The Gateway System: Uniform Web Based Access to Remote Resources

Tomasz Haupt, Erol Akarsu, Geoffrey Fox, Choon-Han Youn
Northeast Parallel Architecture Center at Syracuse University

Abstract:
Exploiting our experience developing the WebFlow system, we designed the Gateway system to provide seamless and secure access to computational resources at ASC MSRC. The Gateway follows our commodity components strategy, and it is implemented as a modern three-tier system. Tier 1 is a high-level front-end for visual programming, steering, run-time data analysis and visualization, built on top of the Web and OO commodity standards. Distributed object-based, scalable, and reusable Web server and Object broker middleware forms Tier 2. Back-end services comprise Tier 3. In particular, access to high performance computational resources is provided by implementing the emerging standard for metacomputing API.

1. Introduction

The last few years have seen the growing power and capability of commodity computing and communication technologies largely driven by commercial distributed information systems. These can be all abstracted to a three-tier model with largely independent clients connected to a distributed network of servers. High performance can be obtained by combining concurrency at the middle tier with optimized parallel back-end servers. The resultant system combines the needed performance for large-scale HPCC applications with the rich functionality of commodity systems.

In each commodity technology area, we have impressive and rapidly improving software artifacts. Perhaps even more importantly than raw technology, we have a set of standards and open interfaces enabling distributed modular software development. These interfaces are at both low and high levels and the latter generate a very powerful software environment in which large preexisting components can be quickly integrated into new applications. We believe that that there are significant incentives to build HPCC environments in a way that naturally inherits all commodity capabilities so that HPCC applications can benefit from the impressive productivity of commodity systems. We termed such approach High Performance Commodity Computing (HPcc).

In several related papers [1] we have described NPAC's HPcc activity that is designed to demonstrate that this is possible and useful so that one can achieve simultaneously both high performance and the functionality of commodity systems. One of these activities is a specific high-level programming environment developed at NPAC - WebFlow [2] - which offers a user-friendly visual graph authoring metaphor for seamless composition of world-wide distributed high performance dataflow applications from reusable computational modules.

One of the most spectacular applications of the WebFlow is Quantum Monte Carlo Simulations [3] developed in collaboration with the NCSA Condensed Matter Physics Laboratory. Here, a chain of high performance applications (both commercial packages such as GAUSSIAN or GAMESS, and 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. The output file of one application in the chain is the input of the next one, after a suitable format conversion.

In spite of the success of the WebFlow project we see that the original implementation, based on Java web servers, suffers form severe limitations. Two the most obvious areas of improvement we want to achieve are fault tolerance and security. However, instead of adding complexity to already complex and to large extend custom protocol of exchanging data between the servers, we have re-implemented the WebFlow middle-tier using industry standard distributed object technologies: JavaBeans and CORBA and industry standard secure communication protocols based on SSL.

The development of the new middle-tier of our system coincides with the JavaGrande [4] initiative to develop international standards for seamless Desktop Access to Remote Resources (DATORR). These standards replace the remaining two custom WebFlow interfaces: computational graph generated by the WebFlow front-end by the Abstract Task Specification and specific Globus[5] interface by the universal metacomputing API.

The new implementation of WebFlow is a part of Aeronautical Systems Center (ASC) Major Shared Resource Center (MSRC) Gateway project at Wright-Paterson Air Force Base, sponsored by DoD HPC Modernization Program, Programming Environment and Training. The objectives of this project are to provide seamless and secure access to computational ASC MSRC resources through web-based interfaces. The functionality of the Gateway system is specified in section 2. Section 3 presents the system architecture and provides a high level description of its major components. In section 4, we discuss the Gateway security model and in section 5 we reveal the middle-tier implementation details. Section 6 provides links to the related research. The paper is summarized in section 7.

2. Overview of the Gateway functionality

The Gateway system offers a particular programming paradigm implemented over a virtual Web accessible metacomputer. A (meta-) application is composed of independently developed modules. The modules are implemented in Java. This gives the user the complete power of Java, and object oriented programming in general, to implement the 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.

The module developers have only limited knowledge of the system on which the modules will run. They not need to 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.

