CS 349: Networks Lab
(January-May 2019)

Assignment – 3: Socket Programming

Submission Deadline: 11:55 pm on Friday, 15th March 2019 (hard deadline)

This assignment is a programming assignment where you need to implement an application using socket programming in C programming language. The assignment will be solved in groups where each group is comprised of 3 members. The group membership information is given in pages 11-13 of this document. The applications' description and requirement specification are given in pages 2-10 of this document.

Instructions:

1. Each group needs to implement one application assigned to it and make one single submission on Moodle. Only one member from a group needs to make the submission. The information about the assignment of applications to groups is contained in **Table 1** given below.
2. The application should be implemented with socket programming in C programming language only. No other programming language other than C will be accepted.
3. Submit the set of source code files of the application as a zipped file on Moodle (maximum file size is 1 MB) by the deadline of **11:55 pm on Friday, 15th March 2019 (hard deadline)**. The **ZIP file's name should be the same as your group number**, for example, "Group_4.zip", or "Group_4.rar", or "Group_4.tar.gz".
4. The assignment will be evaluated through viva voce in your lab during your lab session on **Wednesday, 27th March 2019 (ML-3: 9:00 am to 11:55 am)** where you will need to explain your source codes and execute them before the evaluator (evaluation schedule and TA allocation will be notified in due time).
5. **Write your own source codes and do not copy from any source. Plagiarism and use of unfair means will be penalised by awarding NEGATIVE marks (equal to the maximum marks for the assignment).**

Reference Text Book:

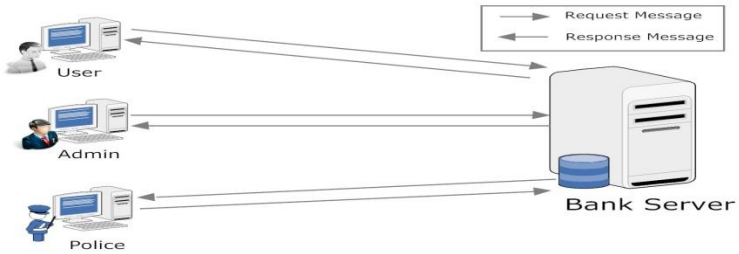
- "*Unix Network Programming*", Volume 1, by W. Richard Stevens (publisher: Prentice Hall) (refer to first few chapters)

Table 1: Assignment of applications to groups

Application Number	Application Name	Group Numbers
1	Banking System	1, 10, 19, 28, 37
2	Trading System	2, 11, 20, 29, 38
3	Base64 Encoding System	3, 12, 21, 30, 39
4	DNS Resolving System	4, 13, 22, 31, 40
5	TCP and UDP Sockets	5, 14, 23, 32, 41
6	Peer-to-Peer System	6, 15, 24, 33, 42
7	Point-of-Sale Terminal	7, 16, 25, 34, 43
8	Error Detection using CRC	8, 17, 26, 35, 44
9	File Transfer Protocol	9, 18, 27, 36, 45

Application #1: *Banking System using Client-Server socket programming*

In this application, you require implementing two C programs, namely Client and Bank Server, and they communicate with each other based on TCP sockets. The goal is to implement a simple Banking System.



Initially, the client will connect to the bank server using the server's TCP port already known to the client. After successful connection, the client sends a Login Message (containing the Username and password) to the bank server. The client side, we can have three different types of user modes namely, Bank_Customer, Bank_Admin and Police. The bank server has the following files with him: Login_file (contains the login entries, assume limited number of static entries only), Customer_Account_files (Assume the bank has 10 Bank_Customers only and one file for each customer, which maintains the transaction history. Refer to Login_file and Customer_Account_files formats for more details). Once the Bank server receives the login request, it validates the information and performs the functionalities according to the user mode type. The system must provide the following functionalities to the following users:

- **Bank_Customer:** The customer should able to see AVAILABLE BALANCE in his/her account and MINI STATEMENT of his/her account.
- **Bank_Admin:** The admin should be able to CREDIT/DEBIT the certain amount of money from any Bank_Customer ACCOUNT (as we do it in a SBI single window counter.☺). The admin must update the respective "Customer_Account_file" by appending the new information. Handel the Customer account balance underflow cases carefully.
- **Police:** The police should only be able to see the available balance of all customers. He is allowed to view any Customers MINI STATEMENT by quoting the Customer _ID (i.e. User_ID with user_type as 'C').

