Emulating End-to-End Reliable Flow Control over Unreliable Communication Channels

21CS30037 P. Datta Ksheeraj
21CS10041 Mayukha Marla

Introduction:

The provided code implements a socket library called **MTPSocket** using shared memory and semaphores for inter-process communication. This report provides a detailed analysis of the implementation, including its functions, usage, and potential improvements.

Project Structure:

The project directory consists of the following files.

1. **msocket.h** - The main header file containing macros, structures and function prototypes.

2. **msocket.c** - This implements the body for each of the function declared as a prototype in msocket.h

3. **Initmsocket.c** - This file declares shared memory, semaphores, and runs R and S threads (receiver and sender) and also runs garbage thread when necessary. This file also includes the function **dropmessage**.

4. **Makefile** - for running the whole application.

5. **Application purpose files**:
    a. **user1.c** , **user2.c** - This two files use the library for sending 20-25 hardcoded words

b. **user3.c**, **user4.c** - These two files use the library for sending an arbitrary text file or any between them

c. **user5.c**, **user6.c** - These two files use the library just for testing purpose i.e to check the functionality of created system over multiple users together

d. **Sample.txt** - file used for transfer as mentioned earlier.

Description:

msocket.h:

a. **Macros** defined:

i. *SOCK_MTP*: For defining a new socket type.
ii. *MAX_SOCKETS*: For maximum number of sockets which is 25 according to the assignment
iii. *MAX_MESSAGE_SIZE:* For maximum message size which is 1024 bytes
iv. *MAX_WINDOW_SIZE*: For maximum window size which is initially 5 for both sender and receiver.
v. *EINVAL*: Error macro for invalid value
vi. *ENOBUFS*: Error macro for no space in buffer
vii. *ENOTBOUND*: Error macro for the case if the socket is not given a bind call
viii. *ENOMESG*: Error macro for indicating the receiver buffer is not having any message to extract
ix. *T*: Timeout for a message mentioned as 5sec in assignment.
x. *P*: probability for receiver accepting the message

b. **Structures**:

i. **MTP Header**: Consists of 2 fields
.

1. Sequence number: This ranges from 1 to 15 and is assigned to messages in order.

2. Flag: This field comes in handy in receiver thread to deliver messages in order to application. This flag indicates whether this

message is delivered in order to the receiver. So the sender will be setting it to 0 and sends and later the receiver handles it.

ii. **Message :** Consists of 2 fields

1. A header (*MTPHeader*): This forms the header of the message to be sent

2. Data (*char[]*): A character array of max size 1024 bytes which holds the actual data or message of the packet.

iii. **Window**: Consists of 4 fields

1. Window size (*int*) : a variable of int type which holds the current size of the window.

2. Sequence array (*int[]*) : This array holds the sequence numbers present in the window. This makes different sense in receiver and sender. In sender this represents those messages which are sent but not yet acknowledged. On the receiver side this represents the messages which are expected to come from the sender. The array just contains numbers in sorted order 1 2 3 4 5 6 7 8 9 10 11 12 16 14 15 1 2 3 …….

3. Left pointer (*int*) : Stores the index of starting position of window and is of int type.

4. Right pointer (*int*) : Stores the index of the ending position of the window and is of int type.

iv. **MTP Socket**: Consists of 11 fields

1. Is_free (*int*) : This indicates whether the socket entry is free or not. 0 indicates not free and 1 indicates free.

2. Process_id (*pid_t*) : This indicates the process id of the process which created the socket.

3. Udp_socket_id (*int*) : This indicates the udp socket id assigned to this socket when made a call to socket().

4. Bound (*int*) : This indicates whether the socket is binded or not. 0 indicates not free and 1 indicates free.

5. Destination_address (*struct sockaddr_in*) : stores the address of the destination of this socket i.e the other side of the socket.

6. Send_buffer (*Message[]*) : This is an array of messages of size 10 which stores the messages sent from the application side which are to be sent to the receiver.

