Deign Document: ASGN-1 `HTTP Client and Server`

Hang Yuan
CruzID: `hyuan3`

1 Goals

The goal of this small project is to implement a simple HTTP client and server. The server is single-threaded, as is the client. The server will use simple “`GET`” and “`PUT`” commands to read and write (respectively) “files” named by 40-character hexadecimal names.

Other than that, the client and server can handle different type of errors and communicate with a web browser. And the server is able to provide the length of the data as an additional response.

2 Design

There are three parts to the design. One is handling all the common components in both client and server. One is all about the HTTP client and the other is about the HTTP server separately.

Basically, the server will read the arguments, set up connection with server by sending out the socket, send the action code and data to server, server understand the action code, dealing with the request and give proper response to client.

Basic transaction flow:

Client (PUT)	**Server**
Understand the arguments passed in	
Set up connection to server	Accept connection and ready to listen & accept
Open and read the file (data)	
Send request with data to server	Receive and understand the request, open (create may be needed) and store data
Finish sending and waiting for response	Stop storing and give response
Send another request or terminate	Waiting for next connection

Client (GET)	**Server**
Understand the arguments passed in	
Set up connection to server	Accept connection and ready to listen & accept
Open (create may be needed) and ready to write	
Send request to server	Receive and understand the request
	Send status code response with Content-Length Open, read and send the file (data)
Open (create may be needed) and store data	

Receive and store data into file	Finish sending
Stop storing	Waiting for next connection
Send another Request or terminate	

All the header part ends in a final control code "\r\n", the elements of a header ends in "\r\n". All the data is sent following the response header without control code.

Data Transfer

Client (PUT)	**Server**
`PUT httpname HTTP/1.1\r\n\r\n`	
`data`	
	`HTTP/1.1 status code\r\n\r\n`

Client (GET)	**Server**
`GET httpname HTTP/1.1\r\n\r\n`	
	`HTTP/1.1 status code\r\n`
	`Content-Length \r\n\r\n`
	`data`

2.1 HTTP Client

The client should handle the arguments indicating the server address, port number and the actual actions that the server should execute. Client will first set up connection, understand the request and generate appropriate request header. Based on different action, different processes will be executed.

Client will only be called once for sending request so after sending and printing out the execution result (if needed), then client should be terminated automatically.

`PUT`

Client will first try to open and read the filename, and if any error occurs, print out the error. Then generate an appropriate request header with "`PUT httpname HTTP/1.1\r\n\r\n`" to the server. The send all the data to the server. Waiting for the server's response and print it out if any error response received. After receive the `200 OK or 201 Created` response, the client will send the rest data of the file. Close both `fd` and `sock`.

`GET`

Client will first send an appropriate request header "`GET httpname HTTP/1.1\r\n\r\n`" to server. Then receive the response of status code, `Content-Length`, and part of data in one response. If error message is received, then print it out and close sock. If `200 OK` is received, then try to read `Content-Length`. After analyzing response header, open and write data into the local file. If any open or write error occurs, print it out. If none, then write all data into the local file. Either receive 0 characters or write-in-total touch the `Content-Length` can stop this transaction and close `fd` and `sock`.

```
Input   : Array of arguments: <address>:<port_number> <request>
Input   : Array length: arg_count
1.  char file [4 KB], file_buf [4 KB]
2.  if argc == 1 then
3.      fprintf("SET UP FAILED")
4.  end
5.  obtain the address and port number
6.  Generate sock to set up connection to server
7.  for i ← 1 to arg_count do
8.      dissemble the request into  action_code, httpname, and filename
9.      if action_code!= s or r || strlen(httpname)!=40 then
10.         send(400 Bad Request status code \r\n\r\n)
11.         continue
12.     end
13.     connect(sock, &addr, sizeof(addr))
14.     if strlen(httpname)!= 40 then
15.         fprintf("400 Bad Request")
16.     end
17.     if connect() == -1 then
18.         fprintf(stderr, erno)
19.     end
20.     if acition_code == s then
21.         PUT(httpname, filename)
22.     else if action_code == r  then
23.         GET(httpname, filename)
24.     else
25.         fprintf("400 Bad Request")
26.     end
```

Algorithm 1: Client side main program loop

```
function PUT(httpname, filename)
  if open(filename, O_RDONLY) == -1 then
    fprintf(stderr, erno)
  else
    if read(fd, file_buf, sizeof(buf)) == -1 then
      fprintf(stderr, erno)
    else
      send(sock, "PUT httpname HTTP/1.1\r\n\r\n")
      send(sock, file_buf, read_result, 0)
      file_size += read_result
      if recv(sock, buf, sizeof(buf), 0) == 0 then
        fprintf(stderr, errno)
      else
        write(1, buf, strlen(buf) - 2)
      end
      do
        read(fd, file_buf, sizeof(file_buf)
        send(sock, file_buf, read_result, 0)
        file_size += read_result
      while read_result != 0
    end
  end
```

Algorithm 2: Server side PUT function

```
function GET(httpname)
   send(sock, "GET httpname HTTP/1.1\r\n\r\n")
   recv(sock, buf, sizeof(buf)-1, MSG_WAITALL)
   obtain status code
   if open(filename, O_WRONLY | O_TRUNC) == -1 then
      If open(filename, O_CREAT | O_WRONLY | O_TRUNC) == -1 then
         fprintf(stderr, errno)
         continue
      end
   end
   obtain Content-Length -> file_size
   do
      if write(fd, file_buf, strlen(file_buf)) == -1 then
         fprintf(stderr, errno)
         break
      end
      file_size -= write_result
      if file_size == 0 then
         break
      end
      recv(file_buf, 0, sizeof(file_buf) - 1, 0)
   while recv_result != 0 && write_result != 0
```

Algorithm 3: Client side GET function

2.3 HTTP Server

The HTTP server handles the file containing and deal with the request message sent from the client and give a reasonable reply to the request as well as the data the client requested (if needed). In general, the server end is in an infinite loop listening to the port for the requests until it is being called to terminate.

