The system is designed to simulate interactions between a web server and web clients following a HTTP-like request – response model. Clients asynchronously send requests to the server, which processes them and returns corresponding responses. To support concurrency and manage multiple clients simultaneously, the system uses Haskell's threading capabilities and *MVar* for thread-safe data sharing. Each interaction (request and response) is logged to an external file, and the server terminates once a limit of 100 requests is reached.

A modular structure, consisting of four main components: *Main*, *Client*, *Server*, and *Types* modules is used to structure the system. The *Main* module initialises and coordinates the system, managing *MVars* and controlling thread pools for clients. The *Client* module Manages client threads, sending requests asynchronously to the server via the *addRequest* function. The *Server* processes requests from the client module and adds them to a queue, generating responses, and logs the interactions. Finally, the *Types* module defines shared data structures like Request and Response, logEntries that define the core data definitions throughout the system.

This modular approach enhances clarity, scalability, and maintainability by isolating responsibilities into distinct modules, making the system easier to debug, modify when adding the extension feature.

The *Request* and *Response* types are defined in the *Types* module as record types. This design allows for structured and ordered data representation. The *Request/Response* records includes the following fields:

- *timestampRequest/Reesponse* field - stores the *UTCTime* of when the request/response was created.
- *contentRequest/Response* field - holds a string describing the request/response details.

Both types are derived as instances of the *Show* typeclass, to allow easy logging and debugging. A type alias was also introduced for the *MVar* managing the queue of pending requests. This design decision ensures that requests are processed in a FIFO (First In, First Out) manner.

The system leverages *MVar* as it enables thread-safe, mutable state management. A *MVar* (Mutable Variable) is a location that can hold a single value or remain empty. Two key operations *putMVar* (to fills the *MVar* if empty) and *takeMVar* (to empty the MVar if full) – ensuring synchronisation between threads through FIFO.

However, in this system, the *modifyMVar_* function was used to manipulate the MVar. This approach still offers a thread-safe approach to appending the queue in the MVar without the need to empty it. Additionally a separate *MVar* was introduced to track of the number of processed requests, ensuring the total does not exceed the 100-request limit. The *System.Exit* module is then implemented to terminate the server once the request limit of 100 requests has been reached.

The system logs all requests and response using *appendFile* function, which appends new entries to the log file without overwriting existing data. An extra feature – latency calculation – was added to compute the time difference between the request and response timestamps. Implementing this feature required significant changes to the programs structure of the basic requirements.

A new record for logged entries was introduced, along with a dedicated *MVar* to track these entries independently of the request queue. This change required the redefining of the handler function to include the new *MVar*[*LogEntries*], which in turn required modifications to the *processRequest* function. Also, since *forkIO* does not provide access to thread IDs, unique IDs for the request and response were manually created instead. This presented a significant challenge, as I aimed to label which client was making each request. In the current output, multiple threads share the same request ID (count), making it difficult to identify which client's request was processed first.

Despite these challenges, the system successfully calculated the latency of each interaction and identified the fastest latency, which was 0 seconds, along with associated requests and responses responsible. In the future I am to be able to optimise the system further by exploring alternative concurrency models like *TVar* to compare performance and scalability with *MVar*.

.