Often, the modules 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 delegating the actual implementation of the module functionality to a back-end DBMS. We follow a similar approach to provide access to high performance resources: a Gateway module "merely" implements an API of a back-end metacomputing services such as those provided by the Globus metacomputing toolkit. In particular, a module that serves as the GRAM (Globus Resource Allocation Manager) proxy generates a resource allocation request. The request essentially defines an executable, its standard input, error and output streams, and the target machine where the executable is to be run. The application represented by the executable is developed independently of the Gateway system (for example, it may be a legacy parallel code written in Fortran + MPI). The role of the Gateway module written in Java is reduced to generating the request following the low level Globus Resource Specification Language (RSL) syntax. In this sense, the Gateway system can be regarded as a high level, visual user interface and job broker for the Globus system.

The Gateway system supports many different programming models for the distributed computations: from coarse-grain dataflow to object oriented to fine-grain data-parallel model. In the dataflow 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. This model is generalized in our new implementation of the Gateway system. Thanks to the fact that modules behave as distributed JavaBeans, each module may invoke an arbitrary method of the other modules involved in the computation.

3. Gateway Architecture

The Gateway system is implemented as a modern three-tier system, as shown in fig. 1. Tier 1 is a high-level front-end for visual programming, steering, run-time data analysis and visualization, built on top of the Web and OO commodity standards. Distributed object-based, scalable, and reusable Web server and Object broker Middleware forms Tier 2. Back-end services comprise Tier 3. In particular, high performance services are implemented using the metacomputing toolkit of Globus.

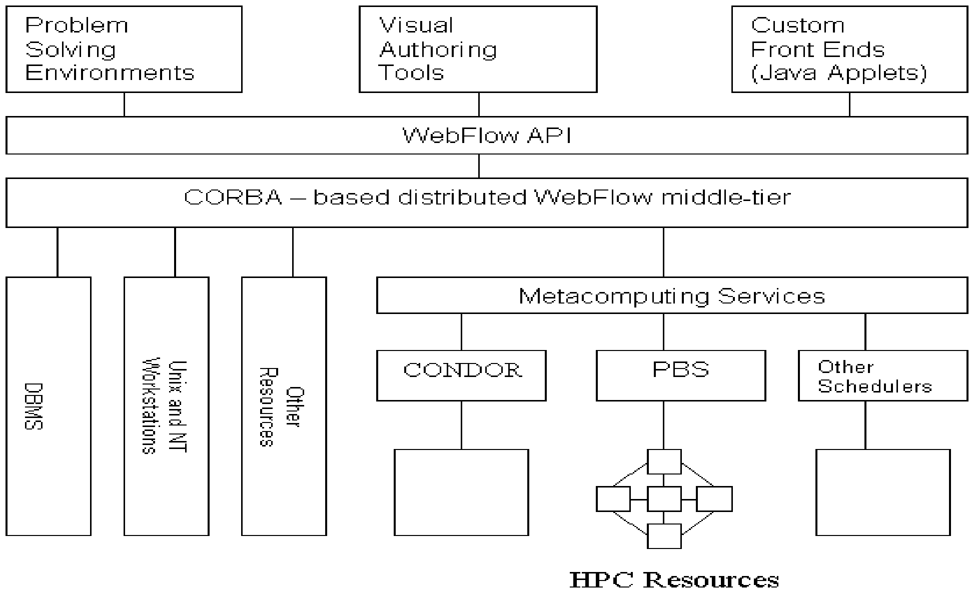

Fig 1: Gateway system architecture

Front End

Different classes of applications require different functionality of the front-end. Therefore we designed the Gateway system to support many different front-ends: from very flexible authoring tools and problem solving environments (PSE) that allows for dynamical creation of meta-applications from pre-existing modules, to highly specialized front-ends customized to meet the need of particular applications. Also, we support many different computational paradigms, from general object-oriented to data-flow to a simple "command line" approach. This flexibility is achieved by treating the front-end as a plug-in implementing the Gateway API.

Gateway API

The Gateway API allows specifying the user's task in the form of the Abstract Task Descriptor (ATD), following the current DATORR recommendations. The ATD is constructed recursively, and may comprise arbitrary number of subtasks. The lowest level, or atomic, task corresponds to the atomic operation in the middle-tier, such as instantiation of an object, or establishing interactions between two objects through event binding. However, in many cases such details should be hidden from the end-user or even the front-end developer. Therefore, the Gateway API provides interfaces to higher level functionality, such as submit a single job or make a file transfer.

