AU/ACSC/0158/97-03

INTERNET WARGAMING WITH DISTRIBUTED PROCESSING USING THE CLIENT-SERVER MODEL

A Research Paper

Presented To

The Research Department

Air Command and Staff College

In Partial Fulfillment of the Graduation Requirements of ACSC

by

Maj. Gregory L. Tarr, Ph.D.

March 1997

Report Documentation Page

Report Date	Report Type	Dates Covered (from... to)
01MAR1997	N/A	-

Title and Subtitle Internet Wargaming with Distributed Processing Using the Client-Server Model	**Contract Number**
	Grant Number
	Program Element Number
Author(s) Tarr, Gregory L.	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Air Command and Staff College Maxwell AFB, Al 36112	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es)	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** UU
Number of Pages 45	

Disclaimer

The views expressed in this academic research paper are those of the author and do not reflect the official policy or position of the US government or the Department of Defense.

Contents

Page

Illustrations

Page

Tables

Page

Preface

Recent technological developments involving the Internet are changing the way we use computers. Modern computers are becoming more of a communication device than a computing engine. This trend will prove to be as great a change as the invention of the telegraph. The purpose of this paper is to examine the tradeoffs in software and hardware selection for complex modeling and simulation projects. This project will demonstrate Internet communication software can be used to relay not only information but situation awareness.

Abstract

The development of a multi-player wargame, accessible on the Internet, is presented. This paper discusses how the client-server model of the World Wide Web (WWW) can be used to implement the five functions of an interactive game. These five functions are registration, interaction, synchronization, adjudication, and graphic display. The techniques used to implement these functions include client-side scripting, server-side computation using the Common Gateway Interface (CGI), and graphical user interface design using the Hyper Text Markup Language (HTML).

The strengths, weaknesses and applicability of the client-server techniques are examined within the context of the game functions. Critical to this analysis is the current state of the software available for implementing the chosen client-server methods. Browser software and the available computer language programming environments are examined for portability, utility and end-user acceptability.

Based on this analysis, the Air Force Employment Exercise (AFEX) was "ported" to the Internet. The engineering solution is chronicled here. The WWW changed dramatically over the course of this project and several recommendations for future work are presented to capitalize on these changes.

Chapter 1

Introduction

This report will try to answer a single question. Can you design, with limited development resources, a computer war game exercise that keeps all players fully engaged? The constraints are simple as well. Each player should be able to interact with the game using their own computer keyboard without waiting for other people to take their turn or to pass a computer keyboard between players. The player should conceivably be able to be physically separated by rooms, using an Intranet or by continents using the Internet.

Although the problem has been solved by a number of software development companies using direct client-to-client socket programming, their approach is too demanding and time consuming for casual programmers. This report presents an approach suitable for the computer-literate subject area expert who wishes to create an exercise with minimal development time and resources. The solution is presented in the form of a set of computer modules written in JavaScript and C++, with an analysis of alternative methods and tools. The code presented in the Appendix converts the Air Force Employment Exercise (AFEX) from a two-player, shared single computer, turn-based game into an asynchronous, multi-player game suitable for distance learning courses.

Approach

The design approach is based on the client-server model developed by Tim Berners-Lee at the Computer Emergency Response Network (CERN) in Switzerland in 1991[1]. This report develops a design method based on the communications capabilities and display properties of WWW browsers and servers. This research focuses on means to utilize computer client-server techniques for classroom exercises, as opposed to the current approach of using a single computer for turned-based play. The final product is a template for the development of a new generation of educational war games.

The thrust of this research is to build multi-user war games from what is essentially document display software. With this in mind, the next questions are: What software tools are available? How can they be used? What are the advantages of one approach over another? What are the disadvantages of a particular design approach? This report tries to answer these questions. The next section details how this report analyzes the multiple solutions available and the criteria used to determine their relative merit.

Overview

The next chapter discusses the methods and tools used for building war games on the WWW. The examination begins with the WWW client-server model and the techniques it makes available for distributing the game's computational load. Of particular interest is the emerging area of client-side processing. Client-side techniques rely upon extensions of the basic WWW standards that enable one to run scripts or fully self-contained programs on the browser. The client-side extensions are complemented with server-side programming using the more traditional common gateway interface (CGI). Both methods require

programming. The next chapter concludes by looking at the available languages and programming environments for both the client and the server. The available programming languages and development tools will be judged based on complexity of use, cross-platform portability, security, and utility for implementing specific game functions.

The third chapter discusses how AFEX implements the five functions required of an Internet war game. Specific client-server methods, discussed more generally in the previous chapter, will be identified for each game function. The methods must work together harmoniously to achieve smooth game play. Just as importantly, the techniques must be implemented with the tools chosen. As technology advances though, the tools change.

In the conclusion, the rapidly changing technical environment is documented and I discuss how improvements in HTML will affect future programming projects.

The problems discussed in this project are the same problems encountered in developing tools for computer aided war preparation. War games have been used throughout history to pre-fight battles. Technology advances may cause computer war-gaming to be distinguishable from computer execution of war only in the fact that one involves blood and the other doesn't. The future of war planning, preparation, and execution depends on the same computer communications issues presented here. The objective of this research is to provide a method for war game exercise development which allows operations training to mimic operations execution. As taken from the joint doctrine, we must train the way we fight, and fight the way we train.[2]

Notes

[1] History of the Internet at http://www.pbs.org/internet/history.

Notes

[2] Global Engagement: A Vision for the 21st Century Air Force.

Chapter 2

Methods and Tools

The client-server model is the foundation for document retrieval on the WWW. The WWW started out as a platform independent method for retrieval of archived image and text documents from centralized servers for display on local machines, called clients. Over time, this paradigm has grown to allow the retrieval of customized documents, generated in real time, and even complete, platform independent applications that allow full user interactivity on the client machine. These developments have spawned a full network programming environment.

