EE122: Introduction to Computer Networks — Fall 2007

Project 1: Build a Web Server

Milestone 1 — DUE SEPT 26, 2007, 11PM

Vern Paxson / Lisa Fowler / Daniel Killebrew / Jorge Ortiz

Introduction

The goal of this project is to build a functional server and client for a subset of HTTP version 1.0. Your server will be able to serve content to web browsers and clients anywhere in the world. Similarly, your web client will be able to receive content from any web server. This project will teach you the basics of distributed programming and client/server architectures, and allow you to gain a functional understanding of the HTTP protocol. This effort will be broken down into smaller stages, and will be due in two milestones. This is the project description for the first milestone.

At a high-level, a web server is simply a program that listens for connections on a particular socket (bound to a specific port on the server), receives commands from clients via this socket, processes the commands, and sends the results of the processing back to the client as a response to the command.

In this abstract view of a web server, a server is a program that executes a logically infinite loop wherein it receives and processes commands, structured something like the following:

```
Loop forever:
Listen for incoming connections
- Accept new client connection
- Parse HTTP/1.0 request
- Ensure well-formed request (return error otherwise)
- Determine if target file exists and if permissions are set properly (return error otherwise)
- Transmit contents of file to client (by performing reads on the file and writes on the socket)
- Close the connection
```

For this milestone, you will create both a web server that behaves as described above, along with a client to interface with any web server using HTTP/1.0.

Milestone 1: Client-Server + Beginning HTTP

The goal of this project is to learn socket programming for both clients and servers. There are two halves to this milestone. You may approach this milestone by starting from either half, but it is essential that you complete both halves.

Part A: Client-Server System

The client-server model is a very common way to build network applications. For the first step in this project, you will build a client program that sends user input to a server program that echoes the data back to the client. Note that in this initial step, the application-layer *protocol* is trivial (the server just copies whatever bytes it receives back to the client, without parsing them in any fashion. You will extend this in the next step, but for now, no additional processing is required.).

Client Specifications

1. The client should compile to a binary called `http_client` and will be executed using the command, `http_client` <*server*> <*port*>, where <*server*> and <*port*> indicate the appropriate port on the particular server to which you wish to connect. In particular, your client should be able to connect to the server at `ilinux3.eecs.berkeley.edu`, TCP port 7788, run by the EE122 staff. (Note: an announcement will be sent when this server is available for testing.)
2. The client program reads lines from the keyboard (`stdin`), accepting input up to 2048 B in size.
3. After the user enters one line, the client sends this line to the server.
4. The client outputs the data received from the server to the console (`stdout`).
5. The client repeats the above steps until it is killed with a `Ctrl-C` or `stdin` is closed (user enters end-of-file, `Ctrl-D`).
6. The client uses TCP to communicate with the server.
7. The client must gracefully handle error conditions. For example, the server might not be running, might not respond to you, or might unexpectedly close the connection.

Server Specifications

1. The server should compile to a binary called `http_server` and will be executed using the command, `http_server` <*port*>, wherein <*port*> is the port number to listen on.
2. The server listens for TCP connections on a port, specified as a command line argument on startup.
3. The server echoes back the first line it receives on the TCP connection, prints the line to standard output, and prints the components of the request or prints an appropriate error for invalid requests.
4. The server closes the connection with the client.
5. The server must be able to handle concurrent connections from multiple clients. This will entail use of `select()`, as will be discussed in section. The server cannot wait on a single client to finish sending while ignoring the other clients.

Both your client and server programs should be prepared to deal with socket-related errors and attempt to recover from the errors when possible. Your program should be able to handle basic socket errors such as one side of the connection closing the connection unexpectedly or a client being unable to connect to a server. If a client or server encounters a socket error, it should print a meaningful error message to standard error. If the client or server is unable to recover from the socket error, it should terminate.

Part B: Beginning HTTP (*i.e.* Doing Useful Work)

Part A of this milestone focuses on the network interface between a client and a server. Here we turn to the higher-level logic that will allow us to do useful work with our client/server system in the future. The task in this part of the milestone has nothing to do with networking and is simply a parsing problem.

Grammar for HTTP/1.0 GET

For this part, you are to write a program that will determine if a command received by a server is a valid HTTP GET command. An HTTP GET command is simply a line of input that looks like the following:

```
GET /Research/Areas/OSNT/index.html HTTP/1.0
```

The HTTP GET request is part of the larger HTTP protocol. Protocols such as HTTP are typically specified using a precise specification notation (in the form of an augmented Backus-Naur Form [BNF] grammar). These notations are, in essence, a textual form of the syntax diagrams that are used to specify the formal syntax of a programming language. For example, the formal definition of the HTTP GET "request line" that you are to implement is given by the following grammar:

```
 Request-Line  =  Method +Space Request-URI +Space HTTP-Version *Space CRLF
       Method  =  "GET"

  Request-URI  =  Absolute-Path
Absolute-Path  =  "/" *FileNameChar
 FileNameChar  =  ALPHA | DIGIT | "." | "-" | "_" | "/"

 HTTP-Version  =  "HTTP" "/" +DIGIT "." +DIGIT

        Space  =  SP | TAB
```

The additional tokens and terminals are described in section "2.2 Basic Rules" in RFC 1945.[1] (We have left out some of the full syntax. For example, in reality FileNameChar can include escape sequences and alternate character encodings.)

Note:

- Anything in quotes is interpreted as a string literal and must appear *exactly* as described.
- The vertical bar, "|", represents a mutually exclusive OR. "+" preceding a token indicates one-or-more instances of the token, and "*" indicates zero-or-more instances of the token.

