The **New-module** command with the parameters **ClientID** and **ClassName** for the module to be created is sent to the SM after the user drops a visual icon for the module to the Graphic Editor Frame. Then the SM finds the corresponding MM for this module and sends the **INITIALIZE** command with the parameter module **ClassName** to it. The SM gets the **MetaModule** object for the created module, creates its **ModuleRepresentation** object by using information inside the **MetaObject**, and inserts this created object into the **ModuleRepresentation** table. The SM also inserts
the **htmlString** of this module into the **ViewerList** table. The **MetaModule** object actually includes the state (description), ModuleID, port list, and **viewerWinName** for the new module. The SM forwards the contained information inside the received **MetaModule** object to the applet. If there is a matching front-end applet for this module, the applet sends a "**new_viewer**" command with parameters **htmlKey** and **ClientID**, receives the **html string**, and pops up the module applet.

![Diagram](image)

Creating and initializing one module

Figure 4.5. Creating a new module

### 4.5.3 Making a connection between modules

Creating a connection between modules is more complex than initializing modules. As seen in Figure 4.6, to establish a connection from module **fromModule** to another module **toModule**, we need port IDs (**fromPortId** and **toPortID**) and Modules
IDs (fromModuleID and toModuleId) of the module pair. Module IDs and their port IDs are received during the initialization of modules. We send all of this information together with ClientID as a Connect command to the SM. Then the SM finds the CMHost (CM host) and CMPort (CM Port) for module toModule and sends them with port IDs of two modules inside the ESTABLISH command to peer CM for module fromModule.

CM keeps a portList table, indexed by a port ID, that includes a PortRepresentation for each module port. The CM also has two ports: one for requests and the other for connections, which is drawn in Figure 4.6 as a filled small circle at the left CM. In the connection process, the CM got fromPortID (for the source module) and toPortID (the target module). The user puts the module Ports obtained by creating OutputPort and InputPort objects (inside the initialize method), where the PortRepresentation object is created for each port and registered that into the CM by calling CM's static method, registerPort. The function of the CM job is to look up PortRepresentation objects for port pairs by means of fromPortID and toPortID and to make connections between these two ports and set the storedPort field (type of Port object) of PortRepresentation objects.

Returning to the connection process, the CM at fromHost sends the RECEIVE command with arguments fromPortID, toPortID, CMHost, and the connections port for fromHost. The CM at host toHost makes a socket connection (as client) to the connections port for host fromHost (used only to make connections), sets PortRepresentation of toPort to this established connection, and sends an OK message to the CM at fromHost. The CM of fromHost sets PortRepresentation of
fromPort to a returned connection from the accept method call of the ServerSocket object and sends an OK message to the SM, which also sends an OK to the applet. This scenario is that of the general case. If two CMs are on the same host, steps 3 and 5 are not needed. Here the user doesn't send any command to CM to register its ports, but gives its Port objects to the CM, because the host names of the CM and MM of one module have to be the same.

Figure 4.6 Creating a connection between modules
4.5.4. Run/stop/destroy modules

The current WebFlow supports only the running of modules together, not separately. After the user finishes building a visual graph, he may send a run command to the SM with parameter **ClientID**. As seen in Figure 4.7, an SM looks up the **WFlowSession** object. For each **ModuleRepresentation** inside this object, the SM sends a **RUN** command with **ModuleID** to the appropriate Module Manager (MM). Each MM finds a **ModuleWrapper** corresponding to the received **ModuleID** and calls its **runModules** method, which then calls an actual run method from the user module implementation. Stop/Destroy command is used in the same way as the **Run** command.

![Image of diagram showing the process of running, stopping, and destroying modules]

**Run/Stop/Destroy of Modules.** Here user has n modules. Session manager sends the commands to module managers. Some of module managers may be same.

4.7. Run/Stop/Destroy modules
4.6 Limitations of Original WebFlow and Other Alternative Solutions

To summarize, we have developed a platform-independent, three-tiered system: the visual authoring tools implemented in the front end, integrated with the middle-tier network of servers based on industry standards and following a distributed-object paradigm, facilitate a seamless integration of commodity software components. In particular, we use the WebFlow as a high-level, visual user interface for GLOBUS. This not only makes the construction of a meta-application much easier for an end user, but also allows this state-of-the-art HPCC environment to be combined with commercial software, including packages available only on Intel-based personal computers.

Although our prototype implementation of WebFlow proved to be very successful, we are not satisfied with it as a complete solution to the problem. Its advantages are the following:

1. It follows the industry-proved standard of a three-tiered model.
2. Developing back-end WebFlow modules and their front-end controls can be achieved independently of each other.
3. WebFlow supports a session concept for each working user that doesn’t interfere with other users.
4. Module developers don’t need to deal with low-level issues such as allocating computing resources and connecting modules.

However, WebFlow doesn’t have many of the features a scientist typically expects to see during the development of a distributed application from previously generated modules. These properties have to be developed and the failings must be corrected:
1. The user must construct a visual graph to represent the back-end application first and start all of the modules at once; the flexibility of adding and starting modules incrementally is also needed.

2. Whenever the user brings a corresponding icon into the front-end palette for a back-end module, there is middle-tier processing to allocate system resources for this new module, and there is no “undo” operation for this intervening middle-tier process.

3. When the user makes a connection between two modules in the front-end visual graph, the front-end applet sends the connection command to the middle tier automatically. There is no way to recover this connection process.

4. There is no utility to save and restore a finished, distributed application. The entire visual graph must be rebuilt every time you want to create it.

5. There is no security and transaction model, and it is difficult to put a security model on top of the current WebFlow.

6. WebFlow doesn’t provide for replacing an old running module with a new one. An entire application must be created out of new modules every time a module needs to be replaced with a new one.

7. Even though we tried to extend the original WebFlow so that the user front-end controls for back-end modules can communicate, much multithreading coding and many complex operations are required. There has to be an easy way for the front-end control to talk with the matching back-end module, such as CORBA method call.
8. Even though we successfully integrated WebFlow with the DARP system by representing DARP as an WebFlow module, much coding is needed and that includes more synchronization and multithreading issues. We have to find a way to strongly couple DARP with WebFlow, possibly by incorporating DARP’s functionality into WebFlow’s middle tier.

9. The original WebFlow doesn’t support the assigning of attributes of user modules or saving and recovering them from/to the database.

10. Because developing distributed applications is a complex process, it is very easy to make a mistake. Therefore, there has to be a monitoring environment for all interactions among front-end and back-end user modules. The original WebFlow doesn’t support this facility.

11. WebFlow doesn’t allow one user to construct and test multiple distributed applications concurrently.

12. WebFlow allows only a data-flow model, but we sometimes need event-driven associations among user modules, which we need to provide.

13. WebFlow doesn’t have a service concept. In the high-performance community we have to provide some services such as a Database server, a Directory Server, a Batch Job submitter, various metacomputing services such as MDS (Metacomputing Directory Service), GRAM (Globus Resource Allocation Manager), GSS (Globus Security Service), and GASS (Globus Access to Secondary Storage). We need a simple way to attach these kinds of services to and detach them from the WebFlow system so that user modules can reach it easily.
Pursuing HPcc goals, we would like to base our implementation on the emerging standards for distributed objects and take full advantage of the possible leverage realized by employing commercial technologies. Our research led us to the following observations: The "Java Platform" or "100% Pure Java" philosophy is being advocated by Sun Microsystems, while the industry consortium led by the OMG pursues a multi-language approach built around the CORBA model. It has been observed recently that the Java and CORBA technologies form a perfect match as complementary enabling technologies for distributed-system engineering. In such a hybrid approach, referred to as the Object Web, CORBA offers the base language-independent model for distributed objects and Java offers a language-specific implementation engine for CORBA brokers and servers.

Meanwhile, other total-solution candidates such as DCOM by Microsoft or WOM (Web Object Model) by the World-Wide Web Consortium for distributed objects/components are emerging. However, standards in this area and interoperability patterns between various approaches are still in the early formation stage. A closer inspection of the distributed object/component standard candidates indicates that, while each of the approaches claims to offer a complete solution, each in fact excels only in specific selected aspects of the required master framework. Indeed, it seems that WOM is the easiest, DCOM the fastest, pure Java the most elegant, and CORBA the most realistic complete solution.

Consequently, to implement the new WebFlow system, we chose CORBA as the base distributed-object model at the Intranet level, and the Web as the worldwide
distributed-object model instead of patching the original WebFlow to solve the above problems.
5 Distributed Object-Based Gateway Architecture

5.1 Impact of Gateway on Seamless Access to HPCC Resources

Seamless access creates an illusion that all the resources needed to complete the user tasks are available locally. In particular, an authorized user can allocate resources without explicit login to the host controlling the resources. We have many examples of this scenario in other platforms such as NSF-mounted disk or a network printer. An user thinks his or her home directory files are located at a single location, but they are distributed across several disks that are NSF-mounted. The same command is used to print a file on any printer and the user doesn’t need to know where the printer in a network is installed.

A Web browser has become a centerpiece of the desktop. The rapidly evolving Web technologies add functionality to this ubiquitous tool, and what is perhaps even more important, new technologies add functionality to the Web servers. This in turn opens new opportunities for the content providers. Nowadays, the Web is not just a collection of static html documents, but also offers numerous services, from on-line shopping and banking to collaboratory environments used for distance training and for sharing scientific data.

The Gateway system offers a specific programming paradigm implemented over a virtual Web-accessible metacomputer. A meta-application is composed of independently developed modules implemented in Java that follow the distributed,
modified JavaBeans model, somewhat similar to the EJB model. This gives the user the complete power of Java and of object-oriented programming in general with which to implement module functionality. However, the functionality of a module does not have to be implemented entirely in Java. Existing applications written in languages other than Java can be easily encapsulated as JavaBeans.

Module developers have only limited knowledge of the system on which the modules will run. They need not concern themselves with issues such as allocating and running the modules on various machines, creating connections among the modules, sending and receiving data across these connections, or running several modules concurrently on one machine. The Gateway system hides these management and coordination functions from the developers, allowing them to concentrate on the modules being developed. In addition to seamless access to modules running across networks, Gateway allows users to construct and work on many different applications composed on several modules concurrently.

Modules often serve as proxies for particular back-end services made available through the Gateway system. For example, an access to a database is provided through JDBC API that delegates the actual implementation of module functionality to a back-end DBMS. We follow a similar approach in providing access to high-performance resources: a Gateway module "merely" implements an API of back-end services.

The Gateway system supports many different programming models for distributed computations, from coarse-grained data flow to object-oriented to a fine-grained, data-parallel model. In the data-flow regime a Gateway application is given by a computational graph visually edited by the end users. The modules comprising the
application exchange data through input and output ports in a way similar to that used in AVS. This model is generalized in our new implementation of the Gateway system. Thanks to the fact that the modules behave as distributed JavaBeans, each module may invoke an arbitrary method of the other modules involved in the computation.

**Gateway** has three-tiered architecture just as **WebFlow** does (Figure 5.1). A stand-alone application or Web browser-based graphical user interface that assists the researcher in the selection of suitable applications, the generation of input data sets, specification of resources, and the post-processing of computational results, comprises tier 1. The distributed, object-oriented middle tier maps the user-task specification onto back-end resources, which form the third tier. In this way we hide the underlying complexities of a heterogeneous computational environment and replace it with a graphical interface through which a user can understand, define, and analyze scientific problems.

In our design we built on our experience of applying the **WebFlow** system to real-life applications, as described in our earlier papers [WebFlowSC98et, WebFlowHPCN99te]. It is important to note that we use the Globus metacomputing toolkit to provide access to high-performance resources in tier 3. Conversely, the Gateway system can be regarded as a high-level, Web-based user interface and job broker for Globus.
5.2 Gateway Middle Tier

5.2.1 Motivation

In an object-oriented approach, applications are made of components and containers. One builds a Java applet by placing AWT components – buttons, labels, text fields and so forth – into frames or panels, that are object containers. This idea can be easily extended to non-graphical components and is implemented as a hybrid of Enterprise JavaBeans and JavaBeans. An important element of the JavaBeans approach is a standardized model for interactions between components through event
notification. Information that is to be shared between components is encapsulated as events and passed to all registered events listeners.

It follows that within the Gateway environment, building a distributed, high-performance application is a process similar to that of building a distributed applet (Figure 5.2). Note that we will use “application,” “context,” and “Gateway context” interchangeably throughout the thesis.

![A distributed Applet](image)

Figure 5.2. A hypothetical distributed applet. Each panel (a container) of this applet is placed on a different host.

Like the distributed applet in Figure 5.2, we have **Gateway Context** and **User Module** corresponding to AWT container and AWT component, respectively, as shown in Figure 5.3 in which an application actually is represented with Gateway Context. A user creates the user context, which is a container for all of his or her applications. The application context holds other application contexts or modules and a user can be represented by Gateway context that can contain an arbitrarily complex hierarchy of
containers and objects in the middle tier. The web-based client tier provides tools for visual composing and for distributing the hierarchy.

![Diagram of a distributed Gateway application]

**Figure 5.3.** A distributed Gateway application

### 5.2.2 GateWay Middle Tier

As shown Figure 5.4, a Gateway middle tier that actually represents the distributed application in Figure 5.3 consists of WEB servers, Gateway contexts, and proxy objects. Even though we showed user modules in Figure 5.4, they actually play a role in the back end. The context residing at the top of tree is referred to as **master context**. Only the master context can hold proxy objects. We call contexts other than the master context **slave context**. Contexts can run as a separate process, called the **Gateway server**, or as an object embedded in another object or **Gateway server**. As shown in Figure 5.4, WEB servers are used to configure and start **Gateway** and they
include descriptions of the user modules. The WEB server holding the master context is called the master WEB server.

