Networks Lab Assignment-5

Group Members:
Simma Pavan Kumar —> 21CS10060
Choda Y B V Anjaneya —> 21CS10020
==========

Introduction:

msocket.h is a custom socket library designed for reliable communication over UDP. It provides features such as error detection, message sequencing, and flow control. This report provides an overview of the data structures used in msocket.h and the purpose of each field, along with explanations of the functions implemented in the library.

Instructions for running our code:

The directory contains three subdirectories *init_process, library_files, user_processes*

Init_process contains initmsocket.c file that initializes the MTP socket by starting the R and S threads and the garbage collector process. A makefile is provided to create the executable to run initmsocket.c

library_files contains custom socket library msocket.h and msocket.c files and a makefile to generate libmsocket.a

user_processes contains user1.c, user2.c, user3.c and user4.c files and a makefile to create executables to run them.

Here is the detailed description to run on terminal:

- Create 4 terminals
- In 1st terminal goto library_files and run make
- In 2nd terminal goto init_process and run make
- In user_processes run make
- To check message transfer and receiving in run user3 and user4 executables in terminals 3 and 4
- To check files transfer run user1 and user2 executables

Table for different p values:

Probability(p value)	No of transmissions made to send	No of messages sent	Average no of transmissions per message
0.05	100	93	1.08
0.10	129	101	1.27
0.15	132	95	1.38
0.20	134	92	1.45
0.25	164	105	1.56
0.30	156	98	1.59
0.35	173	97	1.78
0.40	213	101	2.10
0.45	211	99	2.13
0.50	248	103	2.41

Data Structures in msocket.h:

1. **sbuf**:
    - Description: Represents a send buffer entry containing a message, its sending time, sequence number, and occupancy status.
    - Fields:
        - `msg`: Character array to store the message.
        - `sent_time`: `struct timeval` to store the time at which the message was sent.
        - `sent`: Integer flag to indicate whether the message has been sent.
        - `seq`: Sequence number of the message.
        - `occupied`: Integer flag to indicate whether the buffer entry is occupied.
2. **SEND_B**:

- Description: Represents the send buffer containing multiple `sbuf` entries and additional metadata related to sending messages.
- Fields:
  - `recv_buf_size`: Size of the receive buffer.
  - `last_ack_seq`: Sequence number of the last acknowledged message.
  - `tot_msgs`: Total number of messages sent.
  - `tot_sends`: Total number of message sends attempted.
  - `send_buffer`: Array of `sbuf` entries representing the send buffer.

3. **rbuf**:
   - Description: Represents a receive buffer entry containing a received message, its receiving time, sequence number, occupancy status, and reception status.
   - Fields:
     - `msg`: Character array to store the received message.
     - `recv_time`: `struct timeval` to store the time at which the message was received.
     - `occupied`: Integer flag to indicate whether the buffer entry is occupied.
     - `seq`: Sequence number of the message.
     - `recvd`: Integer flag to indicate whether the message has been received.
4. **RECV_B**:
   - Description: Represents the receive buffer containing multiple `rbuf` entries and additional metadata related to receiving messages.
   - Fields:
     - `last_inorder_seq`: Sequence number of the last received in-order message.
     - `flag_nospace`: Flag indicating if there's no space in the receive buffer.
     - `recv_buffer`: Array of `rbuf` entries representing the receive buffer.
     - `outoforder`: Array of `rbuf` entries representing out-of-order received messages.
5. **swnd**:
   - Description: Represents the send window containing the size of the send window, start index, and sequence numbers of messages sent but not yet acknowledged.
   - Fields:
     - `send_wnd_size`: Size of the send window.
     - `start_index`: Start index of the send window.
     - `sequence`: Array of sequence numbers of messages sent but not yet acknowledged.
6. **rwnd**:
   - Description: Represents the receive window containing the size of the receive window, start index, and sequence numbers of messages received but not yet acknowledged.
   - Fields:

