HTTP Client and Server System Design

-- Author: Brian Reid

This document is intended to provide a description of the overall design of the HTTP server and client, a description of "how it works," and some design tradeoffs that were considered when building this program.

Client Module

Overall design

First and foremost, the HTTP Client is written in Python and designed to be exposed as the command line interface for the user. That is, the user can simply install the Python modules shipped within this repository using `pip` and immediately being using the client within that environment. Once the module is installed, requests can be made to any server over any destination port number. At this time, two HTTP methods are supported through the client: GET and PUT. A GET request will attempt retrieve the specified file from the server if it exists. The output will vary based on whether that file exists and how the server handles requests for documents that do not exist. If the file exists, the content will be logged out to the console.

How it works

The `myclient` module utilizes the builtin `socket` module (https://docs.python.org/3/library/socket.html) to establish a low-level networking interface.

First, we parse the arguments from the command line. This gives us the server, destination port, HTTP method, and filename that the user desires. This operation is done with the widely used `argparse` builtin (https://docs.python.org/3/library/argparse.html).

Once we have the required information from the user, we declare a socket object so that we can begin making network requests either to the locally-running server (if it was started) or externally to another server. We specify with AF_INET that we wish to communicate with IPv4 addresses and also pass SOCK_STREAM to indicate that we wish to communicate using TCP. We also allow the socket to reuse the same address in case the client makes and ungraceful exit and the socket from the previous run remains open on the host.

Next, the code forks to handle either a GET or POST request. In the case of a GET request we simply prepare a string representing the HTTP request protocol. Only one header is added, and it is the `Host:` header. For example, if the user is requesting index.hmtl from www.cnn.com over port 80: `"GET index.html HTTP/1.1\r\nHost: www.cnn.com\r\n\r\n"` Before sending the request to the server over the TCP socket, we need to encode this request message into a bytes-like object. Common practice is to use UTF-8 encoding. If not explicitly provided, the methods in this module will default to UTF-8 encoding and decoding. Finally, we receive the response in chunks of the buffer size from the server and log that content out to the console.

In the case of a PUT request, the `myclient/static` directory that ships with the module is scanned for the filename specified. If the file is not found, the client outputs a useful error message informing the user of this and recommending that they check the spelling of the file name. Assuming the file is found, we prepare the HTTP request message similar to how the GET request was formed. For example: `"PUT`

`index.html HTTP/1.1\r\nHost: www.umass.edu\r\n\r\n"` After the base of the request has been formed, we must add the file data to the body of the request (after the double carriage return / line feed). We do this in a convenient way using Python's context manager (`with open("file.txt", "r"), as f:`) to easily open the file, stream the data to the end of our request message, and close the file. Once the full request has been formed, we break the data into chunks based on the buffer size and begin sending that data to the server. Finally, we wait for a response and log it to the console, as is done with GET requests.

In both the GET and PUT cases, the socket is closed following the transaction.

Tradeoffs

One important tradeoff I had to consider is what to do if there is an error when receiving the data after the request goes out to the server. In practice, I think the natural thing to do is retry the server with the same request and see if you can get the data you originally requested. Many high-level http client libraries (python requests, etc.) do implement configurable retry logic. For example, if the stream coming back from the server suddenly stops, the client should retry and either receive the whole document again, or better yet design a way to only request data from the point after the last received byte. This is ideal if there are large amounts of data being transferred from the server to the client. In my case, I made the decision to simply move on, stop trying to receive data from the server, and log the data that we did receive out to the console. Of course in practice this in unsatisfactory, as the client wants the whole document. As an exercise though, I felt that this decision aided my ability to get through a few roadblocks I encountered on the way and write working software more quickly by not having to think about the long list of things that can go wrong during this connection.

Another tradeoff to consider was from which directories the client should allow the user to select files to post to the server. Let's again think first about the practical solution. If this were an http client that was going out to customers, they would want to be able to select any file from their system. To support this, the client should bring up the OS's file selecting system to allow the user an easy selection. We could also allow the user to provide an absolute file path in the command line. Even better, we might want to allow them to connect to a remote system and select files from that system (obviously, that is out of the scope of this assignment). However, this program is really about testing the HTTP client's ability to send data to another host and receive a response. For that reason, I decided to only allow files within the `myclient/static` to be valid choices for PUT requests. If a user wants to test a special file, it's not unreasonable to ask them to upload those files there since we are only testing.

Finally, you may have noticed the decision to allow the client to reuse the same address, effectively allowing to kill another socket process that might be running in order to re-open the socket and form a new connection. Obviously in a production environment this would be dangerous and you would never want an application process to blindly force it's way onto a socket. However, for development purposes I felt that this was acceptable and worth the time it saved me trying to figure out which processes to kill on my machine to free up the socket if the program exited in an ungraceful manner.