Login_file entry format:

User_ID	**Password**	**User_Type (C/A/P)**

Customer _Account_files entry format:

Transaction_Date	**Transaction Type (Credit/Debit)**	**Available Account_Balance**

Implement the functionalities using proper REQUEST and RESPONSE Message formats. After each negotiation phase, the TCP connection on both sides should be closed gracefully releasing the socket resource. You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number). Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

NB: Please make necessary and valid assumptions whenever required.

Application #2: *Client Server Trading System using socket programming*

A Client-Server Based Trading System is to be designed with the following specifications. There will be a set of traders who will trade with each other in the automated system. There will be a Server which will register requests from traders for buying and selling quantities of Items. The Server will also match the buy with the sell requests from different traders based on certain price rules (as listed below). Traders will log on to the trading system through the trading client (assume Trader ID and password is stored in a file). They will have the option to view the currently available items (for buy/sell), their quantities and their prices. They will also send requests for buying and selling items and specify the quantity and price. Traders will also have the option to view their matched trades at any time. There are ten known items which the traders can trade in with their codes from 1 to 10. There will be a maximum of 5 traders (with codes from 1 to 5) who can log on to the system and work. One trader should work from one client at a time only.

The functionalities of any client will be:

- **Login to the System:** The trader will execute the client, give the trader number and will be logged in. After that he/she will have the following options in a menu. Several clients will login (from different terminals) and assumed they don't trade simultaneously to reduce the complexity.
- **Send Buy Request:** The trader will send a buy request by stating the item code, the quantity and unit price.
- **Send Sell request:** The trader will send a sell request by stating the item code, the quantity and unit price.
- **View Order Status:** The Trader can view the position of buy and sell orders in the system. This will display the current best sell (least price) and the best buy (max price) for each item and their quantities.
- **View Trade Status:** The trader can view his/her matched trades. This will provide the trader with the details of what orders were matched, their quantities, prices and counterparty code.

There will be only one server which will be running and perform the functions of order processing and trade matching in addition to acknowledging logins by clients and servicing their requests. The order processing will be as follows. There will be a buy and a sell order queue for each item. On receiving buy/sell order request from a trader, the server will put it in the appropriate order queue. If there is a possibility of a trade match, then that trade match will take place, the traded items will be appropriately updated and the result of the trade along with the details of the counterparties, item, quantity and price will be stored in the traded set. The matching rule is as follows:

1. On a buy Request at price P and quantity Q of an item I, the server will check if there is any pending sell order for the same item at price P' ≤ P.
2. Among all such pending sell orders, the match will be made with the one having the least selling price.
3. If both have same quantity, i.e., Q'= Q, then both these orders will be removed from their respective queues and the result will be put into the traded set.
4. If Q' > Q then the buy request will be fully traded and the remaining part, i.e., Q' – Q of the sell order will remain in the sell queue at the same price P'.
5. On the other hand, if Q' < Q then the sell order will be fully traded and the remaining buy order will be tested for more matches.
6. If the buy order cannot be matched, it will be put into the buy queue.
7. A similar rule will apply for a sell request and all these requests will be handled in a FCFS basis.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number). *Please make necessary and valid assumptions whenever required.

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

Application #3: *Base64 encoding system using Client-Server socket programming*

In this application, you require to implement two C programs, namely server and client to communicate with each other based on TCP sockets. The aim is to implement simple Base64 encoding communication protocol.

Initially, server will be waiting for a TCP connection from the client. Then, client will connect to the server using server's TCP port already known to the client. After successful connection, the client accepts the text input from the user and encodes the input using Base64 encoding system. Once encoded message is computed the client sends the Message (Type 1 message) to the server via TCP port. After receiving the Message, server should print the received and original message by decoding the received message, and sends an ACK (Type 2 message) to the client. The client and server should remain in a loop to communicate any number of messages. Once the client wants to close the communication, it should send a Message (Type 3 Message) to the server and the TCP connection on both the server and client should be closed gracefully by releasing the socket resource.

The messages used to communicate contain the following fields:

Message_ Type	**Message**

1. Message_type: integer
2. Message: Character [MSG_LEN], where MSG_LEN is an integer constant
3. <Message> content of the message in Type 3 message can be anything.

