Live Multimedia over HTTP

Jonathan C. Soo *
Telemedia, Networks and Systems Group
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

The World Wide Web is currently not well oriented towards distributing stream-oriented media such as audio and video. The limitation is not in HTTP [2] itself, but in currently existing browsers. After opening an HTTP connection to a server, most browsers write all data to a local file before passing it to an external viewer. While this works well for text and graphics, it makes viewing stream-oriented media impractical because of the long delay before start of playback, and because the entire file must be stored on the local host. In addition, it is not possible to send "live" streams of data.

This paper describes a prototype browser designed to solve some of these problems. It supports a subset of HTTP 1.0.

1 Introduction

The World Wide Web is a popular mechanism for distributing many types of data. In the last few years, it has become a major source of traffic on Internet and an important part of the academic and commercial information infrastructure.

While it currently works very well for text and graphics, is is not as well suited for distributing stream-oriented media such as audio and video. Even for users on a local Ethernet, the time needed to download a complete audio or video segment discourages browsing.

In addition, it is currently not possible to access "live" media streams through Web browsers. Live media are media of possibly indeterminiate duration, where capture, transmission, and playback are overlapped, and latency is nearly constant and relatively short; for instance, a live newscast might have latency of a few seconds. In current browsers, such as NCSA Mosiac and tclwww, it is not possible to overlap transmission and playback, resulting in a delay proportional to the length of the segment being transmitted.

This paper describes a scriptable, extensible, and embeddable browser that allows live multimedia to be transmitted using HTTP. Examples of live audio and video are presented, and approaches to live multimedia are discussed. The paper also describes the performance of the browser on prerecorded multimedia, and presents some applications using the browser as a platform for distributed computing.

*The author can be reached at: MIT Laboratory for Computer Science, Room 503, 545 Technology Square, Cambridge, MA 02139; Tel: (617) 253-4731; Email: jcsoo@mit.edu

2 Approach

2.1 Current Web Browser Implementations

Most browsers today are designed primarily for browsing through text and graphics over fast network connections, using a "store and forward" approach.

In a typical transaction, a browser first generates an HTTP request on behalf of a user. The URL provided is parsed to find the host and port of a server, and a TCP connection is opened. The request header is sent over the connection, and the browser then waits for the HTTP reply header.

On the server side, the header is parsed, the appropriate data is located, a HTTP reply header with the data type and length is generated and sent, and the data is written directly to the network connection.

The browser then reads and parses the HTTP reply header, determining the length and type of the data being sent. If the data is of a type that is supported natively by the browser, it is read and processed. If it is not, the data is copied from the TCP connection to a temporary file on local storage, and the name of this file is then passed to an user-defined application though a command line parameter.

This basic model is very effective. Most importantly, it is extensible; adding new data types is fairly simple, usually requiring a small modification to a configuration file and not requiring compilation of the browser. Sub-applications can be designed and tested completely seperately from the browser, and there is an established procedure to promote experimental data types to generally accepted ones.

However, there are some problems with the details of the implementation. The most significant is the step where the data is copied from the network connection into a temporary file before a sub-application is spawned. For small text and graphic files where transmission time is short compared to the connection setup time, there is little effect on response time.

For large audio and video files, however, this extra copy means that the entire file must be transferred before the sub-application can be spawned. Several problems result directly from this. The most important is that it is not possible to send live media streams; playback cannot be overlapped with transmission or capture.

Even for prerecorded media, response times for long segments may be measured in minutes, and the client must have enough local storage to store the entire file whether it will be played completely or not. Another disadvantage of this approach is that it is inefficient; not only is the data being copied one more time than necessary, it probably is being copied to a relatively slow mechanical storage device. As network connections become faster, this may become a serious bottleneck.

2.2 Proposed Changes

The approach taken to solve these problems was to restructure the client to avoid the copy to disk. Instead of immediately reading the data stream and writing it to a file, the connection is passed directly to the sub-application. The sub-application can then read data from the server as it is needed, and overlap processing with I/O. In fact, the sub-application has a TCP connection to the HTTP server process, and can send and receive data simultaneously.

On the server side, no modifications are necessary to take advantage of this approach to playing media files. The server sends data to the client as fast as the TCP connection will accept, and blocks when the TCP connection does.

For live multimedia, some server-side scripts are necessary to create live media sources from media capture applications. Fortunately, both the NCSA and CERN HTTP servers have a gateway interface that make it easy to do this by passing TCP sockets directly to sub-applications.

2.3 Goals

The design goal of this project was to create a client that would implement the above approach, as well as form a platform for future research in related areas. It was desired to have a client that would be easily understandable, portable, extensible, and embeddable. Although an existing browser could have been modified, this was not done for several reasons.

- Most browsers have a large fraction of code dedicated to HTML presentation and user interfaces, and support of alternate protocols such as FTP and NNTP. Excluding this code simplifies development, reduces bugs, and reduces the number of constraints put on the HTTP implementation.
- Most browsers are written in C. While this may provide good performance, it makes experimentation more difficult compared to using an interpreted language.
- Most browsers are designed as stand-alone programs, controlled only by human users. Modifying one to be embedded in another program or to be otherwise controlled by other programs or scripts would have been difficult.