        - `recv_wnd_size`: Size of the receive window.
        - `start_index`: Start index of the receive window.
        - `sequence`: Array of sequence numbers of messages received but not yet acknowledged.
7. **mtp_socket_info**:
    - Description: Represents the information associated with an MTP socket, including allocation status, process ID, UDP socket ID, peer socket information, send and receive buffers, and send and receive windows.
    - Fields:
        - `alloted`: Flag indicating whether the socket is allocated.
        - `pid`: Process ID associated with the socket.
        - `udp_sock_id`: UDP socket ID associated with the MTP socket.
        - `other`: Peer socket information (IP address and port).
        - `send_buf`: Send buffer (`SEND_B`) associated with the socket.
        - `recv_buf`: Receive buffer (`RECV_B`) associated with the socket.
        - `send_window`: Send window (`swnd`) associated with the socket.
        - `recv_window`: Receive window (`rwnd`) associated with the socket.
8. **sock_info**:
    - Description: Represents socket information including socket ID, IP address, port, and error status.
    - Fields:
        - `sock_id`: Socket ID associated with the socket.
        - `ip`: IP address associated with the socket.
        - `port`: Port number associated with the socket.
        - `err`: Error status associated with the socket.

Detailed Functions in msocket.c:

m_socket Function:

- Retrieve the shared memory segments containing the MTP socket information (`SM`) and socket information (`SOCK_INFO`) using `shmget` and `shmat`.
- Acquire a lock on the mutex semaphore to protect access to shared data structures using `semop` with appropriate semaphore operations.
- Ensure that a free entry is available in the `SM` array to create a new socket. If no free entry is available, set `errno` to `ENOBUFS` and return an error status.
- If any error occurs during the creation of the UDP socket using the `socket` call, set `errno` to the corresponding error code to indicate the failure.
- Perform semaphore operations on `sem1` and `sem2` for synchronization purposes. The specifics of these operations depend on the synchronization requirements of your application.

- After completing the socket creation process, perform final error checks to ensure the socket creation was successful.
- If any error is detected, such as an error number (`err`) being set in the `SOCK_INFO` structure, set `errno` accordingly and return an error status.
- If the socket creation is successful, return the socket descriptor. If any error condition is encountered, return an error status (typically -1).

m_bind Function:

- Retrieve the shared memory segment containing the MTP socket information (SM) using `shmget` and `shmat`.
- Acquire a lock on the mutex semaphore to protect access to shared data structures using `semop` with appropriate semaphore operations.
- Check if the socket descriptor (`sockfd`) is valid. If it is invalid (less than 0 or greater than or equal to `MAX_SOCKETS`), set `errno` to `EBADF` and return an error status.
- Ensure that the socket referenced by `sockfd` is allocated (in use). If it is not allocated, set `errno` to `EBADF` and return an error status.
- Update the socket information (`SOCK_INFO`) structure with the source IP address, source port, destination IP address, and destination port.
- Perform semaphore operations on `sem1` and `sem2` for synchronization purposes. The specifics of these operations depend on the synchronization requirements of your application.
- Check for any errors that might occur during the binding of the UDP socket. If an error occurs, set `errno` to the corresponding error code to indicate the failure.
- After completing the binding process, perform final error checks to ensure the binding was successful.
- If any error is detected, such as an error number (`err`) being set in the `SOCK_INFO` structure, set `errno` accordingly and return an error status.
- If the binding is successful, return the return value stored in `SOCK_INFO`, which likely indicates success.
- If the socket is already in use or if any other error occurs, return an error status (typically -1).

m_sendto Function:(non-blocking)

- Retrieve the shared memory segment containing the MTP socket information (`SM`) using `shmget` and `shmat`.
- Acquire a lock on the mutex semaphore to protect access to shared data structures using `semop` with appropriate semaphore operations.

- Check if the socket descriptor (`sockfd`) is valid. If it is invalid (less than 0 or greater than or equal to `MAX_SOCKETS`), set `errno` to `EBADF` and return an error status.
- Ensure that the socket referenced by `sockfd` is allocated (in use). If it is not allocated, set `errno` to `EBADF` and return an error status.
- Verify if the socket is bound to a destination. If it is not bound, set `errno` to `ENOTCONN` and return an error status.
- Check if there is space available in the send buffer. If the send buffer is full, set `errno` to `ENOBUFS` and return an error status.
- If the message sending is successful, return the number of bytes sent. If any error condition is encountered, return an error status (typically -1).
- Detach the shared memory segment containing `SM` using `shmdt`.
- Release the lock on the mutex semaphore using appropriate semaphore operations.

