Analysis of Java Client/Server and Web Programming Tools for Development of Educational Systems

Tomasz Müldner
solid@dragon.acadiau.ca
Jodrey School of Computer Science, Acadia University
Wolfville, NS, Canada B0P 1X0

Abstract: This paper provides a thorough analysis of old and new programming tools for development of client/server programs, in particular Web based programs. The focus is on development of educational systems that use interactive shared workspaces, to provide portable and expandable solutions.

0. Introduction

Computers have been used in education for many years (Alessi & Trollip 1991; Colbourne & Cockerton-Turner 1989; Maurer & Tomek 1990; Ketinger 1991; Maurer 1988; Maurer & Tomek 1990, Norman 1996), however they have been mostly used in labs, rather than in classrooms. During last several years, various universities have introduced "electronic" classrooms, see for example (Mühlhäuser 1996, Müldner & Nicholl 1996, Shneiderman 1995) in which each student has access to her or his computer, either by using a mobile notebook computer (Wake Forest University 1997, Acadia University 1997) or using a stationary computer (North Carolina). In these classrooms, computers are usually networked allowing students to access the Internet.

Various organizations started to develop integrated educational systems which provide numerous facilities, including access to various kinds of information about courses, such as course descriptions, computerized course materials, discussion groups, chat rooms, on-line tests, etc. The most notable examples of these systems are ACME (ACME 1997) and WebCt (Goldberg 1996). Some software tools support specific disciplines; for example physics courses have benefited greatly from the use of computer based presentations to simulate experiments; see (Holmes & Porter 1996). However, most of the existing systems do not support collaborative and interactive sharing of tools. To support groups of people working towards a common goal, one could design computer-based systems that provide an interface to a shared workspace. Indeed, such systems have been extensively researched; see e.g. (Benford, 1994, and Grudin, 1994).

In this paper, we analyze various software techniques that can be used to develop educational systems, in particular to implement shared workspaces. Our main concern is a development of portable, expandable and maintainable software. We are also interested in efficiency issues; that is building systems that can stand a heavy load of interactive users. Our paper starts with a short description of relevant terms and provides references for those readers that would like to expand their understanding of these terms. We present a traditional approach that uses dynamic HTML pages generated by CGI scripts, and describe drawbacks of this approach. Next, we discuss an object-oriented approach to development of client/server programs, using Java; in particular development of distributed systems with the help of tools such as sockets, Remote Method Invocations (RMI), and Corba. We compare some traditional and new Web servers. Finally, we make some recommendations on which tools should be used for development of educational systems; in particular shared workspaces.

1. Client/Server Paradigm and the Web

A *centralized* application is an application which runs on a single machine. A *distributed* application can consist of a number of components located on several networked computers. For the latter type of an application, a *client* is a component of the application that makes requests to another component of the application, called a *server*. Note that a client and a server terms refer to a role that the software component plays rather than to the component itself; i.e. a client can request information from the server and then the server may ask a client for other information, at which time the server and the client roles are reversed. A useful example of this situation is a technique called *callback*, where a client who makes a request registers with the server, and thus the server can call back the client. When the server calls back the client, it becomes the client.

For each active software component there is a *process* that runs this component (in general, a process is a "program in execution"). For example, a client process runs the client component.

The well known example of a client/server application is any Web browser (such as Netscape) and a Web server (in the next section, we show how a Web browser and server communicate). Here, the roles of the client and server are fixed; the client Web browser always requests information from the Web server. For such systems, it makes sense to use a term a server computer; which is a computer running a server process. The HTML pages, see (HTML 1994), are stored on the Web server's computer and are downloaded to the client machine when requested by the Web browser.

For two or more processes to communicate and exchange information, there has to be some kind of a *protocol* which is followed. For example, users running a talk facility often put "o" (for: over) and the end of each message, and "oo" when they are to terminate the communication. While various distributed programming systems use user-defined ad-hoc protocols, the Web uses a standardized protocol, called HTTP, see (HTTP, 1998).