![Diagram of Gateway middle tier](image)

**Figure 5.4. A simplified representation of the Gateway middle tier**

As shown in Figure 5.5 (proxy objects and WEB servers removed for simplification), Gateway contexts and user modules are represented as ellipses and squares, respectively. The slave context registers itself to the master if the slave stays just below the master or to another slave and usually runs on a host different than the one on which its parent is running. Usually, there is only one instance of master or slave on each machine. Slaves can be introduced into the system dynamically and their parents instantly update the list of available children of Gateway servers. Actually, master and slave contexts are instances of the same object.
The labels on ellipses and squares indicate the Distinguished Name (DN) of the contexts and user modules, respectively, (Figure 5.5). This naming convention is similar to the way in which LDAP directory servers and the CORBA naming service name their components. The component at the tail of a solid arrow has reference to the one at the head.

![Diagram of Gateway contexts and user modules]

Figure 5.5. Naming of Gateway contexts and user modules

5.2.3 Lifecycle Service

This service, compared with all other OMG specifications, is more like a guideline than a set of standard programming interfaces (although it contains both). One of the key concepts in the Lifecycle service is the object factory, the purpose of which is to create other objects. Created objects might be in the same address space or
they could be remote objects somewhere else in the global computing environment. The CORBA architecture and its location transparency help to hide the complexity associated with these differences in location.

The Gateway requires a customized Lifecycle service. In particular, instantiation of a Gateway module or context on a remote or local host (to be run under control of the peer Gateway server) requires the creation of a local proxy module in the master context. Therefore, there is a proxy object in the master server for any object created in the system. Each context actually has an object factory and is responsible for all lifecycle operations of its children objects: contexts and user modules.

5.2.4 Proxy Objects

The master creates and maintains proxies for each component in the hierarchy. The original purpose of proxies is to forward requests from the Web client to remote objects (a Web client cannot contact objects on remote slave servers because of the Java sandbox security restrictions). In addition, proxies simplify the association of the distributed components. In our current implementation we generate proxies for all components, including local objects. This symmetric implementation allows the functionality of proxies to be extended. Among the most interesting is the capability of logging, tracking, and filtering all messages between components in the system. We use these capabilities to implement fault tolerance and security and transaction monitors, as well as for debugging purposes.

5.2.5 Interactions of user modules

For distributed applications we need a mechanism to transport events across address spaces or a distributed-object model. An application is made of contexts and
modules that exchange information between each other through the event-notification mechanism. An event is a CORBA object itself that encapsulates the data to be sent from one module to another. To make this work, a registration mechanism must be provided that allows for “connecting” the modules. By “connection”, we mean here an association of a source object, which fires the event, and the target object whose registered method is called as the response to the event.

Since Gateway modules are developed independently of each other and are connected only at runtime, we need a dynamic mechanism for event binding. This functionality is offered by the CORBA event service that defines an event-channel object. Event suppliers subscribe to an event channel as event producers, and event consumers subscribe to a channel. This subscription is achieved by first getting ConsumerAdmin (for event supplier) and SupplierAdmin object (for consumers). The event supplier then gets a ProxyPull/PushConsumer object from ConsumerAdmin and starts pumping the event to the channel through this proxy. Event consumers get a ProxyPull/PushSupplier object from ConsumerAdmin and start getting the event from the channel through this proxy, either by pulling the event or pushing the event channel to the event consumer.

Whenever any event comes from any event supplier, the event channel forwards it to all subscribed consumers. There is no event filtering such that one specific type of event is passed to one particular consumer, because the event channel cannot identify the source of the event in order to forward it to one specific consumer. However, we choose not to use these event channels for security reasons. All events in the event channel are “public,” that is, any object can register itself as the listener for an arbitrary
event. The support for point-to-point event exchange will be provided in future releases of CORBA as the event notification service. At this time, we are forced to develop our own event registration service.

Our implementation is based on an event adapter, which is a simple **translation table** maintained by each Gateway context. Each entry of the table contains a source object reference, event identifier, target-object reference, target-method name, and the type of the connection. We support two types of connections, push and pull. In the push model, whenever the source object fires an event, it is intercepted by its parent context, which immediately calls the registered target method. In the pull model, the captured event is kept inside the translation table until the target-object wants to take it by explicitly calling the “pull” method on its parent context. This dynamic event binding is achieved by using CORBA’s dynamic invocation interface (DII) and dynamic skeleton interface (DSI). In Figure 5.6 we illustrate our event model.

In current implementation, all binding information of modules or contexts nested inside one container context are kept inside the container so that whenever it is needed to modify connections from a user module the suitable request must be sent to its parent container. However, it is possible to view this binding table as a CORBA object instead of as internal tables.

This simple configuration (Figure 5.6) shows that we created master (M) and slave (M/S1 and M/S2) servers. Then we created the context (M/S1/UC1) inside slave S1 and the context (M/S2/UC2) inside slave S2. The modules M1 and M2 are put into contexts UC1 and UC1, respectively. Inside the master we created one proxy for each instantiated object. The arrow with dotted lines represents the relationship of the proxy
in the master with the real object. Suppose we associate module M1 and module M2 with one type of event. The arrow with solid lines shows the execution path when module M1 fires one event.

Figure 5.6. Gateway event model.

Module M1 in context UC1 of S1 fires an event e to invoke a method m of module M2 placed in context UC2 of S2. The small blobs inside the master represent the Gateway components’ proxies maintained by the master. Thin dotted arrows show the relation between the proxies and the actual objects.

1. Module M1 fires the event, which is intercepted by the proxy of its parent context PUC1.
2. The proxy forwards it to the actual context UC1.
3. The context finds in its translation table the intended recipient (here, method m of module M2) and forwards the event to the target module proxy PM2.
4. The proxy invokes method \( m \) of module \( M \).

At first, this model may seem unnecessarily complex. The use of proxies indeed adds some overhead. In practice, however, the performance penalty is barely noticeable, while the advantages of this model outweigh any possible shortcomings. First, we use this long path through proxies only to transfer control logic between objects and actual data transfer is carried out at the back end with optimized high-performance libraries such as MPI and PVM. Second, reference to the proxy of the target module instead of to the module itself leads to module location transparency. Third, firing the event by the Web client is the only way in which to access the remote module. Finally, as discussed above, sending events through proxies opens an opportunity for filtering events. These proxies act as general EJB objects when compared to EJB architecture, therefore the proxy for each user module can manage the persistency of the module automatically.

5.3 Gateway Back-End

The back-end consists of user modules and various software and hardware resources. We call these resources **application services**. There are two types of user modules: the **principle module**, which has no dependency on other services, but only on its own computational code, the **delegate module** or **Gateway service**, which serves as a proxy for **application services**. Modules 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 Gateway modules or **delegate modules**.
A typical task submitted to the Gateway system specifies the software components to be used rather than the actual code. Consequently, Gateway modules do not implement the task functionality directly, but instead act as proxies for that functionality. For example, one of the application services, Globus, which has many metacomputing services, provides a user to obtain high performance whenever needed. The ATD (Abstract Task Descriptor) may request to be allowed to submit an executable that can be found at a given location and store the output file at another specified location. In such case, the Gateway delegate module forwards the performing of these tasks to metacomputing services, supplying them with adequate arguments. In other words, the Gateway delegate module implements the metacomputing service interface and marshals the arguments. An application following any programming paradigm (including parallel) and implemented in any language can thus be run under the control of Gateway modules and the Gateway system.

Accessing to a delegate module has two steps. Because each object in the Gateway system has a proxy, a front-end request for any Gateway service is intercepted by its proxy in the middle tier, which forwards the request to the delegate module. In the second step, the delegate module makes forwards one more to the actual application service at back end. A user module running in the back end can request Gateway to obtain any service. This process may cause Gateway to make a traversal on the tree of objects to find specified service if necessary. A user module can register itself for a specific service so that whenever that service is added to the system, the module is informed immediately. In this way, Gateway not only provides an environment for multiple users to work on with their different applications, but also single point service
discovery for user modules. Therefore, a user module can get any service object that is put inside any Gateway context dynamically. Because the hierarchy of contexts and user modules is very similar to LDAP architecture, this feature may open more functionality in the future.

The **LDAP** and **Database** services use LDAP API and JDBC protocol to access to the Directory server and Oracle database, respectively. NT service can access through Windows DCOM to various NT services or COM objects that were developed by the user or by Microsoft.

![Gateway Services Diagram]

Figure 5.7. Gateway services at back end

Globus has various metacomputing services implemented on top of Nexus, and Gateway has built-in **delegate modules** to these services. The services (Figure 5.7)
include secure resource allocation (GRAM), secure file transfer (GASS), metacomputing directory services (MDS), remote I/O for meta-systems (RIO), Metacomputing Directory Service (MDS), and others.

The GASS (Global Access to Secondary Storage) service, using Globus API, helps to transfer files across different machines in a way that eliminates logging onto a remote site and using ftp. The RIO service provides basic mechanisms for tools and applications that require high-performance access to data located in remote potentially parallel file systems. MDS service provides access to MDS, the information infrastructure of the Globus Metacomputing toolkit. MDS stores static and dynamic information about the status of a metacomputing environment. MDS and RIO access back end application services through globus API and LDAP API respectively.

The GRAM service acts as a client to the GRAM (Globus Resource Allocation Manager) server in the Globus system and is used to submit jobs. More specifically, GRAM service sends a request expressed in the Globus RSL (Resource Specification Language) that defines the target machine and the location of executable and input files, as well as instructions for dealing with standard output and standard error streams. Optionally, through GASS 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 Gateway delegate module generates the RSL command, and the Gateway module developer never needs to see the actual application, let alone make an attempt to rewrite the application in Java.
5.4 The Front End

Different classes of applications require different functionality of the front end (Figure 5.1). We therefore designed the Gateway system to support many different front ends, from very flexible authoring tools or problem-solving environments (PSE) that allow for the dynamical creation of meta-applications, from pre-existing modules to highly specialized front ends customized to meet the needs of specific applications (Custom Application GUI). Also, we support many different computational paradigms, from data-flow (Data-Flow Visual Authoring) to general object-oriented (Object-Oriented Visual Authoring Tools) to “a command line” approach (Standalone Application). This flexibility is achieved by treating the front end as a plug-in implementing the Gateway API. We will briefly discuss these front ends.

PSE provides an environment in which a scientist can solve a particular problem by constructing the task through a selection of several subtasks that may be contained in the specified task and the environment and input/output parameters for each subtask. Specifically, for each task the user has to give at least the application executable name with input/output parameters (or files), target host, and specific environment parameters. PSE produces an actual job descriptor from user selections on the fly and sends it to the middle tier.

The user can create an application-specific GUI (Custom Application GUI). We have built two different front ends for LMS simulations.

We may have some applications consisting of components that have simple relationships among themselves such that each component gets several inputs from upward components, runs an actual computational code, and sends any results to
downward components. We have given a simple Web-based interface to QS following the **Data-Flow Visual Authoring** model. We have created several reusable modules where the user clicks and drags the module into a palette in which he makes data-flow connections among modules. This type of front end is suitable only for applications that have inherent data-flow computations.

**Object-Oriented Visual Authoring Tools** are the most generic of front ends and are similar to the JavaBeans Development Environment (like SUN BDK). The user chooses predefined beans and puts them into a bean palette. The user may then make several types of connections between beans after inspecting the properties and fired events of each one. Thus, the user can construct arbitrary connections between beans such that one bean can invoke method of another one when a source fires an event. After completion of an application, it can be saved and resurrected later.

The stand-alone **application** is the lowest level front end because it uses the API of Gateway to build its specific application.

### 5.5 Comparison of Gateway with EJB

EJB defines a server component model for JavaBeans. The current specification of EJB 1.1 doesn’t support attaching Enterprise JavaBeans through event binding. The simplified representation of the EJB model is shown in Figure 5.8. In the EJB model we have a container that provides transaction and security, state management, and the persistency of Enterprise JavaBeans nested inside the container. The container has tools that take Enterprise JavaBeans and produces the necessary helper classes to deploy the EJBs into the container. Usually, in order to create one Enterprise JavaBeans, the user needs to write, for example, an **Account** interface and its implementation, **AccountBean**
(for Session Beans) and AccountHome. Assuming we are using the EJB development environment of the “Acme” company, the container tools get them and automatically produce AcmeAccountHome, AcmeRemoteAccount, and some other classes. AcmeAccountHome has life-cycle methods to implement the user interface, AccountHome. The user first finds AcmeAccountHome in JNDI and uses it to create the AcmeRemoteAccount object, simply an EJB object, which instantiates the real EJB AccountBean at the same time. This EJB object behaves as a wrapper for user EJB, AccountBean. This means that whenever any method call from the user comes to the EJB object, this object performs a transaction operation and security if needed and forwards the call to the real EJB, AccountBean.

In the Gateway system, our Bean is user-module defined in the idl files. Because we are using an interface repository to extract type information from user modules, we have a generic EJB object (AcmeRemoteAccount) called a proxy for any user-defined module. This proxy can get a request from the client for any user module and forward it to the module. In our model we create user modules simply by instantiating an empty constructor of user module implementation, as opposed to “creating” methods of AccountHome implementation classes (AcmeAccountHome). We can again place security and transaction policies into the proxy as EJB does inside the EJB Object.

As EJB tools generate the helper and other wrapper classes from user EJB interface and implementation classes to provide transaction, security, and persistency management of user EJB objects, we automatically generate wrappers for user module attributes. These attribute wrappers consist of the methods necessary to transform
attributes from one type to another, e.g., a string attribute to its native type or vice versa, or from its native representation to the one in a database. These wrappers aid in user module persistency so that whenever one method call crashes for unknown reasons in the middle of its execution, its full state from the database can be recovered automatically. We will say more about persistency in a later chapter.

