CS39006: Assignment 5
Emulating End-to-End Reliable Flow Control over Unreliable Communication Channels

Date: March 01, 2024

Due Date: March 22, 2024

In this assignment, you will be building support for end-to-end reliable data transfer on top of an unreliable communication channel, with a window-based flow control approach.

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 MTP (My Transport Protocol) socket, that will guarantee that any message sent using a MTP socket is always delivered to the receiver in order. Also, like UDP sockets. MTP is message-oriented, and not byte-oriented.

For each MTP socket, the sender and the receiver maintain a sending window (`swnd`) and a receiving window (`rwnd`), respectively. `swnd` is a data structure that contains the sequence numbers of the messages sent but not yet acknowledged, and the window size indicating the maximum number of messages that can be sent without receiving the corresponding acknowledgement (ideally, this should not be more than the available space at the receiver buffer at any instance of time). Similarly, `rwnd` is a data structure indicating the sequence numbers that the receiver is expecting to receive and a window size indicating the maximum number of messages it can receive depending on the available space at the receiver-side message buffer. The messages are of fixed size (assume that message size is 1 KB), and the messages have increasing sequence numbers (sequence number is of 4 bit length). The sender always starts with the sequence number 1. The receiver maintains a message buffer of size equivalent to 5 messages. The sender maintains a message buffer of size equal to 10 messages. Once the receiver receives an in-order message, the message is written to the buffer after removing the MTP header, the free space in the buffer is computed and the `rwnd` size is updated accordingly. The receiver then sends an ACK message to the sender which piggybacks the updated `rwnd` size, and the sequence number of the last in-order message received within `rwnd`. If the receiver receives an out of order message, it keeps the message in the buffer (if the message sequence number is within `rwnd`), but sends an ACK message to the sender with the sequence number of the last in-order message received and the updated `rwnd` size. Note that the MTP receiver also needs to handle the duplicate messages by identifying them with the sequence number and then dropping them if already received once. However, for the duplicated messages also, it sends an ACK message to the sender with the sequence number of the last in-order message received and the updated `rwnd` size. The receiver side application periodically reads the messages (in 1 KB block, the entire message is read atomically) from the message buffer and frees it up. Message are always delivered in order to the application.

At the sender side, MTP maintains a sender-side message buffer where the sender-side application writes data in a block of 1KB. MTP initially sets `swnd` = 5 and starts sending the messages after adding the required header (the header contains the message sequence number; think of if you need any additional field in the header). Once the MTP sender receives an ACK message, it updates the `swnd` accordingly (slides the window till the last message acknowledged if not a duplicate ACK and increases/decreases the window size based on the piggybacked `rwnd` size in the ACK message). The MTP sender also sets a timeout equivalent to T seconds. If it does not receive an ACK message within the timeout duration, it retransmits **all** the messages from the `swnd` (which had been transmitted earlier but not been acknowledged).

Multiple user applications can create MTP sockets simultaneously; however, in your implementation, you can assume that at any instance, there can be a maximum of N = 25 number of active MTP sockets. To implement these MTP sockets, we use the following:

1. UDP sockets through which all transport layer communication happens for the corresponding MTP socket.
2. An init process P that initializes two threads R and S, and a shared memory as mentioned next. Thread R handles all messages received from the UDP socket, and thread S handles sending messages, and the timeouts and retransmissions. More details of R and S are given later.
3. A shared memory chunk SM containing the information about N MTP sockets. SM[i] is a structure indicating (i) whether the MTP socket with ID i is free or allotted, (ii) process ID for the process that created the MTP socket, (iii) mapping from the MTP socket i to the corresponding UDP socket id, (iv) the IP and port address of the other end of the MTP socket, (v) the send buffer, and (vi) the receive buffer of the MTP socket with ID i (fixed size array of messages), (vii) `swnd` (a structure of the sender window size along with an array of message sequence numbers that have been sent but not acknowledged), and (viii) `rwnd` (a structure of the receiver window size along with an array of message sequence numbers that have been received but not acknowledged). The shared memory is assigned a name that is assumed to be known a-priori to all processes/threads that need to access it.
4. The init process should also start a garbage collector process G to clean up the corresponding entry in the MTP socket if the corresponding process is killed and the socket has not been closed explicitly.

You will be implementing a set of function calls `m_socket`, `m_bind`, `m_sendto`, `m_recvfrom`, and `m_close` that implement MTP sockets. The parameters and return values to these functions are exactly the same as the corresponding functions of the UDP socket, except for `m_bind`. The functions will be implemented as a library. Any user wishing to use MTP 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.

- `m_socket` – This function opens an UDP socket with the socket call. The parameters to these are the same as the normal `socket()` call, except that it will take only SOCK_MTP as the socket type. `m_socket()` checks whether a free entry is available in the SM, creates the corresponding UDP socket if a free entry is available, and initializes SM with corresponding entries. If no free entry is available, it returns -1 with the global error variable set to ENOBUFS. If any error is received while creating the UDP socket, errno is set to the corresponding error.