This environment has become a standard for solving problems of automated processing of client computer requests for interaction with server systems over the Internet. It provides the Air Force Personnel Center the ability to accept volunteers for positions over the Internet[1]. It also allows tailored requests for classified information to be passed from one military installation to another using the Global Command and Control System. Client-server processing is used to automatically generate customized price quotations by Internet businesses. These wonderful advances in the application of the WWW client-server model have not come without some growing pains.

These pains currently manifest themselves in security and portability concerns. The need for security precautions has placed a number of limitations on the environment, to

the point where it excludes some very promising approaches for use in this project. Ironically, one of the primary motivations for developing the WWW client-server model was to implement a set of platform independent standards that would allow all operating systems to participate equally. The rush to enhance the capabilities of the basic WWW standards has opened a Pandora's Box of proliferating non-compatible extensions that threaten to sacrifice the original noble intentions for the sake of marketshare. One goal for this project was to find the common ground in the current programming environments and develop a solution that would work on all platforms. While both of these pains are somewhat under the control of the web community, there is a third growing pain that is a result of the WWW's initial design.

One of the original design features of the Web's client-server model was that it was stateless. This means the client-server model has no memory. When making its next request, the client must specify it fully because the server remembers nothing of any previous interaction. Likewise, within the context of the basic client-server model, when the client receives a new document, the previous one is flushed and forgotten and not allowed to affect the new one. This lack of state is a fundamental problem for war games which are a series of turns, each building upon the results of previous moves. This report will discuss techniques for overcoming this weakness.

In order to examine specific techniques for implementing war games, we will present a discussion of the client-server model, which is the technical foundation of the WWW. This requires a functional analysis of the client's browser software, the request server's software, the protocols that link them, and the emerging options for distribution of the computing load between the two. This leads to an examination of the software

development environments currently available, which will be scored based on security, complexity, portability, and utility for solving required war game functions. First, let's examine the basic WWW environment.

What is the Client-Server Model?

The client-server model is another name for the software and communications protocols behind the WWW. Before exploring the various embellishments and extensions to the model, let's look at a typical web exchange.

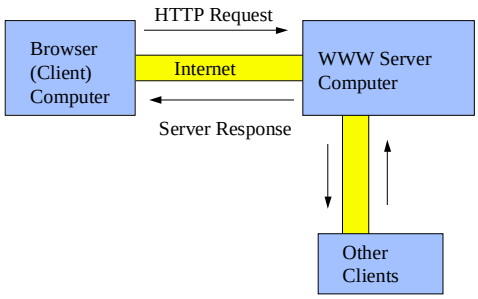

Figure 1. The Client Server Model

A user on the client machine uses his browser software to make a request for a document. The browser sends the request to the server using the hypertext transport protocol (HTTP), now at version 1.0. The server machine, running software called an HTTP server, processes the request and returns the document using the HTTP. The document itself is in hypertext markup language (HTML), currently at version 3.2. At the client, the browser renders the HTML document for display to the user. The client-server model consists then of browser software, sever software and the standard languages and protocols that link them.

The model achieves its platform independence through the use of standard protocols and languages. The browser software and server software are, by necessity, platform

dependent since they run on a particular computer. Virtually all browsers and servers are compliant with the HTTP 1.0. Compliance with the HTML standard is an issue for the browser. Currently there are two browsers that are largely compliant with HTML 3.2. They are Microsoft's Internet Explorer (IE) and Netscape's Navigator (Netscape). With the basics covered by the standards, lets introduce another standard that represents the first embellishment of the client-server model.

Common Gateway Interface (CGI)

The CGI is a specification, currently at version 1.1, that allows a client to request a customized document of the server. To accomplish this, there needs to be three functions added to the basic client-server model.

The first function is for a means for the user to specify what customization options are desired. This is accomplished using FORMs[2], a feature of the HTML since version 2.0. FORMs allow the user to input text, make one-of-many selections or even multiple selections from a list of options. Now, the customized request needs to get to the server.

The CGI provides the mechanism for sending the special request data to the server by extending the HTTP with a couple of specialized messages, more properly referred to as headers. These headers are triggered by the value of the METHOD attribute of the FORM tag in the client's document. The most common values are GET and POST with the POST method being the more preferred practice. The data is sent when the user clicks the FORM's SUBMIT button. With the data on its way, the server must have a mechanism to respond appropriately.

To respond to a FORM request, the CGI establishes a way for a program to run on the server machine. The HTTP server passes the data from the POST or GET header to a separate program on the server machine, called a server-side program. This program acts upon the request data and provides an appropriate response as output. The HTTP server receives the program's output and forwards it to the client. Since the browser on the client machine understands HTML, it becomes apparent that the usual output of a server-side program is an HTML document.

The CGI process of returning a file requires a step not covered to this point. The first line of a server-side program's output must specify the output's MIME type so the browser knows how to render the file. On a standard document request, this step is done for you by the HTTP server. For HTML output, the MIME type is text/html. For a GIF file it is image/gif. For a Microsoft Word document it is application/x-msword. Shockwave, Acrobat, etc. have their own MIME types defined that allow the client browser to recognize the data type that is coming.

The server-side output could be a static document or one generated on-the-fly. If the client only had a limited number of choices with regard to requesting a customized document, the server-side program merely needs to select which of several static documents to return. On the other hand, if there are a multitude of possibilities, such as might exist after several moves in a war game, it is more efficient to compute a response and return it in real time. The client, browser, and user can't tell the difference, and they probably don't care as long as the request was properly served.

