Networks Lab Assignment-5

Group Members:

Meduri Harshith Chowdary 21CS10042

Maddi Nihith 21CS10040

# 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.

# Data Structures

1. **int4**: Represents an unsigned 4-bit integer. Used for sequence number arithmetic.
   * **value**: Internal integer value (0-15).
2. **header**: Represents the header of a message.
   * **seq\_number**: contains the sequence number of the message.
3. **message**: Represents a complete message.
   * **header**: Message header.
   * **data**: Message data (up to 1024 bytes).
4. **send\_window**: Represents the sending window for a socket.
   * **left\_idx**: Index of the left edge of the window.
   * **right\_idx**: Index of the right edge of the window.
   * **new\_entry**: Index of the new entry in the window.
   * **last\_seq\_no**: Last sequence number of the message sent.
   * **last\_ack\_seqno**: Last acknowledged sequence number.
   * **last\_sent**: Last sent sequence number.
   * **last\_active\_time**: Array of the last active time of each message in the window.
   * **last\_ack\_rwnd\_size**: Last acknowledged receiver window size.
5. **receive\_window**: Represents the receiving window for a socket.
   * **full**: Indicates if the buffer is full.
   * **last\_inorder\_received**: contains the sequence number of Last in-order received message received.
   * **last\_user\_taken**: contains the sequence number of the Last user-taken message.
   * **window**: Array of sequence numbers of messages in the window.
6. **mtpSocket**: Represents an MTP (MSocket Transport Protocol) socket.
   * **available**: Indicates if the socket is available (0) or in use (1).
   * **pid**: Process ID of the process that created the socket.
   * **udp\_sockid**: It contains the socket ID of actual UDP socket which is mapped to the MTP socket.
   * **dest\_ip**: It contains Destination IP address.
   * **dest\_port**: It contains the Destination port number.
   * **send\_buffer**: Sender-side message buffer.
   * **receive\_buffer**: Receiver-side message buffer.
   * **swnd**: Sending window.
   * **rwnd**: Receiving window.
7. **SOCK\_INFO:**

# Functions

1. **int4 init\_int4(int value)**: Creates an int4 data structure initialised to given value.
2. **int4 add\_int4(int4 a, int4 b)**: Adds two int4 values following mod 16 arithmetic.
3. **int4 sub\_int4(int4 a, int4 b)**: Subtracts two int4 values following mod 16 arithmetic.
4. **mtpSocket \*get\_shared\_MTP\_Table()**: Gets the shared MTP table from shared memory.
5. **SOCK\_INFO \*get\_SOCK\_INFO()**: Gets the shared SOCK\_INFO structure from shared memory.
6. **void get\_sem1(int \*sem1)**: Gets the shared semaphore 1.
7. **void get\_sem2(int \*sem2)**: Gets the shared semaphore 2.
8. **void get\_mutex(int \*mutex)**: Gets the shared mutex.
9. **void get\_mutex\_swnd(int \*mutex\_swnd)**: Gets the shared mutex for the sending window.
10. **void get\_mutex\_sendbuf(int \*mutex\_sendbuf)**: Gets the shared mutex for the sending buffer.
11. **void get\_mutex\_recvbuf(int \*mutex\_recvbuf)**: Gets the shared mutex for the receiving buffer.
12. **int m\_socket(int domain, int type, int protocol)**: Creates an MTP socket.
    * Checks if the socket type is correct.
    * Initializes semaphores and mutex.
    * Finds an available slot in the MTP table.
    * Sets up the socket and returns its ID.
13. **int m\_close(int socket\_id)**: Closes an MTP socket.

* It retrieves the shared data structures MTP\_Table and sock\_info using get\_shared\_MTP\_Table() and get\_SOCK\_INFO().It acquires a lock on mutex to protect access to shared data structures.
* It checks if the socket with the given socket\_id is available (in use). If it's not available, it returns an error.
* It marks the socket as available, indicating that it is no longer in use.It resets the UDP socket ID to -1.It clears the send buffer of the socket by setting all sequence numbers to -1 and clearing the data.It resets the send window of the socket to its initial state.It clears the receive buffer of the socket by setting all sequence numbers to -1 and clearing the data.It resets the receive window of the socket to its initial state.
* It performs semaphore operations on sem1 and sem2 to synchronize with another thread or process. The purpose of these operations depends on the specific implementation.
* It checks the sock\_info structure for any error number (error\_no). If an error is detected, it sets errno to the error number, resets the socket availability flag, and returns an error.
* If the socket is successfully closed, it returns the return value stored in sock\_info, which indicates success.
* It detaches the shared memory segments containing MTP\_Table and sock\_info.It releases the locks on mutex and other mutexes used for protecting send and receive buffers.
* If the socket is already closed or if any other error occurs, it returns an error.