- `m_bind` – binds the socket with some address-port using the bind call. Bind is necessary for each MTP socket irrespective of whether it is used as a server or a client. This function takes the source IP, the source port, the destination IP and the destination port. It binds the UDP socket with the source IP and source port, and updates the corresponding SM with the destination IP and destination port. If any error is received while binding the UDP socket, errno is set to the corresponding error, and SM is not updated. The idea here is that a single MTP socket is used to communicate with another pre-specified MTP socket only (different from how an UDPP socket behaves, even though MTP uses UDP underneath).

- `m_sendto` – writes the message to the sender side message buffer if the destination IP/Port matches with the bounded IP/Port as set through `m_bind()`. If not, it drops the message, returns -1 and sets the global error variable to ENOTBOUND. If there is no space is the send buffer, return -1 and set the global error variable to ENOBUFS. So the `m_sendto` call is non-blocking.

- `m_recvfrom` – looks up the receiver-side message buffer to see if any message is already received. If yes, it returns the first message (in-order) and deletes that message from the table. If not, it returns with -1 and sets a global error variable to ENOMSG, indicating no message has been available in the message buffer. So the `m_recvfrom` call is non-blocking.

- `m_close` – closes the socket and cleans up the corresponding socket entry in the SM and marks the entry as free.

The thread R behaves in the following manner. It waits for a message to come in a `recvfrom()` call from any of the UDP sockets (you need to use `select()` to keep on checking whether there is any incoming message on any of the UDP sockets, on timeout check whether a new MTP socket has been created and include it in the read/write set accordingly). When it receives a message, if it is a data message, it stores it in the receiver-side message buffer for the corresponding MTP socket (by searching SM with the IP/Port), and sends an ACK message to the sender. In addition it also sets a flag `nospace` if the available space at the receive buffer is zero. On a timeout over `select(),` it additionally checks whether the flag `nospace` was set but now there is space available in the receive buffer. In that case, it sends a duplicate ACK message with the last acknowledged sequence number but with the updated `rwnd` size, and resets the flag (*there might be a problem here – try to find it out and resolve!*). If the received message is an ACK message in response to a previously sent message, it updates the `swnd` and removes the message from the sender-side message buffer for the corresponding MTP socket. If the received message is a duplicate ACK message, it just updates the `swnd` size.

The thread S behaves in the following manner. It sleeps for some time (< T/2), and wakes up periodically. On waking up, it first checks whether the message timeout period (T) is over (by computing the time difference between the current time and the time when the messages within the window were sent last) for the messages sent over any of the active MTP sockets. If yes, it retransmits **all** the messages within the current `swnd` for that MTP socket. It then checks the current `swnd` for each of the MTP sockets and determines whether there is a pending message from the sender-side message buffer that can be sent. If so, it sends that message through the UDP `sendto()` call for the corresponding UDP socket and updates the send timestamp.

Design the message formats and the shared memory properly. Note that different fields in the shared memory are accessed by different threads and processes and would require proper mutual exclusion.

Testing your code:

To test the program, write two programs user1.c and user2.c. user1.c creates an MTP socket M1, binds it to IP_1, Port_1 (local IP and Port), IP_2, Port_2 (remote IP and Port). User2.c creates an MTP socket M2 and binds it to IP_2, Port_2 (local IP and Port), IP_1, Port_1 (remote IP and Port). M1 then uses the m_sendto() call to transfer a large file (size > 100 KB), and M2 uses the m_recvfrom() to receive the contents of that file and write it in a new file. You can run user1.c and user2.c with different IP/Port pairs to create multiple MTP sockets.

You should first start with one pair of MTP sockets, i.e. with a single instance of user1.c and user2.c, and then extend it for multiple applications. If your code supports only two statically assigned sockets with the support of all other functionalities, then you'll be evaluated out of 75% of the total marks.

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 `recvfrom()` 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 5 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 file, count the average number of transmissions made to send each message (total number of transmissions that are made to send the message / no. of messages generated from the file). 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 libmsocket.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_MTP` 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 msocket.h) and the .c file (name it msocket.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.

The value of the parameter T should be `#defined` as “`#define T 5`” in the msocket.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:

- msocket.h and msocket.c
- initmsocket.c that initializes the MTP socket by starting the R and S threads and the garbage collector process
- user1.c and user2.c
- a makefile to generate libmsocket.a
- a makefile to create the executable to run initmsocket.c
- 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 msocket.h and msocket.c, as well as for initmsocket.c, give a list of all data structures used and a brief description of their fields and their purpose, and a list of all functions along with what they do in msocket.c and initmsocket.c.
    - The table with the results for varying p values as described earlier.