Also note that all HTTP requests (and replies) are terminated by a blank line. The blank line comes after information such as the command, status codes, and HTTP headers. For this project, you do not need to include any such headers, but you do need to include the blank-line termination.

Server Behavior

The parsing program on the server-side will read lines of characters from the socket and determine which lines, if any, are legal HTTP GET requests. For each line of input, your program should:

[1] `http://www.w3.org/Protocols/rfc1945/rfc1945`

- Echo the line of input to both the client, and standard output.
- For valid requests, print to standard output the components of the request (these should appear after the line of input).
- For invalid requests, print an error message on standard output indicating which token is missing or illformed.

For a valid request, the output to standard output would be (*i.e.*, the server should return):

```
GET /Research/Areas/OSNT/index.html HTTP/1.0
Method = GET
Request-URI = /Research/Areas/OSNT/index.html
HTTP-Version = HTTP/1.0
```

There are four possible errors that you can detect during a parse. Your program should format its output exactly as shown below, and for each error encountered, should print one of the corresponding error messages:

An invalid method token	`ERROR -- Invalid Method token.`
An invalid path token	`ERROR -- Invalid Absolute-Path token.`
An invalid HTTP version token	`ERROR -- Invalid HTTP-Version token.`
Spurious text appearing between the version token and the end of the line	`ERROR -- Spurious token before CRLF.`

The server needs to be able to handle multiple clients connected simultaneously. This means that if you only receive part of a message, you must store it and handle other clients. You can't receive a partial message and keep attempting to receive from that client, while ignoring the other clients.

Client Behavior

The client program at this stage will be a program that requests an object from a web server and echoes the received data to standard output. The program should take a "full" URL (*e.g.*, a URL of the form `http://ilinux3.eecs.berkeley.edu:7788/index.html`) as input from standard input; translate the URL into a valid HTTP 1.0 GET request as specified above; and send it to the server specified in the URL on the port number specified in the URL (or port 80 if no port number is specified). Your client need not generate any header messages as part of its GET request. Be sure to end your requests to the server with a blank line, as mentioned above.

The client is responsible for validating URLs, which will follow this BNF grammar:

```
    HTTP-URL  =  "http://" Host ?Port ?Path
        Host  =  Hostname | Hostnumber
    Hostname  =  ALPHA *AlphaNum ?("." Hostname)
  Hostnumber  =  +DIGIT "." +DIGIT "." +DIGIT "." +DIGIT
        Port  =  ":" +DIGIT
        Path  =  "/" *FileNameChar
FileNameChar  =  ALPHA | DIGIT | "." | "-" | "_" | "/"
    AlphaNum  =  ALPHA | DIGIT
```

Note:

- You should be lenient with the grammar only in so much as you *do not* require case-sensitivity from the user. *e.g.* `HtTP://sOMeDom41n.com` should be a valid URL.

- "?" indicates zero or one instances of the token.
- Tokens enclosed in parenthesis are grouped together and are to be treated as a unit. The tokens share the same cardinality, *e.g.* `?("." Hostname)` means one instance of `"."` followed by `Hostname`, or neither of the two tokens.

If an invalid URL is encountered, the client should print the following message on standard error:

`ERROR -- Invalid URL:` <*the invalid URL*>

After sending the request the client should await the server's response. The program should print to standard output the response message from the server. The client need not do any other processing of the message other than echo them to standard output.

After the server closes the connection, read the next full URL from standard input. When end-of-file is reached on standard input, the client program should terminate.

For any network errors that occur, print the following message on standard error:

`ERROR -- Network Error:` <*description of error*>

General Instructions

1. You may choose from C or C++ to build your web server, but you must do it in a Unix-like environment. You will want to become familiar with the interactions of the following system calls to build your system: `socket(), select(), bind(), listen(), accept(), connect()`.
2. The code must be clean, well-formatted, and well-documented.
3. The programs must compile and run on the UNIX instructional account machines. We will not be able to grade programs that work only on Windows.
4. Your program needs to be robust to ill-formed or otherwise stressful input.

Helpful Links

- *Beej's Guide to Network Programming*: `http://www.beej.us/guide/bgnet/`
- *Reference Links on Sockets*: `http://www.stanford.edu/class/cs244a/socket-links.html`
- Socket Programming Slides from discussion section: `http://inst.eecs.berkeley.edu/~ee122/fa07/notes/Sockets_discussion.ppt`

Submission Guidelines

1. You will need to submit the source code for both the server and the client, along with an appropriate Makefile to compile both the programs. The `gmake` command should work with your Makefile.
2. The client should compile to a binary called `http_client`.
3. The server should compile to a binary called `http_server`.

4. Please submit a `README.txt` file that documents how to use your `http_client` and `http_server` programs.

5. Submission Steps:

   (a) Make sure you are registered with the EECS instructional account system (different from registration for the class). You would have been prompted to register the first time you logged in to your instructional account. You can check your registration status by running the command `check-register`. If you find yourself not registered, run the command `register`.

   (b) Log in to your instructional account.

   (c) Create a directory called `proj1a`: `% mkdir proj1a`

   (d) Change to that directory: `% cd proj1a`

   (e) Copy all the files that you wish to submit to this directory.

   (f) Run the submit program: `% submit proj1a`

   (g) Make sure that no error messages are displayed. If some error occurs, retry. If it still occurs, contact the TAs.

Grading

- This project milestone is worth **10%** of the course grade.
- The client program will be worth 50% of the project grade. The server accounts for the remaining 50%.
- Your submission must satisfy all the specifications in order to receive 100% of the project grade.
- Programs will be tested on `c199.eecs.berkeley.edu` (a Solaris machine), so your final submitted code should work on that machine.
- You will be penalized if the submitted code is shabbily written (difficult to extend or for us to understand, or brittle in terms of error handling).