COMP3331 Report Assignment 1

z5592060

1. **Programming Language and Code Organisation**
    - State the programming language used (i.e., C, Java, or Python).

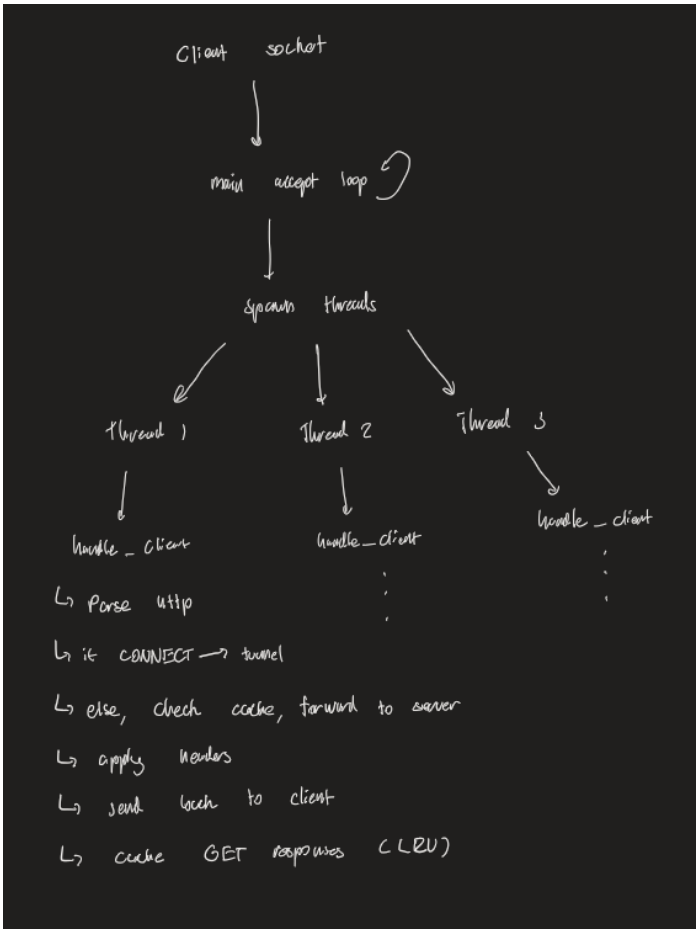
      Used Python
    - Briefly describe your code structure (e.g., key directories, files like Makefile).

      Proxy server runs entirely on proxy.py. Logs are stored in log.log file, which is created at runtime.
2. **High-Level Design**
    - Provide a brief overview of your program's main components and their interactions.

      Firstly, we have models of HTTPResponse, HTTPRequest, and CacheEntry as a standardised way of handling responses, requests and caching within the code. Then we have several helper functions to simplify repetitive tasks. recv_until receives data from a socket until a delimiter is found – typically used to extract HTTP headers. Recv_exact receives data from a socket until it reaches an exact number of bytes. It is typically used to read the body once we know Content-Length after using recv_until. We also have parsing functions such as _split_head_body which just splits raw HTTP head and body apart. Parse_http_request and parse_http_response for fitting raw bytes into HTTPResponse and HTTPRequest models. Split_url for splitting url into host:port and path, which is useful for converting url into origin form to forward to the server. Normalise_url for the keys for the cache. For handling client connections, I used a main loop first which opens a socket and waits to accept connections. Upon accepting a connection, I create a new thread which runs the handle_client loop for concurrency. This loop consists of the main proxy logic of receiving client requests, forwarding it to the server and sending the response back to the client. This loop also includes error handling, logging (which is also handled in the send_log_error_response and generate_clf_entry functions), handling chunked responses, editing headers, CONNECT, persistence, caching and more. Every successful and error transaction is logged in CLF format. The CONNECT tunnel opens a raw socket to the target server and uses a select() loop

bidirectionally until the connection is closed. We store the cache and do all LRU with an OrderedDict structure. We have two simple methods to adjust the cache – cache get (which simply returns based on the key) and cache_put, which allows new entries into the cache (and automatically does LRU). With concurrent connections, we also use a threading.Lock to ensure that shared resources like the cache and logfile are not corrupted.

- You may include a simple diagram if it helps clarify the design.



3. **Data Structures**

   - Summarise the key data structures used in your proxy (e.g., for HTTP message handling, caching).

     I use an HTTPResponse model which stores all information in an HTTP 1.1 response in a structured format. It stores the version, status code, reason, header and body. I also use an accompanying HTTPRequest model, storing the method, url, version, header and body. It is accompanied by helper functions to parse the raw bytes from a socket into these models. We also have a simple cacheentry model for storing responses and their size for LRU calculations, and max_cache_size/max_object_size compliance.

- Focus on the most important structures and how they support your design.

  The HTTPResponse and HTTPRequest models enable a structured manner to create, update and get HTTP 1.1 requests and responses. The pipeline involves parsing raw bytes into these structures, which allow for the easy manipulation of headers as well as logging, caching, and forwarding to origin. The cacheentry model builds upon these two fundamental models to effectively store entire requests in the OrderedDict structure.

4. **Limitations**

   - Mention any known limitations of your implementation (e.g., incomplete feature support, specific conditions where the program might fail).

     The code is not fully RFC compliant, and we simplify caching. For instance, we ignore HTTP caching directives such as Cache-control, expires, pragma, etc. We unconditionally store any 200 OK GET that is within the size limits. Not only this, but we process chunked responses naively and simply. The code is not able to handle anything beyond the simple base case, and is not able to handle multiple transfer encodings like gzip. Also in responses without content length or transfer encoding we simply read until the connection is closed, which might violate some RFC 9112 rules. Furthermore, I use novel methods in the code to parse messages which might not be up to RFC standards.

5. **Acknowledgments**

   - Indicate any external code or resources you've used, with links or brief citations.

     These resources helped me get started

     https://realpython.com/python-sockets/#python-socket-api-overview

     https://stackoverflow.com/questions/39541706/basic-understanding-of-simple-socket-server

     https://stackoverflow.com/questions/7155529/how-does-http-proxy-work