Computer Networks Assignment 4 Documentation

March 2024

1 Group Details

1. Apoorv Kumar (21CS10008),
2. Swarnabh Mandal (21CS10068)

2 Data structures used

2.1 msocket.h

1. **MessageHeader**

   ```
   typedef struct {
       char msg_type;
       int seq_no;
   } MessageHeader;
   ```

   The provided code snippet defines a structure named `MessageHeader` to represent the header of a message. It consists of two fields:

   - `msg_type`: This field indicates the type of the message and is of type `char`. It can take one of three values: 'D' for data, 'A' for acknowledgment, or 'P' for Probe.
   - `seq_no`: This field holds the sequence number of the message and is of type `int`.

   This structure encapsulates the header of a message. It allows us to distinguishing between different types of messages and gives the sequence number of the message.

2. **Message**

```
typedef struct {
    MessageHeader header;
    char msg[MAX_MSG_SIZE];
} Message;
```

The provided code snippet defines a structure named `Message` representing a complete message. It consists of two fields:

- `header`: This field is of type `MessageHeader` and represents the header of the message.
- `msg`: This field is an array of characters representing the body of the message, with a maximum size defined as `MAX_MSG_SIZE`.

This structure encapsulates both the header and the body of a message, allowing for the transmission and processing of complete messages.

3. **Message Info**

```
typedef struct {
    int ack_no;
    time_t time;
    int sent;
    Message message;
} send_msg;
```

The provided code snippet defines a structure named `send_msg`, which encapsulates information related to sending a message. It consists of the following fields:

- `ack_no`: This integer field represents the acknowledgment number. If the acknowledgment for the message is received, it's set to -1; otherwise, it's equal to the sequence number of the message.
- `time`: This field of type `time_t` holds the time of sending the message.
- `sent`: An integer flag indicating whether the message has been sent once.
- `message`: An instance of the `Message` structure representing the actual message to be sent.

This structure provides a comprehensive representation for managing and tracking messages being sent, including details such as acknowledgment status, sending time, and the message content itself.

4. **Recieved Message**

```
typedef struct {
    int ack_no;
    char message[MAX_MSG_SIZE];
} recv_msg;
```

The provided code snippet defines a structure named `recv_msg`, which represents the format of a received message. It comprises the following fields:

- `ack_no`: An integer field representing the acknowledgment number associated with the received message.
- `message`: An array of characters (`char`) representing the body of the received message. The maximum size of the message body is defined by `MAX_MSG_SIZE`.

This structure provides a compact and standardized way to handle received messages, encapsulating both the acknowledgment information and the message body itself.

5. **Sender Window**

```
typedef struct {
    int window_size;
    int window_start_index;
    int last_seq_no;
    send_msg send_buff[SENDER_MSG_BUFFER];
} swnd;
```

The provided code snippet defines a structure named `swnd`, representing the sender's window. It contains the following fields:

- `window_size`: An integer indicating the size of the window.
- `window_start_index`: An integer representing the index of the starting point of the window.
- `last_seq_no`: An integer indicating the last sequence number used in the window.
- `send_buff`: An array of `send_msg` structures, serving as a buffer for messages to be sent. The size of this array is defined by `SENDER_MSG_BUFFER`.

This structure provides a mechanism for organizing and managing the sender's messages within a window, facilitating efficient communication protocols.

6. **Receiver Window**

```
typedef struct {
    int window_size;
    int index_to_read;
    int next_seq_no;
    int index_to_write;
    int nospace;
    recv_msg recv_buff[RECEIVER_MSG_BUFFER];
} rwnd;
```

The provided code snippet defines a structure named `rwnd`, which represents the receiver's window. It consists of the following fields:

- `window_size`: An integer indicating the size of the window.
- `index_to_read`: An integer representing the index to read the next message from.
- `next_seq_no`: An integer indicating the next expected sequence number.
- `index_to_write`: An integer representing the index to write the next received message to.
- `nospace`: A flag indicating if there's no space in the buffer.
- `recv_buff`: An array of `recv_msg` structures, serving as a buffer for received messages. The size of this array is defined by `RECEIVER_MSG_BUFFER`.

