My Reliable Protocol - Documentation

Ashutosh Kumar Singh - 19CS30008

Vanshita Garg - 19CS10064

Introduction

We implement our own socket type, called **My Reliable Protocol (MRP)** socket, that has the following characteristics:

- Any message sent using an MRP socket is always delivered to the receiver.
- MRP sockets do not guarantee in-order delivery of messages.
- MRP sockets do not guarantee exactly-once delivery of messages.
- Like UDP sockets, MRP is message-oriented, and not byte-oriented.

MRP sockets guarantee reliable delivery using a simple ACK and re-transmit mechanism, and by storing all unacknowledged messages in an unacknowledged message table.
We also maintain a received-message table that acts as a buffer for all the messages that have been received till now.

Frame Format

Message Type	Message ID	Message Data
1 byte (char)	2 bytes (short)	100 bytes max (char array)

- **Message Type**: A single character which will be either 'D', denoting a data message, or 'A', denoting an ACK message.
- **Message ID**: The identifier (sequence number) for the message.
- **Message Data**: The actual data to be transmitted. Note that this part remains empty for an ACK message.

Data Structures Used

Received Message Table

All the incoming messages into our socket (which is internally a UDP socket) have to be stored in a buffer so that when the user issues a `r_recvfrom` call, the first message in the buffer can be returned. All these messages are stored in the received message table. The memory for the received message table is allocated dynamically at runtime when a new socket is created using the `r_socket` call.

```
typedef struct _recvd_table_entry {
    short msg_id;
    char *msg;
    size_t msg_len;
    struct sockaddr src_addr;
    socklen_t addrlen;
} recvd_table_entry;
```

Each entry of the received message table has the above mentioned type and has the following fields:

- `short msg_id`: An identifier for the message.
- `char *msg`: The body (data) of the message. It is stored as a `char` pointer, and the memory is dynamically allocated and freed as and when needed.
- `size_t msg_len`: The number of bytes in the message.
- `struct sockaddr src_addr`: The structure storing the details like the IP address and port of the sender.
- `socklen_t addrlen`: The size of the `struct sockaddr`.

The received message table itself is implemented as a FIFO (First In First Out) queue, since we always want the user to get the message which was received first. The queue is implemented using a circular array of fixed size. It has the following structure:

```
typedef struct _recvd_table {
    recvd_table_entry *messages[MAX_TBL_SIZE];
    int in;
    int out;
    int count;
    pthread_mutex_t mutex;
} recvd_table;
```

- `recvd_table_entry *messages[MAX_TBL_SIZE]`: An array of pointers to entries in the table. If a message exists at a particular location in the array, then the pointer at that location points to the appropriate `recvd_table_entry`. If a location in the array is empty, the pointer is set to `NULL`.
- `int in`: The next free position to insert a message in the queue.
- `int out`: The next filled position from where a message should be retrieved.
- `int count`: The number of elements in the queue at any time.
- `pthread_mutex_t mutex`: A mutex lock to ensure mutual exclusion as the received message table will be shared across multiple threads.

Unacknowledged Message Table

To ensure reliable delivery, we use an ACK and re-transmit mechanism, and this is is aided by the unacknowledged message table. The sender sends a message using a UDP socket, and stores the message and the time it is sent in the unacknowledged message table. The receiver, on receiving the message, sends an ACK message back to the sender. The sender, on receiving the ACK for a message, removes that message from the unacknowledged message table. If either the message or the ACK is lost, a timeout occurs at the sender, and the sender resends the message, and resets the sending time of that message in the unacknowledged message table.
The memory for the unacknowledged message table is allocated dynamically at runtime when a new socket is created using the `r_socket` call.

```
typedef struct _unackd_table_entry {
    int msg_id;
    char *msg;
    size_t msg_len;
    int flags;
    struct sockaddr dest_addr;
    socklen_t addrlen;
    struct timeval sent_time;
} unackd_table_entry;
```

Each entry of the unacknowledged message table has the above mentioned type and has the following fields:

- `int msg_id`: An identifier for the message.
- `char *msg`: The body (data) of the message. It is stored as a `char` pointer, and the memory is dynamically allocated and freed as and when needed.
- `size_t msg_len`: The number of bytes in the message.
- `int flags`: This is the `flags` argument required for a `sendto` call to convey additional requirements and infornation. However, for our purposes it will always be 0.
- `struct sockaddr dest_addr`: The structure having the details, like IP address and port, of the destination to which the message has been sent, and might be sent again.
- `socklen_t addrlen`: The size of the `struct sockaddr`.
- `struct timeval sent_time`: The last time when a message was sent. It is needed to determine if a timeout has occurred.

