CSE 3300 – Computer Networks and Data Communication

Professor Bing Wang

Assignment 1 – HTTP Client/Server & Jumble Game

Nicholas Lambourne

ndl17004 – 2749404

# Part 1 – HTTP Client/Server

## Description

### Program Design

#### Client

The HTTP client is provided as a self-contained entity using an Object-Oriented design, which allows the BasicHTTPClient to be imported by other programs but can also be run as a script from the command line. All custom methods are contained within the BasicHTTPClient object for ease of use. The client

#### Server

The HTTP server is also provided as a self-contained object available for import as well as for single run use as a stand-alone script. The server is significantly more complicated than the client because it utilises multithreading in order to server multiple clients concurrently.

### How it works

#### Client

On running the client script in its stand-alone mode, an instance of the BasicHTTPClient class is created. As part of its initialisation, it will examine the command line arguments and attempt to parse them into a host address, an object to retrieve, a port to access on the target host and a set of headers. Once this information has been collected (and if there are no errors in the provided data) the Client will use the object, host and headers to create a properly formatted HTTP request string. It will then attempt to connect to the provided host (and port) over TCP/IPV4 using raw sockets. The connected socket will then be returned to the object to wait for an explicit call of the object’s make\_request method, which is responsible for actually sending the previously created request over the pre-prepared socket object. This method prints out the HTTP response object that is received as a result of the sent request and then exits the program.

If at any time the client encounters an error in performing its core functions, it will print out an error message specific to the issue and proper usage instructions for the user.

#### Server

Running the server in “script” mode will create an instance of the BasicHTTPServer class. In its initialisation method it will perform four key functions. First, it will parse the command line arguments provided to the process and retrieve a port number and root directory (both of which are checked for validity). The program will then change the process’ working directory to that provided. At this point the setup is complete and the script requires an explicit call to the run\_server method, which commissions a raw TCP socket to listen on and begins an infinite loop to accept client connections. On connection by a client, a new thread is spawned to deal with the client using the handle\_client method. This method will read an entire request before parsing the request header and assessing the validity of the request (does the file exist etc.). If the file exists the contents are retrieved and attached to a 200 OK HTTP response and sent back to the client. If the file does not exist, then a 404 Not Found message is constructed and sent instead. Once the response is sent that connection is closed and the responsible thread will terminate. Like the client, it has catches for incorrect usage and will inform the user on how to properly use the script.

## Trade-offs

There were a number of trade-offs made in the production of this code in design choice, use (or lack of use) of certain libraries and in the features that were (or were not) included. Several notable examples are listed below.

* OOP Design/Class Structure: the decision to base both the client and server around an OOP paradigm involved several trade-offs in itself. While it did require a non-trivial amount of additional code and structure that weren’t strictly required to fulfil the requirements of this assignment, it does provide a much nicer structure and significantly increased reusability to the code.
* Solely Internal Methods: One design choice that was made fairly early was to include all methods as part of the HTTPServer and HTTPClient classes, despite the fact that several of the methods could be provided as static, or “external” methods. In the use case of this assignment this decision had very little impact, but it would be useful if the classes were ever imported to another script.
* Print vs. Logging: Both scripts make extensive use of print statements instead of using Python’s the logging module. Given that this client is used as a command line application it makes little difference, but proper form would have been to use the logging module, however this would have required a non-trivial amount of configuration in both scripts that was unnecessary.
* HTML Template Responses: Both the default index.html and 404 Not Found responses are included as separate HTML files. While they could have been provided as in-memory strings, it proved to be neater and more modular to provide them as-is. This means that *the templates will need to be stored in the directory provided at the command line in order for the code to word as intended*.
* Configuration vs. Run: The last major trade-off that was (consciously) made was to explicitly separate the “configuration” of the client and server from the “running” aspect by requiring both classes to make an explicit call to the run\_server and make\_request methods. This seemed like a natural choice at the time but could potentially lead to confusion in their use, though the functionality does add a finer level of control to the system.

