

WebFlow: A Middle-Tier Solution to Support Web-Based Environments for Distributed, High-Performance Computations

(Extended abstract)

Erol Akarsu, Geoffrey Fox, Tomasz Haupt
Northeast Parallel Architectures Center at Syracuse University

Abstract

In this paper we present design and discuss implementation issues of the middle-tier needed to build a Web-based environment for scientists and engineers that enable secure and seamless access to high-performance resources. We choose a CORBA-based solution, with server-side objects implemented in Java. The WebFlow system can be regarded as a high-level, Web-based user interface and job broker for Globus metacomputing toolkit.

Introduction

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. It offers numerous services, from on-line shopping and banking, to collaboratory environments used for distance training and sharing scientific data.

Our goal is to build on these emerging Web technologies, standards, and commodity software components and develop a Web-based environment for scientists and engineers that enable secure and seamless access to high-performance resources. 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, 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 by which a user can understand, define, and analyze scientific problems.

This paper discusses the requirements and implementation issues of the middle-tier needed to build such a system. In our design we build on our experience of applying the preliminary versions of WebFlow system to real life applications, as described in our earlier papers [1,2]. It is important to note that we use Globus metacomputing toolkit[3] to provide access to high performance resources in tier 3. Conversely, the WebFlow system can be regarded as a high-level, Web-based user interface and job broker for Globus.

The Model

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, which are object containers. This idea can be easily extended to non-graphical components and is implemented as 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 event listeners.

For distributed applications, we need a mechanism to transport the events across address spaces, or a distributed object model. There are several competing models available. Among the most known are OMG's CORBA, Microsoft's DCOM, Sun's Java RMI, and WWW consortium's DOM. Each of them has its merits and limitations. We believe that any of these technologies can be used to implement the WebFlow. In this paper, we describe a CORBA-based solution, with server-side objects implemented in Java, and which resembles model of Sun's Enterprise JavaBeans.

WebFlow provides an environment in which the user can create an arbitrarily complex hierarchy of containers and objects in the middle-tier. The web-based client tier provides tools for visual composing and distributing the hierarchy. The middle tier objects can encapsulate executable codes, but typically they act as proxies to services rendered by the back-end, such as high-performance computing or database access.

It follows that within WebFlow environment, building a distributed, high-performance meta-application is a process similar to that of building a distributed applet, as shown in Figure 1.

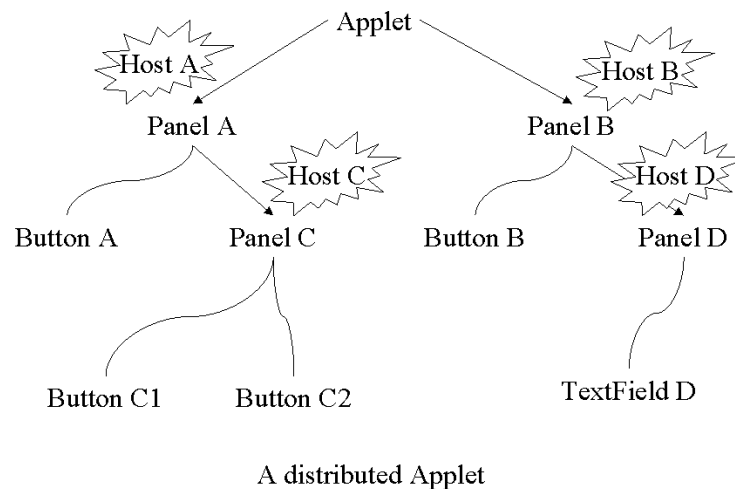


Figure 1. A hypothetical distributed applet. Each panel (a container) of this applet is placed on a different host.

WebFlow containers (also referred to as contexts) serve many different purposes. A WebFlow user creates the user context, which is a container for all his or her applications. The application context holds the application's components or modules. And the module itself can be hierarchically composed of subcomponents. This way a WebFlow server, which is the root of the context hierarchy, may host simultaneously many users, and each user may run several independent, composite applications. Figure 2 shows an example of the WebFlow object hierarchy.

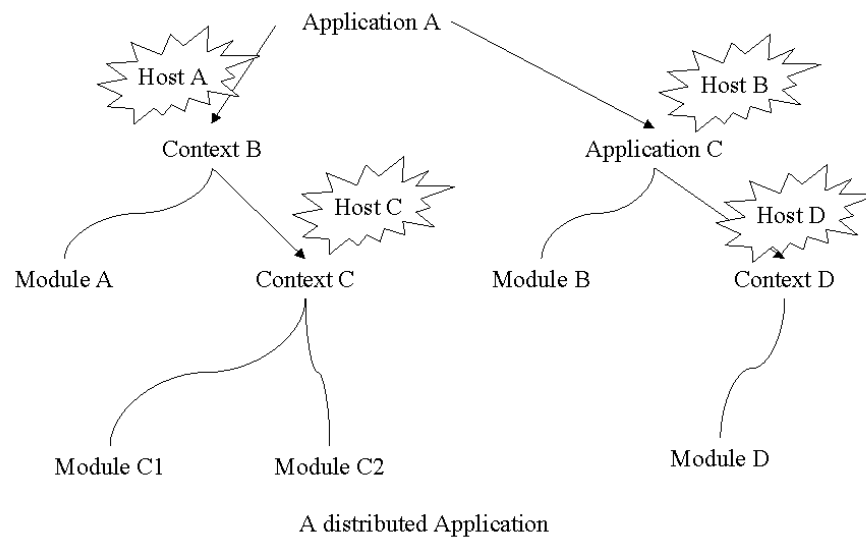


Figure 2: A distributed WebFlow application

WebFlow Servers

The WebFlow system is given by a network of WebFlow servers. One of the servers, called the Master, plays a special role. It is the root of the WebFlow hierarchy of objects, and serves as a WebFlow point of presence (POP). Typically, it is accompanied by a secure web server (httpd). The user starts a WebFlow session by connecting to the web server. After a mutual authentication, the authorized user is returned the reference to the Master, and applet to control WebFlow objects. By placing the Master and the web server on the same host, we avoid some of the Java sandbox restrictions: the applet can connect to the Master.