When specifying the task, the user does not have to specify resources to be used to complete the task. Instead, the user may specify requirements that the target resource must satisfy in order to be capable of executing the job. The identification and allocation of the resources is left to the system discretion. Typically, the middle-tier delegates it to the metacomputing services (such as Globus) or an external scheduler (such as PBS). Once the resources are identified, the abstract task descriptor becomes a Job Specification.

Middle Tier

The middle tier is given by a mesh of CORBA-based Gateway servers. A Gateway server maintains the users sessions within which the users create and control their applications. The middle-tier services provide means to control the lifecycle of modules and to establish communication channels between them. The modules can be created locally or on remote hosts. In the latter case, the task of the module instantiation and initialization is transparently delegated to a peer Gateway server on the selected host, and the communication channels are adjusted accordingly. The services provided by the middle tier include methods to submit and control jobs, methods for file manipulations, method providing access to databases and mass storage, as well as methods to query the status of the system, status of the users applications and their components. More comprehensive description of the middle tier is given in Section 5.

Gateway Modules

The Gateway modules are CORBA objects implemented in Java. The functionality of a module is implemented either directly in the body of the module or the module serves as a proxy of specific backend services, such as DBMS or HPCC services. We developed a collection of generic modules that provide basic access to back-end resources such as file systems, batch systems, and XML parsers. We refer to these modules as Gateway services. By implementing the Gateway module interface, the user can develop his or her own modules as needed.

Meta-computing Services

The metacomputing service is yet another standard being developed within the DATORR initiative. It specifies all mandatory functionality of a metacomputing system and its interfaces. The Globus toolkit is an example of such metacomputing services. The functionality it provides include secure resource allocation (GRAM), secure file transfer (GASS), metacomputing directory services (MDS), heartbeat monitor (HBM), and more.

4. Gateway Security Model

The Gateway system supports a three-component security model. The first component is responsible for a secure web access to the system and establishing the user identity and credentials. The second component enforces secure interactions between distributed objects, including communications between peer Gateway servers, and delegation of the credentials. The third component controls access to back-end resources.

The delegation of credentials is an important issue here. There are many known solutions for secure access in two-tier systems. In the Gateway system, they correspond to communications between the front end and the middle tier, or between the middle-tier and the back end. Web access over open Internet is usually implemented using SSL (secure socket library) pioneered by Netscape Communications. SSL is based on the public key infrastructure (PKI), including X.509 digital certificates. Secure access to back-end resources is typically provided using Kerberos system or PKI based Global Security Services (GSS) developed by the Globus team. For a three-tier system, we must provide solution where the front-end user credentials are accepted by the back end.

When SSL is selected as the underlying security mechanism, we can adopt the Globus delegation model. It is based on the idea of proxy credentials. A proxy credential is a temporary certificate created on the user behalf on the server side. A new key pair is generated, and the resulting certificate is sent back to the user to be signed by his or her private key. This mechanism allows creating user credentials on the server side without sending any sensitive data, such as password or pass phrase, over an open network.

The NPACI HotPages project[6] offers an alternative solution. The credentials needed to access back-end resources (that is, Globus certificates) are permanently stored on the server side. The user accesses the system through a secure (SSL) web server. After a mutual authentication, the user Globus certificate is retrieved, and presented to the back end.

It is difficult to implement delegation of credentials in kerberized environments, in particular when Kerberos in used in conjunction with SecurID protection. Because SecurID forces change the user’s pass phrase every two minutes or so, it is impossible to generate proxy credentials seamlessly, that is, without prompting the user for a new pass phrase each time the back end resources are requested. This violates our goal of providing a seamless access to resources.

Therefore, in the Gateway system we adopted yet another approach based on CORBA security services. The user generates his or her Kerberos ticket (TGT) locally on his workstation before a Gateway session is started. The session is initialized by connecting to the Gateway gatekeeper web server. The server returns a web page with the Gateway control applet. The applet serves as a secure CORBA client. The applet connects to the Gateway server through SECIOP (secure version of IIOP – Internet Inter-ORB Protocol). This involves uploading the user's TGT to the server and standard Kerberos authentication and authorization procedures. From that moment on, all communications between the front end and the middle tier go through SECIOP rather than http or https. We use a commercial, secure version of ORBacus[7] ORB, called ORBAsec, developed by Adiron [8].

The user ticket is then forwarded to the back-end using modified Globus mechanisms. The modification involves recompilation of GRAM (Globus Resource Allocation Manager) against the Kerberos libraries (as opposed to standard SSLeay). As a result, instead of generating a proxy credential, the original user ticket is presented to the GRAM server and used for authentication.