Figure 5.8. Basics of an EJB environment
CHAPTER 6 – Gateway Interfaces and Services

6.1 Gateway Interfaces

6.1.1 BeanContextChild interface

We started with Sun’s JavaBeans model, used its classes, and customized and updated them for a distributed environment. Later, we wrote our own Gateway interfaces on top of the Sun BeanContext interface and implemented them in CORBA. We will define interface methods and skip the ones used internally.

```java
interface baseAttribute;
Interface BeanContextChild {
    fireEvent(baseAttribute eventObj);
    string saveStateInXml(in string saveCase);
    void restoreXMLProperties();
    void saveStateInDataBase();
    void savePropertiesWithJDBC(in string tableName);

    void setEntityFlag(in boolean flag);
    boolean getEntityFlag();
    void destroy();
    void removeMyself();
    void changeImpl(in string objName);
    membersArray pull();
    void setObjectID(in string objectId);
    string getObjectID();
    void setMyProxy(in Object proxyObj);
    Object getMyProxy();
    void setBeanContextChildPeer(in Object peer) raises(NullPointerException);
    void setBeanContext(in Object bc) raises(event::PropertyVetoException);
    Object getBeanContext();
    Object getBeanContextChildPeer();

    void addPropertyChangeListener(in string name, in Object pcl);
    void removePropertyChangeListener(in string name, in Object pcl);
    void addVetoableChangeListener(in string name, in Object vcl);
    void removeVetoableChangeListener(in string name, in Object vcl);
};
```

Table 6.1. BeanContextChild interface

Our **BeanContextChild** interface (Table 6.1) is an extended version of SUN’s **BeanContextChild** interface with our special methods. We have an implementation
class, **BeanContextChildSupport**, of this interface. Each Gateway module implementation must extend from the **BeanContextChildSupport** class so that the module code is able to access to the **fireEvent** method to fire any event. The Gateway context will use these interface methods to control easily life cycle and run-time information of the user modules.

**saveXMLProperties** saves the current state of a user module object by generating an XML document of its defined attributes and returning it as a result value where **saveCase** argument is the method for storing attributes in two forms, “ASCII” for text form and “binary” for CORBA CDR (Common Data Representation). Currently, we support basic data types and sequence and structure types; **restoreXMLProperties** restores the previous state of the called object in XML. Actually, it reads the values of all the attributes defined for the object and sets them. These method calls are intercepted by the proxy of the object and forwarded to the configuration server.

The **destroy** method disconnect from ORB. The **removeMyself** method removes caller object from the children list of its parent context and removes all of the incoming and outgoing connection entries for this object that are in the binding table of the parent, and finally calls the **destroy** method. The **changeImpl** method first calls the **removeMyself** method and instantiates a new user module with the name **objName** parameter by sending an **addModule** method call to the parent of the object. Finally, **changeImpl** substitutes this new instance into the vacant place of this object.

The **pull** method collects all of the events targeted to this object in an array at a particular time and returns the array. This method should be called inside the
implementation of the user module after the module is attached to another module with the “pull” type of connection as event consumer.

The methods `setObjectID` and `getObjectID` set and get, respectively, the object identifier of the called object (we note that CORBA doesn’t have an exact solution for identifying objects). The methods `setMyProxy` and `getMyProxy` set and get, respectively, the proxy created for the called object. These methods are called internally by other methods such as `addModule` and `addContext` of the `Gateway` interface.

The method `setBeanContextChildPeer` sets the implementation object of this interface to the peer object and is called internally during the instantiation of this implementation; `setBeanContext` sets the parent context of this object to the bc object; `GetBeanContextChildPeer` and `getBeanContext` get the implementation peer of this interface and parent context, respectively.

The methods `addPropertyChangeListener` and `removePropertyChangeListener` add and remove the property listener object pcl with a name for fired property events of the called object, respectively; and the methods `addVetoableChangeListener` and `removeVetoableChangeListener` add and remove the vetoable property listener object vcl with the name for the fired vetoable property events of the called object. These event-adding and removing methods are called automatically on behalf of this object during attaching events.

The `FireEvent` method will be explained later in this chapter, and the `saveStateInXml`, `restoreXMLProperties`, `saveStateInDataBase`, `savePropertiesWithJDBC`, `setEntityFlag`, and `getEntityFlag` methods are explained in Chapter 6.3.
6.1.2 BeanContext interface

| Interface Iterator{                  |
|  Boolean hasNext();                |
|  Object next();                   |
|  Void remove();                   |
|}|
| interface Collection{             |
|  boolean add(in Object o) raises  |
|   (IllegalArgumentException,IllegalStateException,NullPointerException) ;  |
|    boolean addAll(in Collection c) ;  |
|  void clear() ;                   |
|  boolean contains(in Object o) ;  |
|  boolean containsAll(in Collection c) ;  |
|  boolean equals(in Object o) ;    |
|  boolean isEmpty() ;             |
|  Iterator iterator() ;           |
|  Boolean remove(in Object o) ;    |
|  Boolean removeAll(in Collection c) ;  |
|  long size() ;                    |
|  membersArray toArray() ;         |
|}|

Table 6.2. Iterator and Collection interfaces

The Gateway interface (Table 6.3) is the IDL definition of the so-called “Context” or “Container” that holds the other containers or user modules. The implementation of this interface, GatewayContextOps, is instantiated whenever one context is added inside another one through the addContext call of its parent container, or when the outermost context-acting master or slave server is started by hand or by URL. This interface provides nested components with a single point of service discovery and a logical environment for them to live in.
Interface DARP;
Interface GatewayContext:BeanContext, DARP
{
    void recoverDownObjects(in string xmlfileName);
    void setChildDeleted(in string childID);
    boolean isAllChildrenDeleted();
    Object getContext(in string bindingName);
    void deactivate();
    void attachPushProperty(in Object source, in string eventID, //Push Events
        in Object targetObject, in string targetMethod);
    void detachPushProperty(in Object source, in string eventID,
        in Object targetObject, in string targetMethod);
    void attachPushVetoableProperty(in Object source, in string eventID,
        in Object targetObject, in string targetMethod);
    void detachPushVetoableProperty(in Object source, in string eventID,
        in Object targetObject, in string targetMethod);
    void attachPushEvent(in Object source, in string eventID,
        in Object targetObject, in string targetMethod);
    void detachPushEvent(in Object source, in string eventID,
        in Object targetObject, in string targetMethod);
    //We give only attach and detach methods for pull type of events of
generic events // we don’t list the other similar methods for customized property event and
vetoable events.
    void attachPullProperty(in Object source, in string eventID,
        in Object targetObject);
    void detachPullProperty(in Object source, in string eventID,
        in Object targetObject);
    //pull events that came into my mailbox
    membersArray pullEvents() ;
    void propertyChange(in event::PropertyChangeEvent evt)
        raises (event::PropertyVetoException); //Event/property adaptor methods
    void vetoablePropertyChange(in event::PropertyChangeEvent evt)
        raises(event::PropertyVetoException);
    Object addNewModule(in string productName);
    Object addNewContext(in string contextName)
        raises(event::PropertyVetoException,NullPointerException);
    StringArray getModuleList();
    long getMyColor();
    string createObjectID(in string productName);
    //remove module
    void removeModule(in Object source) ;
    void removeContext(in Object source) ;
    void removeLocalChildren();
    void copyModuleBinding(in Object sourceModule,in Object there);
    //Gateway services methods
    Object getService(in string serviceName);
    void addService(in string serviceName);
    void revokeService(in string serviceName);
    void serviceAvailable(in Object serviceEvt);
    void serviceRevoked(in Object serviceEvt);
    boolean hasService(in string serviceName);
    void addBeanContextServicesListener(in Object bcsListener);
    void removeBeanContextServicesListener(in Object bcsListener);}
Table 6.3. GatewayContext interface extending Beancontext and DARP interfaces

As a containment service, the Gateway interface, extending the Collection interface in Table 5.10, includes add/remove methods to add or remove an object, specified in their parameters, to or from the children of the called object. It also includes addAll and removeAll to add and remove a collection of objects, and the clear method to remove all of its children. In addition, it will provide inquiry methods to check whether one object or collection of objects exists as a child with contains and containsAll methods, or whether it has empty children with the isEmpty method. The toArray and iterator methods return the children in an array and in an Iterator object, respectively. We used these methods to implement our high-level methods inside the Gateway interface.

We assign object identifier (objectID) to each object instantiated in the system with method createObjectID that takes the abstract name (relative name) of an object in the productName parameter and returns its ID. For example, if a user adds a module with the name “fileManager” to Context with the objectID “uc_1/uc-2”, then the object ID of this module will be “uc_1/uc_2/fileManager”; getObjectId returns the ID of the called object.

The addNewModule method assigns an object ID and the module with the name productName, specified in IDL definition of user modules as string constant, inserts into this Gateway context. This method creates the proxy object for this new module instance at the Gateway server holding the downloaded front-end applet. In a similar way, the addNewContext method adds the new context inside the called context by instantiating the Gateway object.
**RemoveModule** and **removeContext** remove a module and a context, specified with parameter source from their parent contexts respectively, and disconnect from the CORBA ORB object. These methods also remove the incoming and outgoing connections of the source object, if any, and update the related binding tables. If the removed object is an outermost context, master, or slave server, it will be deactivated or shut-down immediately; **removeContext** also recursively (Depth First Traversal-DFS) visits the children of the object and calls the **removeModule** or **removeContext** method for each one, according to the visited child; **removeMyself** is a general method that will perform the same function as the **addNewModule** or **addNewContext** method, depending on which object **removeMyself** is called on. The **removeLocalChildren** method should be called on a context object, makes DFS traversal over the children, and calls the **removeModule** or **removeContext** method for each one, according to the visited child.
Figure 6.1. The details of how **addNewModule** is executed in the Gateway system

As shown in Figure 6.1, the **addNewModule** method request has an eight-step process. First, the user sends the **addNewModule** request to the proxy object of the context named “uc_1” into which he wants to add a module named “M.” The proxy forwards this call to context “uc_1”, which then instantiates the specified module from Module Factory. The context assigns a unique identifier (objectID) for this created module object, **Mobj**, and returns it to its proxy. The proxy creates a new proxy for **Mobj** and adds it to the children of its real corresponding object, context “uc-1”; **addNewContext** has the same semantics as **addNewModule**, but it instantiates a context object, not a user module.
The `removeModule` operation (Figure 6.2) is the most complicated one of the Gateway methods. As indicated in Figure 6.2, the M1-M2 and M2-M3 connections were established before. First, the user sends a `removeModule` method call with a parameter Module M2 object to the proxy of context UC1. Context UC1 then removes local connections going to or from module M2--currently, there is only an M2-M1 connection. During its `firePropertyChangeEvent` method call, the UC1 removes Module M1 and its proxy, PM1, and calls the `propertyChange` method of parent context from M1, UC0. Actually, UC1 sends the request to the proxy of UC0, PUC0, which forwards it to UC0. Finally, UC0 removes the association entry, M1-M2, from the binding table.

Figure 6.2. Interaction of objects during the removal of Module M2
The **deactivate** method is called internally when **removeContext** encounters a context acting as a server, and will shut down the server. The **copyModuleBinding** first completely copies all incoming and outgoing connections of **sourceModule** to the target object **there**, makes a call **removeModule** method for module **sourceModule**, and puts the object **there** into the resulting vacant position.

There are plenty of event-attaching methods in the Gateway interface for two types of events: **push** and **pull**. Each event category has three subcategories--generic event, property event, and vetoable event. The push type of event has four parameters: the event source object is **source**, the identifier of the event is **eventID**, the event target object is **target**, and the called method of **target** when the event is fired from the **source** object is **targetMethod**. The fired event is encapsulated and automatically delivered in a CORBA object on behalf of the fired object.

For a generic push event, **attachPushProperty** and **removePushProperty** attach and detach event binding, respectively. The property push event has **attachPushProperty** and **detachPushProperty** methods to connect and disconnect this type of event. The vetoable push property has **attachPushVetoableProperty** and **detachPushVetoableProperty** methods for event connection and disconnection.

The attaching (**attachEvent**) and detaching (**attachEvent**) of all types of events can be pictured as seen in Figure 6.3-6.4. As shown in Figure 6.3, to create an event binding from module M1 to M0 for event “propName,” the user sends to the proxy of context UC1 an **attachEvent** request with four parameters, PM0 and PM1 (proxies of M0, and M1), “propName,” and target method of M1, “targetMethod.” Context UC1
removes the association entry of M1-M0 in its binding table and adds itself as property “beanContext” change listener of modules M0 and M1.

**DetachEvent**, as shown in Figure 6.4, is the opposite of the **attachEvent** method call. However, instead of adding itself as a property listener to modules M0 and M1, UC0 removes itself as a listener for Modules M0 and M1 if there are no connections to these modules.

![Figure 6.3. Making an association between Modules M1 and M2.](image-url)
In the pull type of event, an event is not sent automatically to the target but captured and stored in a buffer area of the parent context of the firing user module. Therefore, the event target object has to pull an event explicitly to get it through the pull method call of the BeanContextChild interface. This pull method actually calls the pullEvents method of the Gateway interface; pullEvents first detects the requestor object calling this method and checks any events waiting for this object, collects them in an array and returns it. Because the target method is not needed in this type of event call, we dropped the targetMethod parameter. The other three parameters have the same semantics as the push type of event method.
For a generic pull event, `attachPullProperty` and `removePullProperty` attach and detach event binding, respectively. The property pull event has `attachPullProperty` and `detachPullProperty` methods to connect and disconnect this type of event. The vetoable pull property has `attachPullVetoableProperty` and `detachPullVetoableProperty` methods for event connection and disconnection.

