CS 39006: Assignment 4
File Transfer Using Sockets

Date: 14-Feb-2022
Deadline: 7-March-2022, 2 pm

In this assignment, you will be building support for reliable communication over an unreliable link. We will keep some parts simple for now. For ex., you will not have to worry about message ordering or receiving duplicate messages.

You have been introduced to the function calls *socket*, *bind*, *sendto*, and *recvfrom* to work with UDP sockets. Assume that the TCP sockets are not there. We know that UDP sockets are not reliable, meaning that there is no guarantee that a message sent using a UDP socket will reach the receiver. We want to implement our own socket type, called MRP (*My Reliable Protocol*) socket, that will guarantee that any message sent using a MRP socket is always delivered to the receiver. However, unlike TCP sockets, MRP sockets may not guarantee in-order delivery or exactly-once delivery of messages. Thus messages may be delivered more than once (duplicate messages) or out of order (later message delivered earlier). Also, like UDP sockets. MRP is message-oriented, and not byte-oriented.

MRP sockets guarantee reliable delivery using a simple approach. The sender sends a message using a UDP socket, and stores the message and the time it is sent in a table called the ***unacknowledged-message table.*** The receiver, on receiving the message, sends an ACK message back to the sender. The sender, on receiving the ACK for a message, removes that message from the ***unacknowledged-message table***. If either the message or the ACK is lost, a timeout occurs at the sender, and the sender resends the message, and resets the sending time of that message in the ***unacknowledged-message table***. This process repeats for each message.

To implement each MRP socket, we use the following:

1. One UDP socket through which all actual communication actually happen.
2. Two threads R and S. Thread R handles all messages received from the UDP socket, and thread S, handles the timeouts and retransmissions. More details of R and S are given later.
3. Two tables, ***unacknowledged-message table*** and ***received-message table***. The first table contains the list of all messages that have been sent but not yet acknowledged by the receiver, along with the last sending time of the message. The second table contains all messages received in the socket.

The threads are killed and the data structures freed when the MRP socket is closed. For simplicity, in this assignment, you can assume that a program will create only one MRP socket (otherwise, you will have to create the two tables for each socket, and S and R also become a bit more complex).

You will be implementing a set of function calls *r_socket*, *r_bind*, *r_sendto*, *r_recvfrom* , and *r_close* that implement MRP sockets. **The parameters and return values to these functions and their return values are exactly the same as the corresponding functions of the UDP socket, *except for r_socket*.** The functions will be implemented as a library. Any user wishing to use MRP sockets will write a C program that will call these functions in the same sequence as when using UDP sockets. A brief description of the functions is given below.

- *r_socket* – This function opens an UDP socket with the *socket* call. It also creates the 2 threads R and S, and allocates initial space for the tables. The parameters to these are the same as the normal socket( ) call, except that it will take only SOCK_MRP as the socket type.
- *r_bind* – binds the socket with some address-port using the *bind* call.
- *r_sendto* – sends the message immediately by *sendto*. It also adds a message sequence no. at the beginning of the message and stores the message along with its sequence no. and destination address-port in the ***unacknowledged-message table*** before sending the message. With each entry, there is also a time field that is filled up initially with the time of first sending the message.
- *r_recvfrom* – looks up the ***received-message table*** to see if any message is already received in the underlying UDP socket or not. If yes, it returns the first message and deletes that message from the table. If not, it blocks the call. To block the *r_recvfrom* call, you can use *sleep* call to wait for some time and then see again if a message is received. *r_recvfrom*, similar to *recvfrom*, is a blocking call by default and returns to the user only when a message is available.
- *r_close* – closes the socket; kills all threads and frees all memory associated with the socket. If any data is there in the ***received-message table***, it is discarded.

The thread R behaves in the following manner. It waits for a message to come in a recvfrom() call. When it receives a message, if it is a data message, it stores it in the ***received-message table***, and sends an **ACK** message to the sender. If it is an **ACK** message in response to a previously sent message, it updates the ***unacknowledged-message table*** to take out the message for which the acknowledgement has arrived.