In the following sections, we provide some additional information on designing the Gateway security model.

Secure Web Transactions: Authentication and Authorization

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 and she must present 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).

The authorization process is controlled by the AKENTI server [9]. It 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 access to the back-end services is controlled independently by the other two components of the Gateway security services, and it is based on credentials generated during the initial contact with the gatekeeper.

Secure CORBA: middle tier security

Security features of CORBA are build directly into ORB and therefore they are very easy to use. Once the user 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).

The access control is based on the access control lists (ACL). These provide 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. This way, the 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 credentials. This way access to the back-end resources can be controlled by the owners of the resources and not the Gateway system. The Gateway system merely forwards the user credentials.

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

Distributed objects are inherently less secure than traditional client-server systems. Enhanced risk level comes, among other factors, from the fact that objects often delegate parts of their implementation to the other objects (which may be dynamically composed at runtime). This way objects 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. Therefore, the policy of privilege delegation is a very important element of the system security. CORBA is very flexible here, and supports a 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 the intermediary privileges. However, if the application applies its own security measures, we make the initiator's credentials available to it.

Control of Access to Back End Resources

There are no widely accepted standards for a secure access to resources. Different computing centers apply different technologies: SSH, SSL, Keberos5, or other. The design goal of the Gateway system is to preserve the autonomy of the resources owner to define and implement its security policies. At this respect, we are in a very similar situation as 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 the future standards on top of the GSS-API specification (and thus to support simultaneously private and public key based technologies). The Globus project pioneered this approach, and therefore we use Globus GRAM to provide a secure access to the 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).

5. Middle Tier

In an object-oriented approach, applications are made of components and containers. Components are software objects that provide implementation of a set of standard behaviors. These behaviors are defined by the component model to ensure that components can be composed and interoperate efficiently and without conflict. One can build a Java applet by placing AWT components – buttons, labels, text fields and so forth – into frames or panels, which are object containers, following JavaBeans component model. This idea can be easily extended to non-graphical components. 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.

The Gateway system introduces a distributed component model, making building a distributed, high-performance application a process similar to that of building a distributed applet (cf. Fig 3).

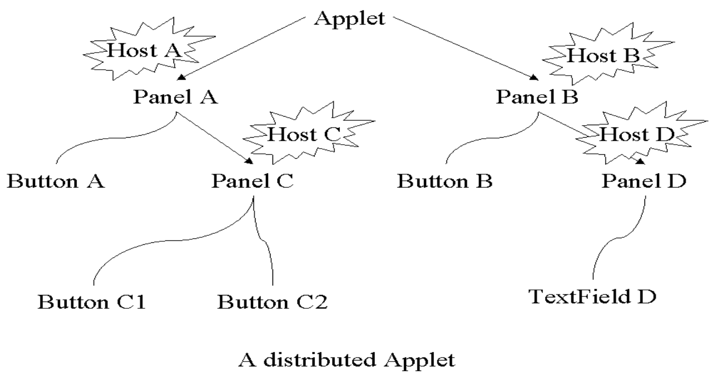

Figure 3. A hypothetical distributed applet. Each panel (a container) of this applet is placed on a different host.

As in the case of the applet, we introduce container objects, which we refer to as **Contexts,** and **Modules.** They correspond to AWT container and AWT component, respectively. Figure 4 shows a distributed in application as implemented in the Gateway environment. Typically, a Gateway object serves as a proxy for a service rendered by the back end resources. A web-based client tier provides tools for visual composing and distributing the hierarchy of Gateway objects.

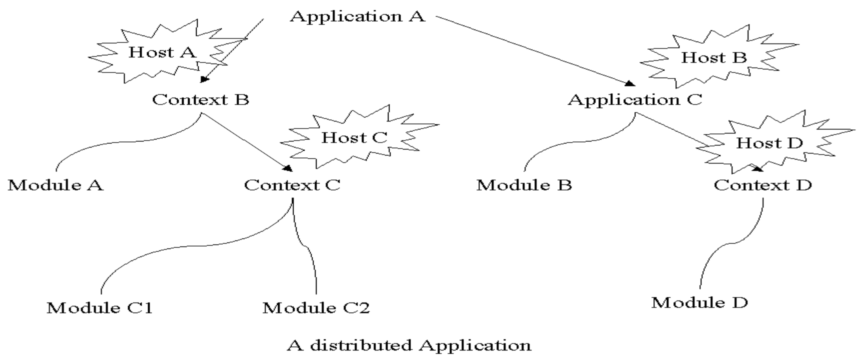

Figure 4: A distributed Gateway application

The Gateway context keeps information on all its children (contexts and modules), and is responsible for the life cycle of the modules, inter-module communications, and communications between modules and the front end. The contexts are distributed, that is, they can live in different address spaces. We have chosen CORBA to facilitate inter-process communications. Consequently, all Gateway components are CORBA objects, currently implemented in Java.