## Potential Extensions

### Client

The BasicHTTPClient class is not an exhaustive HTTP client implementation (hence the name) and there are definitely areas in which its functionality could be extended:

* Interactive Client: The client could be made interactive by providing the existing functionality in a while loop where the user is prompted for more addresses/resources to retrieve.
* Saving Output: The client could be extended by asking the user for a file in which to store the output (though this same functionality can be achieved by piping at the command line). This would require opening a file and writing to it, rather than printing to the command line.
* Exit Without Interrupt: the client could also be improved by accepting a command to shut down, rather than requiring an explicit interrupt. This could be handled as part of the parsing of the command line arguments.

### Server

Like the BasicHTTPClient, the BasicHTTPServer could also be improved though the addition of enhanced functionality, for example:

* Handling Other HTTP Request Types: the server could be modified to provide access to other HTTP request types like POST, PUT, DELETE and HEAD. The request type could be passed as another command line argument and the HTTP header template made more modular to accommodate this. This is definitely possible but would require significantly more parsing to accomplish (though Python certainly has the capability to fulfil the tasks of those types).
* Handling Other HTTP Response Types: the server in its current form is unable to deal with advanced functionality like providing security/rejecting requests and providing appropriate error codes (it only provides 200 OK and 404 Not Found) in its current form. This could be achieved with Python libraries by examining the permissions on each file before providing them and offering more template responses.
* Interactive Root Directory: the server could be enhanced by allowing the changing of the root directory while in use by prompting the user, taking input and then using Python’s os.chdir() function to move the active directory.

## Test Cases

Overall Design:

The test cases were designed to test common usage situations, including likely misuse.

### Client

* Test Case 1: no input (python3 http-client.py)
  + Rationale: It is likely that a new user will just try and run the script without CLA to begin with.
  + Output:

Incorrect number of arguments!

Client startup failed!

Usage: python3 http-client.py <source> <object> [port] [headers...]

N.B: Headers must be provided in the form HeaderName:HeaderValue

(space separated, but not between Name and Value).

* Test Case 2: python3 http-client.py gaia.cs.umass.edu index.html 80
  + Rationale: given test case
  + Output:
    - the full HTML document and header from the requested page (too long to include).
* Test Case 3: python3 http-client.py gaia.cs.umass.edu index.html 443
  + Rationale: incorrect port (should still output response, even if rejected by host).
  + Output:

400 Bad Request

* Test Case 4: python3 http-client.py gaia.cs.umass.edu bob.html 80
  + Rationale: object does not exist (should still return 404).
  + Output: 404 Not found message
* Test Case 5: Host not alive (local)
  + Rationale: Client should notify the user when it can’t connect to the specified host.
  + Output:

Connection failed, are you sure of your host?

Client startup failed!

Usage: python3 http-client.py <source> <object> [port] [headers...]

N.B: Headers must be provided in the form HeaderName:HeaderValue

(space separated, but not between Name and Value).

* Test Case 5: no
  + Rationale:
  + Output:

### Server

* Test Case 1: no input (python3 http-server.py)
  + Rationale: user will run the script without command line arguments until they know how to use.
  + Output:

Incorrect number of arguments!

Server startup failed!

Usage: python3 http-server.py <port> <root\_directory>

* Test Case 2: python3 http-server.py 50007 .
  + Rationale: normal use
  + Output:

Server started, (listening on 127.0.0.1:50007) waiting for connection...

* Test Case 3: invalid directory (python3 http-server.py 50007 /etk)
  + Rationale: what will happen if the user picks an invalid directory?
  + Output:

Provided root directory is invalid!

Server startup failed!

Usage: python3 http-server.py <port> <root\_directory>

* Test Case 4: invalid port (python3 http-server.py 80 .)
  + Rationale: what will happen if the user picks an invalid port?
  + Output:

Provided port must be greater than 5000 to avoid conflicts!

Server startup failed!

Usage: python3 http-server.py <port> <root\_directory>

* Test Case 5: Connection and provision of existing resource ()

# Part 2 – Jumble Game Client/Server

## Description

### Program Design

#### Client

The Jumble Client script is also provided in Object form for greater mobility/ease of use. The JumbleClient object needs to be instantiated, at which time it will use Python standard libraries to parse the arguments provided and set up a connection to the server. It then offers a continuous interactive command line experience to the user, contacting the server when required for new jumbles and correct answers/confirmation of responses.

