# Networks Laboratory (CS39006)
# Assignment-4 Report

**Suhas Jain (19CS30048) and Parth Jindal (19CS30033)**

## 1 Data Structures in `rsocket.h` and `rsocket.c`

### 1.1 Unacknowledged-Message Table Entry

This structure stores all the information in a particular entry of the unacknowledged-message table. Below given are the data fields and then a brief description about functionality of each of them.

```
struct unack_message {
    void *data;
    time_t timestamp;
    int seq_num;
    struct sockaddr_in addr;
    int len;
    int sent_count;
    short status;
};
```

Brief description about purpose and functionality of each field:

- `void* data` : This stores the message that is being sent and receive between different users. This message is stored as `void*` in the table and is also communicated as `void*` and is explicitly converted to `char*` as and when required.
- `time_t timestamp`: This denotes the time stamp at which this particular message was sent last time. Every time this message is sent again (due to no `ack` being recieved), this data field is updated.
- `int seq_num`: This denotes the index in the unacknowledged-message table at which this `unack_message` entry is currently stored.
- `struct sockaddr_in addr`: The `sockaddr_in` structure specifies a transport address and port for the `AF_INET` address family. This particular field stores address of the receiver (server).
- `int len`: This field stores the length of the message i.e. the number of characters in the message or size of message in bytes.
- `sent_count`: This is a counter which stores how many times a particular message has been sent to the receiver (server) because of sender not receiving an `ack` message. This field will be used to make the table for varying `P` values in the later part of the report.

- `status`: This field stores 1 or 0 depending on weather this message has been acknowledged or not. If this field is set as 1 that means it has been acknowledged and a new entry can replace the current entry at this position in the table.

## 1.2 Unacknowledged-Message Table

This is a structure storing the unacknowledged-message table. Below given are the data fields and then a brief description about functionality of each of them.

```
struct unack_mtable {
    struct unack_message msgs[_WINDOW_SIZE];
    int count;
    pthread_mutex_t mutex;
} * unack_mtable;
```

Brief description about purpose and functionality of each field:

- `struct unack_message msgs[_WINDOW_SIZE]`: The main array storing all the information in the table. Each entry of the array is of the form `struct unack_message`, which has been described earlier. `_WINDOW_SIZE` is the size of this array which has been set to 63, as it is mentioned in the assignment that there will not be more than 50 entries at a time in the table.

- `int count`: Total number of unacknowledged entries in the table.

- `pthread_mutex_t mutex`: Mutex lock corresponding to this table so it can be locked and unloacked as and when required.

## 1.3 Received-Message Table Entry

This structure stores all the information in a particular entry of the unacknowledged-message table. Below given are the data fields and then a brief description about functionality of each of them.

```
struct recv_message {
    void *data;
    int len;
    struct sockaddr_in addr;
};
```

Brief description about purpose and functionality of each field:

- `void* data`: This stores the message that is received from the client. This message is stored as `void*` in the table and is also communicated as `void*` and is explicitly converted to `char*` as and when required.

- `int len`: This field stores the length of the message i.e. the number of characters in the message or size of message in bytes.

- `struct sockaddr_in addr`: The `sockaddr_in` structure specifies a transport address and port for the `AF_INET` address family. This particular field stores address of the receiver (server).

## 1.4 Received-Message Table

This structure stores all the information in and about the unacknowledged-message table. Below given are the data fields and then a brief description about functionality of each of them.

```
struct recv_mtable {
    struct recv_message msgs[_WINDOW_SIZE];
    int count;
    int in, out; // circular buffer
    pthread_mutex_t mutex;
} * recv_mtable;
```

Brief description about purpose and functionality of each field:

- `struct recv_message msgs[_WINDOW_SIZE]`: This array stores all the received messages in the from of `struct recv_message`. As we have to continuously add a new value in the front and remove the last value we treat this as a circular array. `_WINDOW_SIZE` is the size of this array which has been set to 63, as it is mentioned in the assignment that there will not be more than 50 entries at a time in the table.
- `int count`: This field stores the number of entries in the table currently.
- `int in`: This signifies the index in the array at which the new element will be added in the circular array.
- `int out`: This signifies the index in the array at from which the element will be popped from the circular array.
- `pthread_mutex_t mutex`: Mutex lock corresponding to this table so it can be locked and unloacked as and when required.

## 1.5 Message Transfer Format

Here, we will discuss about the particular format we follow in which we transfer message packets between client and server. We will discuss the structure of a general packet in the protocol, which can store both `ack` message that is sent from the receiver as well as the original message that was sent from the sender.

| | |
|---|---|
| 32-bits integer | string storing the message |

- **In case of `ack` message**: In this case second part of the packet is empty and it is only 32-bit so we can directly assume that it is an `ack` message. The value of the integer signifies which message number it is acknowledging.

- **In case of normal message**: In this case second part of the packet is not empty and it is more 32-bit so we can directly assume that it is normal message being sent from sender to receiver. The value of the 32-bit integer header signifies the message number to be acknowledged by the receiver.

## 2 Functions in `rsocket.c`

### 2.1 `int add_unack_msg(const void *data, ssize_t len, struct sockaddr_in *addr)`

When sender (client) sends a message to the receiver for the first time a new entry is made in the unacknowledged-message table. This function creates a new object of type `struct unack_message` and finds a index in the table whose message has **ACK** in the status field.
**Return Value**: The index at which new entry was made. If table is full, it returns -1.