Object Hierarchy
The context can run as a separate process (in such a case we refer to it as a **Gateway Server**), or it can be embedded in an existing context ("subcontext"). It follows that the complete Gateway middle-tier may be deployed as a single process or as a distributed, multi-process system. In the former case, the Gateway Server plays a similar role as a JavaBeans Bean Box. In the other case, the Gateway resembles an Enterprise JavaBeans system.

Building an object hierarchy starts with launching a Gateway server, referred to as the **Master Server**, that becomes the root of the hierarchy. New contexts and modules within the Java Virtual Machine (JVM) that runs the Master are recursively added by invoking the context methods. New contexts on different hosts are created by launching new Gateway servers (**Slave Servers**) on these hosts. Once a slave server is running on a given host, new contexts and modules can be added to it by invoking the context method in the same way as in the master server case.

The master server publishes its CORBA Interoperable Object Reference (IOR) on a Web server that runs on the same host. To connect to the server, a client reads the master's IOR from the known URL, and converts it to the actual object reference using ORB (CORBA Object Request Broker) methods. The IOR file name and path is specified in the master server configuration file.

By contrast, a slave server does not publish its IOR. Its configuration file contains the URL of the master server and the name of its parent server. The slave server connects to the master and registers itself as a child of the specified parent. As the result, the difference between a "subcontext" and the Gateway server is hidden from the client.

Example: the Master server acts as a gatekeeper and it is started by the Gateway administrator. The user connects to it, and once authenticated it starts a new Gateway slave server (the user context) on a host different from that on which the master is running. Within his or her context, the user creates a distributed application by adding an application context to the user context, and starting new slave servers on hosts that have access to particular resources such as GRAM client, JDBC services, specific file system, etc. These slave servers are children of the user context, which in turn is a child of the master. As explained below, the user has access to the entire object hierarchy through the Master server.

The Gateway object naming convention is similar to that of LDAP and CORBA Naming Service. Each Gateway component has a unique name, equivalent to a LDAP distinguished name (DN), that is constructed by concatenation of its common name and names of all its ancestors. For example, the DN of a module "M" inside an application context "A" of a user "Joe" is MasterName/Joe/A/M. The DN does not depend on whether the context Joe is a Gateway server, or it is a subcontext of the MasterName server. The Gateway context names make the use of the CORBA Naming Service redundant.

Proxy Objects
The root of the context hierarchy, the Master server, plays a special role. This comes from the fact that we allow to implement the CORBA clients (front-end) as Java applets, and therefore we have to deal with the Java sandbox security restrictions. In order to allow for IIOP (or SECIOP)

connections between the applet and the Gateway Server, the server must run on the same host as the Web server from which the applet has been downloaded. In order to enable the communication between the client and the slave servers running on different hosts, the master creates and maintains proxies for each component in the hierarchy. The main purpose of proxies is to forward requests from the Web client to remote objects. 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.

Gateway Lifecycle Service

The Gateway requires a customized Lifecycle service (as compared to the OMG specification). In particular, instantiation of a Gateway object (module or context) on a local or remote host (to be run under control of the peer Gateway server) requires the creation of a local proxy module in Master context.

Each context has an object factory and is responsible for all lifecycle operations of its children objects. The user creates and destroys a Gateway object by invoking the context methods (addNewContext, addNewModule, removeModule, removeContext, removeAll, removeItself, etc.). When an object is destroyed, its proxy kept by the Master is destroyed as well. When a context is destroyed, all its children are recursively destroyed, too.

Interactions between modules

CORBA makes it easy to invoke methods of remote objects. Given an object reference, the methods of the object can be invoked exactly the same way as a local object (within the same JVM). The complexity of inter-process communications, argument marshalling, etc., is hidden by an ORB. The only difference between local and remote objects is how the object reference is obtained.