#### Server

The Jumble Server script uses a similar design to the HTTP server, using threads to deal with concurrent connections, but sends raw bytes over TCP, rather than relying on a communication standard like HTTP. Like the HTTP client/server and the Jumble Client, it is structured in a self-contained way in that all methods are included as part of the JumbleServer object which must be instantiated and explicitly started.

### How it works

#### Client

The JumbleClient class needs to be instantiated, at which time it will parse the command line arguments, picking out the server address and the port at which it should expect to find the JumbleServer running. It will then use these details to set up a raw TCP socket connection to the server. At this point the client is ready to start the game, which is done by explicitly calling the play\_game method. This will start sending messages to the server in order to identify it as a new connection. It will then enter an infinite “game loop” where it will continuously ping the server for new words and present those at the command line to the user, accepting guesses which are then sent to the server for confirmation or rejection, which is relayed to the user before the cycle starts again. The game stops when the user explicitly calls a Keyboard interrupt at which time the connection is closes and the program ends.

#### Server

Like the Jumble Client, the JumbleServer needs to be instantiated, which is done in the program’s main loop. This instantiation includes the retrieval of the list of words from the provided word list file, the parsing of command line arguments (port) and the creation of an array for storing connection objects. The server is then ready to be explicitly started with the start\_server command which will create a raw listening TCP socket and start an infinite loop for connections. On receipt of a new connection a new thread is spawned and the client is vetted and a game loop is initiated. On the close of connection with a client, the socket is closed and the thread terminated. Each connection is stored so that on the deliberate close of the JumbleServer each connection can be explicitly closed (even if the game is still running) through a Keyboard Interrupt.

## Trade-offs

There were a number of trade-offs made in the production of this code in design choice, use (or lack of use) of certain libraries and in the features that were (or were not) included. Several notable examples are listed below:

* No communication standard: due to the simplicity of the game and communication it was decided that raw text would be sent over the wire rather than using a protocol/standard like HTTP to encode messages. This runs the risk of messages being confused in transit or misinterpreted, but this was not deemed to be serious enough a risk to warrant the additional effort/resources.
* OOP Design: as with the HTTP server and client, the choice to use an Object-Oriented approach added the requirement for more code overhead, but with the added advantage of portability and reuse.

## Potential Extensions

### Client

Possible extensions for the JumbleClient class include:

* Reconnection: on loss of connection to the server, the client would reconnect and identify itself, ensuring that the server would be able to provide the correct answer to a jumble if connection was lost in the middle of a game. This would require both the server and the client to keep additional state in the form of some sort of identifying key.
* Keeping Track of Score: the client could be modified to keep track of a players score, which could be just a simple counter incremented when the player wins, and perhaps decremented on losses.

### Server

Possible extensions for the JumbleServer class include:

* Reconnection to lost clients: see above.
* Prior filtering of word list: at present the word list (some 69000 odd words) is filtered at runtime by repeated request. This could definitely be improved by filtering the word list once before running and removing words that are unsuitable.
* Command Line Arguments as Keyword Arguments: by providing the command line arguments as keyword arguments (e.g. port=50007) they could be processed in any order. This would require a slight adjustment to the current parsing algorithm.

## Test Cases

Overall Design:

The test cases were designed to test common usage situations, including likely misuse.

### Client

* Test Case 1: no input (python3 jumble-client.py)
  + Rationale: user will run without arguments to test.
  + Output:

Client Startup Failed

Incorrect number of arguments

Usage: ./http-client <server-address> [port]

* Test Case 2:
* Test Case 3:
* Test Case 4:
* Test Case 5:

### Server

* Test Case 1: no input (python3 jumble-client.py)
  + Rationale: user will run without arguments to test.
  + Output:

Server started, (listening on 0.0.0.0:50007) waiting for connection...

* Test Case 2:
* Test Case 3:
* Test Case 4:
* Test Case 5: