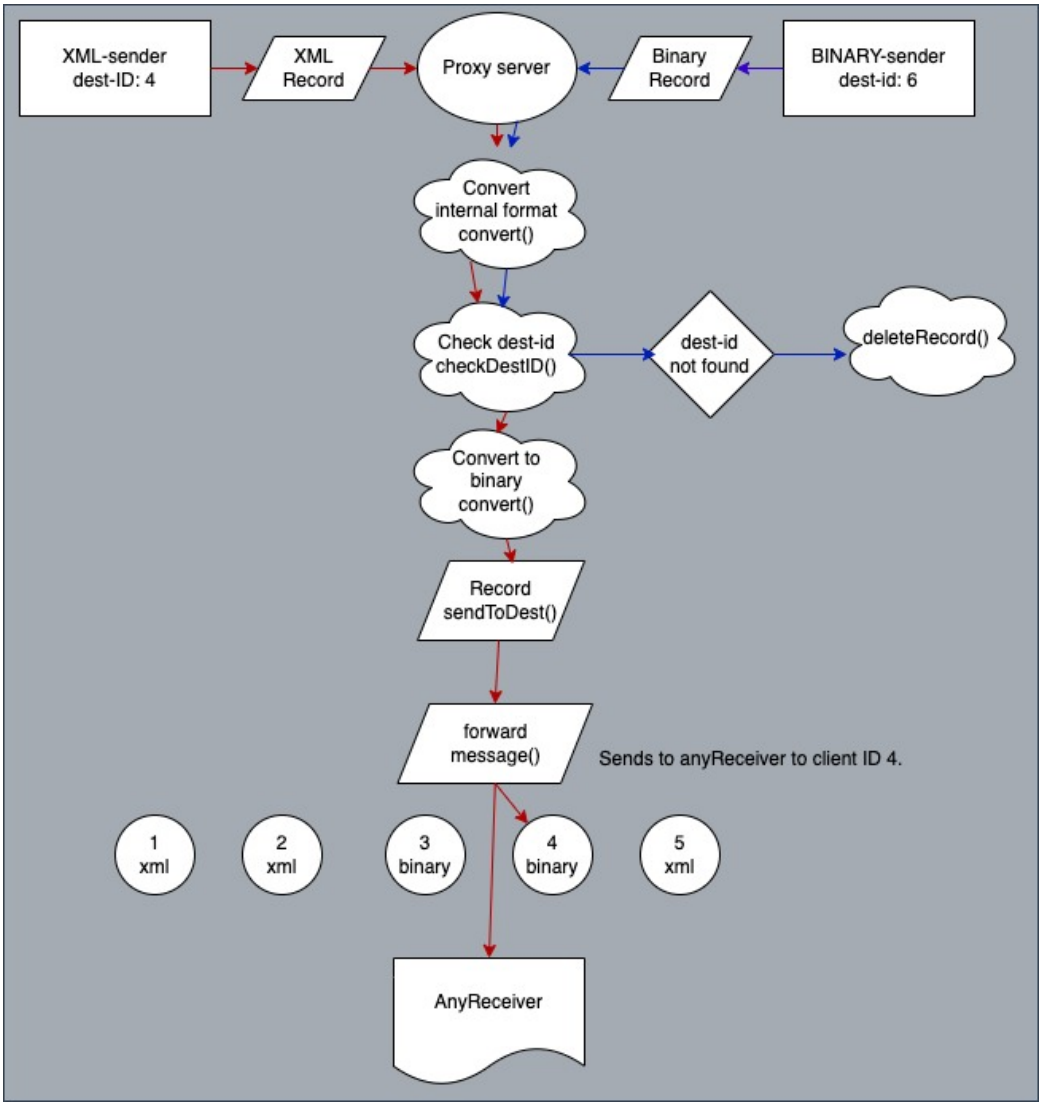
Design Document

This design document provides a detailed view of the code for a simple TCP proxy server, its event loop and conversion process between XML and binary formats. I have attached a picture of how the program works:



TCP Connection Handling (connection.c)

The connection.c file is designed to handle TCP connection methods. It includes operations such as connect, read, write, close, accept, and listening on a socket.

All of these functions are encapsulated in this file for code organization and reusability. This enables easy debugging and helps keeping the code in other files such as proxy.c concise and focused on its primary responsibility of client handling and message forwarding. The check_error() makes it easy to log if something with the TCP handling goes wrong. If an error occurs, it provides an appropriate error message. This facilitates better debugging and helps to quickly identify network connection issues.

Proxy Event Loop (proxy.c)

The proxy.c file contains the event loop that constantly monitors for new client connections and handles message forwarding. It maintains a list of connected clients, their sources, and the format they support (XML or Binary).
The event loop has been designed with the 'select' method with the tcp_wait_timeout, which allows for asynchronous connection of clients. This is used to monitor multiple file descriptors to see if they're ready for I/O. It enables the server to manage multiple clients simultaneously without blocking each other. Upon detecting activity on the server socket, a new client is accepted, initialized and checked for format type and source id, and then added to the linked client list. In contrast, activity on any other socket implies data transfer from a client. In this case, the incoming data is read, and if it completes a record, and then it's forwarded to the appropriate client based on the destination id.
There are multiple reasons why I chose a linked list over an array. The most important reasons are effectiveness when it comes to insertion and deletion, having a dynamic size rather than an arrays fixed size and that linked lists allocate memory for elements dynamically when they are added, whilst arrays require contiguous memory allocation. If we do not know the size of the list, a linked list can be more efficient.

XML to Binary Conversion and Vice Versa

The conversion between XML and binary formats is implemented in the recordToFormat.h, recordFromFormat.h, recordToXML, recordToBinary, XMLtoRecord, and BinaryToRecord functions. This design allows for a modular approach where format conversion is abstracted away from other functionalities, promoting code reusability.
When a client sends a message, the format type specified by the client is used to determine how to convert the incoming message into a Record data structure. The 'recordToXML' function converts Record to an XML format message, and the 'recordToBinary' function converts Record to a binary format message.
If a client's format type is 'X', the XMLtoRecord()' function is used to convert the XML message to a Record. If the format type is 'B', the BinaryToRecord() function is used for the conversion. After the Record is created, it is forwarded to the appropriate client using the forward_message() function. In this method the premade methods converts the record to the appropriate format of the client’s format type.

What works and does not

The proxy server is capable of receiving and sending data, with either multiple sender or multiple receivers. With the included test files the proxy server was capable of running 8/11 tests, with correct expected files being written to. If I had more time, I would implement a buffer inside the struct Client, and send client->buffer as parameter to for example XMLtoRecord(), so that the proxy can successfully keep track of each client's buffer. Everytime a new activity happens on the socket, handle_new_client() is called, and a new buffer is created.
Also, I have a problem with closing the program itself. I have tried using while(1) and then breaking when there is no activity on the socket. When I implement this the anyReceiver gets in an infinite loop. Hence, I have some bytes that are not freed, but I do not have any memory leaks or errors when I run valgrind –leak-check=full and when I compile the only warning I get is from binfile.c, which I did not write.

Conclusion

Overall, this design promotes separation of concerns and modularity, with each part of the program being responsible for a specific task. This makes the code more readable, maintainable, and easier to debug. It is also designed to handle multiple clients concurrently, which is a critical requirement for any scalable server application!