You also require implementing a "***Concurrent Server***", i.e., a server that accepts connections from multiple clients and serves all of them *concurrently*.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

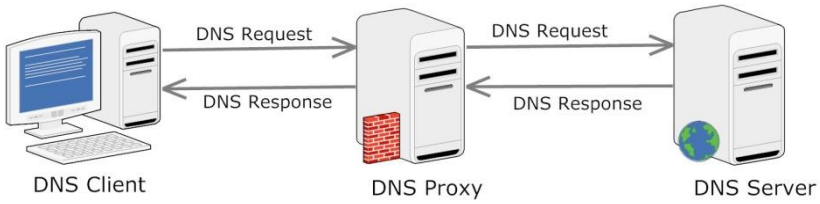
NB: Please make necessary and valid assumptions whenever required.

Base64 Encoding System Description:

Base64 encoding is used for sending a binary message over the net. In this scheme, groups of 24bit are broken into four 6 bit groups and each group is encoded with an ASCII character. For binary values 0 to 25 ASCII character 'A' to 'Z' are used followed by lower case letters and the digits for binary values 26 to 51 & 52 to 61 respectively. Character '+' and '/' are used for binary value 62 & 63 respectively. In case the last group contains only 8 & 16 bits, then "==" & "=" sequence are appended to the end.

Application #4: *Multi-stage DNS Resolving System using Client-Server socket programming*

In this application, you require implementing three C programs, namely Client, Proxy Server (which will act both as client and server) and DNS Server, and they communicate with each other based on TCP sockets. The aim is to implement a simple 2 stage DNS Resolver System.



Initially, the client will connect to the proxy server using the server's TCP port already known to the client. After successful connection, the client sends a Request Message (Type 1/Type 2) to the proxy server. The proxy server has a limited cache (assume a cache with three IP to Domain_Name mapping entries only). After receiving the Request Message, proxy server based on the Request Type (Type 1/Type 2) searches its cache for corresponding match. If match is successful, it will send the response to the client using a Response Message. Otherwise, the proxy server will connect to the DNS Server using a TCP port already known to the Proxy server and send a Request Message (same as the client). The DNS server has a database (say .txt file) with it containing set of Domain_name to IP_Address mappings. Once the DNS Server receives the Request Message from proxy server, it searches in its file for possible match and sends a Response Message (Type 3/Type 4) to the proxy server. On receiving the Response Message from DNS Server, the proxy server forwards the response back to the client. If the Response Message type is 3, then the proxy server must update its cache with the fresh information using FIFO scheme. After each negotiation phase, the TCP connection on both sides should be closed gracefully releasing the socket resource.

Request Message Format:

Request_Type	Message

- Type 1: Message field contains Domain Name and requests for corresponding IP address.
- Type 2: Message field contains IP address and request for the corresponding Domain Name.

Response Message Format:

Response_Type	Message

- Type 3: Message field contains Domain Name/IP address.
- Type 4: Message field contains error message "entry not found in the database".

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

NB: Please make necessary and valid assumptions whenever required.

Application #5: *Client-Server programming using both TCP and UDP sockets*

In this application, you require to implement two C programs, namely server and client to communicate with each other based on both TCP and UDP sockets. The aim is to implement a simple 2 stage communication protocol.

Initially, server will be waiting for a TCP connection from the client. Then, client will connect to the server using server's TCP port already known to the client. After successful connection, the client sends a Request Message (Type 1 message) to the server via TCP port to request a UDP port from server for future communication. After receiving the Request Message, server selects a UDP port number and sends this port number back to the client as a Response Message (Type 2 Message) over the TCP connection. After this negotiation phase, the TCP connection on both the server and client should be closed gracefully releasing the socket resource.

In the second phase, the client transmits a short Data Message (Type 3 message) over the earlier negotiated UDP port. The server will display the received Data Message and sends a Data Response (type 4 message) to indicate the successful reception. After this data transfer phase, both sides close their UDP sockets.

The messages used to communicate contain the following fields:

Message_Type	**Message_Length**	**Message**

1. Message_type: integer
2. Message_length: integer
3. Message: Character [MSG_LEN], where MSG_LEN is an integer constant

<Data Message> in **Client** will be a **Type 3** message with some content in its message section.

