Project 1

Curtis Duvall and Nathan Wilkins

HTTP server and client

Protocol description

Our system works by using HTTP protocols on top of a TCP transport protocol. Our server and client both rely on proper HTTP header formatting to properly communicate back and forth,

The server begins by creating a socket, and binding it to an address, so that the server can then be set to listen for communication coming from a client system. This client system could be a browser, or it could be the terminal based client application that we have implemented. When a connection with a client has been established, the server receives the request message into a buffer. Below is a simple example of what our server expects to see.

**GET /index.html HTTP1.1** 🡨Request code

**Host: 127.0.0.1:60023** 🡨Host Identifier

🡨BLANK LINE

This buffer is then tokenized into individual components, so that individual elements of the process can be processed properly. Our server is mainly concerned with the filename following the GET command.

The server will then check that the requested filename is formatted properly for the server. This includes ensuring all characters are lowercase, and that filenames with no extension, or the .htm extension, are given the .html extension instead.

The server then checks if this file exists. If it does, its size is included as part of the response header. If it does not exist, then the server will instead try to include the 404.html page. If this page is not found, then the body size is set to 0.

Once the body size is established, the server will try to figure out what type of media the file is. The type could be text (html, txt, css, js), image (jpg, png, bmp, gif, ico), audio (mp3, wav, wma, ogg), or video (mp4, wmv). If the server is unable to determine the data type, then it is sent as text, and the client can decide how to handle it.

Below is a simple response that could be produced by our server application.

**HTTP/1.1 200 OK** 🡨Status

**Content-Type: text/html** 🡨Type

**Content-Length: 1024** 🡨Size

🡨BLANK LINE

**<HTML>** 🡨HTML code

**\*\*\*\*\*\*\*** 🡨HTML code

**</HTML>** 🡨HTML code

Once the server has sent the message header and body, it waits for another request. This process will loop until such time as the server has been forcefully stopped, either through the host system shutting down, or the user entering ctrl+c.

The client application is user driven, and thus begins by asking for the user to specify an IP address, in the form of “127.0.0.1:60023”. Once you have typed this in, the server separates the IP address from the port, and verifies that they are in a valid format. In addition, if a port is not specified, the client application defaults to port 80.

The client then attempts to create a socket, and connect to the server through said socket. Once the connection has been established, the application prompts the user for a file to request. The file must be formatted as “/filename.ext”.

The HTTP request will then be sent to the server, formatted as follows:

**GET /index.html HTTP1.1** 🡨Request code

**Host: 127.0.0.1:60023** 🡨Host Identifier

🡨BLANK LINE

The client will then wait for the response from the server. The basic response that is expected will look like the following:

**HTTP/1.1 200 OK** 🡨Status

**Content-Type: text/html** 🡨Type

**Content-Length: 1024** 🡨Size

🡨BLANK LINE

**<HTML>** 🡨HTML code

**\*\*\*\*\*\*\*** 🡨HTML code

**</HTML>** 🡨HTML code

The user will then be asked if they want another file. If yes (y), they will be prompted for another file; if no (n), they will be asked if they want to connect to another server. If yes (y) to this second question, the user will be asked for another server IP address and port (127.0.0.1:60023); if no (n), then the process will end.