The thread S behaves in the following manner. It sleeps for some time (T), and wakes up periodically. On waking up, it scans the ***unacknowledged-message table*** to see if any of the messages timeout period (set to $2T$) is over (from the difference between the time in the table entry for a message and the current time). If yes, it retransmits that message and resets the time in that entry of the table to the new sending time. If not, no action is taken. This is repeated for all messages in the table every time S wakes up.

Design the message formats and the ***unacknowledged-message table*** and the ***received-message tables*** properly. Note that the tables are sometimes shared between different threads and would require proper mutual exclusion. You can assume that the maximum size of these tables will not be more than 50 at any time.

Testing your code

To test the program, write two programs *user1.c* and *user2.c*. The program in *user1*.c will create a MRP socket M1 and bind it to the port 50000+2*<last 4 digits of your roll no> (for ex., if your roll no. is 1001, the port no. is 52002). It then reads a long string from the keyboard (25 < string size < 50), and sends each character of the string **in a separate message** to *user2.c*. The messages are sent using M1 by making the *r_sendto* calls. The program in *user2.c* will create a MRP socket M2 and bind it to the port 50000+2*<last 4 digits of your roll no> + 1 (for ex., if your roll no. is 1001, the port no. is 52002 + 1 = 52003). It then receives each message from *user2.c* using the *r_recvfrom* call on M2 and immediately prints the character received on the screen. If some character is received more than once due to retransmission, it will be printed more than once and characters may also be printed out of order because of retransmission. For example, if the string entered is "Madagascar", the string printed can be (just one possibility), "Madaagggscaar". But each character must be printed at least once.

As the actual number of drops in your machine or in the lab environment will be near 0, you will need to simulate an unreliable link. To do this, in the library created, add a function called *dropMessage()* with the following prototype:

int dropMessage(float p)

where *p* is a probability between 0 and 1. This function first generates a random number between 0 and 1. If the generated number is < *p*, then the function returns 1, else it returns 0. Now, in the code for thread R, after a message is received (by the *recv_from()* call on the UDP socket), first make a call to *dropMessage()*. If it returns 1, do not take any action on the message (irrespective of whether it is data or ack) and just return to wait to receive the next message. If it returns 0, continue with the normal processing in R. Thus, if *dropMessage()* returns 1, the message received is not processed and hence can be thought of as lost. Link the programs in *user1.c* and *user2.c* with this new library. **Submit your code with the *dropMessage()* calls in R, do NOT remove these calls from your code before you submit.**

The value of *T* should be 2 seconds (do not hardcode it deep inside your code, specify it in a .h file (see below)). The value of the parameter *p* (the probability) should also be specified in the same .h file (see below). When you test your program, vary *p* from 0.05 to 0.05 in steps of 0.05 (0.05, 0.1, 0.15, 0.2….,0.45, 0.5). For each *p* value, for the same string, count the average number of transmissions made to send each character (*total number of transmissions that are made to send the string / no. of characters in the string*). Report this in a table in the beginning of the file *documentation.txt* (see below).

Even though it is not needed for this assignment, you should try to also vary T (especially low values) and see its effect. This is an extremely important parameter that affects the transmission. If you do this, set *p* to 0, and use the nanosleep() call of Linux in S as the sleep() call has a resolution of seconds only, so you cannot set T to anything less than 1 second using a sleep() call.

What to submit

The five required functions, plus the function *dropMessage()*, should be implemented as a static library called *librsocket.a* so that a user can write a C program using these calls for reliable communication and link with it (the function *dropMessage()* will not be called by the user but you will call it when you test your program). 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_MRP 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 *rsocket.h*) and the *.c* file (name it *rsocket.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 *librsocket.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.*

The value of the parameter *T* should be #defined as “#define T 2” in the *rsocket.h* file. The value of the parameter p should also be #defined in this .h file.

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

- *rsocket.h* and *rsocket.c*
- *user1.c* and *user2.c*
- a *makefile* to generate *librsocket.a*
- a makefile to create the two executable files to run from *user1.c* and *user2,c* respectively
- a file called *documentation.txt* containing the following:
    - For *rsocket.h* and *rsocket.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.
    - The table with the results for varying *p* values as described earlier.