3 Implementation

The implementation was divided into two parts; an HTTP library and a simple browser. The HTTP library implements a subset of HTTP, and the browser supports several of the common data types found on the Web, and several common sub-applications used to view those types.

The language platform that was chosen was TCL 7.3b [4], with the TCL-DP networking extensions and some C extensions to handle HTTP header parsing. The computing platform was a DEC Alpha running OSF/1 1.2.

The size of the final code was approximately 300 lines of TCL, and 50 lines of C.

3.1 HTTP Library

The HTTP library is written almost completely in TCL. It uses the socket management functions of TCL-DP to establish TCP connections to HTTP servers, and generates an HTTP header. Because TCL has difficulty using non-UNIX line delimiters, it also uses a function written in C to read the HTTP reply header.

When called with an URL and method, the http function first parses the URL to determine the name of the server to access, and the port number to use. The library then opens a TCP connection and sends and receives the appropriate HTTP headers.

The TCP connection is read only until the end of the HTTP header, so that the next byte read would be the start of the data requested. At that point, the HTTP library returns the handle of the TCP connection and passes the parsed list of MIME headers to the calling function.

3.2 Browser

The browser accepts user input and determines how to present the data received from the server to the user. After calling the HTTP library, a dispatch function determines the type of data being received by examining the Content-type entry of the MIME header. For each type, there is a user-defined handler procedure that the dispatcher calls.

For some types of data, such as "text/plain" and "app/x-tcl-script", the data may be read and processed within the browser. For others, a sub-application is started.

Here, instead of writing the data to a file before starting the sub-application, the sub-application is started as a child process, and the exisiting TCP connection that was returned to the browser is redirected to be the sub-application's standard input and output.

At this point, the browser's handle to the TCP connection is closed, and there is no further interaction with the server by the browser.

Most applications in the UNIX environment can easily be configured to read from standard input. In many cases, this is the default behavior. In TCL, starting these applications with a socket redirected standard input and output can be handled in a single command.

The browser currently supports most of the common content-types typically found on the Web, including text/plain and text/html, image/tiff, image/jpeg and image/gif, audio/basic, and video/mpeg. In addition, it supports VuSystem [3] streams, which are interleaved audio, video, and text streams used in the MIT TNS VuSystem project.

3.3 Server

At the server side, no modifications are needed for sending existing files. For live media, however, some scripts are neccesary to run the live media capture applications.

3.3.1 The Common Gateway Interface

Both the NCSA and CERN HTTP servers have a sub-application interface known as the Common Gateway Interface (CGI) [1]. This interface provides a standard way of passing details of HTTP transactions to sub-applications. Two slightly different variants exist on each server. A regular CGI script needs only to write a short content-type header before sending data to the client. The data from the CGI script is first read by the server, which calcluates it's length before sending it to the client. The data is not sent to the client until the CGI script signals the end of the data by terminating itself. Live media sources are not implementable using regular CGI scripts.

3.3.2 NPH-Header Scripts

An NPH-Header script is similar to the regular CGI script except that the sub-application is required to generate a complete HTTP header itself, and the sub-application is given the direct TCP connection to the client rather than a pipe to the server. This makes live media possible.

The NPH-Header scripts used to invoke the media capture applications are very similar to those used in the browser. Most UNIX applications are easily configured through command line options to write their output to standard output rather than a file.

3.3.3 Content-length

It is important to note that in many cases, no use is made of the content-length value returned by the server. In fact, none of the sub-applications used have a provision for a length parameter.

Some data formats such as the various image formats such as GIF and JPEG have the content length essentially encoded in the header, or may have some other way of determining end-of-file. Some other applications such as audio stream players may simply accept data until the connection is closed, which happens when the HTTP server has no more data to send.

For completeness, a convention was adopted where a negative content-length signified an indefinite-length data stream.

3.4 Distributed Applications

Two simple distributed applications were also created. Both of these used the direct TCP connection to transfer live data.

3.4.1 A simple access log monitor

The first application written was a simple access log monitor. A simple server-side nph-header script was written to copy data written to the server's access log to standard output. On the client side, the data was written to the user display as it was received.

3.4.2 A complex access log monitor

The second application was similar to the access log monitor, but was implemented using RPC and was interactive. A server-side nph-script was written with a new output content-type of x-tcl-rpc. On the client side, a handler was written that read from the connection, and evaluated the TCL code.

For this application, the browser was run with the TK extensions, a popular windowing system built around TCL. The server application transferred a program to the client that created the user interface. Then, for every log access, it sent a TCL command to the client, modifying the client's state and updating the client's display.

The server application also monitored the connection for commands sent by the client. For instance, the client could specify to the server whether it was interested in all connections logged, or only connections made by previously unseen hosts.

4 Preliminary Results

The browser was tested with a variety of sub-applications, file types, and servers over a local Ethernet and the Internet. Due to time restrictions, detailed measurements could not be made; however, the subjective results are fairly clear and reported here.

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.