You also require implementing a "***Concurrent Server***", i.e., a server that accepts connections from multiple clients and serves all of them *concurrently*.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

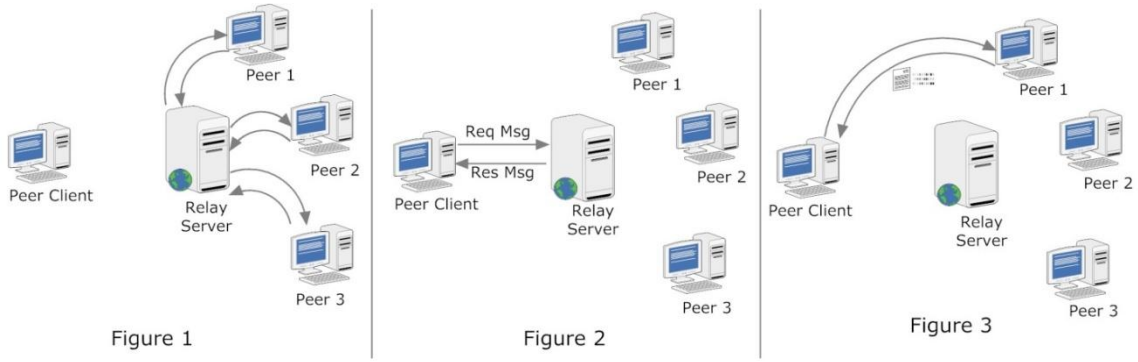
Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

NB: Please make necessary and valid assumptions whenever required.

Application #6: *Relay based Peer-to-Peer System using Client-Server socket programming*

In this application, you require implementing three C programs, namely Peer_Client, and Relay_Server and Peer_Nodes, and they communicate with each other based on TCP sockets. The aim is to implement a simple Relay based Peer-to-Peer System.



Initially, the Peer_Nodes (peer 1/2/3 as shown in Figure 1) will connect to the Relay_Server using the TCP port already known to them. After successful connection, all the Peer_Nodes provide their information (IP address and PORT) to the Relay_Server and close the connections (as shown in Figure 1). The Relay_Server actively maintains all the received information with it. Now the Peer_Nodes will act as servers and wait to accept connection from Peer_Clients (refer phase three).

In second phase, the Peer_Client will connect to the Relay_Server using the server's TCP port already known to it. After successful connection; it will request the Relay_Server for active Peer_Nodes information (as shown in Figure 2). The Relay_Server will response to the Peer_Client with the active Peer_Nodes information currently having with it. On receiving the response message from the Relay_Server, the Peer_Client closes the connection gracefully.

In third phase, a set of files (say, *.txt) are distributed evenly among the three Peer_Nodes. The Peer_Client will take "file_Name" as an input from the user. Then it connects to the Peer_Nodes one at a time using the response information. After successful connection, the Peer_Client tries to fetches the file from the Peer_Node. If the file is present with the Peer_Node, it will provide the file content to the Peer_Client and the Peer_Client will print the file content in its terminal. If not, Peer_Client will connect the next Peer_Node and performs the above action. This will continue till the Peer_Client gets the file content or all the entries in the Relay_Server Response are exhausted (Assume only three/four Peer_Nodes in the system).

Implement the functionalities using appropriate REQUEST and RESPONSE Message formats. After each negotiation phase, the TCP connection on both sides should be closed gracefully releasing the socket resource. You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

NB: Please make necessary and valid assumptions whenever required.

Application #7: *Point-of-Sale Terminal using socket programming*

Use socket programming to implement a simple client and server that communicate over the network and implement a simple application involving Cash Registers. The client implements a simple cash register that opens a session with the server and then supplies a sequence of codes (refer ***request-response messages format***) for some products. The server returns the price of each one, if the product is available, and also keeps a running total of purchases for each client's transactions. When the client closes the session, the server returns the total cost. This is how the point-of-sale terminals should work. You can use a TXT file as a database to store the UPC code and item description at the server end.

You also require implementing a "***Concurrent Server***", i.e., a server that accepts connections from multiple clients and serves all of them *concurrently*.

Request-response messages format

Request_ Type	**UPC-Code**	**Number**

Where

- ***Request_Type*** is either 0 for ***item*** or 1 for ***close***.
- ***UPC-code*** is a 3-digit unique product code; this field is meaningful only if the ***Request_Type*** is 0.
- ***Number*** is the number of items being purchased; this field is meaningful only if the ***Request_Type*** is 0.

For the ***Close*** command, the server returns a number, which is the total cost of all the transactions done by the client. For the ***item*** command, the server returns:

Response_ Type	**Response**

Where:

- <Response_type> is 0 for ***OK*** and 1 for ***error***
- If ***OK***, then <Response> is as follows:
    - if client command was "close", then <response> contains the total amount.
    - if client command was "item", then <response> is of the form <price><name>
      where
      <price> is the price of the requested item
      <name> is the name of the requested item
- If ***error***, then <Response> is as follows: a null terminated string containing the error; the only possible errors are "***Protocol Error***" or "***UPC is not found in database***".

