My Reliable Protocol

- **Himanshu Mundhra**
16CS10057

Table for Average Number of Transmissions per Reliable Packet Delivery

	25	40	55	70	85	100	Theoretical	Calculated
0.05	26	49	62	80	100	109	1.1080	1.1336
0.1	31	56	70	81	94	122	1.2346	1.2326
0.15	33	58	75	97	115	143	1.3841	1.3837
0.2	39	62	92	106	129	170	1.5625	1.5858
0.25	47	74	101	119	150	188	1.7778	1.8185
0.3	48	84	107	134	156	194	2.0408	1.9425
0.35	50	83	137	150	188	240	2.3669	2.2201
0.4	82	113	166	213	226	265	2.7778	2.9125
0.45	85	134	178	246	269	337	3.3058	3.3392
0.5	115	160	221	287	335	390	4.0000	4.0932

The rows here correspond to a particular value of **P**, the drop probability.
The columns here correspond to a particular value of the **Length** of the Transmitted String.

Theoretically, a successful delivery occurs when the Packet and the ACK are NOT DROPPED.
So that occurs with a **Uniform Probability** of **(1-P)*(1-P)**.
So the Expected Number of Transmissions required for a Successful Reliable Delivery is **1/((1-P)^2)** . This is how the Theoretical Value is Calculated.

The Calculated Value is obtained by taking the **Mean** of the Values for each different length.

As we can see, the Theoretical and Calculated Values are coming out to be around the same, And the differences can be attributed to the fact that the rand( ) in C is not truly perfectly a Random number generator.

My Reliable Protocol Sockets or MRP Sockets

- Message Oriented
- Reliable
- Exactly Once

TCP without byte-orientation and in-order delivery OR better yet **UDP with a Reliability Wrapper Around it.** We basically use a UDP socket to perform the message transfer but wrap around it a reliability check which ensures that each message is DEFINITELY DELIVERED to the receiver and EXACTLY ONCE. However, we are NOT bothered with maintaining the ordering of the data packets.

Each MRP Socket contains

- A UDP Socket, through which the actual communication is happening
- Receive Buffer
- Received Message-Id Table
- UnAcknowledged-Message Table
- Thread X

Receive Buffer - Contains the "application" messages which are received.
Contains each message sent exactly once before it is read by the user.
Received Message-ID Table - Contains all the distinct message-ids received once by socket
UnACK Message Table - Contains all those messages' details which are sent by the sender but not ACKed by the receiver.

Maximum Message Size => 100 bytes , Maximum Unique Messages => 100 , Timeout Value => 2 Seconds
DATA STRUCTURES available **globally** to all the functions
Allocation of Memory done **Dynamically** at time of Socket Creation, Freed at time of Socket Closure.
Only one socket is created by a process for send and receive => **one set of variables** shall do.

Creating the Socket:: r_socket()

- **Creates** the UDP Socket
- **Dynamically allocates space** to the global table pointers and **Initialises them.**
- **Creates** the Thread X

Binding the Socket:: r_bind()

- **Binds** the UDP Socket to the specified address-port

Closing the Socket:: r_close()

- **Blocks** till there are UnAck Messages in the **UnACK Message** Table
- **Closes** the Socket
- **Kills** the Thread X
- **Free** the Allocated Memory associated with the socket
    - **Discard** any data present in the **Received-Message** Table

Sending a Message by the user:: r_sendto()

- **Creates** uniqueID (“global” **counter** starting from 0)
- **Appends** uniqueID, **Creates** Packet and **Sends** application message using *one* UDP sendto()
- **Stores** message, id, flags, destination IP-Port, and Time in **UnACK Message Table**.
- Everything done in the main thread, no THREAD X needed here
- **Returns** to user

Receiving a Message by the user:: r_recvfrom()

- **Checks** **Receive Buffer** for the presence of a message
- **Message Absent -** Process is blocked till a message is received & added in the **Receive Buffer** (*in the background by Thread X*)

  How to block () - **sleep()** for some time and **check** after that if something to read has arrived
- **Message Present** - the **First Message is deleted** from records and given to the user.
  - the Function returns the number of bytes in the message to the user.

Thread X waits on select() over the created socket for a **receive** or **a timeout of T**

If it comes out on **a timeout** it calls function **HandleRetransmit()** *(shm Access => MutEx)*

- Scans **complete** UnAck Message Table to check if there is some message whose timeout is over.
  - If **No**:: No Action is taken
  - If **Yes**:: Retransmits message using Data in UnACK Table and sendto()
    Resets time in that table entry to this new time
- Returns to Thread X

Goes back to wait on a select() over the socket with a FRESH TIMEOUT of T

If it comes out on **a receive** it calls function **HandleReceive()** *(shm Access => MutEx)*

- Checks Message for AppMsg (**HandleAppMsg()**) or ACKMsg (**HandleACKMsg()**)

  HandleAppMsg()
  - Checks **Received-MessageID Table** for checking if it is present (duplicate)
    - If **No**:: Message Added to Receive Buffer (with source IP-port, **not ID**)
      ACK is sent out
    - If **Yes**:: Message is dropped
      ACK is sent out
  - Returns to Thread X

  HandleACKMsg()
  - Checks **UnACK-Message Table** for checking if it is present
    - If **No**:: It is a duplicate, it is ignored
    - If **Yes**:: Entry is removed from the table
  - Returns to Thread X

Goes back to wait on a select() over the socket with a REMAINING TIMEOUT of Trem

Message Format

Type (1 byte)	**ID** (2 bytes)	**Message Data** (100 bytes MAX)

Type - Helps to distinguish between an Application Message and an Acknowledgement Message
The Character:: **'P'** standing for a**P**plication Message
The Character:: **'C'** standing for a**C**knowledgement Message
ID - This is a short integer which contains a value ϵ [0, 100) denoting the unique ID of each packet
Message - This portion contains the actual data to be sent over the network.
This portion is empty for an ACK Message

Receive Buffer

This buffer acts as the "storage" of received messages at the receiver side, waiting to be sent to the user via a ***r_recvfrom()*** call. At each call, the First Message is sent to the user :: a FIFO Strategy is implemented.

The **Msg_Info** shall store:: **Message and Length** and **Source IP - Port**.

```
typedef struct __Msg_Info {
        char* Message ; int Msg_Len ; struct sockaddr* Source ; socklen_t SAddr_Len ;
} Msg_Info ;
```

The **recv_buffer** stores a bounded circular array of **Msg_Info** with necessary parameters and mutex locks.

```
typedef struct __buffer {
        Msg_Info* buffer ; int start ; int end ; int size ; int max_size ;
        pthread_mutex_t Lock ; pthread_cond_t Empty ;  pthread_cond_t Full ;
} _buffer ;
_buffer recv_Buffer ;
```

We implement the **Receive Buffer** as a **bounded circular buffer** with a **max-limit of 100 messages**.

The Receive Buffer is Accessed concurrently by the
Thread X - ***HandleAppMsg()*** which **adds packets** to the Receive Buffer
r_recvfrom() - which **removes packets** from it to send to the user (Application Layer)

Each time this shared memory is accessed we **Lock** and enter and then **Wait** on the Condition Variable.
HandleAppMsg() **waits** on the Condition Variable **Full**, waiting for a **signal** from ***r_recvfrom()*** after the latter has removed a packet from the Receive Buffer, and sends a **signal** to **Empty** after processing.
r_recvfrom() **waits** on the Condition Variable **Empty**, waiting for a **signal** from ***HandleAppMsg()*** after the latter has added a packet to the Receive Buffer, and sends a **signal** to **Full** after processing.

In this way we ensure **Mutual Exclusion** in the access of the **Shared Receive Buffer** space such as to prevent **deadlocks** or **race conditions.**

Received Message-ID Table

This table is used to maintain a record as to which message-IDs have been received and are stored in the Receive Buffer. It can simply be implemented as a boolean hash array.
To check if received message with ID 'id' is a duplicate or not, we can check the boolean value of the array[id]
short int **recv_id[100]** ; (all initialised to 0)

UnAcknowledged Message Table

This table is used to keep track of the details of the message which have been sent but we have not yet received an ACK for them.

We store all the details of the message including the **ID, Data, Destination IP - Port, Flags , Time of Sending** as all these will help us in case of a timeout and necessary retransmission.

```
typedef struct __Ack_Data {
        char* Message ; size_t Msg_Len ;
        int Msg_flag ;
        struct sockaddr* Destination ; socklen_t DAddr_Len ;
        struct timeval Msg_time ;
} Ack_Data ;
```

This wrapper stores the Table, Number of Outstanding Messages and the Mutex Lock and Condition variable.

```
typedef struct __UnACK {
        ACK_Data** Table ; int Num ;
        pthread_mutex_t Lock ; pthread_cond_t Exists ;
} _UnACK ;
_UnACK UnACK ;
```

This can be implemented as an array of structure pointers indexed by the ID values. A value of NULL implies no ACK is being awaited for this particular id.

The UnACK Data Structure is accessed by

- **r_sendto()** - sends out packages and **adds elements to the UnACK.Table**
- **ThreadX** - ***HandleRetransmit()*** **modifies the existing UnACK Packages** while retransmitting them
- **ThreadX** - ***HandleACKMsg()*** **receives ACKs** and removes them from the UnACK.Table
- **r_close()** - **Waits for all UnACKs** to be received before safely shutting down the socket.

Each time this shared memory is accessed we **Lock** and enter and then **Wait** on the Condition Variable.
r_close() - **waits** on the Condition Variable **Exists**, waiting for a **signal** from ***HandleACKMsg()*** after the latter has removed a packet from the UnACK.Table (which may be the last Outstanding Message).
HandleACKMsg() - sends a **signal** to ***r_close()*** each time it removes an ACK_Data from the Table.
All other instances properly **Lock** and **Unlock** themselves before accessing any shared memory data.

Code testing specifics

- Since package drops in lab environments are close to 0,
  we simulate an unreliable network with the help of a function dropMessage() whose prototype is added to the header file and implementation in the corresponding c file
  int dropMessage(float p)
    - **‘p’** is the probability of dropping a received packet
    - A uniform random variable **‘r’** is generated between 0 and 1
        - If **r < p** :: the function returns 1
        - If **r >= p** :: the function returns 0
- This dropMessage() is **added to the code of Thread X after a message has been received**
    - If dropMessage() returns **1** :: No Action taken, return to wait on select() with **Trem**
    - If dropMessage() returns **0** :: Continue Normal Processing of Received Packet
- The probability of dropping **‘p’** is defined in header file is varied over 0.05 - 0.5 by 0.05
- The metric **Ave No. of Transmissions** or **“**No. of transmissions / No. of Packets**”** is calculated for all the different values of **‘p’** and tabulated for comparison purposes.
- To calculate the Number of Transmissions, we add a counter in the code of Thread X to calculate the number of times a packet is received in the **HandleReceive()** function.