The server should handle the transaction of `PUT` and `GET`. The `PUT` part is similar to the `GET` part in client side, and vice versa for `GET` in server and `PUT` in client.

The server side will first receive, read and understand the request sent from the client, dividing them into different labels (action code, httpname, version). The control code "`\r\n\r\n`" will be used to identify the end of request header.

PUT:

Following the request header, server will check the httpname's validation via `OPEN(2)` with flags: `O_WRONLY`, `O_TRUNCT`, and `O_CREATE`. Send error message back if `OPEN(2)` or `WRITE(2)` encounters an error, and shut down the connection. Then receive data and store it in the file until meets the Content-Length restriction. After all, send a status code (200 OK or 201 Created) back to client to notify everything is fine on the side of server. Close the file descriptor, shut down connection and clean buffer for next iteration.

GET:

Following the request header, server will check the httpname's validation via `OPEN(2)` with flags: `RDONLY`, sending appropriate status code if `OPEN(2)` or `READ(2)` encounters any error. Shun down the connection and start the next loop iteration, waiting for next connection request. If no error appears, send response back with status code (200 OK). A `Content-Length` will also be sent to server, following control code `"\r\n\r\n"` to indicate header is over. Then send all the data to client until no more words in the file. In this case, no more control code is needed after data. Finally, close the file descriptor, shut down connection and clean buffer for next iteration.

```
1.  Input   : Array of arguments: <address>:<port_number>
2.  Input   : Array length: arg_count
3.  char file [4 KB], file_buf [4 KB]
4.  if argc != 2 then
5.      fprintf("SET UP FAILED")
6.      exit(EXIT_FAILURE)
7.  end
8.  Set up connection: if error occurs, fprintf(error message) and exit(EXIT_FAILURE)
9.  for infinite loop do
10.     accept connection - cl
11.     recv_result = recv(cl, buf, sizeof(buf), 0)
12.     obtain action_code, httpname, version, and Content-Length
13.     if action_code!= PUT or GET || strlen(httpname)!=40 || version!= "HTTP/1.1" then
14.         send(400 Bad Request status code \r\n\r\n)
15.         continue
16.     end
17.     if acition_code == PUT then
18.         PUT(httpname)
19.     else if action_code == GET then
20.         GET(httpname)
21.     else
22.         send(400 Bad Request status code \r\n\r\n)
23.     end
```

Algorithm 4: Server side main program loop

```
function PUT(httpname)
   if recv_result == strlen(request_header) then
      strcpy(file, data)
   else
      recv(cl, file_buf, sizeof(file_buf), 0)
   end
   if open(httpname, O_WRONLY | O_TRUNC) == -1 then
      if open(httpname, O_CREAT | OWRONLY | O_TRUNC) == -1 then
         send(403 Forbidden status code \r\n\r\n)
         continue
      end
   end
   if write(fd, file_buf, strlen(file_buf)) == -1 then
      send(403 Forbidden status code \r\n\r\n)
   else
      send(200 OK or 201 Created status code\r\n\r\n)
      count += write_result
      if count == Content-Length then
         break
      end
      while recv(fd, file_buf, sizeof(file_buf)) != 0 do
         write(fd, file_buf, recv_result)
         count += write_result
         if count == Content-Length then
            break
         end
      end
   end
```

Algorithm 5: Server side PUT function

```
function GET(httpname)
  if open(httpname, O_WRONLY | O_TRUNC) == -1 then
    send(403 Forbidden status code)
  else
    if read(fd, file_buf, sizeof(buf)) == -1 then
      send(400 Bad Request or 403 Forbidden status code)
    else
      send(200 OK status code \r\n)
      send(Content-Length \r\n\r\n)
      send(cl, file_buf, read_result, 0)
      do
        if read(fd, file_buf, sizeof(file_buf) != 0 do
          write(fd, file_buf, strlen(file_buf))
        end
      while read_result != 0
    end
  end
```

Algorithm 6: Server side GET function

2.3 Modularity

Regarding of the principle of modularity, client and server should work individually without impact on each other. In other words, client and server should follow on one protocol but even one of them don't follow it, the other one should also have ability of handle it without any side effect.

Based on this situation, no matter how client work, server should always be in the infinite loop regardless what information it receives. To achieve this, both client and server has a strictly serious process to check the incoming data from outside to make sure it absolutely follows the protocol and won't affect itself.

For server side, the incoming request needs to go under detection. It must follow "action httpname version\r\n\r\n". Both three parts including control code \r\n\r\n will be checked. If any of them doesn't follow the format, an error will be sent.

For client side, the incoming response will be checked. It must follow the standard response header "version status code\r\n\r\n" or extra response header "version statsus code\r\n Content-Length\r\n\r\n" format. The important part is to check each part by '\r\n' for sub-header and a final \r\n to indicate the end of header.