All other WebFlow servers act as “slaves”. The slave registers itself to the Master and usually runs on a host different than that where the Master is running. Slaves can be introduced into the system dynamically and the Master instantly updates the list of the available Gateway servers.

Each WebFlow server runs as a separate process (with associated interface repository and module factory), and maintains its own object hierarchy. The user can add components, contexts and modules, to the context defined by the server using the context methods such as `addContext` and `addNewModule`.

WebFlow Component Proxy

The Master is the root of the WebFlow hierarchy of objects. Furthermore, 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 association of the distributed components. In our current implementation we generate proxies for all components, including the local objects. This symmetric implementation allows extending the functionality of proxies. Among the most interesting is capability of logging, tracking and filtering all messages between components in the system. We use these capabilities to implement fault tolerance, security and transaction monitors, as well as for debugging purposes.

Interactions of components

An application is made of modules that exchange information between each other through the event notification mechanism. An event is a CORBA object itself, which 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 the connection, we mean here an association of a source object, which fire event, and target object whose registered method is called as the response to the event.

Since the WebFlow modules are developed independently of each other, and connected only at runtime, we need a dynamical mechanism for event binding. This functionality is offered by the CORBA event service. 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 the 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 a WebFlow context. Each entry of the table contains 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 source object fires event, it is intercepted by its parent context and called the

registered target method immediately. In pull model, captured event is kept inside translation table until target object wants to take it by explicitly calling “pull” method on its parent context. This dynamical event binding is achieved by using CORBA dynamic invocation interface (DII) and dynamic skeleton interface (DSI). In Figure 3 we illustrate our event model.

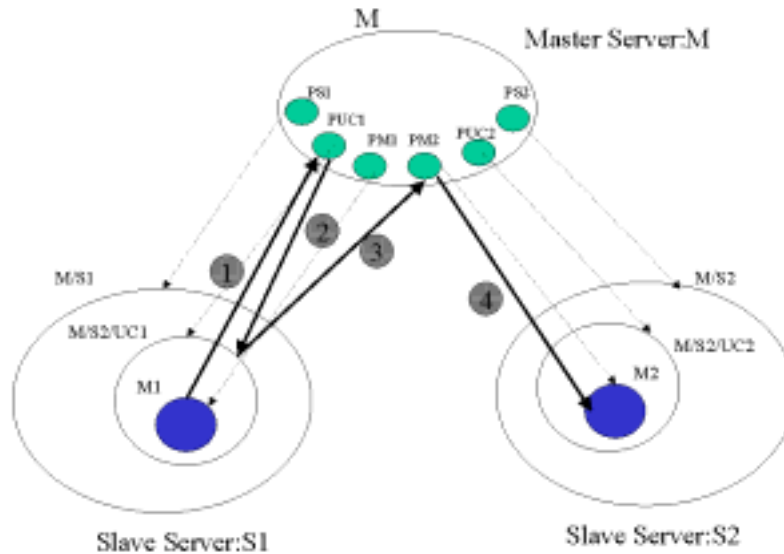


Figure 3. WebFlow event model.

In this example, we have the Master server M, and two slave servers, S1 and S2. Module M1 in context UC1 in S1 fires an event e to invoke a method m of module M2 placed in context UC2 in S2. The green blobs inside the Master represent WebFlow components' proxies maintained by the Master. Thin dotted arrows shows relation between the proxies and the actual objects.

1. Module M1 fires the event. It is intercepted by 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 to you 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 overweight possible shortcomings. Firstly, reference to the proxy of the target module instead of the module itself leads to the module location transparency. Secondly, when the event is fired by the Web client, this is the only method to access the remote module. Finally, as discussed above, sending events through proxies opens opportunity for filtering events.

WebFlow applications

The WebFlow system has been successfully applied to provide a web-based interface for two class of applications, one developed within NCSA Alliance Team B effort, and the other for CEWES MSRC, within DoD Modernization Project, as described in our previous paper[2]. Currently, it is used in the Gateway project[4], and publicly available[5].

[1] E. Akarsu, G. Fox, W. Furmanski, T. Haupt, "WebFlow - High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing", in proceedings of Supercomputing '98, Orlando, November'98.

[2] T. Haupt, E. Akarsu, G. Fox, "WebFlow: a Framework for Web Based Metacomputing", in proceedings of HPCN Europe'99, Amsterdam, the Netherlands, April'99.

[3] Globus home page: <http://www.globus.org>

[4] Gateway project home page: <http://www.osc.edu/~kenf/theGateway>

[5] WebFlow home page: <http://www.npac.syr.edu/users/haupt/WebFlow>