APPLET-BASED DISTRIBUTED COMPUTING ON THE WEB

JAMES F. CARLSON, DAVID V. ESPOSITO, NATHANIEL J. SPRINGER, DAVID FINKEL AND CRAIG E. WILLS
Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609
USA Telephone: +1 (508) 831-5416 Fax: +1 (508) 831-5776
E-Mail: [dfinkel | cew]@cs.wpi.edu

Abstract

We describe a client-server framework that allows chunks of large computations to be distributed among many computers on the Internet. We describe a framework of Java classes and Internet servers that allow a user on the Web to participate in the distributed computation. The executed chunks can be either be strictly computational or may involve exploration of the Web itself. The system is designed so that even non-technical users can participate.

Keywords: Distributed Computing, Java Applet

1. INTRODUCTION

Several different paradigms have been proposed for distributed computing on the Web. Our paradigm is based on a distributable Java class and a Java helper applet downloaded by a Web user [3]. The applet then downloads and executes the distributable class. This work builds on our own previous work, described in [5], in which the Helper program was written as a Java application instead of an applet. The use of an applet in our new work allows programmers to build distributed applications that can be broken into small sections, or chunks, and conceivably executed by any user on the Internet through a Java-enabled browser. The work creates a framework for an electronic marketplace where computations in need of solution can be matched with otherwise idle computers to do the work.

In addition, this work begins to explore a new type of distributed computation where, instead of being strictly computational in nature, a chunk may involve exploration of a piece of the Web. This type of computation raises exciting possibilities for Web crawler applications, data mining applications, and distributed performance monitoring applications.

The first component of our system is a distributable class. We have designed a Java abstract base class, called DistributableClass, which includes several abstract methods. In order to implement a distributed computation, a programmer must write code for a Java class which extends the class DistributableClass. In particular, the programmer must implement the abstract methods in DistributableClass. We call the class written by the programmer a distributable class or a distriblet.

69
The second component of our system is the Distribution Server. The Distribution Server is a CGI script located on a Web server on the Internet. A Web user who wishes to participate in the distributed computation contacts this Web server and downloads a Java applet called the Helper Applet. The Helper Applet contacts the Distribution Server to identify a computation to participate in.

The third component of our system is the Computation Server. The Computation Server is a program which may be run on the same machine as the Distribution Server or on a different machine. After the Helper Applet downloads the distriblet from the Computation Server, the Helper Applet calls pre-defined methods in the distriblet to download arguments from the Computation Server; the arguments define what portion of the distributed computation this Helper Applet will execute. The Helper Applet then calls other pre-defined methods in the distriblet which cause the execution of the computation and the return of results to the Computation Server.

This approach has advantages both for the programmer of a distributed computation and a user wishing to share an otherwise idle machine. The programmer of the distributed computation simply has to write a distriblet to perform the computation, and to define the sets of arguments which will be downloaded to different users. The overhead of the distributed computation is handled by pre-written programs: the Computation Server, the Distribution Server, and the Helper Applet.

The user of an otherwise idle machine is freed from the details of downloading specialized code to perform a computation or locating a computation to share in performing. Rather, the user simply visits the Web page of the well-known Distributed Server and selects a computation to perform.

In this paper we discuss details of our work. Section 2 discusses related work in which users download programs in order to participate in Internet-based distributed computations. Section 3 provides additional details on the design and implementation of our system. In Section 4, we discuss distributed applications we have implemented using our system. Finally, Section 5 summarizes our work and points at future directions.

2. PREVIOUS WORK

We have identified a number of other efforts that provide a framework for distributed computing on the Web. Some employ special-purpose software which a user must download from the Internet and install on their own computer ([2], [4], [7]). This is in contrast to our approach, which employs a general-purpose distributable class.

The project most similar to our current project is the POPCORN Project [1]. POPCORN's goal is to provide any programmer on the Internet with a simple virtual parallel computer. In order to motivate user participation, a market-based payment mechanism for CPU-time underlies the system. The system is implemented as a Java applet that provides a wide-scale safe participation of remote processors. There are three distinct entities in the POPCORN
system [1]: CPU-time buyers, CPU-time sellers and a market which serves as a meeting place for buyers and sellers.