1. **int m\_bind(int socket\_id, char \*src\_ip, unsigned short int src\_port, char \*dest\_ip, unsigned short int dest\_port)**:

* It retrieves the shared data structures MTP\_Table and sock\_info using get\_shared\_MTP\_Table() and get\_SOCK\_INFO().
* It acquires a lock on mutex to protect access to shared data structures.
* It checks if the socket with the given socket\_id is available (not in use). If it's not available, it returns an error.
* It updates the sock\_info structure with the source IP address and port, and the destination IP address and port.
* It performs semaphore operations on sem1 and sem2 to synchronize with another thread or process. The purpose of these operations depends on the specific implementation.
* It checks the sock\_info structure for any error number (error\_no). If an error is detected, it sets errno to the error number, resets the destination IP address and port in MTP\_Table, and returns an error.
* If the binding is successful, it returns the return value stored in sock\_info, which likely indicates success. If the socket is already in use or if any other error occurs, it returns an error (ERROR).
* It detaches the shared memory segments containing MTP\_Table and sock\_info.It releases the lock on mutex.

1. **int m\_sendto(int sockfd, const void \*buf, size\_t len, int flags, const struct sockaddr \*dest\_addr, socklen\_t addrlen)**:

* It retrieves the shared data structure MTP\_Table using get\_shared\_MTP\_Table().It obtains the mutex identifiers for mutex\_swnd and mutex\_sendbuf using custom functions get\_mutex\_swnd and get\_mutex\_sendbuf.It acquires locks on both mutex\_swnd and mutex\_sendbuf to protect access to shared data structures.
* It copies the data (buf) to the send buffer (send\_buffer) of the specified socket at the new entry index (swnd.new\_entry).
* It assigns a sequence number to the data packet based on the last sequence number used by the socket (swnd.last\_seq\_no) and increments the last sequence number for the socket.
* It updates the new entry index for the socket's send window (swnd.new\_entry) to point to the next available slot in the send buffer.
* It releases the locks on mutex\_sendbuf and mutex\_swnd.It detaches the shared memory segment containing MTP\_Table.
* It returns 0 to indicate success.

1. **int m\_recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen)**: Receives a message on an MTP socket.

* It retrieves the shared data structure MTP\_Table using get\_shared\_MTP\_Table().It obtains the mutex identifier for mutex\_recvbuf using the custom function get\_mutex\_recvbuf.It acquires a lock on mutex\_recvbuf to protect access to the receive buffer of the specified socket.
* It calculates the minimum sequence number (min\_seqno) that has not been taken by the user yet. This is based on the last sequence number taken by the user (rwnd.last\_user\_taken) and ensures that the function retrieves the next expected data packet.
* It iterates through the receive buffer of the specified socket to find the data packet with the minimum sequence number (min\_seqno).
* If a matching packet is found, it copies the data from the packet to the provided buffer (buf) and marks the packet as read by setting its sequence number to -1.
* It updates the last sequence number taken by the user (rwnd.last\_user\_taken) to min\_seqno.
* If no matching packet is found in the receive buffer, it sets the error number ENOMSG and returns an error.If a packet is successfully retrieved, it returns the size of the message (MESSAGE\_SIZE) to indicate success.If no message is available, it returns an error (ERROR).

1. **printTable():**

* This function is used to print the contents of the MTP (Multithreaded Transport Protocol) table. It iterates through the table and prints information such as PID, Socket ID, UDP Socket ID, Destination IP, and Destination Port for each entry that is not marked as available.

1. **max\_window(int x, int y, int size):**

* This function calculates the maximum sequence number between x and y, considering them as sequence numbers in a circular buffer of size "size".

1. **sigint\_handler(int signum):**

* This function is a signal handler for the SIGINT signal (interrupt signal, typically generated by pressing Ctrl+C). It cleans up shared memory and semaphores used by the program before exiting.

1. **logs():**

* This function is used to log information about the sockets in the MTP table. It prints the PID, Socket ID, UDP Socket ID, SEND BUFFER SEQ NUMS, RECV WINDOW, and RECV BUFFER for each socket that is not marked as available.