The `getContext` method gives the context object whose object ID is given in the `bindingName` parameter, which must include the full path of this object. For example, in order to find the context with the relative name “myContext” inside the context with ID “uc_1/uc_2/uc_3”, the user has to give “uc_1/uc_2/uc_3/myContext” for `bindingName`.

For persistency, the Gateway interface has only one method, `saveStateInXml`. The way the current state of this object is saved is given by the `saveCase` parameter. The state of any context object includes the properties of objects nested recursively inside this context and the connections among these objects. The input “ascii” of `saveCase` indicates that the value of properties is stored as different XML elements that include them in string type. “Binary” means that properties are saved as binary; `saveStateInXml` makes a DFS traversal over its nested children objects and generates an XML document to save their state.

The user also can save the distributed state with `saveStateInDataBase`, which stores all the related information about the application in the database. (It really makes a traversal on the object tree.) When it encounters a user module, it first saves its attributes such as object ID, CORBA IOR, parent IOR, and some other information with the method call `savePropertiesWithJDBC`. When it finds a context, it will first
save the information about the context object itself, second recursively its children objects consisting of other contexts and modules and third translation table holding outgoing connections from nested modules.

The Gateway system supports a single point of service discovery. We treat service objects as normal user modules. For this, we have several methods: `addService` adds the service with the name `serviceName` into the children’s list of this Gateway object by calling the `addModule` with the parameter `serviceName`. After adding the service object, we call the `serviceAvailable` method of all the listeners registered by the `addBeanContextServicesListener` method call in order to inform them of this new service object.

The `getService` method looks for the service with an absolute relative name specified in `serviceName` and makes a DFS traversal of the whole hierarchical tree from the root node of the master server in order to find this service. If it can’t find any, it instantiates one with the `addService` call and returns the created service. During DFS it will try to find one service whose relative name matches the `serviceName` parameter; `revokeService` is the opposite of the `addService` method, but it will call the `serviceRevoked` method, with the input of the removed service object, of all the listeners and remove these listeners with the `removeBeanContextServicesListener` from their parent context; `hasService` checks whether this service with `serviceName` is available or not.

### 6.1.3 DARP Interface

Two major experiments were our principal accomplishment through our development of DARP and Gateway. Although we succeeded in using two systems together in the
SC97 demo, we were not able to present a clear architecture of how these interact with each other. Because WebFlow doesn’t have the capability of adding DARP functions directly into its middle tier, we will describe the complete architecture of how DARP and Gateway can work together.

As stated in Section 3, we have a **DARP manager** that abstracts a parallel application consisting of multiple nodes into a single node. Through the DARP manager, the user can control a specific processor node independently of other nodes. If looked at carefully, it is not difficult to deduce that the manager functionality should be incorporated into **GatewayContext**’s object behavior; that is, GatewayContext and the DARP manager are merged into a single object. As a result, the manager functionality is implemented as a CORBA object, and thus DARP’s front end will interact with the manager through CORBA requests instead of through TCP/IP protocol with XDR encoding. Similarly, the HPF server gets DARP commands through a combination of **send_deferred** and **poll_response** method calls of the CORBA Object instead of through TCP/IP protocol.

The complete path for sending DARP requests from the front end to the processor nodes is a somewhat complicated process. For a processor pX, the front end calls the DARP method, **putNextCommand**, of the Gateway context object, which stores this request with its parameters inside a tampon buffer, the **requestBuffer**. This buffer is also called as a DARP request buffer that holds requests for each processor. This method call is based on the asynchronous DII methods, **send_deferred** and **poll_response**. An example implementation of the **putNextCommand** of the manager can be like the method in Table 6.4.
/* Put a simple DARP command without parameters for processor pn into request buffer */

public synchronized void putNextCommand(String command, int pn) {
    synchronized(requestTable) {
        Vector list = (Vector) requestTable.elementAt(pn);
        List.add(command); /* append new incoming command to the list of commands for pn */
        RequestTable [pn] = true;
        NotifyAll(); /* make up the sleeping method getNextCommand */
    }
    while(!resultAvailable[pn]){ /* loop back until this DAPR method is processed by processor pn */
        try{
            wait();
        }catch(InterruptedException ex){};
    }
    resultAvailable[pn] = false;
    notifyAll(); /* enable another DARP command */
    /* Get the result from resultTable that came for this method request, 
    return the result in either parameters or as return value of this method depending on this method */
    /* Remember this method is just a simple representative of DARP methods */
}

Table 6.4. A simple representative DARP method

Processor pX calls the asynchronous communication method send_deferred, 
which actually sends the getNextCommand method request to the manager to receive 
the next user command after completing the processing of the previous DARP 
command. A possible realization of the manager method, getNextCommand, is shown 
in Table 6.5.

In TCP/IP-based implementation of DARP, the HPF server checks any DARP 
command before each executable statement of the user code by making a select 
function call. A new, object-based implementation of the HPF server makes a 
poll_response method call to check whether a response for previous DII request 
through send_deferred is available.
Table 6.5. How the DARP server gets the next command from the middle tier

The HPF server gets the DARP command(s) and processes, if any are available, and calls the putNextResult method of the manager to put the previous DARP command result into the resultBuffer of the manager. The resultBuffer, like the DARP request buffer, keeps results and has as much size as clients registered into this manager. A possible implementation of putNextResult can be made as follows:

After the result arrives at the manager and is put into the slot specified in the DARP result table, the current DARP request, waiting for this result, wakes up and gets the result from the resultBuffer. Then the current request returns the result either in parameters or as a return value, depending upon which DARP method is being used. This is a possible solution because of the Java security sandbox problem. If you sign the front-end applet, we can interpose a CORBA event channel between the manager and the applet, which runs a CORBA server, registers the server with the channel, and
gets automatic notification of the command results. All of these invocation sequences are illustrated in Table 6.6.

```java
public synchronized void putNextResult(String result, int pn) {
    synchronized(resultTable) {
        /* find the entry inside resultTable for processor pn */
        Vector list = (Vector)resultTable.elementAt(pn);
        list.add(result); /* append the result of previous command into this entry */
        resultAvailable[pn] = true;
        notifyAll(); /* wake up the client waiting */
    }
}
```

Table 6.6. How the DARP server puts the result of a previous client command into the middle tier

As shown in Figure 6.5, multiple clients can connect and register with a running parallel application organized and orchestrated by the DARP manager, and each client can access the state of application through the DARP data-access methods and steer and control the application through collaboration with each other.

Notice that for each DARP command in the previous TCP/IP-based implementation we have a CORBA method in the WebClowContext interface. All the semantics of the DARP commands are reflected in the matching interface method. Configuration and the starting and stopping of parallel applications are achieved through the DARP manager. SciVis visualization and instrumentation servers can be attached into the Gateway system as a Gateway service. As a result, we have a completely distributed program development environment. Notice also that instrumentation can be performed by directly using YACC and LEX with necessary actions attached to the end of matching grammar productions instead of by using
SAGE++. These actions will produce the instrumented program, inserting HPF server calls and the function calls for registering program variables.

Figure 6.5. How the DARP server, middle tier, and client interact with each other to fully control the distributed execution
Interface DARP

```c
typedef sequence<long> longArray;
typedef sequence<any> anyArray;
typedef struct basicCommandDataDef {
    String commandName;
    Long lineNumber;
    Long pid;
    Long errorCode;
} basicCommandData;
typedef struct dataAccessResultDef {
    basicCommandData basicData;
    longArray dataDesc;
    any dataValue;
} dataAccessResult;
typedef struct protoTypingResultData{
    basicCommandData basicData;
    any dataValue;
} protoTypingResult;

any getNextReqFromManager();
void putPrevReqResult(in any theResult);
// Data access methods
void displayVariable(in long pn,
in string varName, out
dataAccessResult theResult);
void displayArraySection(in long pn,
in string varName,
in string arraySectionStmt,
out dataAccessResult theResult);
void setVariable(in long pn, in string varName,
in any varValueout ,
out basicCommandData theResult);
void setParallelVariable(in long pn,
in string varName, in any varValue
out basicCommandData theResult);
// Prototyping commands
void createInterpretedStmts(in long pn,
in string stmt,
out protoTypingResult theResult);
void interpretStmts(in long pn,
in string functionFromStmts,
out protoTypingResult theResult);
void setSciVisEnv(in long pn,
in long port,
in string ,
out protoTypingResult theResult);
void getSciVisFuncList(in long pn,
in string funcName,
in long lineNumber,
out protoTypingResult theResult);
void deleteSciVisFunc(in long pn,
in string funcName,
in long lineNumber,
out protoTypingResult theResult);
void getFunctionParamNames(in long pn,
in string funcName,
in long lineNumber,
out protoTypingResult theResult);
void storeCurrentConfiguration(in long pn,
in string configFileName,
out protoTypingResult theResult );
void getProfilingInfo(in long pn,
out protoTypingResult theResult);
void getConfigurationFileNames(in long pn,
out protoTypingResult theResult);
void readParseTreeInfo(in long pn,
out protoTypingResult theResult);
// Control commands
void pause(in long pn,
out basicCommandData theResult);
void putBreakOn(in long pn, in long lineNum,
out basicCommandData theResult);
void putBreakOff(in long pn, in long lineNum,
out basicCommandData theResult);
void stepInto(in long pn ,
out basicCommandData theResult);
void stepOver(in long pn ,
out basicCommandData theResult);
void multipleStepInto(in long pn,
in long numOfSteps,
out basicCommandData theResult);
void multipleStepOver(in long pn,
in long numOfSteps,
out basicCommandData theResult);
void multipleLoppIteration(in long pn,
in long numOfSteps,
out basicCommandData theResult);
void multipлеStepInto(in long pn,
in long numOfSteps,
out basicCommandData theResult);
void moveStackDown(in long pn,
out basicCommandData theResult);
void moveStackUp(in long pn,
out basicCommandData theResult);
```

Table 6.7. DARP interface
We briefly describe the DARP methods that correspond to commands in a TCP/IP-based implementation. When a method of the DARP manager is called, it gets the input parameters, constructs CORBA DynAny data from them, and inserts this into the requestBuffer. Each user module running on a processor gets this DynAny through asynchronous calling of the getNextReqFromManager method and extracts information from it, depending on the called DARP method and the process being used. After the user module finishes processing, it constructs special struct data, fills up its member values, and sends the struct to the manager inside any CORBA data by calling the putPrevReqResult method, which, for example, creates struct data of protoTypingResult for prototyping commands. PutPrevReqResult, finally, puts the received result data into the resultBuffer. The wakened-client DARP method pulls this result of CORBA any from the resultBuffer, sets the special struct of this DARP method, and returns.

6.2 Persistency and Configuration Service

We have a prototype of Gateway persistency and configuration model based on XML and database (ORACLE). The user can choose either XML or database, or switch back-and-forth between these two choices (Figure 6.6). Our persistent model is not a general model, but for each module the user can define which attributes should be stored in its IDL files. We use attributes as a way of notifying the state changes of objects to others with a JavaBeans event model. State updates are encapsulated in event objects. We define three types of properties: general property, customized property, and vetoable property.
The user can define the attribute types of his modules by extending from the corresponding base interface in the idl file. The base interfaces of three attributes are shown in Table 6.8. Each extends from the `baseAttribute` interface, which has a group of management methods for attributes sent in events. The `parseXmlValue` method of the `baseAttribute` interface produces an XML definition of the indicated attribute. The `parseSimpleAttribute`, `parseStructAttribute`, and `parseSequenceAttribute` methods take stringified values of the attributes of simple, struct, and sequence types, respectively and convert them to CORBA `ANY` data type. The Gateway middle tier calls these methods to start or save a distributed application description in XML; `setNewValue` and `getNewValue` help to set and get the attribute value in CORBA.
ANY data format. Gateway has an IDLtoXML translator that takes the user attribute definitions in the idl file and produces implementation helper classes for each user-defined attribute.

