# Design document – 2140 Home exam

## TCP code in connection.c

The code in connection.c implements basic operations on TCP sockets, and work as expected.

**tcp_connect()** the function creates a TCP socket using socket() and converts hostname to a network address with inet_pton(), then attempts to connect to a server socket at the specified address with connect().
**tcp_read()** uses read() to read n bytes from the sock to the buffer.
**tcp_write()** uses write() to write 'bytes' number of bytes from the buffer to the sock.
**tcp_write_loop()** uses write() in a loop to write 'bytes' number of bytes from the buffer to sock. It continues writing until all the bytes are successfully written.
**tcp_close()** simply calls the close() function to close the socket.
**tcp_create_and_listen()** creates a server socket with socket(), and then binds it to the specified port with bind() and then starts listening for incoming connections with listen().
I have chosen a backlog size of 26 for the listen function. This decision is based on the fact that the clients' unique IDs range from A-Z, allowing for a maximum of 26 connections.
**tcp_accept()** accepts an incoming connection on the server socket by using accept()
**tcp_wait()** wait for socket events using the select() function without a timeout
**tcp_wait_timeout()** wait for socket events using the select() function with a timeout of 'seconds' seconds.

## Design choices in Proxy.c

**Bufsize and memory handeling**
I chose a BUFSIZE of 500 because it provides enough space to hold typical records without allocating excessive memory. In handleClient() I free the allocated buffer space whenever a complete buffer is processed, minimizing unnecessary memory allocation. Although this approach may result in frequent mallocs and frees if senders transmit many records, it efficiently manages allocated data by minimizing simultaneous allocations.

**Datastructure for client list**
I decided to use a double linked list to manage the clients. This choice offers several advantages. Firstly, it enables fast insertion of new clients. Additionally, the double linked list does not consume extra memory when there are only a few connected clients. Furthermore, when removing a specific client, the previous and next clients can be efficiently linked without the need to iterate through the entire list or shift elements in an array. This improves the overall performance and flexibility of client management operations.

**Helper functions**
Proxy works as expected for every predefined method. In addition there are four helper functions, add_client that adds a client c to the linked list of clients, which is called from handleNewClient when a new client is added. Free_clients() that iterates through the client list and closes all the fds and frees all the allocated memory, which is called every time the program is shut down. Find_client() finds a client based on the id and returns it, otherwise it returns NULL, which is used in forwardMessage() when trying to find the client that the message is being sent to. Remove_bytes_from_buf removes n bytes from a buffer of size s, which is used in handleClient() when removing already read records from the buffer when there is more to read.

**Event loop design**

Initializing variables
The event loop begins by initializing the necessary variables: max_fd to track the highest file descriptor, fds as the file descriptor set, and rc to store the return value of tcp_wait.

Setting up the file descriptor set
The file descriptor set is then set up by clearing it with FD_ZERO and adding server_sock to monitor new client connections. Then it iterates through connected clients, and each file descriptor is added to the set. This is done inside the loop to ensure that the set accurately reflects the current state of connected clients.

Waiting for socket events
Next, the program calls tcp_wait to wait for activity on any file descriptor in the set.

Handling possible new client connection
When activity is detected, it checks if the server socket (server_sock) has activity, indicating a new client connection. If so, it calls handleNewClient. The rc value is then decremented by one, as one event has been handled.

Handling possible new client data
If there are still unhandled events (rc > 0), the program iterates over the client list and checks for activity on each client's file descriptor. If activity is detected, handleClient is called, where either data is read or the client is disconnected and subsequently removed.

Loop termination + cleenup
The loop continues until all clients have disconnected (num_clients = 0). Once this condition is met, the server socket is closed, and the program returns 0 to indicate successful execution.

## Conversion between XML and binary formats.

The program supports conversion between XML and binary formats by firstly converting the XML or binary to an internal structure Record, and then when sending the message to the right destination it converts the Record to the needed format.

When a client has activity the handleClient function is called. The function tries to read what the client has written to the socket into a buffer, then based on if the client is supposed to send XML or binary the corresponding functions BinaryToRecord and XMLtoRecord are called. These functions convert the bytes in the buffer to an internal structure Record. When a Record is found the program calls the forwardMessage() function. This function reads the Records destination id and tries to find a client with this id. If it finds this client, it checks if the client is supposed to receive XML or binary and calls the corresponding functions recordToBinary or recordToXML to convert the Record structure into binary or XML again. Then this converted message is sent to the client.

**XMLtoRecord**
This function reads bytes from a buffer and interprets them as an XML representation of a Record. It first creates a new Record and then looks for the record start tag, then looks for any other predefined attributes, source, destination, id, group, semester, grade and courses and fills the corresponding attributes in the Record. Then at the end it looks for a record end tag, if this is found the Record is returned. If the information in the buffer is not sufficient to create a record, for example that there is incomplete information about a present attribute or that the record start or end tag is missing, the function returns NULL.

**BinaryToRecord**
This function reads bytes from a buffer and interprets them as a binary representation of a Record. It firstly creates a new Record and then it reads the first byte in the buffer and sets the different flags in the record. With this information it tries to read the next bytes according to which attributes it is supposed to have and fills the Record with this information. At any point if the buffer does not contain the information it is supposed to it returns NULL. If not it returns the Record at the end.