Networks Lab Assignment 5

Chinmay Bhusari (21CS30015)
Ronit Nanwani (21CS30043)

Average Number of Transmissions per message for different probability values

Probability of drop (p)	Total transmissions	Total messages	Ratio
0.05	209	99	2.1111
0.10	225	99	2.2727
0.15	246	99	2.4848
0.20	266	99	2.6868
0.25	279	99	2.8181
0.30	293	99	2.9595
0.35	337	99	3.4040
0.40	426	99	4.3030
0.45	457	99	4.6161
0.50	496	99	5.0101

Structs in msocket.h

Structure to represent message header

```
typedef struct {
   int sequence_number;      //stores the sequence number of the message
   int ty;          //1 for message and 2 for ack
   time_t lastsenttime;      //stores lastsenttime for messages
} message_header;
```

lastsenttime is necessary in cases of ACK loses. Whenever this value exceeds a certain limit (T), we will retransmit the messages.

Structure to represent a message

```
typedef struct {
   int ismsg;      //0 if not msg 1 if msg
   message_header msg_header;  //stores the message header as described
   char data[1024]; // Assuming message size is 1KB
} Message;
```

Structure to represent a message in receiver buffer

```
typedef struct {
   int ismsg;       // 0 if not msg 1 if msg
   char data[1024]; // Assuming message size is 1KB
} MessageRecv;
```

Structure to represent the window

```
typedef struct {
   int size;     //represents size of window
   int ptr1;     //represents starting point of window
```

```
   int ptr2;      //represents ending point of window
}Window;
```

Structure to represent a MTP Socket

```
typedef struct {
   int is_free;         //to determine whether this socket entry is free
   int is_close; //to determine whether this socket was closed (m_close)
   int process_id;//stores pid of process which requested creation of
socket
   int udp_socket_id;  //stores the file descriptor id
   char ip_address[INET_ADDRSTRLEN]; // Assuming IPv4 address (e.g.,
"xxx.xxx.xxx.xxx\0")   //stores ip address of destination
   unsigned short port; //stores port of destination
   // Other necessary fields
   int curr;       //next sequence number to assign to a message coming
inside the send buffer;
   int wrs;        //current index to write to in the sender buffer;
   int str;         //which index to start reading from in the receiving
buffer
   int wrr;        //current index to write to in the receiver buffer;
   Message send_buffer[MAX_BUFFER_SIZE_SENDER];//sender buffer
   MessageRecv receive_buffer[MAX_BUFFER_SIZE_RECEIVER];//receiver buffer
   Window swnd;//sender window
   Window rwnd;//receiver window
} MTPSocket;
```

Structure to represent shared memory consisting of 25 sockets

```
// Structure to represent shared memory containing information about N MTP
sockets
typedef struct {
   MTPSocket sockets[MAX_SOCKETS];
} SharedMemory;
```

Structure to store info about socket created in initmsocket

```
typedef struct{
   int sockid;//sockid of the socket
```

```
   char ip[INET_ADDRSTRLEN];//ip address the socket is to be bound to
   short port;//port the socket needs to be bound to
   int err_no;//stores the err_no in case of error
} SockInfo;
```

Functions in msocket.c

```
int dropMessage(float prob)
```

Function to simulate the probability of dropping a message

```
int m_socket(int domain, int type, int protocol)
```

Function to create an MTP Socket for the application program.
This function eventually signals the initmsocket to create a UDP socket for it and then sends the details back to it using semaphore synchronisation. These details are finally stored in shared memory for MTP Sockets.

```
int m_bind(int sockfd, char* srcip,short srcport,char* destip,short
destport)
```

Function to bind the socket to an IP and port as specified the user. This is done by semaphore synchronisation between initmsocket and msocket.

```
ssize_t m_sendto(int sockfd, const void* buf, size_t len, int flags,
struct sockaddr* dest_addr, socklen_t dest_len)
```

Function to put the message into the sender buffer if empty. If the destination address and port does not match it returns with ENOTBOUND error. If the sender buffer is full then it returns with ENOBUFS error. Otherwise it returns the length of the message written to the sender buffer. This call is non blocking.

```
ssize_t m_recvfrom(int sockfd, void *buf, size_t len,int flags,struct sockaddr* sender_addr,socklen_t* sender_addr_len)
```

Function to receive from the receiver buffer. It stores the required (len) number of bytes in the buf which is delivered to the application program. If there is no message in the buffer then it returns with ENOMSG error. This function call is non-blocking.

```
int m_close(int sockfd)
```

This function sets the is_close value to 1 which is then useful in the garbage collector to further close the udp socket.

Functions in initmsocket.c

```
void remove_shared_memory()
```

Function to remove the shared memory created by initmsocket upon receiving SIGINT signal

```
void remove_semaphore()
```

Function to remove all the semaphores created by initmsocket upon receiving SIGINT signal

```
void signal_handler(int signal)
```
Signal handler for SIGINT

```
void* thread_R(void* arg)
```
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 (we have used select() to keep on checking
whether there is any incoming message on any of the UDP sockets, on timeout check we have checked whether
a new MTP socket has been created and have included 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!). For this we have maintained the lastacksendtime for a socket and if this exceeds a certain limit (2*T = 10) then we send the ACK again. This prevents a deadlock in the code.

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.

```
void* thread_S(void* arg)
```

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.

```
void* garbage_collector(void* arg)
```

Function to release resources whenever an application process terminates or a m_close() call has been made.