SOCKET PROGRAMMING ALOGRITHMS

Experiment-6: A simple UDP Server-Client program which displays the current calendar time.

Server:

1. Create a UDP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Bind the socket to the server address using the bind() system call.
6. Check if the bind operation is successful and print a message.
7. Receive a message from the client using recvfrom() system call.
8. Print the received message.
9. Get the current date and time using time() and ctime() functions.
10. Format the date and time string and copy it to a buffer.
11. Send the formatted date and time back to the client using sendto() system call.
12. Close the server socket.

Client:

1. Create a UDP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Prompt the user to enter a message and store it in a buffer.
6. Send the message to the server using the sendto() system call.
7. Receive the response from the server using recvfrom() system call.
8. Print the received message from the server.

9. Close the client socket.

Experiment-7: A simple TCP Server-Client program where the client provides the username and password as request and the server authenticates the request and returns the result.

Server:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Bind the socket to the server address using the bind() system call.
6. Check if the bind operation is successful and print a message.
7. Listen for incoming connections with the listen() system call.
8. Accept a connection from a client using the accept() system call.
9. Check if the connection is accepted and print a message.
10. Receive the username from the client using the recv() system call.
11. Print the received username.
12. If the username is "admin," receive the password from the client.
13. Print the received password.
14. If the username and password are both "admin," send a confirmation message.
15. If the username or password is incorrect, send an appropriate error message.
16. Close the server socket.

Client:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Connect to the server using the connect() system call.

6. Print a prompt for the username and read it from the user.
7. Send the username to the server using the send() system call.
8. Print a prompt for the password and read it from the user.
9. Send the password to the server using the send() system call.
10. Receive the authentication result from the server using the recv() system call.
11. Print the authentication result.
12. Close the client socket.

Experiment-8: A simple TCP Server-Client program implementing a dictionary with meanings and antonyms.

Server:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Bind the socket to the server address using the bind() system call.
6. Check if the bind operation is successful and print a message.
7. Listen for incoming connections with the listen() system call.
8. Accept a connection from a client using the accept() system call.
9. Check if the connection is accepted and print a message.
10. Receive the word to be searched from the client using the recv() system call.
11. Print the received word.
12. Search the dictionary for the word and send its definition and antonym to the client using send() system calls.
13. If the word is not found in the dictionary, send an appropriate message to the client.
14. Close the server socket.

Client:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Connect to the server using the connect() system call.
6. Print a prompt for the user to enter a word to be searched in the dictionary.
7. Read the word from the user and send it to the server using the send() system call.
8. Receive the meaning of the word from the server using the recv() system call.
9. Print the received meaning.
10. Receive the antonym of the word from the server using the recv() system call.
11. Print the received antonym.
12. Close the client socket.

Experiment-9: A simple TCP Server-Client program that gets the MAC address and IP address of the client connected.

Server:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Bind the socket to the server address using the bind() system call.
6. Check if the bind operation is successful and print a message.
7. Listen for incoming connections with the listen() system call.
8. Accept a connection from a client using the accept() system call.
9. Check if the connection is accepted and print a message.

10. Receive the client's IP address from the client using the recv() system call.
11. Print the received IP address.
12. Receive the client's MAC address from the client using the recv() system call.
13. Print the received MAC address.
14. Close the server socket.

Client:

1. Create a TCP socket using the socket() system call.
2. Check if the socket creation is successful and print a message.
3. Initialize a sockaddr_in structure for the server address.
4. Set the server address family, port, and IP address.
5. Connect to the server using the connect() system call.
6. Get the client's IP address using the inet_ntoa() function and send it to the server using the send() system call.
7. Print a message indicating that the client's IP address has been sent to the server.
8. Use ioctl() to obtain the client's MAC address and send it to the server using the send() system call.
9. Print a message indicating that the client's MAC address has been sent to the server.
10. Close the client socket.

Experiment-10: Implementation of multiple chat program

Server:

1. Initialize an array to store client sockets (clients[]).
2. Initialize a variable n to keep track of the number of connected clients.
3. Initialize a mutex (mutex) for thread synchronization.

4. Implement a function sendtoall() to send messages to all connected clients except the sender.
5. Implement a function recvmg() to handle receiving messages from a specific client and broadcast them to others.
6. In the main function:
    a. Set up the server address structure (ServerIp) with appropriate values.
    b. Create a socket (sock) for communication.
    c. Bind the socket to the server address.
    d. Start listening for incoming connections.
    e. Inside an infinite loop:
        i. Accept a new client connection (Client_sock).
        ii. Lock the mutex to update the clients array and increment the client count.
        iii. Store the new client socket in the array.
        iv. Increment the client count.
        v. Create a new thread (recvt) to handle the communication with the new client.
        vi. Unlock the mutex.
7. Continue listening for new client connections and handling their messages.

Client:

1. Initialize a message buffer (msg) to store messages.
2. Implement a function recvmg() to continuously receive and display messages from the server.
3. In the main function:
    a. Create a socket (sock) for communication.
    b. Set up the server address structure (ServerIp) with appropriate values.
    c. Connect to the server.
    d. Create a client thread (recvt) to handle receiving messages from the server.
    e. Inside a loop:

i. Read a message from the console and prepend the client's name.
ii. Send the formatted message to the server using the write() system call.

f. Close the client thread using pthread_join() when the loop is exited.
g. Close the socket.

Experiment-11:Implementation of Echo Server using TCP

Server:

1. Create a socket using the socket() system call.
2. Check if the socket creation is successful.
3. Initialize the server address structure with appropriate values.
4. Bind the socket to the server address using the bind() system call.
5. Check if the binding is successful.
6. Start listening for incoming connections using the listen() system call.
7. If a connection is requested, accept the connection using the accept() system call.
8. Fork a child process to handle the communication with the client.
    a. In the child process:
        i. Close the original socket.
        ii. Continuously receive messages from the client using recv().
        iii. If the received message is ":exit," print a disconnection message and break the loop.
        iv. Otherwise, print the received message and send it back to the client using send().
    b. In the parent process, close the new socket.

9. Continue listening for incoming connections in the parent process.

Client:

1. Create a socket using the socket() system call.
2. Check if the socket creation is successful.
3. Initialize the server address structure with appropriate values.
4. Connect to the server using the connect() system call.
5. Check if the connection is successful.
6. In a loop:
    a. Prompt the user to enter a message.
    b. Send the entered message to the server using the send() system call.
    c. If the message is ":exit," close the client socket and exit.
    d. Receive the response from the server using the recv() system call.
    e. Print the received message.
7. Close the client socket.