# Data Networks
# Project 1 - A client and a server

## Handed out: Monday, April 30

## Due: Wednesday, May 16 at 11:59pm

### I. Introduction

The goal of this project is to get some experience with networked applications and protocols, including how to establish connections, format data, and use sockets.

In this project, you are to develop a simple client and a simple server. The client should establish a **TCP** connection with the server. Once the connection is established, the client should send a "ping" message (described in Section II.a) to the server. Upon receiving the "ping" message, the server should reply with that same message, i.e. a "pong", immediately. The client waits for the reply, then repeats the message exchange for a total of N such exchanges, then closes the connection. The client should measure the latency of N such transactions and report the total time and average round-trip time and then terminate. In addition to responding to your "ping-pong" client's messages, the server should have an alternate execution mode in which it acts as a simple web server. This functionality is described in Section III. Your server must be able to handle multiple concurrent connections from clients.

### II. The Ping-Pong Client and Server (60 %)

The ping-pong client should take 4 command line parameters, in the following order:

- -h *hostname*

  The host where the server is running. You should support connecting to hosts by domain name (e.g. forest.mpi-sws.mpg.de).

- -p *port*

  The port on which the server is running (e.g. 4321).

- -s *size*

  The size in bytes of each message to send (e.g. 50).

- -c *count*

  The number of message exchanges to perform (e.g. 10).

The server should take 1 mandatory parameter and 2 optional parameters, in the following order:

- -p *port*

  The port on which the server should run (e.g. 4321).

- -m *mode* (optional)

The mode of the server. If this is "www", then the server should run in web server mode (described in Section III). If it is anything else, or left out, then the server should run in ping-pong mode.

- -r *root directory* (required for www mode)

  This is the directory where the web server should look for documents. If the server is not in web server mode, ignore this parameter. (e.g. /home/student/comp429/project1/html)

The client opens a TCP connection, sends a ping packet of the appropriate size, waits for the reply, repeats the exchange "count" times, and closes the connection. It should measure the total latency and using this to calculate average round-trip latency. You can use gettimeofday(), which gives microsecond-granularity in timing. The client should finally print the average round-trip latency and terminate.

## II.a. Format of the Ping-Pong Packets

The ping packet is formatted as follows:

| size (4 bytes) | data (size bytes) |
|---|---|

The size is in network-byte order. The data is unordered. The "size" lets the server know how much data there is to read in the ping message.

The server should reply with the exact same message.

## III. The Web Server (40 %)

When the server's optional *mode* parameter is set to "www", the server should run in web server mode. In this mode, the server should answer HTTP GET requests from web browsers. The server should open files relative to the root directory specified as a parameter to the server, in order to respond to GET requests. The full specification for HTTP can be seen at http://www.w3.org/Protocols/. Because HTTP is an extremely complicated protocol, we only expect you to implement a subset, described here.

### Special Cases

Note that you want all requests to be restricted to under the root directory. This means you should disallow requests containing "../" path elements which could potentially cause the server to send documents not under the root directory. You can allow symbolic links which leave the specified root, since the user explicitly put them there and knows what to expect. Return error message with code 400, if "../" appears in the path.

If a file is not found, you need to return an appropriate HTTP error code and message. How you handle directories is largely up to you; you can return a file-not-found message, a more specialized error message, or convert the request to index.html within the specified directory. It is **NOT** OK to just open the directory as a file and return the contents.

## IV. Administrivia

This project is due on **Wednesday, May 16, 2005 at 11:59pm**. To submit the project, create a single compressed file of your project directory (use tar and gzip), name it with the following format *proj1_lastname* (e.g. project1_gummadi.tgz), and send it by e-mail to the course staff at datanets-projects@mpi-sws.mpg.de.

In the project directory, there should be a file entitled README which documents how you tested your client and server, any known problems, and anything the graders should know. You should use *make* to build your project (there should be a Makefile in the project directory, and running make in the project directory should build both the client and the server). The grading will be done by testing how well your code runs on the student computer pool (CIP) machines, so make sure that your code works on these machines.