When a new module is created by a CORBA client, the context's addNewModule method returns its reference that can be immediately used to invoke methods of the module. There are several methods of obtaining references to existing objects. One is publishing the module IOR. The module reference can be passed to the client as an argument. If the module distinguished name is known, the reference can be obtained form the Master server. Finally, since a Gateway context keeps record of all its children, the entire object hierarchy can be searched to localize the module.

Since the modules are developed independently of each other, and they can play a role of both client and server, none of the methods is adequate for their interactions. The IOR may be stalled, that is, the serialized reference corresponds to an object that no longer exists. The reference must be known by the caller to be passed as an argument (chicken and egg problem). The module DN is usually unknown, as the user may put the module in a different context each time the module is needed. Finally, querying the context hierarchy would require a detailed knowledge of the Gateway internals, making development of custom modules difficult for a module developer. Therefore, we added yet another way of method invocation: event notification.

The idea behind the Gateway event model is very simple. An event is a CORBA object itself, which encapsulates the data to be sent from one module (a client) to another (a server). To invoke a method of the server, the client "fires" an event; this actually is an invocation of a parent context method. The context implements an event adapter, which is a translation table. Each entry of the table contains a source object reference, event identifier, target-object reference, target-method name, and the type of the connection (push or pull). In other words, each event type (defined by an identifier) fired by a client module is associated ("connected") with a particular method of the target module. Consequently, firing an event results in the prescribed method invocation made by the context on behalf of the client module. The detailed description of the Gateway event model, including the role of the proxy modules is shown in Figure 5. The event object is passed as an argument to the invoked method. Our implementation is based on CORBA specific features: interface repository (IR), dynamic invocation interface (DII) and dynamic skeleton interface (DSI).

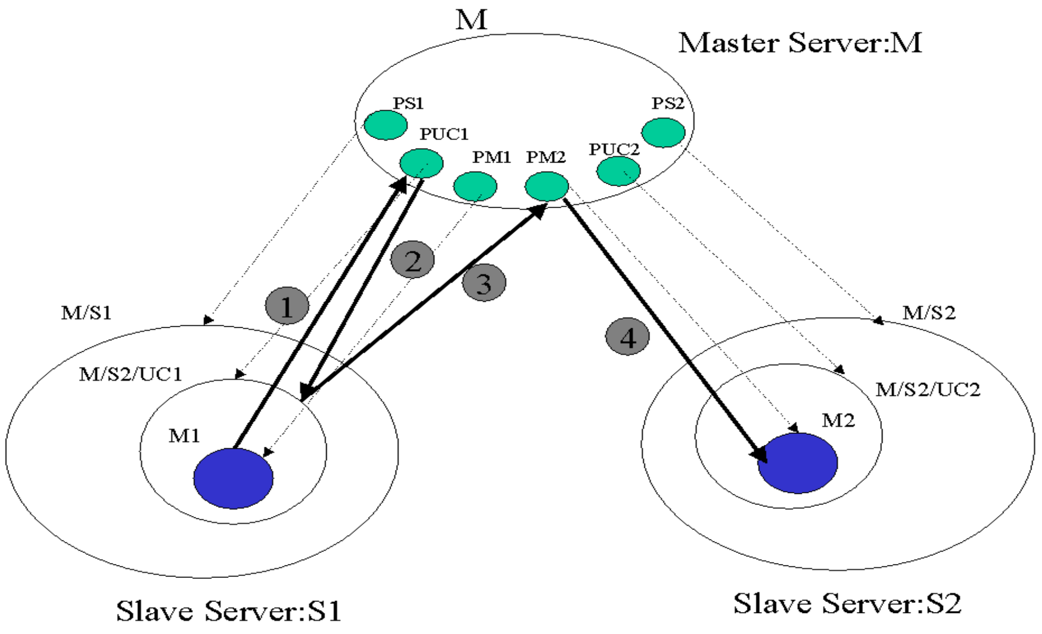

Figure 5. Gateway event model.

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 green 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.
Module M1 fires the event, which is intercepted by the proxy of its parent context PUC1. The proxy forwards it to the actual context UC1. 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. 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 overweigh any possible shortcomings. First, we use this long path through proxies only to transfer control logic between objects while the actual data transfer is carried out at 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. Finally, sending events through proxies opens an opportunity for filtering events.