7. Receive_buffer (*Message[]*) : This is an array of messages of size 5 which stores the messages received by the receiver and are to delivered to the application

8. Swnd (*Window*) : This maintains the current sender window of this socket.

9. Rwnd (*Window*) : This maintains the current receiver window of this socket.

10. lastSent (*int*) : This stores the timestamp of the last sent message from this socket

11. noSpace (*int*) : This is a flag which gets set if the current free size of the receive buffer is 0 and vice versa.

v. **SOCK_info**: Consists of 4 fields

1. Sockid (*int*) : Stores the sockid of the currently used socket (for creation and binding purposes)

2. Port (*int*) : Stores the port to which it gets connected.

3. Ip (*char[32]*) : Stores the ip address with which it gets identified

4. Errno (*int*) : Stores any error if it occurs during the execution of the respective function.

c. **Prototypes:**

i. **m_socket(int family, int type, int protocol) :** For creating the socket. It returns the UDP sock id of the created socket

ii. **m_bind(int sockfd, const char *src_ip, int src_port, const char *dest_ip, int dest_port)** : Binds to the src ip and src port and destination address gets filled in the socket entry in the table. Returns 0 if successful else -1

iii. **m_sendto(int sockfd, char *buf, size_t len, int seqno, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)** : Takes buffer and its length and sends it to the destination address. Returns the length of the message on success and -1 on failure and also sets the corresponding error message

iv. **m_recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t * addrlen)**: Receives the buffer from the source address and returns len of message on success and -1 on failure.

v. **m_close(int mtp_socket_id)** : Closes the respective socket and frees the entry in the table.

msocket.c:

Implements 5 functions which are mentioned in header file

a. **msocket**() function: The function begins by acquiring shared memory segments using **shmget** for storing **MTPSocket** structures (**SM** table) and **SOCK_info** information. These shared memory segments are identified using unique keys generated by **ftok**

Next, the function iterates through an array of **MTPSocket** structures **SM** to find a free slot for the new socket. If no free slots are available then the respective error is set. If a free slot is available then it waits on a semaphore and asks **initmsocket's** main function to create a socket for this. After that function creates a socket it releases the semaphore and then this function fills that free slot with this information about socket using filled up info in **SOCK_info** structure.

Finally it returns the UDP sockid of the new socket created.

b. **mbind()** function: This function similarly first gets access to the shared memory **SM** and **SOCK_info.** It takes in the destination address and source address as parameters. It releases semaphore initially to indicate the initmsocket's main function to make a bind call from the source address available and waits on a semaphore to let that function complete the UDP bind process. It then updates the socket entry with the destination address given. Returns 0 on success else -1.

c. **m_sendto()** function: The function begins by acquiring shared memory segments Next, the function verifies whether the socket specified by **sockfd** is bound and exists within the shared memory array. It iterates through the array of `MTPSocket` structures to find the socket matching the provided socket file descriptor (**sockfd**) and destination address (dest_addr). If the socket is not bound or does not exist, the function sets **errno** to *ENOTBOUND* and returns -1 to indicate failure.

   If the socket is bound and exists, the function proceeds to search for an available slot in the socket's send buffer. It iterates through the send buffer to find an entry with an unused sequence number (initialized to -1). Upon finding such an entry, the function assigns the provided sequence number (**seqno**) to the packet header and copies the data (**buf**) into the send buffer.If an available slot is found and data is successfully copied, the function returns the length of the data sent. Otherwise, if no available slot is found in the send buffer, indicating that it is full, the function sets errno to *ENOBUFS* and returns -1 to indicate failure.

d. **m_recvfrom()** function: The function begins by acquiring shared memory segments. Also it checks the boundedness of the socket as described in earlier function

If the socket is bound and exists, the function proceeds to search for a received packet in the socket's receive buffer. It iterates through the receive buffer to find an entry with a sequence number matching the expected sequence number to deliver (**toDeliver**) inorder to deliver messages in order. Upon finding such an entry, the function copies the data from the receive buffer to the provided buffer (**buf**), updates the sequence number and flag of the received packet, and clears the data in the receive buffer entry.

If a valid packet is found and data is successfully copied, the function returns the length of the data received. Otherwise, if no valid packet is found in the receive buffer, indicating that there is no message available, the function sets errno to *ENOMESG* and returns -1 to indicate failure.

e. **mclose()** function : First, the function acquires the shared memory segments Next, the function iterates through the SM table structure to find the socket with the specified socket file descriptor (**sockfd**). If the socket is found and is not marked as free, indicating it is in use, the function closes the socket using UDP close and sets the **is_free** flag of the socket to 1 to mark it as available for reuse. It then returns 0 to indicate successful closure.

   If the specified socket file descriptor is not found or the socket is already free, the function sets **errno** to *EINVAL* (Invalid Argument) and returns -1 to indicate failure.