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

The connection to the server should be gracefully terminated. When the server is terminated by pressing ***Control+C***, the server should also gracefully release the open socket (Hint: requires use of a signal handler).

NB: Please make necessary and valid assumptions whenever required.

Application #8: *Error Detection using Cyclic Redundancy Code (Using CRC-8)*

In this application, your aim will be to implement a simple Stop-and-Wait based data link layer level logical channel between two nodes **A** and **B** using socket API, where node **A** and node **B** are the client and the server for the socket interface respectively. Data link layer protocol should provide the following Error handling technique in Data Link Layer.

- Error Detection using Cyclic Redundancy Code
(using CRC-8 as generator polynomial, i.e. **$G(x) = x^8+x^2+x+1$**)

Operation to Implement:

- Client should construct the message to be transmitted (T(x)) from the raw message using CRC.
- At the sender side T(x) is completely divisible by G(x) (means no error), send ACK to the sender, otherwise (means error), send NACK to the sender.
- You must write error generating codes based on a user given BER or probability (random number between 0 and 1) to insert error into both T(x)and ACK/NACK.
- If NACK is received by the sender, it should retransmit the T(x) again following the above steps.
- In the client side also implement Timer Mechanism to detect the timeout (in case of error in ACK/ NACK) and retransmit the message T(x) again once time out happens.

You also require implementing a "***Concurrent Server***", i.e., a server that accepts connections from multiple clients and serves all of them *concurrently*.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

The connection to the server should be gracefully terminated. When the server is terminated by pressing ***Control+C***, the server should also gracefully release the open socket (Hint: requires use of a signal handler).

NB: Please make necessary and valid assumptions whenever required.

Application #9: *File Transfer Protocol (FTP) using Client-Server socket programming*

In this assignment, you require to implement two C programs, namely server and client to communicate with each other based on TCP sockets. The goal is to implement a simple File Transfer Protocol (FTP). Initially, server will be waiting for a TCP connection from the client. Then, client will connect to the server using server's TCP port already known to the client. After successful connection, the client should be able to perform the following functionalities:

- **PUT:** Client should transfer the file specified by the user to the server. On receiving the file, server stores the file in its disk. If the file is already exists in the server disk, it communicates with the client to inform it. The client should ask the user whether to overwrite the file or not and based on the user choice the server should perform the needful action.
- **GET:** Client should fetch the file specified by the user from the server. On receiving the file, client stores the file in its disk. If the file is already exists in the client disk, it should ask the user whether to overwrite the file or not and based on the user choice require to perform the needful action.
- **MPUT and MGET:** MPUT and MGET are quite similar to PUT and GET respectively except they are used to fetch all the files with a particular extension (e.g. .c, .txt, etc.). To perform these functions both the client and server require to maintain the list of files they have in their disk. Also implement the file overwriting case for these two commands as well.

Use appropriate message types to implement the aforesaid functionalities. For simplicity assume only .txt and .c file(s) for transfer.

You should accept the IP Address and Port number from the command line (Don't use a hard-coded port number).

Prototype for command line is as follows:

Prototypes for Client and Server

Client: <executable code><Server IP Address><Server Port number>
Server: <executable code><Server Port number>

NB: Please make necessary and valid assumptions whenever required.

Group Number	Group Members	
	Roll Number	Name
1	160101008	Abhishek Ranjan
	160101018	Avinash Uchchainiya
	160101034	Hemant Yadav
2	160101019	Tushara Langulya
	160101020	Bedadhala Manoj Reddy
	160101040	Bhargav Mallala
3	160101048	Nitin Kedia
	160101005	Abhinav Mishra
	160101049	Rohit Pant
4	150123023	Md Imtiyaz
	150101008	Ankit Vyas
	150101030	Krishna Kumar
5	160101066	Shivam Kumar
	160101067	Shreyanshi Bharadia
	160101059	Samyak Jain
6	160123047	Atharva Amdekar
	160123046	Rajat Paliwal
	160123051	Saurabh Rai
7	160123032	Rakshit Tiwari
	160123035	Satyam Kumar
	160123008	Eswar Modala
8	160101045	Lakshmi Sai Durga Myneni
	160101064	Pradeepa Seelam
	160101029	Ajay Ram Gudala
9	160123054	Deepak Kumar Gouda
	160101085	Akul Agrawal
	160123044	Yash Kothari
10	160101076	Vivek Raj
	160101039	Kapil Goyal
	160101057	Sahib Khan
11	160123024	Neelabh Tiwari
	160123048	Uddeshya Mathur
	160123049	Himanshu Raj
12	160101071	Siddharth Sharma
	160101054	Ravi Venkata Naga Pavan Kumar
	160101052	Poreddy Sai Kiran Reddy
13	160123039	Shreya Jain
	160123021	Muskan Agarwal
	160123014	Kartikey Kant
14	160101012	Ansh Sood
	160101031	Harshit Gupta
	160101038	Kanika Agarwal
15	160101070	Shubhanker Jauhari
	160101068	Shubhendu Patidar
	160123031	Rajan Sukanth
16	160101037	Kakustham Anurag
	160123015	Kodali Naga Sai Anirudh
	160123053	Shiva Reddy