This service, as previously suggested, could be the adjudication step for a war game. In a turn-based war game like AFEX, adjudication requires the collection of inputs from

multiple clients and the results are computed only after everyone makes their move. Server-side programs can accommodate this requirement by creating temporary files on the server that store the various client's moves until they are all collected. In a similar fashion, the registration process can be done by the CGI. The server maintains a waiting list of players, looking for partners. When a new player arrives, he is offered the option of pairing with someone on the waiting list or being added to the list. Adjudication and registration both require a mechanism for notifying the waiting clients when the process is completed. This can be done using what is called client-pull.

Client Side Processing

Client-Pull

Client-pull is implemented using a feature of the HTML. The HTML META tag can direct a document refresh by specifying a refresh interval and refresh document location.

<META HTTP-EQUIV="Refresh" Content='5;URL='*your-refresh.htm'*>

This META tag is part of the current document. It tells the browser to load the refresh document after the refresh interval, five seconds in the example above, expires. The refresh document could be the same document as is currently loaded or any another document on the WWW. This new document could contain a refresh directive also. In this manner, a series of documents can be loaded in a timed sequence. This series might loop back on itself, setting up an endless cycle. In the case of a series of length one, you update the current document at a specified interval.

Why might one want to continually refresh the current document? Consider the case where that document can be changed by a server-side program. The document, call it

result.htm, might initially contain a wait message for a client during adjudication or registration. The document contains a refresh directive so it is reloaded periodically. When adjudication or registration completes, the server-side program changes the contents of result.htm to announce the real result. This new file would not contain a refresh directive, thus ending the client-pull sequence once the real result is obtained. This technique can be used to synchronize the activities of the multiple players in a war game.

To this point, three of the five war game functions have been addressed by client-server model techniques: registration; adjudication; and move synchronization. The remaining two, interaction and graphic display, are user-interface design issues. The display is controlled by the browser on the user's machine, referred to, up to this point, as the client.

From the user's point of view, the browser actually performs a server function. Simply rendering the HTML document is stretching the concept of the browser as a server, a bit. More credible as a server function is the manipulation of FORM elements such as text input, radio button selections, and submission button clicks. With the advent of client-side programming however, the browser becomes a true server for the user and makes possible the design of custom user interfaces.

Client-side Programming

Client-side programming includes scripting, virtual machine programming, and direct access programming. Scripting includes JavaScript, JScript and VBScript. Virtual machine programming includes Java[3], J++ and ActiveX. Direct access programming includes plug-ins and helpers written for a specific platform to handle unique document

types. Examples of these include Macromedia's Shockwave plug-in or Adobe's Acrobat Reader, available as either a plug-in or helper.

Up to this point, portability has not been an issue. All of the methods described up to and including client-pull can be implemented with any client and server that is compliant with the standards: HTML 3.2; CGI 1.1; and HTTP 1.0. Client-side programming inextricably entwines us with the issue of a particular browser implemented on a particular operating system. As evidenced by the proliferation of scripting languages, client-side programming has no standards. While the browser manufacturers try to make their products function identically across all platforms that they support, even this is not always possible. Yet, the choice of browser, is really the first major choice when considering your client-side options.

Many browsers are available. Some meet the basic requirement for user interaction with the client, to be able to render HTML 3.2 documents. Only two major browsers, Netscape and IE, offer suitable extensions to implement client-side processing.

Considering only simple HTML, the difference between the two browsers is minor[4]. Newsgroup discussions suggest that IE is more stable with fewer bugs. When something doesn't appear to work as documented in IE, the problem is usually caused by a mistake in the document file. The same is not true for Netscape, where many programmers are reporting bugs. The major difference between the browsers is platform support.

Netscape tries to support all of the major platforms, UNIX, PC and Macintosh. Their support tends to be more uniform across the platforms. Microsoft has abandoned the UNIX world, and often lags in deploying new versions of their browser for the Macintosh. Even within the PC market, the Windows 3.1 operating system gets second billing to

Windows 95 by Microsoft. Since the target platform for this project is a PC running Windows 3.1 and IE, we will be limited to a subset of possible techniques.

Once the browser selection is made, several other decisions are fixed. Selecting a browser specifies the scripting language, the programming language and the set of plug-ins. Fortunately, each company has tried to copy the tools of the other, with Microsoft having a few additional choices. JScript is nearly the same as JavaScript, and J++ is the same as Java. The companies are even cooperating to some extent to allow plug-ins written for one browser to work with the other. As the various client-side options are discussed, the common ground between the browsers will be identified.

Client Side Scripts

The SCRIPT tag is a feature of the HTML 3.2 standard. It allows a document to contain code that is interpreted by the browser itself. The code arrives in plain text in-line with the rest of the HTML document. This differentiates the script from virtual machine code which arrives in a binary format. Script code is easily viewed by the user. Scripts are only allowed to affect the browser contents and not allowed access to any of the local machine's operating system functions. From a security perspective, scripts are fairly safe. They are also fairly powerful.

Script code is activated based on events that happen within the browser such as button clicks, an object getting or losing focus, or even the loading or unloading of the document that the script is contained in. What types of actions can scripts cause to happen?

One of the most useful functions of scripts is to do "error checking" on FORMS input elements. Data entered into a text input element can be checked for proper format and value before it leaves the client. This saves effort within server-side CGI programs and the Internet's round-trip delay to have the server respond to bad inputs. When the client-side script detects an input error, it causes an error message to appear in a dialog box. A dialog box is a separate window on most operating systems and scripting extends the idea to allow programmers to open new windows over which they have full control.

These new windows can be used to provide instructions, descriptions or other amplifying information. Instead of clicking a hyperlink and having to fetch an entirely new document from the server, the button click can activate a script that opens a new window with the script-supplied required information in it. This can greatly facilitate interface design, especially when combined with other HTML features.