Initmsocket.c:

Though this is mentioned in assignment, as there are some additional things implemented in code for smooth run and some edge cases, so trying to mention those in detail

Receiver Thread:

The receiver thread does the following in order:

1. Firstly, The thread waits on select call to detect any reading on any of file descriptors present in **readfds.** After a timeout of **10 seconds** if no activity is observed it does the following.

   It iterates through each SM entry which is not free and has noSpace set to 1 i.e free size of receive buffer is 0 , it then computes the freesize again and if it is not 0 now it sends a duplicate ack with updated size and last inorder message. Also it sets the noSpace flag to 0 again

2. If some activity is observed on any socket it goes to the socket and makes a recv call on that socket and then **dropMessage** decides whether the message should be passed down further or not . If that is an ack (****ACK FORMAT MENTIONED AT LAST OF DOCUMENT****) then it is handled this way in order
   a. If the received ack is for some message present in the sender window, let us say for example the window is from 2 to 6 and the ack is received for 4.
   b. Then 2 things need to be updated in code
      i. Pointers to the sender window
      ii. Removing messages from sender buffer
   c. So we update the left pointer to point to the index after the one which points to the present message because there is no point in considering the before messages to be lost as the receiver receives them in order. So rather the sender understands that the acks sent for the previous messages are lost and so can directly push the left pointer to that
   d. Also if we encounter messages of sequence numbers we have passed in window, in sender buffer we remove them and the sender window size is updated from the size we got through the ack.

3. If its a data message we handle it in the following order:
   a. We check whether the message is present in the receive buffer i.e if its expected. If the message sequence number is present on the right side of the right pointer of the receiver window, then it is an out of order message and it is ignored.
   b. We need to check if the message is a duplicate message. There can be 2 types of duplicate message which have to be handled

i. Type1: The message is receiver window but it already got a place in receive buffer then it is unnecessary to store it again, so we drop it and just send the last inorder ack message

ii. Type2: The message sequence number is present on the left side of the left pointer of the receiver window indicating that it was already received and might also have been delivered to the application. So we check for this message if its present in the left side of the window and also not too far Ex: 1…2…5….15…1 Here if we receive 1 as a message we should not think of it as 1 left of 15 . It should be right of 15 because our protocol does not lead to such a long distance discrepancy due to small window sizes. So we check for this also then decide if it's a duplicate message of Type2.

c. Now we update the left pointer of the receiver window if the message received is the very next message expected. Then we store the message if it is neither duplicate nor out of order. Also for the message even if it is duplicate Ack is sent for the last inorder message.

Sender Thread:

It does the following in order

1. First it iterates through each socket and checks the last sent time stamp on that socket , if it is greater than T(5) seconds then we resend all of those messages.

2. It next checks if there are pending messages on any socket , we send them according to the receive buffer space available on the other side which we get through the last ack received

dropMessage function: It simply generates a float number and checks whether it is greater than or equal to the given argument. If it is greater then it means to not drop the packet so the function returns 0 else the function returns 1

Semaphores Used: Semaphores were used wherever we need to access SM table and update it with the received information.Also as mentioned in assignment 2 semaphores were used to implement **msocket mbind** functions to synchronize with **initmsocket** function calls

Garbage Collector thread: As mentioned in assignment It periodically (every 60 sec) checks if any of the processes got killed by sending an empty signal. If the process got killed it will not receive the signal so kill() function returns value less than 0. Then we clear the corresponding socket and reinitialise it to fresh values(whatever there at the beginning.).

ACK FORMAT CONSIDERED FOR THE ASSIGNMENT: **#seqno#recvBufferFreeSize

ALSO THROUGHOUT THE CODE SEQNO IS SENT AND RECEIVED IN BINARY FORM

Makefile : A makefile is present in the project directory which has following

1. **Library**: builds library
2. **init**: builds initmsocket.c
3. **user1**, **user2**, **user3….**: builds respective users
4. **all**: builds everything
5. **clean**: cleans the objective files, .so files and other extra files generated during run of the program

Table showing the probability taken and corresponding number of packets sent:

Probability	Number of transmissions per message
0.05	1.19047619
0.10	1.333333333
0.15	1.761904762
0.20	1.476190476
0.25	1.80952381
0.30	2.761904762
0.35	2.380952381
0.40	2.761904762
0.45	2.857142857
0.50	5
0.55	4.904761905
0.60	6.380952381
0.65	5.428571429
0.70	12.57142857
0.75	18
0.80	15.23809524
0.85	38.0952381

0.90	40.57142857
0.95	149.8571429