Group Number	Group Members	
	Roll Number	Name
17	160123007	Divya Kumari
	160101028	Ekta Dhan
	160123052	Sayani Kundu
18	160101051	Phool Chandra
	160101062	Savinay
	160101063	Savsani Kevin Mukesh Bhai
19	160101001	Aadil hoda
	160101065	Shimona Verma
	160101072	Sparsh Bansal
20	160101035	Inderpreet Singh Chera
	160101042	Mitansh Jain
	160101073	Sujoy Ghosh
21	160123004	Ankam Aman sai
	160123013	Kamana Vishnu Vardhan Reddy
	160123041	Thirthala Udayasri
22	160101003	Abhay Kshatriya
	160123002	Abhishek Dogra
	160101010	Aditya Chouhan
23	160101058	Sahil Garhwal
	160101060	Sanchit Jangir
	160101078	Wakade Yugandhar
24	160123038	Shrey Jain
	160123025	Neha Oraon
	160123050	Naveen Mathew
25	160101021	Rajas Bhadke
	160101007	Abhishek Kumar
	160101027	Durgesh Yadav
26	160101087	Archit Jugran
	160101083	Shubham Goel
	160101079	Yagyansh Bhatia
27	160101044	Mukul Verma
	160101050	Paranjay Bagga
	160101075	Varun Kumar Kedia
28	160101014	Arpan Konar
	160101015	Arpit Gupta
	160101081	Debangshu Banerjee
29	160101030	Harshit Agrawal
	160101032	Harshit Sharma
	160123005	Anurag Barfa
30	160101016	Ashveen Bansal
	160101006	Abhishek Bhardwaj
	160101026	Divyansh Sharma
31	160101009	Abhishek Suryavanshi
	160101082	Ameya Daigavane
	160101084	Nitesh Jindal
32	160123018	MARUPAKA SAITEJA
	160123045	YERNENA SRINIVAS NAIDU
	160101077	VYKUNTAM AKHIL

Group Number	Group Members	
	Roll Number	Name
33	160101002	Aayush Sanjay Agarwal
	160101024	Daman Tekchandani
	160101033	Harshit Srivastava
34	160101043	Mohit Singh
	160101047	Nikhil Kumar
	160101055	Ritik Agrawal
35	160123026	nipendra singh
	160101056	sachin chouhan
	160101069	shubham kumar koul
36	160123010	Harshit Singh
	160123016	Kshitij Nayar
	160123017	Kuldeep Sharma
37	160101046	Namit Kumar
	160101036	Jatin Goyal
	160101061	Saurabh Bazari
38	160101086	Shaurya Gomber
	160101088	Rishabh Jain
	160123036	Shashwat Jolly
39	160123006	Ashish Ranjan
	160123003	Animesh Kumar
	160123020	Mitanshu Mittal
40	160123027	Nishant Jain
	160123028	Pranav Jangir
	160123030	Rahul Kumar Gupta
41	160123011	Himanshu ranjan
	160123001	Aashutosh Agarwal
	160123019	Mohammad zatin meraz
42	160123043	Yash Kumar
	160101023	Chandra Prakash Meena
	160101025	Divyam Agarwal
43	160123009	Garikapati Ganesh
	160123022	Nalgonda Gnaneshwar kumar
	160101022	Boddu Hari
44	160101004	Abhinav Hinger
	160101013	Apurva N Saraogi
	160123012	Ishan Azad
45	150101028	JIGNYASU RASESH CHASMAWALA
	160101080	YASH RATHORE
	160123029	PRANAV SINGH MUKATI
	160101011	AKHIL CHANDRA PANCHUMARTHI

— End of document —