What is involved in client/server *interactions*? A client may request simply some data; for example a Web browser requests text, images, etc. from the Web server. Then, a client downloads these data and uses them, for example to display an image. This type of a client is often referred to as a *null* client; i.e. a client which does no processing. The client can be more active and request an executable program, which after being downloaded, will be executed on the client. For the Web, an applet is an example of an active client; when it appears in the HTML page, then it is downloaded from the Web server's computer and executed on the client's machine. Now, not only the information but also the execution is distributed, which also distributes a total *load* related to an application in question.

At this point it may be relevant to ask whether it is better to use standalone client applications or applets which reside in HTML Web pages? The major advantage of an applet is that most users have a Web browser already installed, and for various reasons they may be unwilling to download and install another application. Also, there is a lot of functionality provided by Web browsers, such as displaying pages that include text, graphics, etc., and this functionality would have to be implemented in user-defined applications. (Note, however, that new java beans may provide this functionality and be easily incorporated into existing applications.)

Another essential issue in a client/server paradigm is the *persistence* of information; when a process produces some data, is this data persistent that is saved when the process terminates? This question should be discussed in the context of security: fetching or saving data means accessing a local file system and this activity clearly creates a security risk. Other examples of such activities contacting any site other than the server's site. While the fixed server, such as a Web server is at a well-known location, a client may come from an unknown destination, and may disguise its identity or even be an impostor. The client may decide to trust no one, and reject any attempt to start the above activities, to trust a selected site, or to trust everybody. Some browsers, Netscape included, make this decision for the client and trust no one (although with some programming it is possible to change this). Other browsers, for example Hot Java, let you choose your trusted sites. The HTTP protocol is *stateless*; when one client interaction is terminated and the next one starts, the server doesn't know about the previous history. The only exception to this rule are the so called *cookies* which are usually short files written by the Web server to the client's machine (the client may be asked if she or he agrees to receiving a cookie and may decline it). Sometimes, Web pages include invisible information used to remember the previous state (used for example, when a client performs an on-line transaction and responds to a series of questions). Therefore, a typical way of making data persistent is to save them on the server's computer; using Common Gateway Interface, CGI. Below, we describe this technique in details.

CGI involves programs that can be invoked by the Web server. Here, we consider one commonly used scenario that involves an invocation of a CGI program; using a POST request specified by the HTTP protocol. When a Web browser (client) submits a POST request (this request specifies a CGI program, for example P), the Web server executes P. Now, P may need some input data. The Web server makes arrangements so that P gets

its input from the data that are also provided in the POST request. This way, a CGI program can process input data and write some results to the server, for example store information in a database. In a similar way, the Web server redirects the output from a CGI program; if P outputs any data, then this data is sent back to the browser. If this data is in HTML format then the browser will interpret them in standard way, that is it will display a page based on the HTML code. This technique can be used to create *dynamic* HTML pages that are created on the fly by the server; unlike static pages that are represented by files, stored on the Web server's side. Dynamic HTML pages are more interactive and useful for security reasons; the client won't be able to access these pages unless she or he starts at a place where a proper authentication can take place. On the other hand, creating dynamic HTML pages may significantly increase the load on the server (see the discussion below). Executing CGI programs is a special case of server-side includes; i.e. programs or scripts that are executed on the server as a result of a special HTML tag.

2. Load

In a distributed environment, with multiple clients and servers it is essential to consider *load balancing*: if all the work is done on the server then clients have to suffer from possibly long delays. In particular, for a single Web server and a number of Web browsers (clients), a Web server can be easily overloaded if the number of hits is too high. Most Web servers that are currently used handle CGI requests in an inefficient way, starting a new process to invoke every new CGI program. There are several possible solutions to problems described above: (1) One can use multiple servers and a specialized hardware which distributes client requests and maintains consistency between clients; see http: //www.zdnet.com/pcmag/features/loadbal/open.htm for details; (2) Load is distributed between clients and servers. This distribution has to take place at run-time, that is we need a dynamic load balancing; and (3) More efficient way of handling CGI programs is used, for example using servlets (see below).