The POPCORN programming paradigm used by the buyer program spawns off subcomputations, termed computelets. The POPCORN system automatically sends these computelets to a market that then forwards them to connected CPU-time sellers who execute them and return the results. The process of matching buyers and sellers in the market is dynamic and often results in a payment from the buyer to the seller. In comparison to our work, the economic aspects of such a paradigm are more fully explored in the POPCORN project, but our work has begun to explore new types of distributed computations that can be implemented with this paradigm.

3. DESIGN AND IMPLEMENTATION

3.1 The Java Class DistributableClass

The Java class DistributableClass has five abstract methods; that is, methods that are named but not implemented. These methods are getArgs, sendArgs, run, sendResults, and getResult. The programmer of the distributed computation must write a class which extends DistributableClass and which provides implementations for these five methods. This structure was successfully used in our previous work [5] and is unchanged for this work.

The getArgs, run, and sendResults methods are used by the Helper Applet to execute the distributed application.

The purpose of the getArgs method is to receive the arguments that this execution of the distributable class will use. By allowing programmers to write their own methods, the programmers have the flexibility to specify the number and type of the arguments to be used. By using a separate method to download the arguments, instead of having the argument values as part of the class definition, the Helper Applet is able to perform many pieces of the distributed application without downloading many classes. The distributable class is downloaded only once, and then the chunks of the distributed application can be executed sequentially by making multiple calls to getArgs.

The run method actually performs the piece of the distributed application, acting on the arguments provided to getArgs. Since the distriblet is executing inside of the Helper Applet, it is constrained by the general security restrictions placed on Java applets. However, it is possible to modify these security restrictions to allow the applet to contact Web sites other than the Web site the applet was downloaded from. This security modification allows our scheme to be used for Web-based applications such as data mining. We discuss these security restrictions further in Section 3.

The sendResults method returns the results of each chunk of the computation to the Distribution Server. Again, the programmer has the flexibility to specify the number and type of values to be returned.
The `sendArgs` and `getResult` are used by the Computation Server to provide arguments to the Helper Applet and to receive the results returned by the Helper Applet.

### 3.2 The Distribution Server

The purpose of the Distribution Server is to keep a list of all available jobs and their corresponding Computation Servers, to store the distributed class files, and to respond to requests from the Helper Applet and Computation Server. The Distribution Server consists of a CGI script located on a Web server. Requests are made to the Distribution Server by Computation Servers and Helper Applets via HTTP requests.

The Distribution Server responds to the following types of requests: `AddJob`, `DeleteJob`, `GetJobList`, and `GetNextJob`.

A Computation Server makes the `AddJob` request when it wishes to register a new computation with the Distribution Server. The Computation Server must provide the hostname and port number of the Computation Server, a unique job identifier, the file name of the distributable class that the Computation Server is providing, and the actual contents of the distributable class (Java bytecode).

The `DeleteJob` request is made by a Computation Server when a computation has been completed. The Computation Server must include the unique job identifier with this request.

The `GetJobList` request is made by the Helper Applet in order to receive information about all of the computations registered with the Distribution Server. The Helper Applet then displays all of the available computations to the user and allows the user to select a distributed computation to participate in. The benevolence rating and the cost figure are returned to the Helper Applet with the results of this request.

The `GetNextJob` request is sent by the Helper Applet prior to the start of the distributed computation. This request gets the most current information from the Distribution Server before beginning the distributed computation.

### 3.3 Computation Server

The task of the Computation Server is to respond to requests from Helper Applets and serve chunks of the distributed computation to them. The Computation Server class created in this project is generic and must be instantiated and initialized using an application written by the developer of the distributed computation. Parameters passed to the Computation Server include the computation name, computation ID, computation benevolence and cost values, the port of the Computation Server, and the address of the Distribution Server. In order to register the computation, the Computation Server sends the `AddJob` request to the Distribution Server via HTTP POST.

When the Computation Server begins, a socket is created that listens for new connections from Helper Applets. Once a Helper Applet connects to the socket, a connection thread is started that handles all of the communication with the Helper Applet. The Computation
Server supports the following requests from the Helper Applet: GetWork, GetClassName, GetData, and SendResults.