### 2.2 `void free_unack_mtable(struct unack_mtable *table)`

Free up memory pointed by `data` pointers of all the entries in the unacknowledged-message table. This function is called when we are closing the socket using `r_close` function.

### 2.3 `void free_recv_mtable(struct recv_mtable *table)`

Free up memory pointed by `data` pointers of all the entries in the received-messages table. This function is called when we are closing the socket using `r_close` function.

### 2.4 `void* handle_S(void* param)`

This is the function which handles the **S** thread of our program. Since this function is called from `pthread_create` return type and parameters are kept as `void*`. After every **T** seconds, this thread iterates through the unacknowledged-message table and checks if there any entries with status as **UNACK**. If the status is **UNACK** and it has been more than **2T** time from when the message in that particular table entry was last sent then the message is sent again and `sent_count` and `timestamp` of the entry are updated.

### 2.5 `int add_recv_msg(struct recv_mtable *table, const void *data, ssize_t len, struct sockaddr_in addr)`

When receiver receives a new message, thus function is called and it makes a new entry corresponding to that new message in the received-messages table. As we treat the array in received-messages table as a queue, if the table has any space left then new entry is made at `in` index in the array.
**Return Value**: The index at which new entry was made. If table is full, it returns -1.

### 2.6 `int get_recv_msg(struct recv_mtable *table, char *data, ssize_t len, struct sockaddr_in *addr)`

After acknowledging a message from the received-messages table it has to be removed from the table and we have to free up the memory corresponding to that table entry. As we treat the array in received-messages table as a queue, if the table has non-zero number of entries then entry is removed from `out` index in the array. We copy the message that was removed into the `char* data` field so that it can be outputted by the receiver.
**Return Value**: We return the length of message that was copied in the `data` parameter of the function, if no message is copied (table is empty), we return -1.

### 2.7 `void* handle_R(void *param)`

This is the function which handles the **T** thread of our program. Since this function is called from `pthread_create` return type and parameters are kept as `void*`. This thread receives messages and then depending on weather it is an **ACK** message or a normal message, it either changes status of an entry in the table from **UNACK** to **ACK** or makes a new received-message table entry.

### 2.8 `int r_socket(int __domain, int __type, int __protocol)`

This function call creates a new socket, allocate space for unacknowledged-table as well as received-message table and also initialise **R** and **S** threads.
**Return Value**: File descriptor value of the new socket, returns -1 for errors.

### 2.9 `int r_bind(int __fd, const struct sockaddr *__addr, socklen_t __len)`

This function call is used to associate the socket with local address i.e. IP Address, port and address family.
**Return Value**: Returns 0 if the operation is successful, otherwise returns -1.

### 2.10 `ssize_t r_sendto(int __fd, const void *__buf, size_t __n, int __flags, const struct sockaddr *__addr, socklen_t __addr_len)`

This function sends the message as mentioned by the function parameters, creates a new unacknowledged-message table entry and coverts the message that is to be sent in the appropriate format as mentioned in section 1.5 before sending it.
**Return Value:** Upon successful completion, it return the number of bytes sent. Otherwise, -1 is returned.

### 2.11 `ssize_t r_recvfrom(int __fd, void *__restrict__ __buf, size_t __n, int __flags, struct sockaddr *__restrict__ __addr, socklen_t * __restrict__ __addr_len)`

Receives message according to the function parameters using `get_recv_msg` function call. It tries to receive message at an interval of 100 microseconds till it is successfully received.
**Return Value**: We return the length of message that was received, if there is an error, return -1.

### 2.12 `int r_close(int __fd)`

Closes the socket, terminates **R** and **S** threads and frees all the allocated memory to both the tables.
**Return Value**: Upon successful completion we return 0, otherwise we return -1.

## 3 Effect of Varying Drop Probability

As in a real UDP environment packet drops are rare and to test if our protocol is working properly or not and also to compare practical results with theoretical values we specify the probability of a message getting dropped as a marco, let us call it **P**. In the algorithm according we drop messages according to the set **P** value. Please note that this applies for both normal messages as well as **ACK** messages.
We vary **P** from 0.05 to 0.5 in the steps of 0.05 and send a set 36-character message multiple times to observe how many packets we have send in order to send the whole message according to our algorithm. We take an average across 3 trials and compare it with the theoretical value. Theoretical value is calculated according to the formula below.

$$\mathbf{Number\ of\ Packets} = \frac{\mathbf{Message\ Length}}{\mathbf{(1-P)^2}}$$

**36-character message sent**: `abcdefghijklmnopqrstuvwxyz1234567890`
Refer to this table for results:

| P | Trial 1 | Trial 2 | Trial 3 | Average | Theoretical Value |
|---|---|---|---|---|---|
| 0.05 | 41 | 40 | 44 | 41 | 39.88 |
| 0.10 | 52 | 42 | 43 | 45 | 44.44 |
| 0.15 | 48 | 52 | 50 | 50 | 49.82 |
| 0.20 | 64 | 55 | 57 | 58 | 56.25 |
| 0.25 | 70 | 59 | 60 | 63 | 64 |
| 0.30 | 65 | 75 | 73 | 71 | 73.46 |
| 0.35 | 100 | 86 | 83 | 89 | 85.20 |
| 0.40 | 109 | 95 | 116 | 106 | 100 |
| 0.45 | 111 1 | 111 | 151 | 124 | 119.01 |
| 0.50 | 148 | 142 2 | 161 | 150 | 144 |

**As we can see from the table above that the observed practical values are close to the theoretically calculated values and hence this validates the protocol we have implemented.**