3. Communication Techniques: Sockets

Any two processes running on two networked computers can communicate using sockets (here, we assume that a network uses the TCP/IP protocol). *Sockets* are communication end-points, see (Stevens, 1994). There are two kinds of sockets: stream sockets and datagram sockets. *Stream sockets* are called connection-oriented because their use resembles a telephone connection: one process (a server) has to listen for a connection and the other process (a client) uses the first process' location (an IP address), and a port number (an integer value) to connect to the server. Thus, an IP address is like the phone number of a switchboard and a port number is like an extension. Once a connection is established, both processes can exchange information in both directions until one of them decides to close the connection (thus, a server process is really a server only when communication is to be established; afterwards it can play both roles; the client's and the server's). Note that in this case, we don't really need to use callbacks; the server has an open line of communication with the client.

Data received over a socket connection can be interpreted by the receiving process as a request to perform some operation; this way a sending process can indirectly "invoke" this operation by the receiving process. Of course, both processes need to follow a certain protocol known to both of them, for example to verify whether or not the required operation has been successfully completed. Stream sockets are reliable, that is any errors are reported and data will be automatically retransmitted if necessary.

Besides stream sockets, there are also *datagram sockets*, for which the communication resembles sending a package (called a datagram) by mail: a client process sends data and provides the address (an IP address and a port number), but no fixed communication link is established and the communication is not reliable, that is the package could be lost. Again, we don't need callbacks because a datagram contains a return address, i.e. an address of the sender, which can be used by the server to call back the client.

In the remainder of this paper, we will only discuss stream sockets and will refer to them simply as *sockets*. The *standard* Web server and the browser use sockets to communicate. The Web server is represented by a so-called daemon process (called an httpd, which stands for an HTTP daemon); i.e. a process which is used to accept requests on behalf of other programs, and then forward these requests to these programs (something like a telephone switching board). HTTPD listens on a selected port (usually port 80), and the Web client

connects to this port. When a connection with the client is established, the daemon starts a separate process to service this client and returns to listening for further connections (to avoid an expense of starting a new process, the server may maintain a pool of available processes, and use any available process, and start a new process only if the pool is empty). For any component of an HTML page, such as an image, a separate socket connection is established in order to transfer this component. As we mentioned above, a standard security requirement is that a Web browser can only communicate with the Web server from which the communication originated, in particular it can only open socket connection with this server and not with any other computer. If the latter option is required, specialized servers are required (see below).

4. Choice of a Programming Paradigm and Language

Programs that implement an educational system are large and complex; therefore they should be developed following standard software engineering techniques, so that not only the code but also the design can be re-used (thus, the use of design patterns, see (Arnold and Gossling, 1998)). Therefore, it appears obvious that an object oriented approach is the only approach that is currently acceptable. Among various object oriented programming languages available, we recommend Java because of the following reasons. First, Java supports threads and also garbage collection, both centralized and distributed. Thus, the programmer does not have to worry about memory leaks; a major issue when using a language such as C++. Second, Java is architecture neutral. Third, Java is more than a programming language, it is a system that has a number of components and built-in techniques such as Java Beans (for more information, see (Javasoft, 98)). Finally, Java supports various communication and distribution techniques, and one does not have to resort to using foreign libraries. Java can be used to develop client applications, server applications and also CGI programs. Currently, most CGI programs are implemented in Perl, which is a low-level, interpreted, procedural language.

5. Techniques to Distribute Execution

Using sockets for remote execution is rather cumbersome because it requires designing a protocol and coding to this protocol. In addition, it is hard to design expandable protocols that would accommodate any future needs. There are two techniques which can be used to directly invoke remote actions: Remote Method Invocation (RMI); and Common Object Request Broker Architecture, CORBA. Below, we briefly describe each of these techniques.