You can write your programs in C, C++, or Java. When programming in C++ and Java, you should not use certain advanced classes and libraries (e.g. Java HTTP library) that make your implementation trivial. If you have questions regarding this, approach the course staff during their tutorial or office hours. Don't forget to check for error return codes to network API calls; it's not only good practice, it can also help you to figure out what is going wrong if your program is not doing what you expect.

---

# HTTP Protocol Overview

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data \ transfer across the Internet. HTTP/1.0 improved the protocol by allowing messages to be in the format of MIME-like messages, containing meta information about the data transferred and modifiers on the request/response semantics. Latest HTTP Version is 1.1, which is documented in RFC 2616; version 1.0 is docume\ nted in RFC 1945.

HTTP operates over TCP connections, usually to port 80, though this can be overridden and another port used. After a successful connection, the client transmits a request message to the server, which sends a reply message back. HTTP messages are human-readable.

The simplest HTTP request message is `GET url`, to which the server replies by sending the named document. If the document doesn't exist, the server will send an error message stating this.

The format for GET command is `GET url HTTP/version`. The client may then send additional header fields, one per line, terminating the message with a blank line. The server replies in a similar vein, first with a series of header lines, then a blank line, then the document proper.

## Here a sample HTTP 1.1 exchange represented using strings in the C programming language:

### Client sends HTTP request:

```
GET /index.html HTTP/1.1\r\n
\r\n
```

### Server replies with HTTP response:

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
\r\n
<HTML document body follows>
```

Note: (a) The line-feeds `\r\n` and the empty line between HTTP response header and HTTP response body are mandatory. (b) All message characters should be encoded in ASCII.

# Simply speaking, HTTP protocol operates in 4 steps:

- **Step 1 'CONNECTION':** Web browser(the client) try to connect to the web server using socket.
- **Step 2 'REQUEST':** Web client sends its request via the established socket. Here we only consider the request using GET command, e.g.:
  `GET path/file HTTP/version \r\n\r\n`
  the 'path/file' indicates the requested file; 'HTTP/version' specifies the HTTP version used by browser.
- **Step 3 'RESPONSE':** After the server receives the request, it will do some necessary processing then send back the results to clients. The the client can display the requested web page. For example, if the client wants to visit the webpage www.mycomputer.com:80/mydir/index.html, it will send a GET co\ mmand: GET /mydir/index.html HTTP/1.1 to the server. The webserver with domain name www.mycomputer.com will search for 'index.html' from its subdirectory 'mydir'. The server needs to send the client some necessary information regarding the requested file. So the server will first transmit some HTTP header followed by the document body. HTTP header is separated from the document body by a blank line.
- **Step 4 'DISCONNECTION':** After the response, web server should disconnect with the client.

# Your implementation must follow at least these specifications:

As there are many extensions to HTTP and HTTP itself is rather complex, we require you to implement at least the following subset of HTTP/1.1

- The supported HTTP Version is 1.1 only. If you find another version in the HTTP request, you have to answer it with the appropriate response message indicating that you do not support that version.
- A request has to be formatted as found in the RFC of HTTP. Your server will not be required to process any header, but must be able to handle headers (i.e. ignoring them). A valid request looks like this(note the required line feed characters at the end of every line and especially the empty line in the \ end:
  `GET /index.html HTTP/1.1\r\n<optional headers>\r\n\r\n`
  If your server sees any other request types (either valid ones as in the RFC or invalid ones), it should reply with the appropriate response type (see below).
- A response has to be formatted as found in the RFC of HTTP. Your server should support at least the following response types:
  - `200 OK` : If the request was valid and could be processed
  - `400 Bad Request` : If the request was malformed
  - `404 Not Found` : If the requested file could not be found
  - `501 Not Implemented` : If the request was valid but the request type is not implemented
- You only need to support the MIME type text/html in responses, that means that the data sent is a HTML document. You do not need to deal with any other type. Note, that you have to place an empty line between the HTTP header and the body (which contains the HTML document).

# Reference:

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

# A tutorial about socket programming can be found at:

http://beej.us/guide/bgnet/

Many more tutorials can be found on the web for every possible programming language. Just ask your favorite search engine.