For example, in defining a user IDL file (Table 6.9) for the timer module below, the user defines three types of attribute: timerEvent, timerProperty, and timerVetoableProperty, which indicate general property, customized property, and vetoable property, respectively, and which are extended from three different interfaces. Each attribute interface has a field of constant string “attributeName,” whose value is used to identify this attribute in the Gateway system. The “value” field holds the actual value of an attribute. As seen, one more level of abstraction for defining user attributes is used to distinguish three types of attribute (Table 6.9), and to assign an abstract name to the attribute in order for the user to access it in his application codes. Once the user has defined the attribute interface, it can be used to define an attribute in the “timer” module interface.

```java
//Base interface for all event types
interface baseAttribute{
    Object getSource();
    void setSource(in Object source);
    string getPropertyName();
    any getNewValue();
    void setNewValue(in any newval);
    any getOldValue();
    void setOldValue(in any newval);
    string parseXmlValue();
    void parseSimpleValue(in string value);
    void parseStructValue(in stringArray values, in longArray dims);
    void parseSequenceValue(in stringArray values, in longArray dims);
}

interface PropertyChangeAttribute: baseAttribute {}; //Property change event
interface VetoableChangeAttribute: baseAttribute {}; //Vetoable property change event
interface GenericAttribute: baseAttribute {}; //Generic event
```

Table 6.8. Base attribute and three different attributes extending from the base
Keeping the module descriptions in an XML document has some benefits. First, the front end will use these descriptions to instantiate modules, make connections between modules, and configure modules. Second, the middle tier mainly uses this document to store and recover the distributed state of modules with their attributes whenever necessary. This makes it possible for the user to feed the initial values of the module attributes and other environmental values through the XML document. Third, it supports the saving/restoring of the distributed configuration that the user created during the application development process. The values of the attributes of running modules can be stored either as binary or text. If stored in text form, the user will have an opportunity to edit and to update the last state of the modules in the XML. Later, the user can incarnate the distributed configuration with these updated values of module properties or connections. A simple module definition in the file “test1.xml” is shown in Table 6.9.
// test1.xml
// an array definition
typedef sequence<long> longArray;

// a structure
typedef struct timerstruct{
    longArray pastValues;
    string currenttime;
} STRUCT_1;

typedef sequence<STRUCT_1> structArray;

// another structure
typedef struct anothertimestruct{
    structArray pastValues;
    double currenttime;
} STRUCT_2;

// for general property
interface timerEvent:event::GenericAttribute{
    const string
    attributeName="timerEventData";
    attribute string value;
};

// vetoable property
interface timerVetoableProperty:
    event:: VetoableChangeAttribute{
    const string
    attributeName = "timerVetoablePropertyData";
    attribute STRUCT_2 value;
};

interface timer_1:BeanContextChild{
    const string
    moduleName="timerModule_1";
    attribute timerEvent genericTimerEvent;
    attribute timerProperty timerPropertyEvent;
    attribute timerVetoableProperty timerVetoableEvent;
    void method_1(in string p1, in long p2);
    void method_2();
};

interface timer_2:BeanContextChild{
    const string
    moduleName="timerModule_2";
    // other user module methods
    void targetMethod(in timerEvent tevent);
};

Table 6.9. An IDL definition file of an example user module.
//store this XML document in file “module_properties.xml”
<UserModule idlFile="test1.idl" componentID="timerModule_1">
  <repositoryID>IDL:Gateway/test/timer_1:1.0</repositoryID>
  <properties>
    <simpleDef attID="timerEvent" eventType="genericEvent">
      <wrapper_interface>IDL:Gateway/test/timerEvent:1.0</wrapper_interface>
      <simple type="string" name="genericTimerEvent">
        <description>this is optional description</description>
        <repositoryID>string</repositoryID>
      </simple>
    </simpleDef>
    <structDef attID="timerProperty" eventType="propertyEvent">
      <wrapper_interface>IDL:Gateway/test/timerProperty:1.0</wrapper_interface>
      <struct type="struct" name="timerPropertyEvent">
        <description>this is optional description</description>
        <repositoryID>IDL:Gateway/test/timerstruct:1.0</repositoryID>
        <sequence type="sequence&lt;long&gt;" name="pastValues">
          <description>this is optional description</description>
          <repositoryID>IDL:Gateway/test/longArray:1.0</repositoryID>
        </sequence>
        <simple type="string" name="currenttime">
          <description>this is optional description</description>
        </simple>
      </struct>
    </structDef>
    <structDef attID="timerVetoableProperty" eventType="vetoableEvent">
      <wrapper_interface>IDL:Gateway/test/timerVetoableProperty:1.0</wrapper_interface>
      <struct type="struct" name="timerVetoableEvent">
        <description>this is optional description</description>
        <repositoryID>IDL:Gateway/test/anothertimerstruct:1.0</repositoryID>
        <sequence type="sequence&lt;alias&gt;" name="pastValues">
          <description>this is optional description</description>
          <repositoryID>IDL:Gateway/test/structArray:1.0</repositoryID>
        </sequence>
        <simple type="double" name="currenttime">
          <description>this is optional description</description>
        </simple>
      </struct>
    </structDef>
  </properties>
  <methodList>
    <method name="method_1" returnType="void">
      <parameter name="p1" type="string"/>
      <parameter name="p2" type="long"/>
    </method>
    <method name="method_2" returnType="void"/>
  </methodList>
</UserModule>
<UserModule idlFile="test1.idl" componentID="timerModule_2">
  <repositoryID>IDL:Gateway/test/timer_2:1.0</repositoryID>
  <methodList>
    <method name="targetMethod" returnType="void">
      <parameter name="tevent" type="IDL:Gateway/test/timerEvent:1.0"/>
    </method>
  </methodList>
</UserModule>

Table 6.10. XML description of user IDL file in Table 6.9
IDLtoXML translates user module IDL definitions with their attribute types to XML documents. Using this translator, we can generate module descriptions in the XML document from the information of the user module IDL files, as is done in Table 6.10 for the user module definition of IDL interfaces in Table 6.9. As seen in the above translation, we currently support basic types (integer and string, etc.), array, and structure types. IDLtoXML also produces helper classes for user-defined attributes, as explained above. We show the definition of two methods of helper class, generated automatically by IDLtoXML (Table 6.11) for the timerVetoableProperty of the user module in Table 6.9.

A user can use the system in two ways: either through direct middle tier API or through the job descriptions in an XML document file (Figure 6.6). It is possible to make a transformation among three different environments: a standalone Gateway application, XML space, and a database with an integrated collection of Gateway objects. The user can create a distributed application through direct use of Gateway API or describe his application in XML and submit it to the middle tier which parses it and makes it run. During the course of the application’s run, he may request the middle tier to translate the distributed state to an XML document or to database tables. The user also can store the XML version of the application into the database. If the user is working on continuous running sessions, after getting each intermediate result, the state can be converted into XML. The result values can be inspected and modified, and another session started with new initial parameter values.
We will give one simple example of using API. We can create the indicated connection (Figure 5.6) between modules M1 and M2, including instantiation of them and their contexts with Gateway.

API calls are shown in Table 6.12. Assume that master M and slaves S1 and S2 were started manually or through URL, and also assume that modules M1 and M2 represent two modules, timer_1 and timer_2, as defined in the IDL file shown in Table 6.9.
public void parseStructValue(String []values, int []dims){
    int next = 0;
    WebFlow.translator_exam1.anothertimerstruct attValue;
    attValue = new WebFlow.translator_exam1.anothertimerstruct();
    attValue.pastValues = new WebFlow.translator_exam1.timerstruct[dims[0]];
    for(int i0=0;i0< dims[0];i0++)
    {
        attValue.pastValues[i0].pastValues = new int[dims[1]];
        for(int i1=0;i1< dims[1];i1++)
        {
            int t_3 = Integer.parseInt(values[next++]);
            attValue.pastValues[i0].pastValues[i1] = t_3;
        }
        String t_4 = values[next++];
        attValue.pastValues[i0].currenttime = t_4;
    }
    double t_5 = Double.parseDouble(values[next++]);
    attValue.currenttime = t_5;
    value(attValue);
}

public String parseXmlValue(){
    String xmls = "";
    xmls += "    <propertyVal attrRef="timerVetoablePropertyData">
    WebFlow.translator_exam1.anothertimerstruct attValue = newValue;
    xmls += "      <structVal>
    xmls += "        <sequenceVal>
    for(int i0=0;i0< attValue.pastValues.length;i0++){
        xmls += "          <structVal>
    xmls += "            <sequenceVal>
    for(int i1=0;i1< attValue.pastValues[i0].pastValues.length;i1++){
        xmls += "              <simpleVal>"+String.valueOf(attValue.pastValues[i0].pastValues[i1])+"</simpleVal>
    }
    xmls += "        </sequenceVal>
    xmls += "    </structVal>
    xmls += "    </propertyVal>
    return xmls;
}

Table 6.11. The definition of two methods for the Helper class of user attributes

After the user constructs a simple, distributed configuration of Figure 5.6, the complete current state in XML can be saved by sending the saveStateInXml method call to the master server. In the opposite direction, the user may assign new values of attributes in the XML document by using the restoreXMLProperties method. If the
user initialized all of the attributes of timer_1 and timer_2 modules (Table 6.9), we can automatically get the XML document in Table 6.13 from the middle tier. Later, the user can restore the old configuration in Figure 5.6 simply by sending the XML document (Table 6.13). Thus the user has the flexibility of choosing API-method calls or specifying that the entire abstract job be in XML in order to construct his application. One of the greatest benefits of saving the state in XML is that the user can later inspect the stored modules’ attributes and update them for the next iterations of the run if he chose the strategy of saving attributes as ASCII. If the user doesn’t want to inspect the saved attributes, they can be stored as binary in an XML document. Notice that the attributes in the XML file shown in Table 6.13 were given random values in user-module code and saved as ASCII.

```java
org.omg.CORBA.Object s1_obj, s2_obj,uc1_obj, uc2_obj,m1_obj,m2_obj;
GatewayContext uc1, uc2,s1,s2;
//get IOR string from fixed url
String ref = getIORFromURL(masterURL);
//convert IOR to corba object
org.omg.CORBA.Object obj = orb.string_to_object(ref);
//narrow corba Object to specific corba server type
GatewayContext master = GatewayContextHelper.narrow(obj);
//get the proxy reference of slave server S1
s1_obj = master.getContext("M/S1");
s1 = GatewayContextHelper.narrow(s1_obj);
//add new user context with name “uc1”
uc1_obj = s1.addNewContext("uc1");
//narrow it to WebFowContext
uc1 = GatewayContextHelper.narrow(uc1_obj);
m1_obj = uc1.addNewModule("timerModule_1");
//get the proxy reference of slave server S2
s2_obj = master.getContext("M/S2");
s2 = GatewayContextHelper.narrow(s2_obj);
//add new user context with name “uc2”
uc2_obj = s2.addNewContext("uc2");
//narrow it to WebFowContext
uc2 = GatewayContextHelper.narrow(uc2_obj);
m2_obj = uc1.addNewModule("timerModule_2");
//Finally make "push" type of connection between modules M1 and M2. Make sure you issue the
"attachPushEvent" //command to the parent of event source where M1 is the source of event.
uc1.attachPushEvent(m1_obj," timerEventData",m2_obj,"targetMethod");
```

Table 6.12. These Gateway API calls construct the configuration in Figure 5.6
Table 6.13. This XML document is saved if the user issues a `saveStateInXML` request to the master server. In addition, the user can construct the configuration in Figure 5.6 with this document.

In order for the job (Table 6.13) to start, all specified Gateway servers have to be started manually or in another way such as through URL. In order to start a Gateway server automatically if it is down during the instantiation of the Gateway
context and the user modules represented in XML, we have another way of configuring entire distributed application that includes Gateway servers. In Table 6.13 we always describe modules before their instantiation in the document. If we are using module declarations in several places, this will cause multiple module descriptions in different XML documents. To solve this problem, we benefited from the XML ENTITY element with which we could put one module description in an URL-addressable location and point to that place every time we used it.

```xml
<?xml version="1.0"?>
<!DOCTYPE distributed_application SYSTEM "webflow.dtd" [ 
]>
<distributed_application>
  <WebFlowContext componentID="ntserver"
    entityFlag="no">
    //insert module declaration sitting another WEB site
    &timer-master-decls;
    <WebFlowContext componentID="uc_1" entityFlag="no">
      <WebFlowContext componentID="uc_2" entityFlag="no">
        <WebFlowContext componentID="uc_3" entityFlag="no">
          <ModuleInstance>
            <componentRef>helloModule_1</componentRef>
            <moduleID>ntserver/uc_1/uc_2/uc_3/helloModule_1</moduleID>
            //Here skipped for clarity necessary module instantiations with their attributes
            //and connections among them
          </ModuleInstance>
        </WebFlowContext>
      </WebFlowContext>
    </WebFlowContext>
  </WebFlowContext>
  <WebFlowContext componentID="mickey"
    entityFlag="no">
    //insert module declaration sitting another WEB site
    &timer-master-decls;
    //Do some module instantiations and connections among them
  </WebFlowContext>
</distributed_application>
```

Table 6.17. Solving multiple declarations of modules with the XML ENTITY element
For example, as shown in Table 6.17, we have two module declarations staying at two different WEB locations and we have declared no modules but put a pointer to their declaration locations. Notice also that for Gateway contexts running as separate processes—Gateway server, we have the servlet_href attribute whose value tells the URL address of the installer servlet for this Gateway server. For this type of configuration, we must make all of the servlets specified under servlet_href it running. When the Gateway middle tier parses the application in a XML document and finds a context that has a servlet_href attribute, it will simply invoke that servlet, which in turn starts the specified Gateway server and returns its IOR. The user will then start any number of WEB servers only one time. Later he can start and configure an arbitrarily distributed application in a tremendously efficient and simple way, which proves the simplicity of configuring and maintaining the Gateway system.

6.3 Gateway Fault Tolerance Model

Gateway has simple fault tolerance only at the time a proxy at the root of the Gateway server or real remote user module is down. This model is not able to solve a problem when a communication link between interacting objects is down. It will use a combination of the resources of a Monitor Servlet, WEB servers, Database, and CORBA Interface Repository to fix a problem that may occur when the user-distributed application is running. We will briefly explain how this process takes place.

Two principal problems may occur in the Gateway system: Either a module proxy living in the root of the Gateway server or in the remote module object may be down. We need to consider the worst possible scenario; which is that multiple proxies
and remote modules can go wrong simultaneously, in which case the recovery algorithm shouldn’t fail.

First of all, the user, who wants to roll back a module when a fault occurs, has to set it as an entity module by calling `setEntityFlag`. The user normally sets the entity flag when it is created and added into a context. Only modules having entity flags set will be recovered. With the entity flag set, the state of a user module is stored across its method invocations with the `savePropertiesWithJDBC` method call.

![Diagram](image)

**Figure 6.7. Recovering proxy module in root Gateway server**

As shown in Figure 6.7, it is deduced the a proxy object is down when the client tries to access to a remote module or when another module wants to send an event to the remote module corresponding to that proxy. Actually, that event is intercepted by
the proxy and forwarded to the remote object when the normal case continues. Transpare
tly from the user, the module that sends an event query the database to find
the previous state of the session and creates a new proxy for the remote module. The
user must invoke a servlet without any parameters when he discovers a proxy is down.

Because in Gateway's current architecture all proxy objects are created in the
master Gateway server, recovery from the state of the master server being down is more
complicated than that described above. This state is detected when a client tries to make
a call for one module. When the host for the master server is down, we may face
single-point failure. If this is the case, we can try to configure the master server at a
different active host. It is possible to distribute proxy objects across different hosts, but
this configuration will not provide a solution for the Java sandbox problem.