- **GetWork** – This is a request for the Computation Server to send the next available chunk of the computation to the Helper Applet. If the computation has been completed, the server returns the NoWork response to the Helper Applet. If there is work remaining, the name of the distributable class and the ID number of the next chunk to be executed are returned. If the Helper Applet does not have a copy of the required class, then the Helper Applet must contact the Distribution Server for a copy of that class’ bytecode. The reason the class is stored on the Distribution Server instead of the Computation Server is that the applet security restrictions do not allow applets to execute classes unless they are downloaded from the same Web server the applet originated from.

- **GetData** – This request is sent from the Helper Applet asking for the initialization data for the chunk. This data is used by the Helper Applet to set the initial state of the chunk before starting work on it. The Computation Server returns the SendData response along with the initialization data. The initialization data is retrieved from the chunk and sent to the Helper Applet via the sendArgs method within the chunk. The data is received on the client side via the use of getArgs method.

- **SendResults** – This request is sent from the Helper Applet when the chunk has finished. The Helper Applet also sends the ID of the chunk that it has completed and the userID of the user who completed the chunk. If the chunk has been previously completed, the Computation Server returns the Job Done response to the Helper Applet. However, if the chunk is still incomplete, the Computation Server returns the GetResults response to the Helper Applet. The Helper Applet then sends the execution results of the chunk to the Computation Server.

The end of the computation is signaled by the call of the shutDown() method of the Computation Server. This method is called by the application written by the distributed computation developer. During execution of the shutDown() method, the Computation Server contacts the Distribution Server to unregister the computation. Included in this request is a list of all users and the number of chunks of the computation that they completed. This information is gathered during the distributed computation as Helper computers return results to the Computation Server.

### 3.4 Helper Applet

The Helper Applet is the module of the distributed computation framework that the end user interacts with. This module allows the user to select which computation they wish to participate in. The behavior of the Helper Applet is separated into two distinct phases, the selection phase and the execution phase.

The selection phase begins when the user connects to the web page containing the Helper Applet. The Helper Applet loads and makes a request to the user asking for permission to contact third party hosts. Under its normal security scheme, an applet may not contact any computer other than the computer it originated from. In order to allow the Helper Applet to contact each Computation Server, the applet must be digitally signed and the user must give
permission to contact third party hosts. Currently, Netscape Navigator 4.06 (and above) employs a per-applet security policy. An API is provided by Netscape to explicitly ask the user for permission to remove a security restriction from the JVM security manager [6]. Netscape replaces the Security Manager employed by the JVM with its own security manager known as Privilege Manager. If the user agrees to the request, Privilege Manager is set to allow the action to be granted at any point during the applet execution. If the request is denied, the applet must have a contingency plan that may include terminating the applet. Alternately, running the distribution and computation servers on the same machine would avoid the need for permission to contact third-party hosts if the computation did not need to contact other hosts.

Regardless of whether or not the user chose to grant the permission, the Helper Applet continues initialization and the list of available computations is displayed to the user. However, if the user chose to deny the request, any attempt to contact third-party hosts will fail.

The user is now presented with several options to select computations to contribute to. The options are Choose Each Mode, Select Attribute Mode, and Continuous Mode. Choose Each Mode allows the user to select from the list of available computations. Select Attribute Mode allows the user to select computations based on the benevolence ranking or the cost figure.

In Continuous Mode, computations are randomly selected from the list of available computations. Once all the chunks of a computation are completed, another computation is selected. The Helper Applet continues to complete computations until the user cancels the action or there are no more computations available on the Distribution Server. This mode is ideal for execution when a computer is idle, at night for example, because it requires no user interaction after the initial computation is started.

When the Helper Applet is initially loaded, it connects to the Distribution Server and retrieves the list of available computations by sending the GetJobList request to the Distribution Server. This list is stored in the applet and the name of each computation displayed to the user. At this point, the user chooses a selection option and optionally a computation from the computation list. Depending on the options selected by the user, the applet selects a computation to download.