One of the HTML's features is the HIDDEN input type within a FORM. This input type does not display and a FORM might only have these in it, thus, it would be invisible. If the document containing a FORM is the output of a server-side CGI program, these HIDDEN inputs can be initialized with customized data. Client-side scripts can access and use the information in these HIDDEN inputs. Thus, a method for achieving client-local data storage is possible. This method is even more lucrative when combined with another HTML feature, frames.

Frames allow a single browser window to be partitioned to display multiple documents. For game design, this allows a more sophisticated user interface and better control over the screen layout. The top-level frame document, called the parent, specifies the layout of the individual child frames and their initial document sources. See Figure 2

for a three frame example. The contents of each of the child frames can be changed independently of the other. More importantly, any scripts located in the parent frame document persist through changes in the children. Parent frame scripts can access data in the child frame documents. This concept when combined with HIDDEN FORMS and client-pull has some very powerful implications.

```
<HTML>
<script language=JavaScript>
// Persistent code goes here.
</script>
<HEAD>
<TITLE>Frame example</TITLE>
</HEAD>
<FRAMESET cols="70%,30%">
<FRAME NOSCROLL  src="wargame.htm"
name="gameframe">
<FRAMESET rows="40%,60%">
<FRAME src="registration.htm" bgcolor=Blue
name="inputframe">
<FRAME src="bform.htm" name="commframe">
</FRAMESET>
</FRAMESET></HTML>
```

Figure 2. Syntax for the Frame Document

Consider the three frame example where a large display frame holds a graphical presentation; a second frame has an array of controls such as buttons and selection lists; and a third frame with HIDDEN FORM inputs that are refreshed by client-pull from a document whose contents are updated by a server-side program. A user could manipulate the controls in the second frame to activate client-side scripts that read data from the third frame and make display changes in the first frame, the main display. With some limitations, you can do exactly that.

In Netscape's JavaScript, the client-side programmer can access just about any object in any document in any frame, including images. This is not the case in Microsoft's JScript. JScript will not allow an image to be replaced or changed once the document is rendered[5]. In Microsoft's defense, JScript was deployed as a catch up measure when Netscape fielded JavaScript. Microsoft did the best they could to maintain compatibility with a competitor's emerging standard. At the same time, Microsoft was developing their own approach to client-side scripting.

A Note About ActiveX and VBScript

ActiveX and VBScript are distributed without cost by Microsoft. Their strengths are in their flexibility and precise placement of graphic elements. Their weakness is in portability. They are only available for Windows 95, NT and Macintosh. VBScript is similar to JavaScript with all the object controls removed. Object control is what makes JavaScript so powerful. VBScript is intended to be used together with ActiveX for programming. Together they are very powerful, building on the syntax of Visual Basic. Without ActiveX, VBScript is relatively weak. So what is ActiveX? To answer that question, we look at virtual machine programming.

Client Side Programming Using Virtual Machines

A virtual machine is a software program that acts like a computer inside your computer. Building a software computer inside your computer attacks one of the fundamental problems of distributed network processing: portability. The virtual machine, being a software application, is not platform independent. It is written for each particular platform according to standards set down by the virtual machine designer. The code that

the virtual machine runs though, is platform independent. A software application developer would only have to write code for the virtual machine, and it would run on any hardware for which the virtual machine had been constructed. This is Sun Microsystems ambition for Java.

ActiveX is less ambitious. It is not intended to be a full fledged virtual machine. Some of the functions of a virtual machine are done with what Microsoft calls Controls. The Controls have to be developed for each supported platform, generally using MicroSoft's Visual Basic[6]. The Web programmer uses VBScript to pass inputs to these Controls. This hybrid of virtual machine programming and scripting is very powerful[7]. Market share may eventually make up for its lack of portability. At present, the full virtual machine is more mature.

Java and Microsoft's J++ are by far the best environments for developing war games. There are two reasons: the powerful programming capabilities, and the ability to precisely place graphics. Unfortunately, neither runs under Windows 3.1, a project requirement. Java virtual machines exist for Windows 95, Windows NT, Macintosh, and various UNIX machines. There are other negative issues besides portability.

Virtual machines still represent a security risk. Security is critical in network software, where that unexpected blinking of the hard drive activity light could be the transmission of all your bank account and credit card numbers to some unknown location. Sun Microsystems has tried to eliminate that concern by preventing any local storage medium access. Even with that, the Air Force may prohibit by regulation the distribution of Java based applications. The other concern is complexity. Virtual machine programming, like Java, requires about the same programming skills as C++.

Direct Access Programming: Plug-Ins and Helpers

Short of using Java, two other possibilities exist: plug-ins and helpers. While these two are about the same, they behave slightly differently. These two elements are used to interpret special file types passed to the browser.

Helpers are applications that reside on the client computer and can be called by the browser to render special files like sound, movies, or 3-D files. They execute independently of the browser in their own window. An example would be Microsoft's PowerPoint Viewer. It would launch and display any PowerPoint slide shows that were downloaded through the browser using the HTTP.

Plug-ins are applications that work with the browser to interpret special files within the browser window. Almost all browsers render GIF and JPEG image files in-line with the rest of the HTML document. The part of the browser code that allows this to happen functions just like plug-in. Similarly, the Acrobat Reader plug-in can mix the proprietary pdf file format in-line with the rest of an HTML document. This is accomplished using the HTML tag, EMBED.

Because plug-ins are so closely integrated with the browser, application programming interfaces (APIs) are almost a necessity for writing them. Borland C++ and Visual C++ both provide libraries to assist in their development. Templates are available for both Netscape and Internet Explorer developers[8]. The process is not difficult for the beginning C++ programmer (the only language option available). Plug-ins provide a good university level solution where the number of operating systems can be limited and student researchers provide a technical personnel pool for development. It is not a viable solution for overworked content area experts with limited programming skill. Be prepared to write

separate software packages for every operating system and supporting every configuration variation.