In the second scenario (Figure 6.8), a remote module is discovered to be down
when the proxy of this remote object tries to call one of its methods. The user faces this
condition when trying to call the method of a remote object or when another proxy tries
to send an event to that same remote object. The proxy gets an exception showing that
the remote object is down and it looks for the previous state in the database and
recreates the remote object and re-sends the previous method call that was interrupted
because of the remote object’s failure.
Figure 6.8. Recovering a remote user module
6.4 Gateway Security Access

The Gateway middle tier is given by a tree of CORBA-based Gateway servers, one of which serves as a **gatekeeper** (Figure 6.9). The **gatekeeper** comprises three logical components: a (secure) Web Server, the AKENTI server, and the CORBA-based Gateway server. The user accesses the Gateway system through a portal web page from the gatekeeper’s web server. The portal implements the first layer of gateway security, the authentication and generation of credentials that eventually will be used to grant the user access to resources. The authorization process is controlled by the AKENTI server. For each authorized user, the web server creates a session (that is, it instantiates the user context in the Gateway server, as described below) and gives permission to download the front-end applet that is used to create or restore, run, and control user applications. The applet communicates directly with the CORBA-based...
Gateway server using IIOP protocol. We currently use secure ApacheSSL and JigsawSSL commodity web servers.

The Gateway system supports a three-layer security model (Figure 6.10). The first layer is responsible for secure web access to the system and for establishing the user’s identity and credentials. The second layer enforces secure interactions between distributed objects, including communications between peer Gateway servers and the delegation of credentials. The third layer controls access to back-end resources. In each case we follow industry standards or participate in the creation of standards.

Security Model (Keberos)

Figure 6.10. Gateway Security Model
6.4.1 First Security Layer: Secure Web Transactions

To implement secure web transactions we use industry-standard https protocol and commodity-secure web servers. The server is configured to mandate a mutual authentication. To make a connection, the user must accept the server’s X.509 certificate (both Netscape and Internet Explorer web browsers support this feature) and must present his or her certificate to the server. A commercial software package (Netscape’s certificate server) is used to generate the user certificates, and they are signed by the Gateway certificate authority (CA). Since the certificates are generated in PKCS#12 format and are supported by Netscape and Internet Explorer web browsers, they are handled transparently by the browser.

The authorization process is controlled by the AKENTI server [AKENTIWeb], which provides a way to express and to enforce an access policy without requiring a central enforcer and administrative authority. Its architecture is optimized to support security services in distributed network environments.

This component of security services provides access for authorized users only to the Gateway server associated with the gatekeeper, following policies defined in AKENTI (and thus representing the stakeholders’ interests). Access to peer Gateway servers and to back-end services is controlled independently by level-two and level-three gateway security services based on credentials generated during the initial contact with the gatekeeper.

6.4.2 Second Security Layer

Security features of CORBA are built directly into ORB and are therefore very easy to use. Once the user’s credentials are established, secure operations on distributed
objects are enforced transparently. This includes authorized use of objects, and optional per-message security (in terms of integrity, confidentiality, and mutual authentication).

Access control is based on the access control lists (ACL), which provide the means to define policies at different granularity from an individual user to groups defined by a role, and from a particular method of a particular object to computational domains. In particular, the role of a user can be assigned according to policies defined in AKENTI. In this way, access to the distributed objects can be controlled by the stakeholders.

In addition, for security-aware applications, the CORBA security service provides access to the user’s credentials, thus allowing access to the back-end resources to be controlled by the owners of the resources and not by the Gateway system, which merely forwards the credentials.

The CORBA security service is defined as an interface, and the OMG specification is neutral with respect to the actual security technology to be used, which can be implemented on top of PKI technologies (such as SSL), private key technologies (such as Keberos), or GSS-API, to mention the most popular ones.

Distributed objects are inherently less secure than traditional client-server systems. An enhanced risk level comes, among other factors, from the fact that objects often delegate parts of their implementation to other objects (which may be dynamically composed at runtime), thus allowing objects to serve simultaneously as both clients and servers. Because of subclassing, the implementation of an object may change over time. The original programmer neither knows nor cares about the changes. The policy of privilege delegation is therefore a very important element of system
security. CORBA is very flexible here, and supports no delegation model (the intermediary object uses its own credentials), a simple delegation model (the intermediary object impersonate the client), and a composite delegation (the intermediary object may combine its own privileges with those of the client). We follow the composite model. For security-unaware applications, we use the intersection of the client and intermediary privileges. However, if the application applies its own security measures, we make the initiator credentials available to it.

6.4.3 Second Security Layer: Control of Access to Back End Resources

There are no widely accepted standards for secure access to resources. Different computing centers apply different technologies: SSH, SSL, Keberos5, or others. The design goal of the Gateway system is to preserve the autonomy of the owner of the resources to define and implement security policies. In this respect, we are in a situation similar to that of other research groups that try to provide a secure access to remote resources. Our strategy is to participate in the process of defining standards within DATORR and the common Alliance PKI infrastructure. It seems that the current preference is to build future standards on top of the GSS-API specification (and thus to simultaneously support private and public key-based technologies). The Globus project pioneered this approach, and we therefore use Globus GRAM to provide secure access to remote resources. To get access to resources available via GRAM the user must present a certificate signed by the Globus CA (currently an additional item of the Gateway user set of credentials).

6.5 Consequences of This Distributed Model

The model offers a simple interface to complete the middle tier so that the user will use the API of the GatewayContext object, both to manage the life cycle of the
nested user modules and to establish an association among the modules. In addition, *GatewayContext* will be used to control and steer user modules at a fine-grained level. This indicates that the complete architecture will provide both fine-grained and course-grained monitoring, and the tuning of applications consists of single or multiple user modules. We have arrived at what we proposed at the beginning of this thesis. Services like databases, visualization, and instrumentation can be attached to any context so that any module in any context can access this service transparently. This transparency is achieved by interpreting an entire meta-application as a single unit, which is a hierarchical tree that composes a heterogeneous application. After the user sets up the application and initiates it, the application may be monitored at both the message level and at the lowest level, statement by statement. A generated proxy for each user module or context performs message-level steering. Line-by-line debugging is achieved through DARP methods of *GatewayContext*. Notice that we have a prototype system that combines distributed-object-level computing and parallel computing with simple, high-level debugger functionality.

Managing and establishing a distributed and parallel system is performed by a high-level facility, CORBA, but high performance is satisfied with the help of commodity parallel libraries, MPI, PVM or Globus whenever needed.
CHAPTER 7 Gateway Applications

We have applied Gateway for two projects: LMS (Landscape Modeling System) and QS (Quantum Simulation). I developed the Gateway infrastructure and the middle tier backbone, and Tomasz Haupt was the principal coordinator in applying it for LMS and QS. However, I also participated in these applications by developing several modules, including the File Browser and File Transfer Progress modules.

7.1 LMS

7.1.1 Description of the Project

The LMS project was sponsored by the U.S. Army Corps of Engineers, Waterways Experiment Station (CEWES), Major Shared Resource Center (MSRC) at Vicksburg, MS, 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 period caused by some short-term disturbance such as fire or human activities. A critical parameter of the vegetation model is the conditioning of soil at the time of the disturbance. This, in turn, is dominated possibly by rainfall, at some time. Consequently, the implementation of this project requires:

- Data retrieval from remote sources, including DEM (data elevation models) data, land use maps, soil textures, and dominant flora species and their growing characteristics, to name a few. The data are available from many sources,
including public services such as USGS web servers and proprietary databases. The data come in different formats and with different spatial resolutions. Without WebFlow, the data must be manually prefetched.

- Data preprocessing to prune and convert the raw data to a format expected by the simulation software. This preprocessing is performed interactively using the WMS [WMSWeb] (Watershed Modeling System) package.

- Execution of two simulation programs EDYS [WMSWeb] for vegetation simulation, including disturbances, and CASC2D [WMSWeb] for watershed simulations during rainfalls. The latter results in the generating of maps of the soil condition after the rainfall. The initial conditions for CASC2D are set by EDYS just before the rainfall event, and the output of CASC2D after the event is used to update the parameters of EDYS. We used test data sets for time periods with at least two rainfalls. Consequently, the data transfer between the two codes had to be performed several times during one simulation. EDYS is not CPU-demanding, and it is implemented only for Windows95/98/NT systems. On the other hand, CASC2D is very computationally intensive and is typically run on powerful UNIX compute servers.

- Visualization of the results of the simulation. Again, WMS is used for this purpose.
The purpose of this project was to demonstrate the feasibility of implementing a system that would allow the complete simulation to be launched and controlled 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, either at CEWES in Vicksburg, MS, or at NPAC in Syracuse, NY. We demonstrated the system using a Pentium II-based laptop running Windows NT located in Washington, DC, (with CASC2D running at Vicksburg) and in Orlando, FL, during Supercomputing ’98 (with CASC2D running alternatively at Syracuse and Vicksburg).
7.1.2 Interaction between Casc2d and Edys simulations

The casc2d [CASCD] and Edys [EDYS] 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 concentrates 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, averaged precipitation data is used 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 which 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. Figure 7.2). The simulation starts 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 continues 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 simulation 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, casc2d exits, and the final run of Edys is performed. The run terminates at a predefined date, typically 20 years after the first rainfall.

Figure 7.2. 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.

* Ten seconds is a negligibly short time as compared to the average time required to complete one casc2d iteration, which is about 15 minutes on an SGI O2 workstation.
7.1.3 LMS Middle Tier

This application requires two computational modules: one encapsulating EDYS to be run on a 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 (Figure 7.3) because we run the client application there. In addition, we run Web servers on both machines which are needed primarily to exchange data between the modules. We also publish the master IOR on the Web server on the WindowsNT side.
The servers can be started manually by feeding the different configuration files into XML documents or through servlet calls to the Web servers on which the WebFlow servers live.

The servers can be accessed by WebFlow API calls that are part of the LMS front end. With the master and slave servers started before, the code segment in Figure 7.4 creates the state in Figure 7.3. 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: WebFlowContext (line 5). Now it can call methods of this object. In lines 9 and 10 it uses the method getContxt(serverName) to retrieve references to both slave servers. It adds module “runEdys” to the nts Server 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, and event “casc2dDone,” fired by runCasc2d, will invoke method “receiveData()” of runEdys. Now everything is ready to start the execution, which is triggered by invoking method run() of module runCasc2d.

Instead of API calls (Figure 7.4), the application can be defined by a static XML document (Table 7.1), available from a Web server, and instantiated dynamically at runtime. Moreover, the modification of the application will be possible just by editing the XML file without introducing any changes to the code.

Notice that to prepare Table 7.1, the user first passes the module IDL interface file through our IDL-to-XML translator and gets a “properties.xml” XML document
which he should include in preparing the XML configuration file. The user manually
starts only the master server that starts the other slave servers. The master parses the
document (Table 7.1) and starts any slave server specified by the WebFlowContext
XML element, if necessary, by making a URL request to the address given by the
`<servlet_href>` attribute.

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, which 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`.

```java
1. ORB orb = ORB.init(args, new java.util.Properties());
2. String masterURL = args[0];
3. String ref = getIORFromURL(masterURL);
4. org.omg.CORBA.Object obj = orb.string_to_object(ref);
5. WebFlowContext master = WebFlowContextHelper.narrow(obj);
6. WebFlowContext slave;
7. try {
8. org.omg.CORBA.Object p1, p2;
9. slave1 = WebFlowContextHelper.narrow(master.getContext("master/ntserver"));
10. slave2 = WebFlowContextHelper.narrow(master.getContext("master/osprey4"));
11. p1 = slave1.addNewModule("runEdys");
12. p2 = slave1.addNewModule("runCasc2d");
13. runCasc2 = runCasc2dHelper(p2);
14. master.attachPushEvent(p1,"EdysDone", p2,"runAgain");
15. master.attachPushEvent(p2,"Casc2dDone", p1,"receiveData");
16. rc.run();
17. } catch(Exception e) {}
```

Figure 7.4. WebFlow API calls to construct the configuration in the Figure 7.3
Table 7.1. Abstract job specification in XML document for configuration in Figure 7.3
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.

The RunEdys module is triggered by the Casc2dDone event that invokes the receiveData() method. First, a file options.txt is generated. This file defines the 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. This step could be avoided by 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.

7.1.4 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 the Java Runtime class to run the
WebFlow modules on the same host on which the WebFlow server runs. As discussed in Section 7.3, we are prepared to provide a secure access to remote, high-performance resources when needed.

### 7.1.5 LMS Front End

For this project we developed a custom front end implemented as a Java application (as opposed to Web-accessible Java applets). There are several reasons for that, one of which 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. Thus it does not really matter if the extensions to WMS are downloaded as applets each time the LMS is run, or downloaded once and stored permanently on the client machine.

![LMS front-end main panel](image)

**Figure 7.5.** LMS front-end main panel

The WMS program (Watershed Modeling System) is a rich collection of tools for pre- and post-processing data. 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 varieties 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 (Figure 7.5).

7.1.6 Data Wizard

The data wizard panel (Figure 7.8) 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.
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.

### 7.1.9 WMS

The WMS button on the main panel starts the WMS program on the local host by the using the Java Runtime class in a separate thread. The WMS controls are available during the entire LMS session.
Figure 7.6. A screen-dump of a WMS session: in the central window just downloaded DEM data are displayed. The raw data must be pre-processed now, including selecting a watershed region, smoothing, and format conversion.

7.1.9 Simulations

The functionality of this part can be deduced from the above. Figure 7.7 shows the front-end panel that is used to start the simulations.
As shown in Figure 7.7, 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 selects 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.

