CS 39006: Assignment 7

Reliable Communication over Unreliable Links
Date: 28-Feb-2019
Due: March 20, 2019, 11 pm

Objective:

The objective of this assignment is to build support for reliable communication over an unreliable link. The unreliable link will be implemented with a UDP socket. You will have to give APIs to the user that will do reliable send/receive over this unreliable link by appropriate implementation of the APIs.

Problem Statement:

In this assignment, you will be building support for reliable communication over an unreliable link. In particular, you will build a message-oriented, reliable, exactly-once delivery communication layer (you can think of it as TCP without byte-orientation and in-order delivery, or UDP with reliable delivery). Message ordering is not needed, so messages with higher ids can be delivered earlier.

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 exactly once. However, unlike TCP sockets, MRP sockets may not guarantee in-order delivery of messages. Thus messages may be delivered out of order (later message delivered earlier). However, a message should be delivered to the user exactly once.

To implement each MRP socket, we use the following:

1. One UDP socket through which all actual communication happen.
2. One thread X. Thread X handles all messages received from the UDP socket, and all timeout and retransmissions needed for ensuring reliability.
3. At least the following three data structures associated with it (you can have others of your own if you need them): ***receive buffer***, ***unacknowledged-message table***, and ***received-message-id table***. The ***receive buffer*** contains the application messages that are received. This will contain each message sent exactly once before it is read by the user. The ***unacknowledged-message table*** contains the details of all messages that have been sent but not yet acknowledged by the receiver, along with the last sending time of the message. This will be used to decide which messages need to be retransmitted if no acknowledgement is

received. The ***received-message-id table*** contains all distinct message ids that are received in the socket so far. This will be used to detect duplicate messages.
You can assume a maximum size of 100 bytes for each message and maximum table sizes of 100 messages for each table. The tables should be **dynamically created when the socket is opened** and **freed when the socket is closed**. The table pointers (or pointers to any other data structure you may need) can be kept as global variables, so that they will be accessible from all API functions directly.

The broad flow on send and receive of messages is as follows:

Sending an application message for the first time: The user calls an API function *r_sendto()* to send its message. The send API adds a unique id (use a counter starting from 0) to the application message and sends the message with the id using the UDP socket (this should happen in a single sendto() call). It also stores the message, its id, destination IP-port, and the time it is sent in ***unacknowledged-message table.*** This is done in the user thread (main process) and the thread X has no role here. The send API function returns after the send to the user.

Handling message receives (application and ACK) from the socket and Retransmission of Messages: The thread X waits on a select() call to receive message from the UDP socket or a timeout T. Given that X can come out of the wait either on receiving a message or on a timeout, its actions on each case are as follows:

- If it comes out of wait on a timeout, it calls a function ***HandleRetransmit()***. This function handles any ***retransmission of application message*** needed. The function scans the ***unacknowledged-message table*** to see if any of the messages' timeout period (T) 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. At the end of scanning all messages, the function returns. On return of the function, the thread X goes to wait on the select() call again with a fresh timeout of T.
- If it comes out of wait on receiving a message in the UDP socket, it calls a function ***HandleReceive()***. The function checks if it is an application message or an ACK.
  - If it is an application message, call a function ***HandleAppMsgRecv()***, which does the following:
    - o check the ***received-message-id table*** if the id has already been received (duplicate message). If it is a duplicate message, the message is dropped, but an ACK is still sent. If it is not a duplicate message, the message is added to the ***receive buffer*** (without the id) including source IP-port and an ACK is sent. The function returns after that.
  - If it is an ACK message, call a function ***HandleACKMsgRecv()***, which does the following:
    - o If the message is found in the ***unacknowledged-message table***, it is removed from the table. If not (duplicate ACK), it is ignored. The function returns after that.

After handling the message, the HandleReceive() function returns. On return of the function, the thread X goes to wait on the select() call again ***with a timeout of T_{rem}, where $T_{rem} \leq T$ is the timeout remaining when the select() call came out of wait because of the message receive***.

Receiving a message by the user: The user calls an API function *r_recvfrom()*, which either finds a message in the receive buffer or not. If there is a message in the receive buffer, the first message is removed and given to the user, and the function returns the no. of bytes in the message returned. If there is no message in the receive buffer, the user process is blocked. So there is again no direct role of the thread X here. The function will return when there is a message in the receive buffer (if a message comes, it will be put in the buffer by X in the background).

You can assume there will be exactly one MRP socket created by a process for send and receive, so only one set of the variables/threads are needed.

You will be implementing an API with a set of function calls *r_socket*, *r_bind*, *r_sendto*, *r_recvfrom* , and *r_close* that implement MRP sockets. **The parameters 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 static library. Any user wishing to use MRP sockets will write a C/C++ program that will call these functions in the same sequence as when using UDP sockets. The library will be linked with the user program during compilation.

A brief description of the functions is given below. All calls should return 0 on success (except r_recvfrom() which should return the no. of bytes in the message) and -1 on error.

- *r_socket* – This function opens an UDP socket with the *socket* call. It also creates the thread X, dynamically allocates space for all the tables, and initializes them. 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 UDP socket with the specified address-port using the *bind* call.
- *r_sendto* –Adds a message id at the beginning of the message and sends the message immediately by *sendto*. It also stores the message along with its id 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.

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.

Testing your code:

1. Write a file ***rsocket.h*** that will contain
   a. #includes for the sockets to work
   b. #define for the timeout T (set to 2 seconds) and the drop probability *p* (see description of ***dropMessage()*** function below)
   c. prototypes of all the functions in the API listed above + a function dropMessage() which is described below.
2. Write a file ***rsocket.c*** that contain all global variable declarations for the thread X and the data structures, and implementation of all the functions in the API + ***dropMessage()***. This should NOT contain any main() function.
3. Create a static library called ***librsocket.a*** (look up the ***ar*** command in linux)
4. Write two test 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*<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 < 100), 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*<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. Insert the ***dropMessage()*** function at appropriate place (see below).
5. #include *rsocket.h* in user1.c and user2.c and compile them with *librsocket.a* (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* byusing the flag –lm during compilation)
6. Run *user1* and *user2* to see if the strings are transmitted correctly. Also measure the metric below by inserting appropriate code in your library.

<u>*dropMessage()* function</u>:

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

<center>***int dropMessage(float p)***</center>

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 X, 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 X. Thus, if *dropMessage()* returns 1, the message received is not processed and hence can be thought about as lost. **Submit your code with the *dropMessage()* calls in X, do NOT remove these calls from your code before you submit.**

The value of *T* should be 2 seconds (#define in *rsocket.h*). The value of the parameter *p* (the probability) should also be specified in the *rsocket.h* file with a #define. When you test your program, vary *p* from 0.05 to 0.5 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*). You can do this by adding a counter in the appropriate part of the code for thread X. Report this in a table in the beginning of the file *documentation.txt* (see below).

What to submit:

You should submit the following files in a single zip file:

1. *rsocket.h*
2. *rsocket.c*
3. *makefile* (this should include all commands to create a library named *librsocket.a* from your source files.)
4. A file called *documentation.txt* containing description of the different files. This file should include:
    a. The table described above for the no. of retransmissions
    b. A list of all messages and their formats, with a brief description of the use of each field
    c. A list of all data structures used and a brief description of their fields

<u>Do not include any other file in your submission, and upload the submission only in zip format.</u>