This structure facilitates the organization and management of received messages within a window at the receiver's end.

7. **Socket Entry**

```
typedef struct {
    int socket_alloted;
    pid_t process_id;
    int udp_socket_id;
    struct sockaddr_in destination_addr;
    swnd send_window;
    rwnd recv_window;
} MTPSocketEntry;
```

The provided code snippet defines a structure named `MTPSocketEntry` representing a socket entry in shared memory SM table. It comprises the following fields:

- `socket_alloted`: An integer flag indicating whether the slot has been allotted for the socket.

- `process_id`: A process ID (`pid_t`) representing the process associated with the socket entry.
- `udp_socket_id`: An integer representing the associated UDP socket ID.
- `destination_addr`: A structure of type `sockaddr_in` representing the destination address.
- `send_window`: An instance of the sender's window (`swnd`) structure.
- `recv_window`: An instance of the receiver's window (`rwnd`) structure.

This structure serves as a comprehensive container for managing MTP sockets, incorporating information about process IDs, socket IDs, destination addresses, and sender and receiver windows.

8. **Socket Info**

```
typedef struct {
    int sock_id;                      // Socket ID
    unsigned long IP;                 // IP address
    unsigned short port;              // Port number
    int errno_val;                    // Error number value
} SOCK_INFO;
```

The provided code snippet defines a structure named `SOCK_INFO`, which is designed to hold socket creation and binding information. It consists of the following fields:

- `sock_id`: An integer representing the socket ID.
- `IP`: An unsigned long integer representing the IP address.
- `port`: An unsigned short integer representing the port number.
- `errno_val`: An integer representing the error number value associated with socket operations.

This structure provides a convenient way to encapsulate essential socket-related information, facilitating socket management and error handling within a program.

These are the various data structures that have been defined in mxocket.h

2.2 initsocket.c

1. **Variables accessed**

- `shmid_SM`: Shared memory ID for the `MTPSocketEntry` shared memory segment.

```
int shmid_SM = -1;
```

- `shmid_sock_info`: Shared memory ID for the `SOCK_INFO` shared memory segment.

  ```
  int shmid_sock_info = -1;
  ```

- `SM`: Pointer to the shared memory segment for `MTPSocketEntry`.

  ```
  MTPSocketEntry *SM = NULL;
  ```

- `sock_info`: Pointer to the shared memory segment for `SOCK_INFO`.

  ```
  SOCK_INFO *sock_info = NULL;
  ```

- `Sem1`, `Sem2`: Semaphores for synchronization, used for `m_socket` and `m_bind` calls.

  ```
  sem_t *Sem1 = NULL;
  sem_t *Sem2 = NULL;
  ```

- `SM_mutex`: Semaphore for mutual exclusion, used for accessing the shared resource `SM`.

  ```
  sem_t *SM_mutex = NULL;
  ```

- `num_messages`: Number of messages sent.

  ```
  int num_messages = 0;
  ```

- `num_transmissions`: Number of transmissions made.

  ```
  int num_transmissions = 0;
  ```

2. **Data structure defined**

   (a) **Persistence Timer**

   ```
   typedef struct {
       int flag;
       time_t last_time;
       int ack_seq_no;
   } Persistence_Timer_;
   ```

   The provided code snippet defines a structure named `Persistence_Timer_`, which represents the persistence timer. It comprises the following fields:

   - `flag`: An integer field used to indicate the number of timeouts and to set the exponential backoff waiting time (within an upper limit).
   - `last_time`: A field of type `time_t` representing the latest time when an ACK with window size 0 was received.

- `ack_seq_no`: An integer representing the sequence number of the ACK message with a window size of 0.

  This structure facilitates the management of the persistence timer, allowing for handling timeouts and adjusting waiting times based on received acknowledgments.

