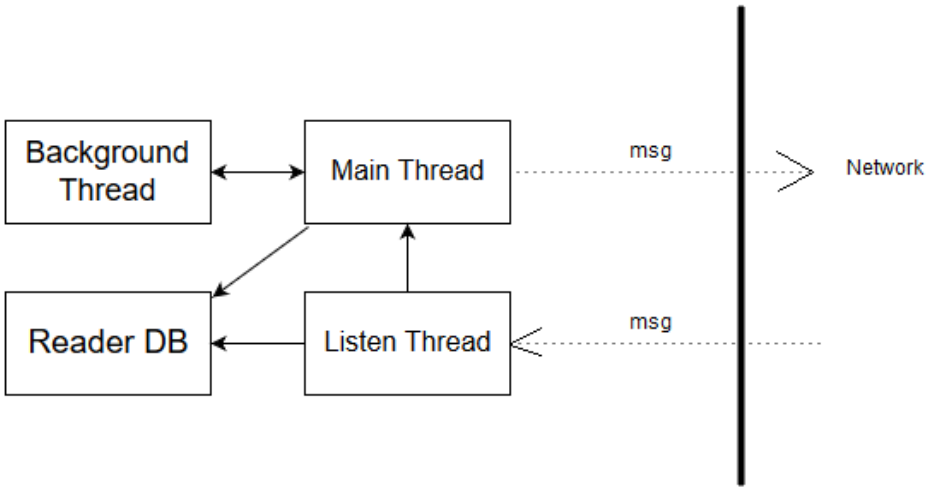
COMP3331 - Networks Assignment: Networking eDocuments (Report)

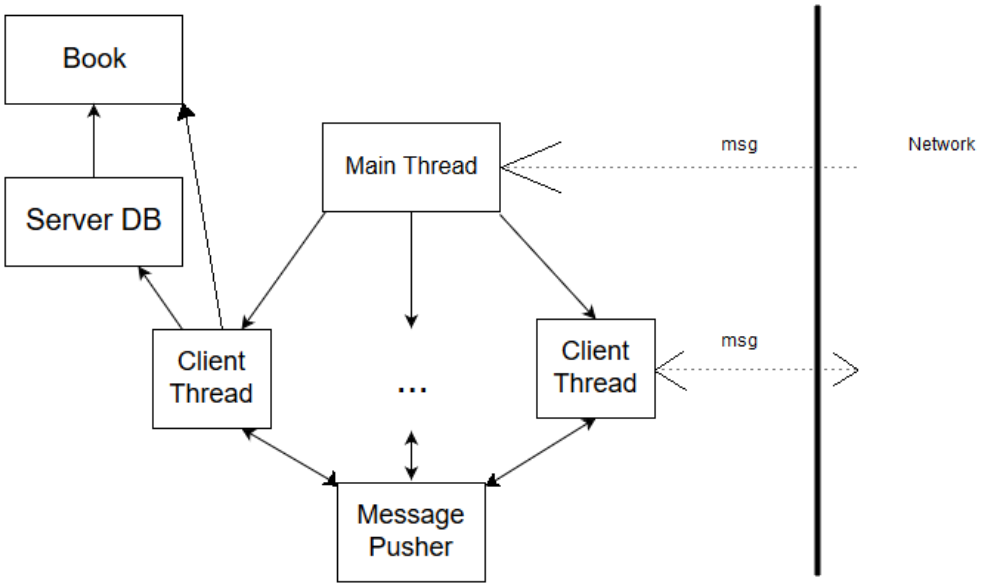
Program Design

The following diagrams illustrate the design of each program (detailing classes, dependencies, and information flow).

- Reader.py



- Server.py



How System Works

For the reader, the database stores all information about a post (namely: sender, book, page, line, read status, and content), and provides methods for how information can be retrieved and stored. The 'Main Thread' runs continuously, reading in user commands and carrying out the appropriate procedures. The background thread is a constantly running routine that executes indirect tasks, based on the user

command. In particular, for the pull and push mode of the reader, it uses functions in 'Main' to send query messages to the server to update the reader's database of forum posts. The 'Listen Thread' listens to messages from the TCP connection with the server continuously. It decodes the messages, and carries out the appropriate procedures.