Plug-ins and helpers allow the complex graphics display routines to remain resident on the client computer, while only small message traffic passes between client and server. Application specific plug-ins are the basis for most commercial Internet games as DOOM and PYST. They work by allowing a minimal amount information to pass over the network such as a player's position and orientation within the game environment. The plug-in knows how to use this information to construct a textured 3-D world from the player's viewpoint complete with opponents in their proper positions.

One might consider using a commercial plug-in for war game development. There are tool kits for writing binary files that are interpreted by a particular plug-in. The most popular plug-in set is a commercial product called Shockwave, but a number of others exist based on commercial draw packages, for example Autocad. Many like VTML, include a three dimensional capability. The major disadvantage is that each requires a costly development kit to write client files. Using plug-ins complicates access to most CGI client server interaction. Consequently, the games must be written entirely in the development kit, which raises synchronization problems. They do provide a commercial quality solution for single player games.

Server Side Programming Environment

As the various client-side techniques were developed, the programming environments for each were discussed. Let's revisit server-side programming to discuss its programming environment.

Server-side programs are activated by and return their results through the CGI. Server-side programs can also create server local files since security is not as great an issue. These files might be the targets of URLs within a client document or documents that are refreshed using client-pull. One could even generate special files that would require client plug-ins to handle if you had access to the proprietary file formats required. The GIF image file format is well published and a set of C language libraries actually exist to aid in creating and writing GIF files. The basic requirement for a server-side program is that it be an executable program on the server. This gives great latitude in the selection of a programming environment.

The programming language can range from formal compiled languages to operating system level scripting languages. Formal languages include the likes of C, C++, Fortran, or ADA. Shell scripting languages like the UNIX csh or Bourne shells or even DOS batch files can be used as server-side programs. In between these extremes are interpreted scripting languages like the Practical Extraction and Report Language (Perl) or awk. In selecting an environment, the primary considerations should be portability, utility and simplicity.

The ANSI-standard versions of C/C++ are probably the most portable with the Perl being a close second. The arguments between advocates of the two camps take on the fervor of religious fanaticism. The Perl excels in string manipulation tasks, a common chore in generating HTML documents. The C/C++ camp enjoys a much better programming environment with utilities like make files and revision control systems. The Web community seems to favor the Perl, especially since its version 5.0 release which

takes it closer to being an object-oriented language. The major reason for Perl's ascendancy is that in many cases the developer does not own the server machine.

Perl has the feature of being interpreted instead of compiled. This is more acceptable to system operators because it doesn't require game authors to have access to the server's operating system. PERL scripts are also uploaded to the CGI directory or sub-directory as text files, which makes checking their system interactions easier. Perl represents the only choice for most people, as operating system shell accounts are generally not available except in the educational environment. Internet service providers no longer provide shell accounts because of the security risk.

If you own the server, a very reasonable choice is C++, or C. The next chapter will discuss how to develop complex graphics using C and the gd libraries. The availability of these libraries and the unsuitability of the other client-side graphics placement techniques drove this project to adopt C/C++ as its primary server-side programming environment.

Table 1 shows the relationship between programming possibilities and several estimates of merit[9]. Security considers the ease with which distributed programs could be replaced with malicious programming. Complexity measures the relative level of technical training required with ten equated to the level of competent but casual programmer. Zero implies a professional programmer is required. Utility is a measure of the ability to precisely place and display graphic elements, with 10 the highest. Portability is a measure of what attention must be paid to what operating system and hardware is being used. For the table, zero, means hardware differences affect the coding. Five implies that code must be recompiled with only minor changes. A score of ten means that the code is independent of machine.

Based on the evaluation of each entry of table 1, JavaScript or JScript was selected for the client-side programming environment

Table 1. Comparison of Software Approaches

	Security	Ease of Use	Utility	Portable	Notes
Assembly language	10	1	1	0	Out of date
Borland C++	10	5	10	10	Good
Visual C++	4	8	10	10	Better
Java/J++	8	5	10	5	No Win 3.1
ActiveX	8	5	10	8	No Win 3.1
Plug-ins	0	5	10	0	Great/but complex
Helpers	0	5	10	0	Straight forward with C++
Java/JScript	10	10	8	9	Best
Tcl	10	12	5	2	For UNIX only Motif/Xwin
Python	10	8	5	10	Extension Language like Tcl, might be wave of the future.
PERL	10	3	10	10	Interpreted
C shell	10	6	2	10	UNIX only
VBScript	10	4	3	5	No UNIX/weak

The two are functionally equivalent except in the way they handle mistakes.

Incorrect syntax is very often ignored in one case but not on the other, making the cross platform debugging difficult. C++ is selected for the server side environment.

This chapter has covered the basic techniques made available by the WWW's client-server model. The programming environments for both client-side and server-side functions has been covered with an eye on portability, complexity, utility, and security. The next chapter will cover how these techniques are deployed in the AFEX exercise.

Notes

[1] http://www.afpc.af.mil/

[2] These techniques are discussed at length in: The HTML Sourcebook : A Complete Guide to HTML 3.0 by Graham, Ian S, Publisher: John Wiley & Sons 2nd Edition, ISBN: 047114242 February 1996.

[3] For an tutorial on Java see: A Brief Introduction to Java at http://smoke.thepipe.com/java.html. by Eric Sorenson

[4] Many of the differences are simply annoying. For example, IE and Netscape reverse the order that the data is sent to the server.

[5] This deficiency has apparently been fixed in the soon to be released IE 4.0.

[6] It is not absolutely necessary to use Visual Basic, Visual C++ can be used as well, but with much more difficulty.