7.2 Quantum Simulation (QS)

As a test application for WebFlow, we selected Quantum Simulations [QSWeb]. The motivation of designing QS algorithms arises from considerations of the computational
complexity of solving the Schrödinger equation for systems of many electrons, resulting in calculating electronic structure, as opposed to the methods of expanding the wave function in a basis. By complexity, we mean simply the computation time of some property of a system to some specified absolute error that must be a “true” error as a combination of the system and statistical errors. In order to get errors accurately, we couldn’t use methods such as the density functional theory within the local density approximation (LDA) or the generalized gradient approximation (GGA). We have to use algorithms based on a complete representation of many-body wave functions that have exponential computation time in the number of electrons. One example algorithm, the configuration interaction (CI) expands the wave function in Slater determinants of one-body orbitals. Whenever an atom is added to the system, an additional number of molecular orbitals must be considered and the total number of determinants to reach chemical accuracy is then multiplied by this factor. As a result, we got an exponential running time in terms of the number of electrons. Quantum simulation constructs the wave equation (or N-body density function) by sampling it and therefore it doesn’t need its value everywhere. The complexity then usually has some power (1-4) of the number of particles. Therefore, we have to use parallel machines to solve QS by evenly distributing the needed computational load across processors.

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 parameters. The output file of one application in the chain is the input of the next one, after a suitable format conversion. The logical structure of the application is shown in Figure 7.8

![Figure 7.8. Logical structure of the Quantum Simulation application](image)

This example meta-application demonstrates the strength of our WebFlow approach. The WebFlow editor provides an intuitive environment to visually compose the chain of data-flow computations from preexisting modules. The modules encapsulate many different classes of applications, from massively parallel to custom-developed auxiliary programs to commodity commercial ones (such as DBMS or visualization packages). The seamless integration of such heterogeneous software components is achieved by employing distributed-objects technologies in the middle tier. The high-performance part of the back-end tier is implemented using the
GLOBUS toolkit. In particular, we use MDS (metacomputing directory services) to identify resources, GRAM (globus resource allocation manager) to allocate resources that include mutual, SSL-based authentication, and GASS (global access to secondary storage) for a high-performance data transfer. The high-performance part of the back-end is augmented with a commodity DBMS (servicing Permanent Object Manager) and LDAP-based custom directory service to maintain geographically distributed data files generated by the Quantum Simulation project.

The diagram illustrating the WebFlow implementation of the Quantum Simulation is shown in Figure 7.9.

Figure 7.9. WebFlow implementation of the Quantum Simulations problem

WebFlow can be applied to many different applications. At Supercomputing '97 [DARPSC97] we demonstrated two other applications, one of which was an AVS-like [AVSWeb] image processing. Here we took advantage of our platform-independent
design (implemented in Java) to integrate computations running on both UNIX and Windows-NT systems. The other application employed HPF-based back end. More specifically, we demonstrated the DARP system [DARP98ACMJava], an integrated environment for compiled and interpreted HPF wrapped as a WebFlow module. A typical WebFlow visual graph of an Quantum Simulation application is shown in Figure 7.10.

Figure 7.10. Example WebFlow Session of QS
CHAPTER 8 - Conclusions

We have developed a platform-independent, three-tiered system with visual authoring tools implemented in the front end and integrated with a middle-tier network of servers. It is based on industry standards and follows a distributed-object paradigm, facilitating a seamless integration of commodity software components. In particular, we used Gateway as a high-level, visual user interface for GLOBUS, which not only makes the construction of a meta-application much easier for an end user, but also allows this state-of-the-art HPCC environment to be combined with commercial software that includes packages available only on Intel-based personal computers.

We used Gateway to provide a seamless access to remote resources for the two applications that required it. For LMS we used Gateway to retrieve data from many different sources as well as to allocate the remote computational resources needed to solve the problem at hand. Gateway transparently controls the necessary data transfer between hosts. Quantum Simulations requires access to HPCC resources and we therefore layered Gateway on top of the Globus metacomputing toolkit. We admit that Gateway does not yet comprise a complete solution for seamless access to remote resources, and many issues --notably security-- still remain to be solved. We expect to leverage from the recent DATORR (Desktop Access to Remote Resources) an initiative of the Java Grande Forum when tackling these issues.

Exploiting our experience in developing the WebFlow system, we designed and implemented a new system, Gateway, to provide seamless and secure access to computational resources at ASC MSRC. While preserving the original three-tiered architecture, we re-engineered the implementation of each tier in order to conform
strictly to the standards. In particular, we used CORBA and the Enterprise JavaBeans model to build the new middle tier, which facilitates the seamless integration of commodity software components. Database connectivity is a typical example of a commodity software component. However, the most distinct feature of the Gateway system is that we apply the same commodity-components strategy to incorporate HPCC systems into Gateway architecture. By implementing the emerging standard interface for metacomputing services, as defined by DATORR, we provide a uniform and secure access to high-performance resources. Similarly, by conforming to the Abstract Task Descriptor specification we enable seamless integration of many different front-end, visual-authoring tools.

In addition to composing meta-application with our visual tools, we provided fine-grained control on each component, written in HPF, of a meta-application with DARP, which we successfully incorporated into the Gateway system. By reusing commodity components and technologies we have built this powerful tool, DARP, for data analysis and rapid prototyping to be used by an HPF application developer. The most important feature of the system is interactive access to distributed data, which, in turn, makes it possible to select and send data to a visualization system at an arbitrary point of the application execution. Also, data can be modified using either native HPF commands or dynamically linked computational modules. If the user spots a place in an application, he can stop the method execution and go into details by using DARP control and prototyping commands.

Consistently with our HPcc strategy, the DARP system implements a three-tiered architecture: The Java front end holds proxy objects produced by an HPF front
end operating on the back-end code. These proxy objects can be manipulated with an
interpreted Web client interacting dynamically with compiled code through a typical
tier-2 server (middleware). Although targeted for an HPF back end, the system's
architecture is independent of the back-end language and can be extended to support
other high-performance languages such as HPC++ or HPJava by replacing the
corresponding parser for target language with an HPF parser. Finally, since we
followed a distributed-objects approach, the DARP system can easily be incorporated
into a collaboratory environment such as Tango or Habanero. There are other solutions
for client-side collaboration that will be explained in Section 8.1.

By adopting the architecture of commodity systems, we have made it easier to
track their rapid evolution and we expect this will give high functionality to HPCC
systems. Whenever high performance is needed, we delegate that task to the high-
performance back end with Globus. As basic commodity technology evolves from the
servicelet and socket-based framework of WebFlow to the “Distributed Object
Technology” architecture of Gateway, we preserve a high performance. We strongly
believe that this comes from our three-tiered commodity-architecture system. The same
scenario occurred in our DARP system. As we reengineered the DARP to couple it
strongly with the Gateway middle-tier from the client-server model, we lost nothing in
performance but gained much in flexibility and functionality.

From the user’s point of view, an XML-powered abstract job specification
makes it easy to establish distributed application. In addition, after a user creates and
starts an application, it can be saved in its complete state in an XML document. By
“state,” we mean the attributes of the user modules in an IDL interface and the
hierarchy of the distributed objects and the connections among them. A user can later restore a saved application and may edit the attributes of the modules and update the next startup.

Other than using XML to accomplish a persistency model, we also have the facility of using the database to store the distributed state. We used JDBC connection protocol to carry out the saving and restoring of a complete picture of a current user session.

There is dependency on the SAGE++ product to instrument user module source code. Because we have many available parsers for most languages, the DARP system can be extended to support the C++ and Java languages by simply annotating the YACC parser of the corresponding language to be instrumented. In this way, we can remove the SAGE++ dependency.

In the integrated environment of Gateway with DARP, it is possible to establish associations among completely different applications in addition to establishing connections among individual modules. For these cases, DARP and Gateway are capable of extracting, and collecting data from the application modules and sending it to another application via the high-performance globus communication lines, TCP/IP, CORBA IIOP, or PAWS [PAWSWeb].

Our architecture of Gateway allows us to put security and transaction protocol into the proxy code. We already have a simple transaction mechanism that allows the proxy to intercept a request before it is delivered to the user module. The proxy code saves or restores the user properties of the back-end module through the JDBC connection to the Oracle database. Therefore, Gateway can be extended to a similar
EJB development environment. By using the newly emerging CORBA facility, POA (Portable Object Adapter), Gateway can have more robust and dynamic behavior.

8.1 A Possible Framework of A Client-Side Collaborative Environment

There are two ways to achieve the sharing and interactions of front-end pages. One is by using the powerful event mechanism of JavaScript 1.3, and another is by implementing the front end as JavaBeans components.

With the new event model, a window or document object can intercept events generated by nested JavaScript objects such as Button, TextField, etc. By putting any JavaScript page into a virtual JavaScript window, we can catch all events fired by components of the original page. We can establish a collaborative environment in which the captured events are sent to our Gateway middle-tier, which keeps them until other collaborators pull them explicitly in an asynchronous way. Each virtual JavaScript window object in a collaborative environment parses the received event and routes it through the calling routeEvent method of a window object. Also, as we establish connections between back-end user modules we can create connections between front-end pages. Again, the Gateway middle tier can easily be employed for this purpose. In this case, the translation table incorporated into Gateway will hold the identifications of front-end pages instead of the assigned names of the user modules, and the rest of the parameters for attaching two front-end pages are the same. Note that the front-end pages behave as clients, as opposed to the back-end user modules, which work as a CORBA server. Because of this, we have to use the pull type of Gateway event-attaching model to connect front-end pages.

The second method of collaboration is just to define the front end as JavaBeans components. The Gateway middle tier will again behave as a centralized place to which
the fired events of front-end pages arrive, and other collaborators will explicitly pull the received events and update their pages. The event transport mechanism is hidden from users. Again, we put the user’s JavaBeans-defined page into a virtual JavaScript or Java window that has the same behavior as defined above. We can use the live-connect facility of Netscape if the window is JavaScript-based. Otherwise, we define a Java-based virtual window, which has the same capability as a JavaScript-based one.

8.2 How do user modules written in COM interact with the ones in CORBA?

We propose a framework in which a user’s COM modules can work together with CORBA modules. As previously noted, we create a proxy in the host of the master Gateway server for each instantiated user module. All of the interactions between user modules and GatewayContext objects pass through these proxy objects. We strongly believe that a proxy can be written as a mixture of the codes of CORBA and COM. This proxy can implement both the DynamicImplementation interface of CORBA and the IDispatch interface of COM; that is, it will have automation facility. Thus, the proxy will function as both a CORBA object and a COM component.

8.3 Gateway can act as firewall to remote objects

Another use for Gateway is as a firewall. We can base an application-level firewall on top of generated proxies that relay GIOP messages between users and their back-end modules. Remember that all of our user modules are always interacting with each other through proxies so that we have a two-way communication mechanism between any two modules, including callbacks. Because we can insert security protocol into the proxy, our firewall already has a secure channel. The security and transaction protocols for user modules can be handled transparently by the Gateway middle tier.
We can configure the proxy generated for each user module so that we can realize two styles of connection through a GIOP Proxy, *normal* and *passthrough*. In a normal connection a proxy can monitor the GIOP traffic, which causes two security issues. First, a client would not want the proxy to examine the traffic. Second, the client and server may be using an authentication and/or encryption mechanism that is unknown to the proxy. Both of these cases can be solved by a *passthrough* connection that simply forwards all of the GIOP messages it receives to the appropriate party.

### 8.4 Comparisons with other Component Models

We have three enterprise component models: CORBA Component Model (CCM), Enterprise JavaBeans (v1.1) (EJB) and DCOM/COM, and ongoing research product DOE Common Component Architecture (DCCA). The CCM specification was released first and EJB has adopted very similar architecture except that EJB is designed specifically for the Java language. Currently, only the COM model and early versions of EJB have been implemented. Therefore, we will compare the Gateway with them by looking at their current specifications.

COM, CCM, DCCA and Gateway have event models, but EJB does not. CCM uses a subset of CORBA’s notification service, which is a more extended version of the previous CORBA event service. Gateway has event service similar to that of CCM. DCCA adopted the “provides/uses interface” rule of CCM where the “provides” interface acts as an event listener and “uses” interface plays as an event subscriber. COM has the “provides/uses” concept for communicating events between COM objects.
CCA, EJB and Gateway have been using XML for persistency, while DCCA does not yet have persistency a model. Because COM is a binary-standard model, it saves components in binary format.

CCA and EJB have the **HomeBase** interface for entity objects, but not for Session objects to create and find objects. For entity objects, the user has to implement this interface with a Naming service or with JNDI. Gateway already has Naming service functionality incorporated into the core Gateway code. CCA and EJB are not able to put hierarchy on several containers and nested components but the most important aspect of **Gateway** is that it permits the user to view many components running on different locations as a single component and can look for any one. COM has **Monikers** as a naming service, but DCCA has no naming service concept.

Gateway, CCA and EJB all have the capability of intercepting user requests before delivering them to a real remote object. CCA and EJB have container tools to generate specific proxy objects for each component, but Gateway has a generic proxy for all components. COM also has some sort of functionality of this kind in its specialized transaction monitor MTS. DCCA doesn’t have this property yet.

CCM and EJB have to work with a database to keep persistency of objects, but Gateway can work with either Database and/or XML files, and both of these formats can be translated to each other to save the state of objects. We haven’t modified IDL syntax for Gateway, but EJB, CCM, and DCCA need to extend the base of IDL grammar. COM already has a mature IDL syntax.
Gateway, CCA, and EJB have the capability of assigning attributes to components by defining them in IDL and giving initial values in XML documents. DCCA does not yet have this functionality.

**Gateway** has nice architecture for adding some user-specific service objects. As explained in previous chapters, Gateway service is nothing more than an object. Therefore, we add service to the Gateway context as we do for a module, except that parent context has to inform all the registered objects waiting for this service by calling their “**serviceAvailable**” method. However, finding a service is a bit different. The user has to ask a Gateway context object to find any object that is reachable from this context in the object tree, and the user can find any service attached to any part of the object tree. COM, EJB, and DCCA do not have this functionality.