These are the various important variables and data structures declared and used in initsocket.c

2.3 msocket.c

msocket.c doesn't have any special data structure/variable defined in it and uses the shared memory, data stuctures defined in msocket.h and initsocket.c mainly

3 Functions

3.1 msocket.c

1. **cleanup**: Function to free resources before exiting the program

   ```
   void cleanup(MTPSocketEntry *SM, SOCK_INFO *sock_info, sem_t *Sem1,
               sem_t *Sem2, sem_t *SM_mutex)
   ```

2. **dropmessage**: Function to drop message with probability p

   ```
       int dropMessage(float p)
   ```

3. **m_socket**: Function to open an MTP socket. This function searches for an empty slot in the SM table and if empty slot is present, it clears all buffers and sets the fields appropriately to the provided information and returns the socket id(a number between 0 to MAX_MTP_SOCKETS). In case of any error case like no available slots or invalid domain/protocol,etc it sets the errno appropriately and return -1. This function also ensures mutual exclusion while accessing shared memory and release of resources, in case of error in accessing the shared variables, it releases resources and returns -1.

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

4. **m_bind**: Function to bind an MTP socket with given sockfd to the given source ip, source port and destination ip, destination port. The function ensures mutual exclusion and also frees any resources after usage and in case of success returns 0 else returns -1;

   ```
   int m_bind(int sockfd, unsigned long src_ip, unsigned short src_port,
               unsigned long dest_ip, unsigned short dest_port)
   ```

5. **m_sendto**: Function to send a message buf of length len through the MTP socet with the provided sockfd to the given destination IP, destination port.The function puts the message in the sender buffer and updates the sender window information and returns length of message len. In case of errors like, the given destination IP and port are different from that with which the socket is bound, or if there are no free slots in sender buffer, the function just drops the message, sets errno appropriately and returns -1.

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

6. **m_recvfrom**: Function to recieve a message buf of length len through the MTP socet with the provided sockfd from the given source IP, source port.The function puts the message in the receiver buffer and updates the receiver window information and returns length of message len. In case of cases like, no available message, invalid sockfd or error during access of shared resources the function sets errno appropriately and returns -1.

```
ssize_t m_recvfrom(int sockfd, const void *buf, size_t len, int flags,
          const struct sockaddr *src_addr, socklen_t addrlen)
```

7. **m_close**: Function to close socket with given sockfd. The function just changes the allocated flag of the given sockfd entry in SM table to 0. In case of cases like, invalid sockfd or if the given sockfd doesn't belong to calling process or error during access of shared resources the function sets errno appropriately and returns -1.

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

3.2 initsocket.c

1. **cleanup_on_exit**: Function to print average number of transmissions required per message at the message drop probability p and finally to destroy all shared resources before exiting.

```
void cleanup_on_exit()
```

2. **signal_handler**: Signal handler for SIGINT, SIGQUIT

```
void signal_handler(int  signum)
```

3. **R Thread**: 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 using select() call , on timeout check whether a new MTP socket has been created and includes 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 table 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, additionally this flag may get lost and hence it sends an ACK, with the current window size, periodically to all the senders. If the received message is an ACK 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_R()
```

4. **S Thread**: The thread S behaves as follows: it periodically wakes up after sleeping for some time (less than T/2), where T represents the message timeout period. Upon awakening, it checks if the timeout period has elapsed for any messages sent over active MTP sockets by comparing the current time with the time when the messages within the window were last sent. If a timeout is detected, it initiates the retransmission of all messages within the current sender's window (swnd) for the corresponding MTP socket. Subsequently, it examines each MTP socket's sender's window to determine if there are pending messages in the sender-side buffer that can be sent. If pending messages exist, it transmits them via the UDP sendto() call for the respective UDP socket and updates the send timestamp for the transmitted messages.

   ```
   void *thread_S()
   ```

5. **G Thread**: Periodically checks all entries of the SM table and determines if the allocated slot should be freed by checking whether the process with the pid (in the SM table entry) is alive.

   ```
   void *thread_G()
   ```

4 Resuts

Table 1: Average Transmissions per Message for Different Values of P

P	Average Transmissions per Message
0.05	1.172414
0.1	1.293103
0.15	1.405172
0.2	1.672414
0.25	1.836207
0.3	2.474138
0.35	2.413793
0.4	2.594828
0.45	3.017241
0.5	3.456897