RMI, see (Javasoft, 1998) is a technique specific to Java; i.e. it can be used if we have two machines running Java Virtual Machines (however, these machines may run different operating systems). A server process exports an implementation of an object, which may support a number of methods. A client process can invoke these methods. This, from the programmer's point of view a call ref.foo(arguments) looks the same no matter whether the object ref is on the client's machine, or it is on a remote machine. For the implementation of this technique, exported objects have to be known to a registry, and a dedicated daemon process, called a registry process, listens on a socket port (typically, port 1099). The client connects to this registry process and then remote invocations made by the client are executed on the server's machine. Callbacks are easy to implement when using RMI; but some browsers such as Netscape require additional coding to enable special permissions. Unfortunately, the RMI programmer doesn't know whether a remote object is really remote; i.e. stored on a different machine, or it happens to be stored on the local machine (in the same address space) and must take extra precautions to compensate for the fact that RMI uses different modes of parameter transmission in these two cases; for details see (Brose and al., 1997).

Corba is an abstract specification, see (OMG, 1995) and its implementation (called *ORB*) provides a more general technique than RMI. Corba is language and operating system independent. Both, the client and the server program can be developed in any programming language, although Java seems to be the easiest to use and it is becoming more and more popular, see for example OrbixWeb, (OrbixWeb, 1997). Unfortunately, due to a very general scope, Corba programming is not easy. For example a Java programmer must additionally know details of mapping of a general specification language, called an IDL into Java, as well as many other technical details. Again, Corba can easily take advantage of callbacks, for an example see an implementation of a chat room in OrbixWeb, (OrbixWeb, 1997). Two machines running Corba use a protocol called IIOP (Internet Inter-

ORB Protocol). It is possible that there will be future implementations of RMI on top of IIOP, which will allow the programmer to benefit both from the simplicity of Java programming and the power of Corba connectivity.

6. Special-purpose Servers

Specialized servers are basically daemon processes that can be used for various reasons. For example, they can be used to deal with security restrictions imposed by Web browsers. If a Web browser wants to establish a socket connection with the socket located on a machine M which is different from the Web server's machine W, then we need an additional server running on W. This *relay* server will accept socket communications on behalf of M, and then forward any messages to M (there are no restrictions on operations of the relay servers). Another type of a specialized server running on the same machine as the Web server can be used to provide various tasks for which a Web server was not designed for. For example, if you want to develop a live chat room which at any given time accepts no more than 10 users, or which accepts only registered users, you may wish to use a *gateway* server, which will accept requests from the client and process them, either rejecting them, or accepting them and handing them over to the Web server. Here, we assumed that a Web server can not be expanded by adding a new functionality. Finally, it may be useful to have a specialized relay server to perform other tasks, such as a communication with other servers (the so-called three tiers architecture, see (Symantec, 1998)). For example, Symantec Café implementation provides a server called dbAnywhere which supports a communication with data base servers, see below.

7. Persistent Information

As described above, a generally acceptable way of saving information is to save it on the server, because there are no restrictions on what the server can do with its files. The best way to save information is to use a commercial data base, such Access or Oracle. To write programs that access data bases in a portable manner independent of a particular database one could use Object Database Connectivity, ODBC. However, ODBC is written in various languages and it is not object oriented. The better solution is to use Java Database Connectivity, JDBC, which is a set of interfaces to either directly access database servers or provide a bridge to ODBC. If an application that uses JDBC is on the same machine as the database server then it can directly "talk" to this server; this is called a *two-tier architecture*. Often, it is useful to be able to have an application on one machine M and a database server on another machine D. This *three-tier architecture* is possible thanks to an intermediate server running on M and communicating with D, and this is how dbAnywhere has been implemented.

8. Expandability

Since it is rarely possible to write an application that will satisfy all current and future needs, we need to be able to expand our applications. One could rewrite the application, or even better use inheritance and proper design patterns to satisfy new needs and then restart this application. However, it is far better to dynamically expand running applications, and any Java application can do this. As the first example, consider a small bootstrapping application that downloads code to perform a task (this code can be changed on the server when a new version becomes available). For instance, an HTML page may contain an icon representing an applet representing a discussion group. When the user clicks on this icon, the applet downloads the discussion group. Web applications can particularly benefit from dynamic expandability. A Java Web browser can access a new protocol and be unable to handle it; but with extendibility it could download the protocol handler from the server and from that point on will be able to understand this protocol (Hot Java Web browser is an example of such an application). A Java Web server, such as Sun's Web server of JigSaw can handle servlets, which are basically server-side applets (for more details see (Javasoft, 1998)). A running server can be expanded to provide new services to the clients by using servlets.