For the server side, book content is stored in 'Book' objects (1 for each book), and forum posts in the database. These two objects provide methods for other classes to retrieve / store information (the latter only for the database). The 'Main Thread' accepts connections from a client (reader), before spawning a thread for that client with the associated socket, which is responsible for listening to ONLY that socket. These client threads are also responsible for decoding received messages, carrying out appropriate procedures, and sending back any messages, using database and book methods as necessary. Also, the first message received is a client's 'intro' message, which is parsed by the 'Client Thread' and bundled into a 'Client Object', which merely carries further information about the host end (eg username). To push data to clients in the 'push' mode, the 'Message Pusher' is triggered (by a client thread) to help deliver the message to all the client threads whose clients are running in 'push' mode.

Error handling is simple: for example, the user wants to display a page from an invalid book. The server will respond with an error message (format of which is described under 'Message Design') that contains "Book is invalid", which is displayed on the reader's end.

Talk about the way server handles errors, and sends them back as error strings to the reader, where it will be displayed. For single responses (eg after submitting a post), it will have the status right after the codeword. If there are errors, the error mesage will follow after a '#' after the status. For streams, the error string will simply be of the format: '#Error#[error msg]'. The errors will be considered in the function that receives the data as a stream.

Message Design

In a message, there is a single phrase, (optionally) followed by parameters. The phrase and parameters are separated using the main delimiter '#', chosen as it is an uncommon character to include in normal messages (eg post content). For parameters which are interpreted as a list (such as when server sends a list of post ID's), a comma is used as the delimiter.

For example, a forum post being exchanged will be contained in two strings of the following format:

#PostInfo#[PostID]#[Sender]#[Book]#[Page]#[Line]
#PostContent#[PostID]#[Post content]

Another example is a request to submit a new post. The following message formats will be used:

Client: #UploadPost#PostInfo...|#PostContent
Server: #UploadPostResp#Success OR #UploadPostResp#Error#[error msg]

Note: All characters are to be treated literally, unless they are in '[]' brackets, which gives special semantics.

Design Considerations and Tradeoffs

- **Reader's Listen Thread:** With the addition of the 'Listen Thread', the reader was always listening for messages (which works for the chat function, when an incoming request is not initiated by the reader itself but a server), at the cost of tasks being carried out by two independent classes. For example, updating data would involve 'Main' to send a request, and 'Listen' to process posts and insert into database (rather than a single step).
- **Storing a forum post in two parts (post info, and post content):** This was to facilitate future updates where the content of the post can be extremely long and contain formatting characters (eg emojis), thus storing it separately to the post's information will make it clearer. However, this leads to more difficult processing, as there are two parts to a forum post that needs to be interpreted.

Possible Improvements

- **Server Database:** A random ID for a post is better than an increasing ID, as it could give the reader an idea of when it was inserted (the ID is only meant to identify the post, not give away sequence ordering)
- **Message format class:** Due to multiple message formats being exchanged, it would be a better idea to have a class that was solely responsible for formatting messages based on the action (eg display, submit post etc). Currently the messages are hardcoded in their format. With such a 'Message Formatter' class, it can encapsulate the message formatting and make maintenance of message formats MUCH easier.

Issues

- **[Formatting issue] Reader.py:** When the reader's threads prints to screen, it will print on the same line as the '>' character for the user input. Partially fixed this by delaying reading user input for a small period (until other printing is completed).
- **[Execution issue] Reader.py:** Any error that causes 'main' to stop execution (including 'Keyboard Interrupt') will cause the listening thread to continue running.

Code References

- Code used as a framework for creating sockets, and making connections between a server and client taken from "Socket Programming HOWTO" (https://docs.python.org/2/howto/sockets.html)
- Code used for handling multiple clients using 'Select' for networks was adapted from "Python Network Programming: Echo Server with Select" (http://ilab.cs.byu.edu/python/select/echoserver.html)
- The template code for constructing and running threads was adapted from "Python Multithreaded Programming" (http://www.tutorialspoint.com/python/python_multithreading.htm