### 8.5 Lessons learned from Gateway

Because Gateway is a component-based development (CBD) environment, it provides the user with many properties of CBD that furnish the following benefits:

1. Fast development that enables programmers to build solutions faster by assembling software from pre-built parts.

2. Lower integration costs that provide a common set of interfaces for software programs from different vendors, meaning that less custom work is required to integrate components into complete solutions.
3. Improved deployment flexibility, making it easier to customize a software solution for different areas of a company simply by changing some of the components in the overall application.

4. Lower maintenance costs, isolating software functions into discrete components that provide a low-cost, efficient mechanism with which to upgrade a component without having to retrofit the entire application.

We have demonstrated the Gateway with the adaptation of legacy software applications such as QS and LMS. Gateway is a highly-generic, multi-purpose and extendible component model that actually has a platform-independent and three-tiered system and can be customized for different environments such as telecommunication, E-commerce, etc. A user who wants to develop a package from scratch with Gateway will first break up the solution for a problem into simple components and their information exchange mechanism. Ready-to-use components will probably be used then, and the user's own new components may also be written in. After that, Gateway can be used to configure, deploy and connect the components. In this way, complex HPCC packages can be developed and deployed by project members working concurrently. Thus HPCC can have quick development and deployment with lower maintenance costs. If monolithic software approach were taken, then integration, maintenance, and customization of different software modules would be difficult, because modules would be very large and difficult to handle. Using Gateway for simple projects might not be of much benefit, but for large projects it will decrease by an incredible amount the time required
for development and deployment. Although training people is very important for selling a product, even non-technical people can use Gateway simply by writing an XML document to customize and define a distributed application.

Gateway will need a visual development environment like BDK (the Javabeans development environment). Currently, it has some browser-based interfaces to customize and run specific projects running across the Internet under the Gateway system. We need a more generalized interface to compose distributed applications visually. We may use VGJ [VGJWeb] or GEF [GEFWeb] to achieve this goal or we can create our own visual front-end. Gateway provides the properties, fired events, and other related information for each user module as an XML document to build this visual interface. All of this information is used by the interface to configure individual modules and connect them through event and property binding to compose complex distributed applications from the modules.
REFERENCES


[CASCD] Fred Ogden, University of Connecticut, ogden@eng2.uconn.edu


http://www.npac.syr.edu/users/haupt/SC97/HPccdemos.html


http://www.npac.syr.edu/users/haupt/HPFI


[EDYS] Edys has been written by Michael Childress, Shepherd Miller, Inc., mchildress@shepmill.com


[GEFWeb] GEF: The Graph Editing Framework home page:
http://www.ics.uci.edu/pub/arch/gef/


[GSSAPIWeb] RFC 1508, RFC 2078

[HabaneroWeb] “NCSA Habanero,”

http://www.ncsa.uiuc.edu/SDG/Software/Habanero/index.html


[QSWeb] Quantum Simulations,
http://www.ncsa.uiuc.edu/Apps/CMP/cmp-homepage.html


[SC98Pres] T. Haupt, "WebFlow High-Level Programming Environment and Visual Authoring Toolkit for HPDC (desktop access to remote resources)," SC’98 technical presentation,


[SSLWeb] SSL, Netscape Communications, Inc,


[WebFlowHPCNJournal] The above HPCN paper has been selected by the program committee as one of the best that was submitted to the proceedings. It is invited to publish in a special issue of the *Elsevier Journal on Future Generation Computer Systems*.


http://www.itl.nist.gov/div895/sasg/websubmit/websubmit.html

[WMSWeb] WMS, EDYS and CASC2D codes has been made available to us by CEWES. EDYS is written by Michael Childress and CASC2D is written by Fred Ogden, http://www.wes.hpc.mil
Vitae

NAME: EROL AKARSU

DATE OF BIRTH: 20 SEPTEMBER 1965

PLACE OF BIRTH: USAK, TURKEY

EDUCATION:

DECEMBER 1999 Ph.D. in Computer Science
Department of Electrical Engineering and Computer Science, Syracuse University,
Syracuse, NY U.S.A.

MAY 1996 M.S. in Computer Science
Department of Electrical Engineering and Computer Science, Syracuse University,
Syracuse, NY U.S.A.

JULY 1993 M.S. in Computer Engineering,
Istanbul Technical University
Istanbul, TURKEY

JULY 1991 B.S. in Computer Engineering, Ege University
Izmir, TURKEY.
EXPERIENCE:

JULY 1 1995 – NOVEMBER 8 1999  Graduate Research Assistant
Northeast Parallel Architectures Center
Syracuse University
Syracuse, NY U.S.A.

SEPTEMBER 1 1991 – OCTOBER 1 1993  Graduate Research Assistant
Information Processing Center
Istanbul, TURKEY.
GLOSSARY

**Applet** A partial Java application program designed to run inside a web browser with help from some predefined support classes.

**ATD** Abstract Job Descriptor written in XML. Usually does not specify hardware resources for jobs defined.

**AVS** A commercial data visualization package provided by Advanced Visual Systems.

**BDK** Javabeans Development Kit. A graphical user interface for composing front ends.

**CGI** Common Gateway Interface. A non-Java technique of sending data from HTML forms in browsers to server programs written in C, Python, Tcl, or Perl. They typically do data base searches or process data in HTML forms and send back MIME.

**CM** Connection Manager. A running servlet. Part of the WebFlow middle tier that works to create associations among user modules.

**COM** Common Object Model. Microsoft's windows object model, which is being extended to distributed systems and multi-tiered architectures. ActiveX controls are an important class of COM objects that implement the component models of software.

**DCOM** Distributed version of COM

**ComponentWare** An approach to software engineering with software modules developed as objects with specific design frameworks and with visual editors both to interface to properties of each module and to link modules together.
**Computational Grid**  A recent term used by the HPCC community to describe large-scale distributed computing that draws on analogies with electrical power grids.

**CORBA**  Common Object Request Broker Architecture. An approach to cross-platform, cross-language distributed objects developed by a broad industrial group, the OMG. CORBA specifies basic services (such as naming, trading, persistence) and the protocol IIOP used by communicating ORBS. It is developing higher level facilities that are object architectures for specialized domains.

**DARP**  Data Analysis and Rapid Prototyping. Protocol that integrates compiled and interpreted environments and provides a web-based interface for the user.

**DII**  Dynamic Invocation Interface. An interface defined in CORBA that allows the invocation of operations on object references without compile-time knowledge of the objects' interface types.

**DSI**  Dynamic Skeleton Interface. An interface defined in CORBA that allows servers to dynamically interpret incoming invocation requests of arbitrary operations.

**EJB**  Enterprise Javabeans. Enhanced Javabeans for server-side operations with capabilities such as multi-user support. A cross-platform component architecture for the development and deployment of multi-tier, distributed, scalable, object-oriented Java applications.

**Event**  A noteworthy state change of an object or signal that involves the behavior of an object. An event can signal the creation, termination, classification, declassification, or change in value of an object. For example, the creation of a new circuit design or the debit of $300 to a particular account.
Exception  An indication that some invariant has not, or cannot, be satisfied. Mechanisms for handling exceptions are often added to OO programming languages and environments. For example, Java, C++, and CORBA all have built-in exception handling.


GEF  A graphics package written in Java.

Globus  A metacomputing infrastructure toolkit that is a bag of of services.

GRAM  Globus Resource Allocation Manager. Provides secure mechanisms to allocate and schedule resources.

GSSAPI  Generic Security Services Application Programmer Interface. Provides a standard programming interface that is authentication-mechanism independent, which allows the application programmer to design an application and application protocol that can be used for alternative authentication technologies, including Kerberos.

HPcc  High Performance commodity computing. An NPAC project that developed a commodity computing-based high performance computing software environment. (Note that we have dropped the word "communications" referred to in the classic HPCC acronym--not because it is unimportant, but rather because a commodity approach to high performance networking is already being adopted. We focus on high-level services such as programming, data access and visualization that we abstract to the rather wishy-washy "computing" in the HPcc acronym.)

HPCC  High Performance Computing and Communication. Originally a formal federal initiative, but even after that ended in 1996, this term has been used to describe
the field devoted to solving large-scale problems with powerful computers and networks.

**HPF**  High Performance Fortran Language. Extended from Fortran90.

**HTTP**  Hyper Text Transport Mechanism. A stateless transport protocol allowing control information and data to be transmitted between web clients and servers.

**IDL**  Interface Definition Language. A language, platform, and methodology-independent notation for describing objects and their relationships. IDL is used to describe the interfaces that client objects call and that object implementations provide.

**IIOP**  Internet Inter Orb Protocol. A state protocol allowing CORBA ORBs to communicate with each other and transfer both the request for a desired service and the returned result.

**IR**  Interface Repository. A container, typically a database, of OMG IDL interface definitions. The interface to the interface repository is defined in the CORBA specification. Implementations of this interface are supplied by CORBA vendors.

**Javabean**  Part of the Java 1.1 enhancements defining design frameworks (particularly naming conventions) and inter-Javabean communication mechanisms for Java components with standard (Bean box) or customized visual interfaces (property editors). Javabeans are Java's component technology and in this sense are more analogous to ActiveX than either COM or CORBA. However, Javabeans augmented with RMI can be used to build a "pure Java" distributed-object model.

**JDBC**  Java Data Base Connection. A set of interfaces (Java methods and constants) in the Java 1.1 enterprise framework that define a uniform access to relational databases. JDBC calls from a client or server a Java program link to a particular "driver" that
converts these universal database access calls (establishes a connection, SQL query, etc.) to the particular syntax needed to access essentially any significant database.

**Jini** Sun's protocol for devices to identify each other using TCP/IP protocol. It will be used in small devices such as telephones to allow a new device to be plugged into the system while everything is running. This device automatically finds out about everything else on the net and allows other devices to find it.

**LDAP** Lightweight Directory Access Protocol. A client-server protocol for accessing a directory service. Initially used as a front-end to X.500, but can also be used with stand-alone and other kinds of directory servers.

**Legacy System** A production system designed for technology assumptions that are no longer valid or that are expected to become invalid in the foreseeable future. When deploying new applications or new system architectures, a legacy system is one that may be accessed, but will typically not be modified to support new architecture.

**LMS** Landscape Modeling Simulation. A legacy code for managing land and water resources.


**MM** Module Manager. A running servlet. A part of the WebFlow middle tier that deals with life cycles of WebFlow modules.

**MPI** Message Passing Interface. Designed as a standard for optimized communication among parallel processors.

**Object** Anything that can be referred to; anything that can be identified, named, or perceived as an object; anything to which a type applies; an instance of a type or class. An instance of a class is comprised of the values linked to the object (Object State) and
can respond to the requests specified for the class.

**Object Interface**  The set of requests that an object can respond to, i.e., its behavioral specification. An object interface is the union of the interfaces of the object's type interfaces.

**Object Web**  The evolving systems' software middleware infrastructure achieved by merging CORBA with Java. Correspondingly, merging CORBA with Javabeans gives Object Web ComponentWare, which is expected to compete with Microsoft's COM/ActiveX architecture.

**OMG**  Object Management Group. An organization of over 700 companies that is developing CORBA through a process of proposal calls and the development of consensus standards.

**ORB**  Object Request Broker. Used in both clients and servers in CORBA to enable remote access to objects. ORBs are available from many vendors and communicate via IIOP protocol.

**ORBacus**  An example implementation of an OMG CORBA specification.

**Proxy**  An object that is authorized to act or take action on behalf of another object.

**QS-Quantum Simulation**  A simulation code for solving the Schrodinger equation or systems of many electrons. Results in the calculation of an electronic structure.

**Request**  An event that is the invocation of an operation. The request includes the operation name and zero or more actual parameters. A client issues a request to cause a service to be performed. Also associated with a request are the results, which can be returned to the client. A message can be used to implement (carry) the request and an results.
**RSL**  Resource Specification Lanaguage. Part of the Globus package. Used to send job requests to GRAM.

**Server Object**  An entity (e.g., object, class, or application) that provides a response to a client's request for a service.

**Servlet**  An application designed to run on a server in the womb of a permanently resident CGI mother-program written in Java that provides services for it, much the way an Applet runs in the womb of a Web browser.

**SM**  Session Manager. A running servlet, a part of the WebFlow middle tier that handles sessions for different users.

**SSL**  Secure Socket Layer. A protocol for communication securely through sockets.

**UML**  Universal Modeling Language. A modeling technique designed by Grady Booch, Ivar Jacobson, and James Rumbauch of Rational Rose. It is used for OOAD (Object Oriented Analysis and Design) and is supported by a broad base of leading industries. It merges the best of the various notations into one single notation style.

**VGJ**  A graphics package written in Java.

**Web Client**  Originally, web clients displayed HTML and related pages but now support Java Applets that can be programmed to give web clients the necessary capabilities to support general enterprise computing. The support of signed applets in recent browsers has removed crude security restrictions that handicapped the previous use of applets.

**Web Servers**  Web Servers originally supported HTTP requests for information, basically HTML pages, but included the invocation of general server side programs using the very simple but arcane CGI (Common Gateway Interface). A new generation
of Java servers have enhanced capabilities, including server-side Java program enhancements (Servlets) and the support of permanent communication channels.

**XDR** External data representation. A protocol for sending data among heterogeneous architectures that was developed by SUN Microsystems.

**XML** Extensible Markup Language. A W3C-proposed recommendation. Like HTML, XML is based on SGML, an International Standard (ISO 8879) for creating markup languages. However, while HTML is a single SGML document type, with a fixed set of element type names (AKA "tag names"), XML is a simplified profile of SGML.