m_recvfrom Function:(non-blocking)

- Retrieve the shared memory segment containing the MTP socket information (`SM`) using `shmget` and `shmat`.
- Acquire a lock on the mutex semaphore to protect access to shared data structures using `semop` with appropriate semaphore operations.
- Check if the socket descriptor (`sockfd`) is valid. If it is invalid (less than 0 or greater than or equal to `MAX_SOCKETS`), set `errno` to `EBADF` and return an error status.
- Ensure that the socket referenced by `sockfd` is allocated (in use). If it is not allocated, set `errno` to `EBADF` and return an error status.
- Verify if the socket is bound to a destination. If it is not bound, set `errno` to `ENOTCONN` and return an error status.
- Check if there are any messages available in the receive buffer. If the receive buffer is empty, set `errno` to `ENOMSG` and return an error status.
- If a message is successfully received, return the number of bytes received. If any error condition is encountered, return an error status (typically -1).
- Detach the shared memory segment containing `SM` using `shmdt`.
- Release the lock on the mutex semaphore using appropriate semaphore operations.

m_close Function:

- Retrieve the shared memory segment containing the MTP socket information (`SM`) using `shmget` and `shmat`.

- Acquire a lock on the mutex semaphore to protect access to shared data structures using `semop` with appropriate semaphore operations.
- Error Handling for Invalid Socket Descriptor.

dropMessage(float p) function:

- The `dropMessage()` function takes a probability p as input, representing the likelihood of dropping a message.
- Inside the function, a random number between 0 and 1 is generated using the `rand()` function.
- If the generated random number is less than the given probability p, the function returns 1, indicating that the message should be dropped.
- Otherwise, if the generated random number is greater than or equal to p, the function returns 0, indicating that the message should not be dropped.

Initmsocket.c:

R Thread:

The R thread is responsible for receiving messages from UDP sockets and handling them according to the specified behavior.

1. **Initialization**:
   - This thread waits for incoming messages using the `select()` function to monitor multiple file descriptors (UDP sockets).
   - It sets a timeout for the `select()` call to detect if there are no incoming messages.
2. **Message Reception**:
   - Upon receiving a message, it checks if it's a data message or an ACK message.
   - If it's a data message, it stores it in the receiver-side message buffer for the corresponding MTP socket.
   - It sends an ACK message to the sender, acknowledging the received data.
   - If the receive buffer is full, it sets a flag `nospace`.
3. **Timeout Handling**:
   - On a timeout, it checks if the `nospace` flag was set but space is now available in the receive buffer.
   - In this case, it sends a duplicate ACK message with the last acknowledged sequence number and the updated rwnd size.

S Thread:

The S thread periodically sends messages and handles message timeouts for active MTP sockets.

1. **Initialization**:
    - This thread sleeps for a certain time period (less than T/2) and wakes up periodically.
2. **Message Timeout Handling**:
    - Upon waking up, it checks if the message timeout period (T) has elapsed for messages sent over any active MTP socket.
    - If timeout occurs, it retransmits all messages within the current send window for that MTP socket.
3. **Sending Messages**:
    - It checks if there are pending messages in the sender-side message buffer that can be sent.
    - The sender window is updated according to the sender-size message buffer and the ACKs received.
    - If so, it sends those messages through the UDP sockets using the `sendto()` call for the corresponding UDP socket and updates the send timestamp.

G Thread:

The G thread cleans up resources associated with terminated processes and closed UDP sockets.

- This thread periodically checks for terminated processes and closed UDP sockets.
- If a process associated with an MTP socket is terminated, it closes the corresponding UDP socket and deallocates resources related to that socket.

Shared Memory and Semaphore Handling:

- Shared memory segments are created and initialized to store socket and MTP socket information.
- Semaphores are created for synchronization between threads and processes.
- Mutexes are used to protect shared data structures from concurrent access.

Conclusion:

MSocket provides a reliable communication mechanism over UDP, implementing features like message sequencing and flow control. It uses shared memory and semaphores for inter-process communication and synchronization. The data structures and functions work together to ensure reliable data transmission in a networked environment.