The execution phase begins when a computation is selected. The user display during the execution phase includes details on the current and completed computation chunks, and a cancel button, which allows the user to cancel the execution of the current chunk.

The Helper Applet now starts a Computation Client thread to handle all network communication with the Computation Server. The Computation Client first connects to the Computation Server associated with the computation selected in the selection phase. After connection, the Computation Server sends its name to the Computation Client. This is to ensure that the Computation Client has contacted the correct Computation Server. Once the connection is established, the Helper Applet sends the GetWork request to the Computation Server. If the Computation Server returns the NoWork condition, the connection to the Computation Server is closed, the Computation Client is terminated, the execution phase
ends, and the Helper Applet returns to the selection phase. However, if work is available, the Computation Server returns the class name of the distributable class and an identification number of the chunk to be executed. If the Helper Applet does not have the bytecode for this distributable, the Helper Applet must download the class from the Distribution Server.

After the class is downloaded and instantiated, the Computation Client requests any arguments associated with the chunk from the Computation Server. The Computation Client reads the data from the Computation Server via the `getArgs()` method of the newly loaded distributable class. The `getArgs()` method uses the data from the Computation Server to set the initial state of the computation. After the initial data is received, the chunk is executed by calling its `run()` method. After the work on the chunk is completed, the Computation client returns the results to the Computation Server via the `sendResults()` method of the distributable class. After the chunk is complete and the data has been uploaded to the Computation Server, the Computation Client requests another chunk from the Computation Server and repeats the process. If no more chunks are available, the Computation Client disconnects from the Computation Server and closes. At this point, the status frame closes, the execution phase ends, and the Helper Applet returns to the selection phase. If the user selected the Continuous Mode option, the Helper Applet reconnects to the Distribution Server, retrieves a new computation, and starts the execution phase again. If any other option was selected, the user may select a new computation.

4. APPLICATIONS

In order to test the functionality of our system, we implemented three distributed applications.

The first, Number Accumulator, simply adds the numbers from 1 to N. The second, Mandelbrot, performs a distributed computation of a Mandelbrot set. The third, Web Crawler, is an example of Web-based applications that could be implemented using our system. In this sample application, each chunk is given a URL. The Web page specified by the URL is downloaded, and the number of links on the page is determined.

We did test runs of these application using various number of client machines and different selection modes provided by the Helper Applet. In one of the tests, seven client machines connected over the Internet were used, and the Helper Applets were run in continuous mode. Our tests demonstrated that it is straightforward to implement a distributed application using our framework, and that the various components functioned correctly.

5. CONCLUSIONS

We had three major goals for our project: to create a system that allowed users to participate in a distributed computation regardless of platform or operating system; to make it easy to use; and to implement a server that provided access to all available distributed computations.

We achieved the goal of platform independence through the use of Java. By writing the Helper software as a Java applet, anyone who has access to a Java-capable web browser can contribute to distributed computations. A current limitation of platform independence is that
the Helper Applet must run inside Netscape Navigator, but in the long-term we expect that other browsers could be used.

We achieved the goal of ease of use by writing the Helper software as a Java applet. No user intervention is required to start the applet, other than connecting to a Web page. The Helper Applet handles all of the communications between the user’s computer and the Computation Servers; therefore, the user does not need to understand the underlying framework in order to contribute to a computation. The use of an applet is an improvement over our previous design, reported in [5], in which the Helper software was provided as a Java application and required more technical expertise on behalf of a user.

We attained the third goal of having a central contact point by implementing the Distribution Server. It contains a list of all of the Computation Servers and their corresponding computations that are available to a user. Thus, the Distribution Server provides a single point of access for the Helper Applet. It also provides a framework for keeping track of the work completed by each user and eventually can be used to implement a method for paying users for participating in a distributed computation.

We believe our work not only demonstrates the feasibility of this distributed computing paradigm, but also provides a framework for additional work in this area. The possibility of computational chunks that do work on a piece of the Web itself provides obvious opportunities for interesting and useful distributed applications. We have only begun to explore this area with this work. In addition, investigation of the economic implications of this paradigm are important.

REFERENCES