Server Module

Overall design

Similar to the client module, the server is written in Python and exposed as a command line interface. Once installed, the user can immediately start the server from the command line. The user has the option to

select on which port number they would like to start the server. Once the server starts, the program enters an infinite loop, waiting to accept incoming connections. At this time, both GET and PUT HTTP requests are supported. When a GET request is received, the server returns either a 200 message with the file or a 404 message. When a PUT request is received, the server writes the supplied file to `myserver/static`.

How it works

The `myserver` module utilizes the builtin `socket` module (https://docs.python.org/3/library/socket.html) to establish a low-level networking interface.

First, we parse the command line argument from the user with the widely used `argparse` builtin (https://docs.python.org/3/library/argparse.html). In this case, we need only the port on which the server should run.

In the same way as the client module , we declare a socket object so that we can begin accepting requests. We specify with AF_INET that we wish to communicate with IPv4 addresses and also pass SOCK_STREAM to indicate that we wish to communicate using TCP. We also allow the socket to reuse the same address in case the client makes and ungraceful exit and the socket from the previous run remains open. Finally, we call listen() so the server is ready to accept connections.

We now enter the infinite loop, listening on the socket. When a connection comes in, it is accepted (assuming the backlog of yet-to-be accepted connections is not full; only one is allowed on this server for simplicity) and data begins streaming in to the server. In a similar way to how the client accepts the response data from the server, the server receives data in byte chunks equal to the buffer size until data stops coming in. At that time, we join the chunks of data together to re-create the original message string. If the flow of data is interrupted, we raise this exception because we may or may not have received enough data to know what to do with the request and deem the connection unstable.

At this point, we can get the method and filename from the incoming message. We can do this because HTTP protocol defines the format of the message string. Assuming we recognize and support the HTTP command (we'll talk about what happens otherwise shortly), we need to respond to either a GET or a PUT request. If the request is a GET, we scan the `myserver/static` directory for a file with the specified name. If the file is found, we generate an HTTP response message with the 200 OK status code and message. We then use Python's context manager to open the file, stream the data and append it to the end of the message after the double return / line feed. Finally, the data is encoded to bytes with UTF-8, sent to the requesting client, and the connection is closed.

If the request is for a PUT, we first check the `myserver/static` directory to see if the file already exists. We do this because it affects the response code and message that we send. Then, we use the context manager again to write the data (the message content sent to the server after the double return / line feed) to `myserver/static/<filename>` where `<filename>` is the file requested by the client. If we just created this file, we send a 201 Content created message, otherwise we send 200 OK, indicating that the resource was updated successfully. Again, these messages are sent UTF-8 encoded and the connection is closed.

If we do not recognize the HTTP method supplied, the server responds to the client with a 400 Bad Request message. This indicates that we do not know what to do with the request data sent. The connection is again closed.

Finally, if anything goes wrong during these steps and an exception is thrown, that exception will be caught in the `except` block. To keep the server alive, we send a 500 Server error message and close the connection. This indicates to the client that the request content was acceptable, but there was some sort of a problem.

Tradeoffs

The tradeoff discussed in the client section to let the program reconnect to the same address when restarted also applied here. In an effort to be less redundant, I will not discuss it again here.

You may have noticed that the server closes the connection with the client no matter what the result of the connection is. In this case, it helps keep the server environment more contained and aides in debugging and siloing connections since the server is not configured to do any concurrent processing of requests. Additionally, we assume that the server should not expect any additional data from the client after sending the response. In this case, closing the connection is reasonable because while testing the server we are only sending adhoc requests. In a production environment, our server may be expecting the server to follow-up with additional requests. If this were the case, it would be wise to keep the connection alive in order to avoid establishing a new handshake connection each time the client wished to send another request.

A small decision that had to be made was whether to return 200 OK or 204 No content when a PUT request has resulted in a successfully updated file. This comes down mostly to a matter of personal preference, but I feel 200 OK is more appropriate for this situation. To me, it indicates that everything is running smoothly and the request has been properly handled.

With more time, I would love to give better feedback to the client when a request results in a 400 error. This system is a simple one, so perhaps its less important, but as a developer I'm constantly frustrated with API's that have both poor documentation and give little or unhelpful feedback to clients. Obviously this situation is a little different since we are purely implementing an HTTP server, but the principles still apply. Attaching a message body to 400 responses that state the reason for the error and/or some information that we gathered from the request is helpful for all parties.

Other improvements

Aside from the improvements listed in the tradeoffs section for both the client and the sever, there are a few other items that would be nice improvements to make. First and foremost, caching would be a great improvement. A very simple idea would be to store the last `N` or most commonly requested `N` documents on the client with a time last modified for the document. With this information, we could send an if-modified-since header to the server. The server would them only need to stream the requested back to the server if it has been recently modified.