The unacknowledged message table is implemented as a fixed size array of pointers to the struct `unackd_table_entry`. An important point to note is that when an ACK is received, we may have to delete elements from the middle of the array. In such cases, we free the allocated memory for that message, and set the pointer at that location to `NULL`. When a position does not have a message, it is set to `NULL`. It has the following structure:

```
typedef struct _unackd_table {
    unackd_table_entry *messages[MAX_TBL_SIZE];
    int count;
    pthread_mutex_t mutex;
} unackd_table;
```

- `unackd_table_entry *messages[MAX_TBL_SIZE]`: A fixed size array of pointers to the struct `unackd_table_entry`. When a position does not have a message, it is set to `NULL`. When we need to insert a message, we look for an empty spot in the array, and set the pointer at that position to the message entry. When we need to delete a message from the table, we free the memory allocated to it, and set the pointer at that position to `NULL`.
- `int count`: The number of messages in the table at any time.
- `pthread_mutex_t mutex`: A mutex lock to ensure mutual exclusion as the unacknowledged message table will be shared across multiple threads.

Utility of Different Threads

The main thread performs the task of executing the appropriate library function for our MRP sockets, as and when requested by the user. However, there will be other tasks which must run concurrently like receiving incoming messages, and handling timeouts and retransmissions. Hence, we need two more threads to achieve these functionalities.

The thread R is responsible for handling all incoming messages into the underlying UDP socket. It waits for a message to come using the `recvfrom` call. When a message is received, if it is a data message, then the thread R stores it in the received message table, and sends an ACK message to the sender. If it is an ACK message in response to a previously sent message, it updates the unacknowledged message table by deleting the message for which the acknowledgement has arrived. More implementation level details are provided in the description of the `recv_thread` function below.

The thread S handles timeouts and retransmissions. It sleeps for a time T, and wakes up periodically. On waking up, it scans the unacknowledged message table to see if any message's timeout period (set to $2T$) is over (by calculating the difference between the time in the table entry for the message and the current time). If yes, then it retransmits that message and resets the time in that entry of the table to the new sending time. If not, no action is taken. This is repeated for all messages in the table every time the thread S wakes up. More implementation level details are provided in the description of the `retransmit_thread` function below.

Description of all Functions

Helper Functions (only in `rsocket.c`)

- `LOCK`: A wrapper around `pthread_mutex_lock` to facilitate error checking and display error messages.
- `UNLOCK`: A wrapper around `pthread_mutex_unlock` to facilitate error checking and display error messages.
- `init_recvd_table`: Allocates memory for the received message table (`recvd_msg_tbl`), sets all pointers to the entries to `NULL`, initializes the variables `in`, `out`, and `count` to 0, and initializes the mutex lock.
- `enqueue_recvd_table`: Inserts a new message to the front of the received message table queue, at the position pointed to by `in`, moves the `in` pointer one position ahead (in a cyclic fashion), and increments `count` by 1.
- `dequeue_recvd_table`: Removes the message pointed to by `out` from the queue, moves the `out` pointer to one position ahead, and decrements `count` by 1. It also frees the dynamically allocated memory for the message (`char *msg`) and the `recvd_table_entry`, and the pointer at that location is set to `NULL`.
- `free_recvd_table`: Frees the memory allocated to the messages and entries in the received message table. It also destroys and releases the mutex lock for the received message table. It is called when the socket is being closed using the `r_close` call.
- `init_unackd_table`: Allocates memory for the unacknowledged message table (`recvd_msg_tbl`), sets all pointers to the entries to `NULL`, initializes the variable `count` to 0, and initializes the mutex lock.
- `insert_unackd_table`: Finds an empty spot in the unacknowledged message table, inserts a new message at that spot, and increments `count` by 1.

- `delete_unackd_table`: Searches for the message with a given `msg_id`, which is passed as a parameter. Deletes that message from the table, and decrements `count` by 1. It also frees the dynamically allocated memory for the message (`char *msg`) and the `unackd_table_entry`, and the pointer at that location is set to `NULL` to indicate that this location in the table is now empty.

- `free_unackd_table`: Frees the memory allocated to the messages and entries in the unacknowledged message table. It also destroys and releases the mutex lock for the unacknowledged message table. It is called when the socket is being closed using the `r_close` call.

