CS 39006: Assignment 5
Message Oriented TCP
Deadline: 17-March-2023, 2 pm

We know that TCP is a byte-oriented protocol, i.e., it treats the data sent as a stream of bytes. In this assignment, you will be building a message-oriented TCP protocol using the standard TCP connection below it. We want to implement our own socket type, called *MyTCP* (*SOCK_MyTCP)* socket, that will guarantee that anything sent with a single send call will be received by a single receive call. So it will behave similarly as UDP, just that it will also be reliable, and messages will be delivered exactly once in FIFO order as in TCP. You will be implementing it by putting a layer of software between the *MyTCP* calls and the TCP socket call. Your program will take the calls of *MyTCP* and convert them to calls of TCP, along with additional bookkeeping that will be needed to ensure the message-oriented behavior.

More specifically, you will be implementing a set of function calls *my_socket*, *my_bind*, *my_accept*, *my_connect*, *my_listen*, *my_send*, *my_recv*, and *my_close* that implement *MyTCP* sockets. The *MyTCP* socket will be implemented underneath by a TCP socket through which all communication will actually happen. **The parameters and return values to these functions and their return values are exactly the same as the corresponding functions of the TCP socket, *except for my_socket***. The functions will be implemented as a library. Any user wishing to use *MyTCP* sockets will write a C program that will call these functions in the same sequence as when using TCP sockets. A brief description of the functions is given below.

- *my_socket* – This function opens a standard TCP socket with the *socket* call. It also creates two threads R and S (to be described later), allocates and initializes space for two tables **Send_Message** and **Received_Message** (to be described later), and any additional space that may be needed. The parameters to these are the same as the normal socket( ) call, except that it will take only SOCK_MyTCP as the socket type.
- *my_bind* – binds the socket with some address-port using the *bind* call.
- *my_listen* – makes a *listen* call.
- *my_accept* – accepts a connection on the *MyTCP* socket by making the *accept* call on the TCP socket (only on server side)
- *my_connect* – opens a connection through the *MyTCP* socket by making the *connect* call on the TCP socket
- *my_send* – sends a message (see description later). One message is defined as what is sent in one *my_send* call.
- *my_recv* – receives a message. This is a blocking call and will return only if a message, sent using the *my_send* call, is received. If the buffer length specified is less than the message length, the extra bytes in the message are discarded (similar to that in UDP).
- *my_close* – closes the socket and cleans up everything. If there are any messages to be sent/received, they are discarded.

So how do you ensure that a message is received as a whole on the other side? You will open a standard TCP connection to actually communicate. This ensures the reliability, exactly once, and in-order delivery you want. However, this does not guarantee that whatever is sent using a *my_send* call on the *MyTCP* socket is sent and received in a single send/recv call on the underlying TCP socket. So you need to somehow remember and send message boundaries and reconstruct the message on the receiving side even if it is not received in one go. Design this.

So to implement each myTCP socket, we need the following:

1. One TCP socket through which all actual communication actually happen.
2. Two threads R and S. Thread R handles all messages received from the TCP socket, and thread S, handles all sends done through the TCP socket.
3. Buffers, **Send_Message** and **Received_Message**, to store messages to send (given by the *my_send* call) and messages received (to be read by the *my_recv* call.

The threads are killed and the data structures freed when the *MyTCP* socket is closed. For simplicity, in this assignment, you can assume that a program will create only one *MyTCP* socket, and the server is iterative. You can also assume that the maximum length of a message can be 5000 bytes.

The thread R behaves in the following manner. It waits on a *recv* call on the TCP socket, receives data that comes in, and interpretes the data to form the message (the complete data for which may come in multiple *recv* calls), and puts the message in the **Received_Message** table.

The thread S behaves in the following manner. It sleeps for some time (T), and wakes up periodically. On waking up, it sees if any message is waiting to be sent in the **Send_Message** table. If so, it sends the message using one or more send calls on the TCP socket. You can only send a maximum of 1000 bytes in a single send call.

The *my_socket*, *my_bind*, *my_listen*, *my_accept*, *my_connect*, and *my_close* calls are easy to understand, they are mostly wrappers around the corresponding TCP calls, with some additional work to create threads/buffers in *my_socket* and clean them in *my_close*. The *my_send* call behaves as follows: it puts the message in the **Send_Message** table if the table has a free entry and returns immediately. If the table does not have a free entry, it gets blocked until an entry is free. The *my_recv* call behaves as follows: it sees if there is any message to receive in the **Received_Message** table. If yes, the message is returned, and the entry is deleted from the table. If not, the *my_recv* call blocks until a message is received. For simplicity, you can implement blocking (in both *my_send* and *my_recv*) by using the *sleep* call to wait for some time and then try again. The *my_send* and *my_recv* calls should return to the user only when either the data to be sent is written in the **Send_Message** table, or a message is received from the **Received_Message** table.

Design the message formats and the **Send_Message a**nd the **Received_Message** tables properly. Note that the tables are sometimes shared between different threads and would require proper mutual exclusion. The maximum number of entries in the tables should be 10.

What to submit

All the functions should be implemented as a static library called *libmsocket.a* so that a user can write a C program using these calls for reliable communication and link with it Look up the **ar** command under Linux to see how to build a static library. Building a library means creating a *.h* file containing all definitions needed for the library (for ex., among other things, you will #define SOCK_MyTCP here), and a *.c* file containing the code for the functions in the library. This *.c* file should not contain any *main()* function, and will be compiled using **ar** command to create a *.a* library file. Thus, **you will write the *.h* header file (name it *mysocket.h*) and the *.c* file (name it *mysocket.c*)** from which the *.a* library file will be generated. Any application wishing to call these functions will include the *.h* file and link with the *libmsocket.a* library. For example, when we want to use functions in the math library like *sqrt()*, we include *math.h* in our C file, and then link with the math library *libm.a*.

You should submit the following files in a single tar.gz file:

- *mysocket.h* and my*socket.c*
- a *makefile* to generate *libmsocket.a*
- a file called *documentation.txt* containing the following:
    - For *mysocket.h* and *mysocket.c*, give a list of all data structures used and a brief description of their fields, and a list of all functions along with what they do in rsocket.c.
    - A description of exactly what happens, with details of which functions are called in what sequence and what tables are used in what manner when (i) a *my_send* call is made, and (ii) a *my_recv* call is made.