[7] To a much more limited extent, JavaScript can call Java applets. The MicroSoft implementation of the paradigm is more elegant, purposeful, and powerful.

[8] See: HTTP://home.netscape.com/eng/mozilla/3.0/handbook/plugins/index.html.

[9] These figures of merit are estimates generated by the author based examples tested and written over the course of this project.

Chapter 3

Implementing Game Functions

Deploying a war game, even one originally done on the computer, to the Internet requires more than just language conversion, such as from BASIC to C++. It requires a possibly full redesign of the interface, provisions for synchronization that were not required before, schemes for graphics display on a broad range of platforms simultaneously, and a plan for the distribution of processing among at least three computers. As an example for illustration, the ACSC war game exercise AFEX was selected. The conversion is discussed in terms of five basic functions that occur during the game play.

The five basic functions required of Internet war games are: registration, interaction, synchronization, adjudication and graphic display. Each requires one or more of the techniques discussed in the previous chapter to implement. This chapter discusses the particular techniques and combination of techniques used to deploy AFEX to the Internet.

Registration

Registration requires that each player (client) register with a central processor (server) to establish the opponent pairings for the game. At the same time, the game state is initialized. For AFEX, the game always begins the same and is completely symmetrical

for both players, like chess. Other games might require a randomization process to set, for example, the initial forces and their positions. This initial environment is defined by data structures and stored in files on the server that are keyed for the registered players. For gaming, this is the electronic equivalent of a game board and pieces. Once the opponent pairing occurs, the same process that communicates the game state after the adjudication phase is used to transmit the initial state of the game to the players. The method for doing the opponent pairing will be covered now, with the game state transmission function coming later.

For the pairing of opponents, there are essentially two cases. You arrive at the game either before or after your desired opponent. All entrants to the game then want to check the list of those waiting. If they arrived after the opponent, the opponent will be on the waiting list, in which case, pair with them. Otherwise, join the waiting list. This is implemented with a CGI call that generates FORMs on-the-fly.

The URL for the registration is a CGI call, register.cgi. The server maintains the waiting list in a server-local file. Register.cgi returns an HTML page with two FORMs on it. The first FORM has a SELECT option list that includes every name on the waiting list. It is a single select list. This FORM's SUBMIT button calls a client-side JavaScript function that makes sure an opponent has been selected from the list. If so, that selection is forwarded to another server-side CGI function, make_pairing.cgi, to make the pairing. The other FORM returned by register.cgi only has a submit button on it labeled. Add to Waiting List. This button's ACTION is a CGI function, add_to_wait.cgi, that appends the name of the new entrant to the waiting list. Both of the ACTION CGI functions use the REMOTE_USER environment variable to identify the new entrant to the game. This

prevents imposters, to some extent. Of interest now is what the ACTION CGIs do and what their output is to the user.

The add_to_wait.cgi appends the new entrant's name to the waiting list, assigns a game number label to this player, and returns a copy of the waiting list. The game number label will be used to name the files that contain the player's game state data structures. This file doesn't exist yet, because opponent pairing has not occurred. The copy of the waiting list is returned by sending a location header that references the CGI program, show_list.cgi, with the game number label passed in the QUERY_STRING environment variable. The show_list.cgi program first checks for the existence of the game number data structure file. If it exists, pairing has occurred. Show_list.cgi removes the waiting player's name from the waiting list file. A page is returned announcing this pairing, with a link to retrieve the game state, which at this time is the game's initial state. If the game number labeled data structure file does not exist, a page with the waiting list is returned. This page has a META refresh directive to show_list.cgi so it updates itself, executing the check for pairing first. One might wonder why the waiting list is returned as the wait message.

The reason for returning a copy of the waiting list is to break a possible race condition. Suppose both opponents arrive at about the same time, calling register.cgi before either of them are placed on the waiting list. Neither sees the other in the option SELECT list and both press the Add to Waiting List button. Both get added to the list, but both will see each other on the list that is eventually returned by show_list.cgi. A link to register.cgi is provided on the page returned by show_list.cgi with the waiting list on it.

The trick now is to avoid a race condition when both try to pair with each other using the make_pairing.cgi.

How does make_pairing.cgi work? It first checks for the existence of the selected opponent on the list. If the opponent is there, the data structure file is created using the opponent's game number label. The opponent will remove himself from the list when his show_list.cgi next executes. Check to make sure you are not on the waiting list. If you are, remove yourself. What if the opponent isn't there? The reason would be that he paired with you. Your game number labeled data structure file will exist. Remove your name from the list and set up to retrieve the game initial state. The reason this works is because you never remove anyone from the waiting list except yourself.

The registration process uses several methods previously discussed. FORMs and the CGI are the basic mechanism. It is supplemented by server-side storage of the waiting list and game data structure files. Client-pull keeps the waiting client advised of the registration status. The race condition is really a synchronization issue that we hadn't discussed before. It is handled by being a bit clever in triggering events by the presence or absence of a given piece of data. This piece of data though must reside on the server, as it is the only common ground tread by all of the clients.

Client Interaction

The success of the game is dependent upon three things: an intuitive interface, a simple interaction module, and accurate graphical presentation of the game state. Synchronization allows these three functions to work together.

The basic interface is the frames interface described in the previous chapter. AFEX uses three frames: a display panel; control panel; and a communications/status panel. In this section, the interaction function, the way the users make their inputs, is described.

The AFEX client interaction is based upon a reasonably complex set of FORM inputs. The basic move in AFEX is the selection of missions and targets for each of 96 aircraft. This requires over 150 pieces of information. JavaScript functions in the parent frame document are used to generate the form in the child display frame when the user decides to generate the air tasking order (ATO). The ATO function is selected by clicking an ATO button in the control panel frame. When the form is completed by the player, the Launch button is pressed which submits the user input to the server. The next step is to synchronize moves between players.