This event-driven mechanism of method invocation solves the problem of obtaining object references. The client invokes a method of its parent context, and the context invokes a public method of the server. That is, neither client nor server is required to know the reference of the other module. It is the client context responsibility to have access to the object references. This is easily achieved, since the module binding can be done immediately after instantiation of modules. In a typical scenario, a front-end client builds the object hierarchy. Each invocation of the addNewModule method returns the reference of the newly created module that can be used to fill in the translation table (by invoking attachEvent method of the context).

The module binding is dynamical, that is, the connections between modules can be added, modified and removed at any time, as long as the object references are known. Another restriction is that the target method must be included in the server module IDL definition, and it must have adequate signature (must accept the event as an argument).

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 calls the registered target method immediately. 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.

Note that we do not make any use of the CORBA event service based on an event-channel object. In the OMG model, event suppliers subscribe to an event channel as event producers. All events in the event channel are “public,” that is, any object can register itself as a listener (event consumer) for an arbitrary event. Whenever an event comes from an event supplier, the event channel forwards it to all subscribed consumers, since the event channel cannot identify the source of the event in order to forward it to one specific consumer. The support for point-to-point event exchange will be provided in the future releases of CORBA as the event notification service. At this time, we are forced to develop our own event service.

6. Related Work

There are several other projects addressed to solving the problem of seamless access to remote resources. A comprehensive list of these is available from the JavaGrande web site [10]. Here we mention the three that are most closely related to this project.

The UNICORE project [11] introduces an excellent model for the Abstract Task Descriptor that most likely will strongly influence the DATORR standard, and consequently we are taking a very similar approach. The UNICORE middle-tier is given by a network of Java web servers (Jigsaw). The WebSubmit project [12] implements web access to remote high performance resources through CGI scripts. Both projects use https protocol for user authentication (as we do), and implement custom solutions for access control. The ARCADE project [13] is in a very early stage, and its designers intend to use CORBA to implement the middleware. As of now, there is no available description of the ARCADE security model.

7. Summary

To summarize, exploiting our experience developing the WebFlow system, we designed a new system, Gateway, to provide seamless and secure access to computational resources at ASC MSRC. While preserving the original three-tier architecture, we re-engineered implementation of each tier in order to strictly conform to the standards. In particular, we use CORBA and the JavaBeans model to build the new middle tier, which facilitates 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 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.

The prototype Gateway system is now available [14] and the fully functional version is expected to be deployed by November 1999.

References

1. G. C. Fox, W. Furmanski, "High Performance Commodity Computing" in "The Grid. Blueprint for a New Computing Infrastructure", a book edited by C. Kesselman and Foster, Morgan-Kaufmann Publishers, Inc., San Francisco, 1998;
   G. C. Fox, W. Furmanski, T. Haupt, "Distributed Systems on the Pragmatic Object Web - Computing with Java and Corba"
2. D. Bhatia, V. Burzewski, M. Camuseva, G. C. Fox, W. Furmanski, G. Premchandran, "WebFlow - A Visual Programming Paradigm for Web/Java based coarse grain distributed computing", Concurrency Practice and Experience, 9, 555-578 (1997) (http://tapetus.npac.syr.edu/iwt98/pm/documents/)
3. E. Akarsu, G. C. Fox, W. Furmanski, T. Haupt, "WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing", in proceedings of Supercomputing '98
4. Java Grande Forum, home page: http://www.javagrande.org
5. Globus Metacomputing Toolkit, home page: http://www.globus.org
6. NPACI HotPages home page: http://hotpage.npaci.edu
7. Object Oriented Concepts, Inc., ORBacus SSL, home page: http://www.ooc.com/ssl/
8. Adiron home page: http://www.adiron.com
9. S. S. Mudumbai, W. Johnston, M. R. Thompson, A. Essiari, G. Hoo, K. Jackson, Akenti - A Distributed Access Control System, home page: http://www-itg.lbl.gov/Akenti
10. http://www-fp.mcs.anl.gov/~gregor/datorr/datorr.html
11. UNICORE: Uniform Access to Computing Resources, home page: http://www.fz-juelich.de/unicore
12. WebSubmit: A Web-based Interface to High-Performance Computing Resources, home page: http://www.itl.nist.gov/div895/sasg/websubmit/websubmit.html
13. ARCADE, home page: http://www.icase.edu:8080
14. Gateway project home page http://www.npac.syr.edu/users/haupt/WebFlow/demo.html