- `recv_thread`: This is the function that is executed by the thread R, and is responsible for receiving messages. It waits for a message by calling `recvfrom`, which is a blocking call. When a message is received, at first `dropMessage` is called, and if it returns 1, then the data/ACK message is dropped, and it waits for the next message. If `dropMessage` returns 0, then an appropriate action is taken depending upon the type of the message. If it is a data message, then we wait till we have a vacant position in the received message table, and then we insert the data message at the front of the received message table queue by calling `enqueue_recvd_table`. During this step, we appropriately lock and unlock the received message table data structure. In this case, we also send back an acknowledgement (ACK) to the sender, for the received data message. On the other hand, if the original message that was received was an ACK message, then we simply delete the message with the received message id from the unacknowledged message table by calling the `delete_unackd_table` function. Again, we take care of mutual exclusion by appropriately locking and unlocking the unacknowledged message table data structure. We have also installed cancellation points at few points in the function (using `pthread_testcancel`), where it is safe to cancel the thread.

- `retransmit_thread`: This function is executed by the thread S, which takes care of timeouts and re-transimissions. It sleeps for a time T, then wakes up and checks if any message in the unacknowledged message table has timed out. This is done by checking if the difference of the current time and the last sent time of any message is greater than the timeout value. If any such message is found that has timed out, then it re-transmits that message, and updates its last sent time to the current time. Here too, we appropriately lock and unlock the unacknowledged message table data structure to ensure mutual exclusion. We have also installed cancellation points at few points in the function (using `pthread_testcancel`), where it is safe to cancel the thread.

Main Library Functions (in both `rsocket.h` and `rsocket.c`)

- `dropMessage`: It first generates a random number between 0 and 1. If the generated number is less than the argument p, then the function returns 1, else it returns 0. It is used during testing to simulate an unreliable link.

- `r_socket`: Opens a UDP socket with the `socket` call. It also creates the 2 threads R and S, and allocates initial space for the tables and initializes them by calling `init_recvd_table` and `init_unackd_table`. The parameters to these are the same as the normal `socket` call, except that it will take only `SOCK_MRP` as the socket type.

- `r_bind`: Binds the socket with an address and a port using the `bind` call.

- `r_sendto`: It first converts the message to the appropriate frame (message) format by prepending the message with the type ('D' for data message), and the message id. It then sends the message using the `sendto` call. It then inserts the message along with its message id and destination address and port (`struct sockaddr`) into the unacknowledged message table by calling the `insert_unackd_table` function. With each entry, there is also a `sent_time` field that is filled up initially with the time of first sending the message.

- `r_recvfrom`: It behaves similar to `recvfrom` by acting as a blocking call, and it returns to the user only when a message is available. It looks up the received message table to see if any message has already been received in the underlying UDP socket or not. If yes, then it returns the first message from the received message table queue by calling `dequeue_recvd_table` and that message is removed from the received message table. If no message is found, it blocks the call. To block the call, we sleep for 1 second and then see again if a message is received.
- `r_close`: It first waits for the count of messages in the unacknowledged message table to become 0, to ensure that all messages have been received by the receiver and acknowledged. It then cancels (kills) the threads R and S by issuing the `pthread_cancel` command. This allows the threads to terminate in an orderly fashion and ensure that all resources are released. It then waits for the threads to terminate using the `pthread_join` command. It then frees the received message table and the unacknowledged message table. If there is any data in the received message table then it is discarded. It then closes the socket using the `close` call which releases all the resources associated with the socket.

Average Number of Transmissions by Varying the Drop Probability p

We send the string `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN` of length 40 from *user1* to *user2*. For every p value from 0.05 to 0.50 (in steps of 0.05), we perform 3 iterations, and take the average of them (rounded down) to be the total number of transmissions required. That divided by the length 40 gives us the average number of transmissions required to send each character.

p	**Iteration 1**	**Iteration 2**	**Iteration 3**	**Total no. of transmissions (averaged)**	**Avg. no. of transmissions per character**	**Theoretical value**
0.05	43	46	46	45	1.125	1.108
0.10	48	51	47	48	1.200	1.234
0.15	56	58	52	55	1.375	1.384
0.20	57	66	57	60	1.500	1.562
0.25	77	75	70	74	1.850	1.777
0.30	88	81	84	84	2.100	2.040
0.35	105	88	87	93	2.325	2.366
0.40	120	122	116	119	2.975	2.777
0.45	131	147	130	136	3.400	3.305
0.50	167	176	169	170	4.250	4.000

Theoretically, a character is successfully transmitted when neither the message nor the ACK is dropped. This happens with a uniform probability of $(1-p)*(1-p)$. Hence, the theoretical expected number of transmissions required for a successful delivery of a single message is $1/(1-p)^2$. We can indeed see that the observed and theoretical values are very close.