Synchronization

A difficult part of the Internet game design is implementing synchronization between players, and triggering the adjudication phase. The communication frame is the key to synchronization. The communication window keeps the players informed of the game status by displaying messages from the server.

A typical method for finding out the status of another client would be to use an Internet socket request. While socket programming is not especially difficult, by using a combination of tricks it can be avoided[1]. The approach taken here is to use client-pull to synchronize on the server as necessary.

The procedure is very similar to the registration process. When the Launch button submits our move, the server-side response is returned to the communications frame. Our

move submission will either be the last within a game round or not. If it's not last, a wait message is returned that has a META refresh tag in it. This starts the client pull in the communications frame. When the last client submits their move in the game turn, adjudication is triggered. The output of adjudication is to change the document that all of the clients communication frames are refreshing on to announce the availability of adjudication results. This announcement document has no refresh directive, so the client-pulls from the communication frames end. In AFEX this requires two separate executables on the server. The C program, cgi-afex, is used to process user form inputs and store the information in a server local file for use by the adjudication routine. The cgi-write program is used to maintain the "wait" document that is pulled by the clients.

Once all of the client moves have been accepted and stored, the adjudication algorithm is called.

Adjudication

Adjudication takes place as a simple evaluation of the environment that brings about changes in the status of the pieces and board, e.g., database. Adjudication in this exercise will be used to determine the effects of the strike packages on the targets.

The adjudication engine must manipulate the data base, which is composed of two lists: a 13 x 96 element array for aircraft and a 22 x 13 element array for targets. When requested by a client, the information is sent to the communication frame and stored in HIDDEN FORM elements. The approach taken here is to store the results on the server and instruct the user, inside the communication frame, to request an intelligence report document.

Adjudication and display in C++ are discussed by Gradecki[2]. While we will use C++, it will only be used to manipulate text files and generate graphics files which can be interpreted by the browser.

Graphic Display

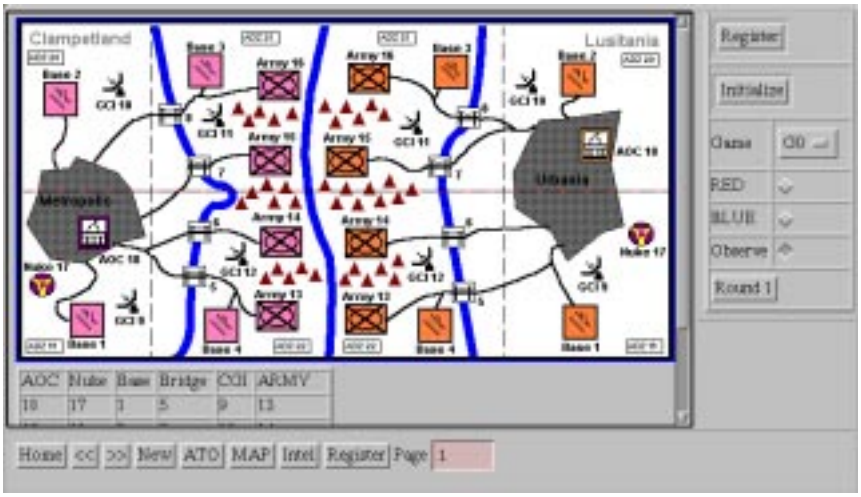

Figure 3. Intelligence Report

The intelligence report is a graphical display using custom GIF files and tables. When the communication window informs the player that the round is complete and the next round is starting, it activates a button allowing a request for the intelligence report. The graphic display file is written by the adjudication engine which also stores the game state in binary form on the server. The graphic file is created using the gd libraries to put dynamic information on a static map.

When developing computer games, embedding computer graphics and multimedia is possible using several approaches. The basic approach is to develop each piece as a separate file and link the file to an HTML document using an <A> tag. The browser has the responsibility to interpret the special file types internally with plug-ins[3] or externally

using helper applications. Multimedia includes graphics, sound, animation, video clips, and special file types developed by the game author. The EMBED container[4] provides direct access to the plug-in and helper applications.

The second approach, which uses elements of the first, is to actually write the graphics files on the server using a programming language and a graphics library. The first view is to look at multimedia as a sort of clip-art, the second is to actually generate the multimedia on the server. The second approach is what is done in AFEX.

Using CGI programs, a GIF file can be created on demand. Most browsers provide for display of GIF, and JPEG files in the browser, so this is a very portable approach. The CGI program writes the GIF image to a file accessible by the HTTP server. The image is included in the HTML document by using the <IMG src=URL> container. Except for a library of graphics tools called the gd libraries, this might be difficult for the casual programmer.

The gd libraries provide all the normal functions such as, open image, circle, line, square, and paste from file, necessary to create an image in real time. It is not intended to replace a drawing program, but is good for creating a specific game board display based on client inputs or the results of the adjudication engine. In AFEX, the primary static display, the map, is displayed using the BACKGROUND attribute of the BODY container. The dynamic information is put into a transparent GIF that overlays the background. The results can look very professional.

The generation and transmission of a GIF is not an optimal solution in terms of network bandwidth. It is the solution forced upon us by the compromises made in

choosing the development environments. The next section examines the distribution of processing between the client and server in AFEX.

Distribution of AFEX Processes

Based on the requirements derived from the discussion above, the game design requires the following sub-modules. The basic game script is in the form of a Uniform Resource Locator (URL) file on the server, and additional pages as needed to provide user instruction in the form of an on-line player manual. Three executables are needed on the server to execute the game. The form processor (cgi-afex in Table 2) collects the user input and stores it in the data directory.

