Project 1: Simple HTTP Client and Server

README.txt (**please read this) and use modified Vagrantfile**

**High Level Design of Web Client**

We begin by parsing the arguments given to the web client. The arguments should be at least one URL. The rest of the code is one big FOR loop to handle one URL. The URL is separated into its protocol, domain, port number, path, and file name, using regular expressions. The regex is done through boost∷regex with an expression specified in an RFC for parsing URIs. As an important note, the client will exit immediately with an error after anything unexpected happens. This includes an incorrect input URL. For example, if the first URL if formatted incorrectly and the ones after are not, the client will exit after reading only the first URL. All URLs must begin with http://.

We resolve the domain’s IP address using the same function as in the web server code, resolveIP() (which calls inet\_ntop()) and also get a mutable Cstring of the domain. We format the request message with “GET,” the file path, and HTTP/1.0\r\n\r\n. We do not use classes for our HTTP messages.

We create a client socket which we connect() to the server and the formatted request is sent. Then the client receives a response from the server, 20-byte buffers at a time until \r\n\r\n is found in the buffer indicating the end of the header. If the buffer happens to contain some bytes past the \r\n\r\n, it’s reached the body of the response and this part of the body is stored for later.

The header of the response is separated by \r\n to get individual lines. We inspect the first header line for a correct HTTP/1.0 (or HTTP 1.1) 200 OK message. Again, the client will exit with an error in the event of something unexpected. This includes the server not being found (so there’s no connection at all), incorrectly formatted responses from the web server, and any responses that are not 200 OK.

Then the client looks through the response message to see if a Content-Length header fieldwas specified and records the value if it exists. The client will ignore any other header fields.

To prepare the file to output, a new file is created into which we can write the contents of the body that will be obtained from the web server. If a file of the name already exists, the new file will be named with an appended number describing the count of existing file names. The new file is opened with ofstream, a stream, so that the number of bytes will not be restricted by the usage of a string. The part of the body that was already received (that was saved from earlier) is then written into the file. To get the rest of the body, we receive the rest of the response message 20 bytes at a time. If Content-Length is defined, the client only receives that many bytes of the requested file from the server. Otherwise, the client keeps receiving until EOF is reached. All the bytes are written to the stream.

A timeout feature has been added to a separate file called web-client-timeout.cpp which was implemented using the SO\_RECVTIMEO and SO\_SENDTIMEO options in the function setsockopt().

**High Level Design of Web Server**

We begin by parsing through the arguments given to the web server. They should be [hostname] [port] [fire-dir]. Otherwise they are set to the default as specified by the spec: localhost:4000. The IP address is retrieved by changing the hostname into a mutable Cstring (using a function that Richard wrote) and passing it into the function, inet\_ntop(). As an important note, we ignore the signal SIGPIPE, so that when a client and server are connected and the client closes, the server does NOT receive an error signal. We do not use any classes for HTTP messages; we just hardcode the formatting messages.

We create a connection socket, bind the server’s port number and IP address to the socket, and start listening for any clients. We have a non-ending FOR loop that keeps listening for requests. All new client socket file descriptors are stored in a vector and then given a new thread to handle the new connection.

For each connection, we create a new socket and have a continuous loop that keeps receiving the client’s request string until we reach a ‘\r\n\r\n’ which indicates the end of request. We split the request string by ‘\r\n’ which indicates the end of header lines using a continuous while loop that only ends when we reach \r\n\r\n. If there isn’t an appropriate first header line, we simply return the client an error. The first header line must be of format similar to GET /index.html HTTP/1.0. We use the boost∷tokenizer to separate the first header line by spaces to retrieve the *method, file\_path,* and *HTTP version.*

Then, we open the file requested (as specified by file\_dir + file\_path). We check that it MUST be a regular file; otherwise it is considered an error. A single slash, ‘/’ is not considered a correct file path and will be treated as an error 404. We also get the file size using fstat.

Afterwards, we move onto formatting our response message to the client. We add the protocol, statusCode (which is set to default 200, and changed to 400 or 404 if there was an error), and corresponding status message. This first line is then appended with a \r\n indicating the end of the first response line. Next, we have the server also add a *ContentLength* header field which is followed by \r\n\r\n to indicate the end of the header and start of the body (the file requested). The server sends the header portion first, then starts sending the file requested a byte at a time until EOF. Afterwards the thread socket is closed. The server still has a main thread that has a connection socket accepting concurrent connections.

A timeout feature has also been added to the server for both receiving and sending in a separate file called web-server-timeout.cpp using the SO\_RECVTIMEO and SO\_SENDTIMEO options in the function setsockopt().

**The Problems you ran into and how you solved them**

Boost is a bait. Using compiled boost libraries sucks. Apt-get does it better.

Bugs and corner-cases coming up through testing and verifying them. We fixed this by testing more.

**Additional Instructions to build your project**

* Please use our modified vagrant file (we need BOOST from apt-get)

**How you tested your code and why**

We tested by passing back and forth some index.html’s and later 1Gb files between our client and server. This was done to check that our code could handle big inputs for which we had to change some of our code from using strings to string streams. We also retrieved files externally, doing a DIF on them to verify their validity. Ricard also tested getting binary (ie. .png, .zip) files and multiple parallel connections to the web server. We used prebuilt tools like $telnet to send incorrectly formatted requests to the web server and see the error codes and messages. We also used commands $printf and $nc to send valid requests to the server.

**Contribution of each team member**

Richard Min (UID: 604451118) and Joanne Park (UID: 104450395)

Richard wrote all of web-server.cpp and the functions: stringToCString() and resolveIP(). He also set up web-server.cpp with argument parsing and socket/connection accepting. Joanne finished web-server.cpp and wrote the function: split\_by\_carriage\_return(). Richard also did the timeout functions in web-server-timeout.cpp and web-client-timeout.cpp. Joanne did the report. Richard did all the extra credit. Richard wrote the README.txt (**please read this** for warnings, directions, notes!)

**File Organization**

Base project:

* web-client
* web-server

Timeout Extra Credit

* web-client-timeout
* web-server-timeout

Async Extra Credit

* web-client-async

HTTP/1.1 Extra Credit

* web-server-1\_1
* web-client-1\_1

README.txt has notes about the files.

Vagrantfile has been modified, use this (or manually run the shell script if necessary).

Makefile has been heavily modified.