1. **thread\_R\_func(void \*arg)**:

The thread\_R\_func function is designed to handle the receiving side of a multi-threaded network application, likely implementing a protocol similar to TCP over UDP. Here's a detailed explanation of how this function works:

* The function begins by obtaining a shared data structure MTP\_Table that stores information about each socket, such as its receive window, send window, and buffers. It then acquires a mutex (mutex\_recvbuf) to protect the access to this shared data structure. For each socket in MTP\_Table (represented by the array index i), the function initializes the receive window (rwnd), setting its state to not full and initializing its sequence numbers. It also clears the receive buffer for each socket.
* The function enters a continuous loop to handle incoming data. It uses the select call to know which sockets in MTP\_Table have data to read.
* If the data is an ACK msg then It extracts the ACK sequence number (curr\_ack\_seqno) and the current receiver window size (curr\_rwnd\_size) from the ACK message.
* It checks if the ACK is a duplicate by comparing the ACK sequence number with the last acknowledged sequence number in the send window (curr\_swnd.last\_ack\_seqno). If it's a duplicate, it ignores the ACK and continues to the next iteration of the loop.
* If it's a new ACK, it updates the send buffer, marking all packets up to the ACKed sequence number as acknowledged. It then updates the send window (swnd) to reflect the new state.
* If the received message is not an ACK, it processes it as a data packet. It extracts the sequence number (seq\_no) from the packet.
* It then checks if the sequence number fits within the receive window (rwnd). If it does, it stores the data packet in the receive buffer (receive\_buffer) at the appropriate index.
* After storing the packet, it updates the receive window based on the sequence numbers of the packets in the receive buffer. It finds the last in-order received packet and updates the receive window accordingly.
* If the receive window has space, it sends an acknowledgment back to the sender, indicating the updated receive window size and the last in-order received sequence number.

1. **thread\_S\_func(void \*arg):**

The thread\_S\_func function is responsible for managing the sending side of a network application, likely implementing a protocol similar to TCP over UDP. Let's break down its functionality:

* It retrieves the shared data structures MTP\_Table and SOCK\_INFO using get\_shared\_MTP\_Table() and get\_SOCK\_INFO() respectively.
* It acquires two mutexes, mutex\_swnd and mutex\_sendbuf, to protect access to shared data structures.
* For each socket in MTP\_Table, it initializes the send buffer (send\_buffer) and send window (swnd).It sets all sequence numbers in the send buffer to -1 and clears the data.It initializes the send window parameters such as the left and right indexes, indicating the range of packets that can be sent, and initializes other parameters such as new\_entry, last\_seq\_no, last\_sent, and last\_ack\_seqno.It also initializes the last\_active\_time array for the sender window.
* The function enters a continuous loop where it periodically checks the status of each socket and sends data packets if necessary. It uses sleep(T/2) to wait for a period of time T/2 before checking the sockets again.
* For each socket in MTP\_Table, if the socket is not available (i.e., actively sending data), it checks the send window (swnd) to determine which packets can be sent.
* It checks for a timeout condition by comparing the current time with the last active time of the packets in the send window. If a timeout occurs, it resends the packets that have not been acknowledged.
* If there is no timeout, it sends new packets starting from the last sent packet + 1 (modulo the buffer size) up to the right index of the send window. It marks these packets as sent and updates the last active time for each sent packet.
* For each packet to be sent, it constructs a data packet (buffer) with the sequence number and data from the send buffer.It then sends the packet using sendto to the destination address (dest\_addr) obtained from the socket information in MTP\_Table.

1. **thread\_G\_func(void \*arg):**

The thread\_G\_func function is designed to act as a garbage collector for a network application, likely cleaning up resources associated with sockets and processes that have been closed or terminated. Here's a detailed explanation of how it works:

* It retrieves the shared data structure MTP\_Table using get\_shared\_MTP\_Table().It enters a continuous loop to periodically check the status of sockets and processes.
* The function enters a continuous loop where it sleeps for 60 seconds before checking the status of sockets and processes.It prints a header for the garbage collection process and then prints the current state of the MTP\_Table using the printTable function.
* For each socket in MTP\_Table, if the socket is not available (i.e., actively in use), it checks the status of the associated process using kill(pid, 0).If the process exists (returns 0), it indicates that the process is still running.If the process has exited (returns non-zero), it marks the socket as available, resets the process ID (pid), and closes the UDP socket associated with the socket index using m\_close.
* Before exiting, the function detaches the shared memory segment containing MTP\_Table using shmdt.

1. **dropMessage(float p):**

* This function simulates message dropping by generating a random number between 0 and 1 and comparing it to a given probability p. If the random number is less than p, the function returns 1 (indicating that the message should be dropped), otherwise it returns 0.

|  |  |  |  |
| --- | --- | --- | --- |
| Probability | Total number of messages sent | Total number of transmissions | Average number of transmissions per message |
| 0.05 | 99 | 108 | 1.09 |
| 0.10 | 99 | 112 | 1.13 |
| 0.15 | 99 | 113 | 1.14 |
| 0.20 | 99 | 133 | 1.34 |
| 0.25 | 99 | 142 | 1.43 |
| 0.30 | 99 | 135 | 1.36 |
| 0.35 | 99 | 143 | 1.44 |
| 0.40 | 99 | 153 | 1.55 |
| 0.45 | 99 | 159 | 1.61 |
| 0.50 | 99 | 188 | 1.89 |

# 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.