9. Summary and Recommendations

It is our opinion that no one should even consider writing large educational Web-based applications and using a traditional, old fashioned technology, that is old Web servers, CGI scripts written in Perl and a file system for persistent data. Instead, one should consistently use Java object oriented techniques to develop a system that consists of components whose behavior and distribution can be easily modified and expanded. In particular, we recommend servlets rather than CGI, servlets and RMI, or Corba rather than sockets, and Java servers rather then using ad-hoc user-defined specialized servers. In our example of a live chat room, changing the protocol or distribution technique will be easy to do with the recommended approach and very difficult to do with a traditional approach. For the same example, saving messages using JDBC is rather trivial, while saving them using a file system is error prone and almost impossible to modify. The cost of this approach is that you need programmers who are highly skilled and experienced in object oriented programming, but the benefit greatly outweighs the cost. The resulting software will be maintainable, portable and modifiable.

References

Acadia University 1997. "Acadia Advantage" http://admin.acadiau.ca/library/acadvant/advhome.htm

ACME 1996 http://plato.acadiau.ca/sandbox/home/present.htm

Alessi, S. M., & Trollip, S. R. 1991. Computer-Based Instruction. (2nd ed.). Prentice Hall.

Arnold, K. Gossling, J. The Java Programming Language, Second Edition. Addison-Wesley 1998.

Benford, S., Bowers, J., Fahlen, L., Mariani, J., and Rodden, T. 1994. Supporting Cooperative Work in Virtual Environments. The Computer Journal, 37(8).

Brose, G., Lorh, K. and Spiegel, A. Java Resists Transparent Distribution. The Object Magazine, Dec. 1997: www.sigs.com

Colbourne, C.J., & Cockerton-Turner T., (1989). Using Hypertext for Educational Help Facilities. University of York,

Goldberg M. W., Salari S and Swoboda P. 1996 World Wide Web Course Tool: An Environment for Building WWW-Based Courses", Computer Networks and ISDN Systems, 28 (1996).

Grudin, J. 1994. Groupware and Social Dynamics.. CACM, 37, 1 (1994), 93-105.

Holmes, M. and Porter, D. 1996. Student Notebook Computers in Studio Courses. ED-MEDIA'96, Boston (June 1996).

HTML: The Definitive Guide, 1994, O'Reilly and Associates.

HTTP 1998: http://www.w3.org

Javasoft 1998: http: //www.javasoft.

Ketinger W.J. (1991). Computer Classrooms in Higher Education. Educational Technology. pp. 36-43.

Maurer, H. A (1988). Report on the COSTOC Project. EATCS Bull. 35.

Maurer, H. and Tomek, I. (1990). Hypermedia in Teleteaching. Computers in Education, Elsevier Science Publishers, IFIP.

Maurer, H., & Tomek, I. 1990. Hyper-G - A Survey. Technical Report 284, IIG Techn. Universitat Graz.

Mühlhäuser, M., Borchers, J., Falkowski, C., Manske, K. 1996. The Conference/Classroom of the Future: an Interdisciplinary Approach. ED-MEDIA'96 Conference, AACE Proceedings, Boston, June 1996.

OMG 1995, Corba specification, New York: John Wiley& Sons, 1995.

OrbixWeb Programmer's Guide 1997. IONA Technologies PLC, November 1997.

Rodden, T. 1996. Populating the Application: A Model of Awareness for Cooperative Applications. In Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work, Cambridge, MA.

Shneiderman, B. and Alavi, M., Norman, K, and Borkowski, E. 1995. Windows of Opportunity in Electronic Classrooms. Communications of the ACM, 38, 11 (1995).

Stevens, R. 1994. Unix Network Programming. Prentice-Hall.

Symantec, 1998: http://www.symantec.com

Wake Forest 1997. http://www.wfu.edu/ThinkPad/Technology-Guide/2intro_index.html