The synchronization program, (in the table, gamewriter.cgi), checks for user input and provides the appropriate documents back to the client at the right time. The adjudication program, called by gamewriter.cgi, provides as its output a new game state in the form of a data file and an image file. The registration processor is actually included in the cgi-afex.c as a set of subroutines. This is the user input processor, which outputs the player database needed to validate moves.

Table 2. Distribution of Processes

Module	Location	Language	Function
GameScript AFEX.html	Server Document/ Download to Client	JavaScript	Download Client Input handler
Information Pages Apage#.htm	Server Document/ Download to Client	HTML	Download Instructions
cgi-afex.c	Server CGI	C++	Process user input
writegame.cgi	Server CGI	C++	Synchronization and call adjudication
Adjudication.exe	Server EXE Locally called and executed.	C++	Execute Adjudication on Sync

Table 2 continued

G#.AFEX.game	Server Data	Text/HTML	Store Game Data
G#.AFEX.register	Server Data	Text/HTML	Store Player ID
Board.gif	Server Data Download to Client	GD lib using C++	Graphical translation of the game state.

The game was implemented on a local area network using a Silicon Graphics computer with a Netscape HTTP server, with a number of Windows 3.1 Pentium computers using Internet Explorer acting as clients. The next chapter will discuss the conclusions and recommendations for similar projects.

Notes

[1] See Copes' discussion for the hardware level of Internet programming. A simpler approach is developed by Marc Loy, in *Java Programming for the Internet*, Prentice Hall, ISBN: 0132707780 1996. Unfortunately it not appropriate for this effort as Windows 3.1 is the standard, which eliminates Java

[2] His approach is excellent for development of games which interact with the client operating system by means of plug-ins or helpers but violates the requirement for simplicity, portability, and security. See *Netwarriors in C++ : Programming Multiplayer for Windows*, Joe Gradecki, John Wiley & Sons, ISBN: 0471113379, Published: February 1996

[3] Browsers typically come built in with several special file types for sound, graphics and multimedia. Specialized file types can be embeded in the document using plug-in components. These are relatively easy to write under C++, as the netscape window is a visible canvas object, using the netscape developers kit. For a full treatment see*: The Netscape Plug-Ins Developer's Kit*, Michael Morgan, Que Publishing Company. 1996.

[4] For a quick introduction to HTML containers, the Yahoo Search Engine provides many standard references at :http://www.yahoo.com/Computers_and_Internet/Internet/-World_Wide_Web/Information_and_Documentation/Beginner_s_Guides/. I recommend the NCSA guide http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html

Chapter 4

Conclusions

The proceeding chapters illustrate methods to implement the five software functions of war game training exercises. These are: registration, interaction, synchronization, adjudication and graphic display. The general approach and the analysis of specific techniques was explained in the chapters above. The implementation details are contained in the Software Appendix. The Software Appendix contains specific examples of each of the techniques described above. In particular, examine the source code for the Net Chess game and the Air Force Employment Exercise game. The concepts presented are intended for the casual programmer and subject area expert as guidance in the development of educational exercises that require student interaction as part of the learning process.

What was not known at the start of the project was if the project was even feasible with the limited resources available. By implementing this design method, and testing it against the Net Chess exercise and the AFEX game, it was demonstrated that relatively sophisticated games could be constructed with the available tools.

Recommendations.

While the information presented here should serve as a guide to multi-player game development, the speed with which the environment is changing should be taken into

account. For this project JavaScript was chosen for the client, and C++ for the server. The future environment may look completely different.

In the next few years, caution suggests attention be paid to server-side developments. Already projects are in the works to develop script programming for the Server. NetscapeOne looks like a means to write both server and client software in JavaScript. Still, with the strength of Microsoft, who is offering an essentially free product, Netscape could easily disappear. Scripting languages are improving at a rapid rate. Many of the limitations encountered in this project will probably be eliminated in future releases. The proposed HTML++ container[1] <FIG> will allow overlaid and offset image files. This, combined with the Netscape <IMG> container, should provide everything a developer might want in the way of precise placement of objects. By then, the virtual machine will probably have recovered from the security problems it suffers from now. Within a few years, fully interruptible operating systems like Windows 95 and Windows NT will have replaced Windows 3.1 and the client-server model will be replaced by a client peer-to-peer model. These developments could lead to plug-ins written in script languages and the methods of this study will be overcome by technical development.

Notes

[1] The difficulty with such a rapidly developing technology is that the technical documentation must be out in front of the actual technical developments to ensure that the documents are not out of date before being published. Althought several HTML++ containers are documented as fact in several books, they in fact do not exist and may never exist. The reference for the FIG container was taken from: *Teach Yourself Web Publishing with HTML 3.0,* Laura Lemay, Sams Publishing, 1996.

Bibliography

Air Force Publication, Global Engagement: A Vision for the 21st Century Air Force.

Cope, Kris A. Internet Programming, Jamsa Press by,. ISBN: 1884133126, April 1995.

Graham, Ian S. The HTML Sourcebook : A Complete Guide to HTML 3.0, Publisher: John Wiley & Sons 2nd Edition, ISBN: 047114242 February 1996.

Morgan, Michael. The Netscape Plug-Ins Developer's Kit, Que Publishing Company. 1996.

Lemay, Laura. Teach Yourself Web Publishing with HTML 3.0, Laura Lemay, Sams Publishing, 1996

Java Programming for the Internet, Prentice Hall, ISBN: 0132707780 1996

Netwarriors in C++ : Programming Multiplayer for Windows, Joe Gradecki, John Wiley & Sons, ISBN: 0471113379, Published: February 1996

DISTRIBUTION A:

Approved for public release; distribution is unlimited.

Air Command and Staff College
Maxwell AFB, Al 36112