A UDP-based Network diagnostic utility for Estimating Round-Trip-Time between computers
Experiment No: AV-341-2025-Lab-5

Saurabh Kumar
SC22B146
March 30, 2025

Date and Time of experiment: February 24, 2025, 15:00 IST

Objectives

- Realize a UDP communication session with echo packets using the concept of sockets.
- Create a simple MyPing utility to compute round-trip-time (RTT) between two processes. Round-trip-time is defined as the time for a process to send a message and receive its corresponding echo packet. we need to send 10 packets, say P_1, P_2, . . . , P_{10}, from client and find RTT for each of them. Finally compute average RTT as
  RTT $= \frac{\sum_{i=1}^{10} RTT_i}{10}$, where RTT_i is the RTT of the i^{th} packet.

Tools Used

- PC: 12th Gen Intel(R) Core(TM) i5-1240P 1.70 GHz, Windows 11 / Ubuntu 24.04 dual-booted, 64-bit, (reduced to) 4 GB RAM
- OS: Ubuntu 24.04
- Software used: VS Code, VS-Code in-built bash terminal, gcc compiler

Procedure

1. The following C program in written in VS code for creating echo server at port 5000 :

```
#include<sys/socket.h>
#include<arpa/inet.h>
#include<string.h>
#include<stdio.h>

#define BUFFERSIZE 1024
#define PORT 5000

int main()
{
  int sock;
  int bind_status, bytes_received, bytes_sent;
  char send_buf[BUFFERSIZE], recv_buf[BUFFERSIZE];

  struct sockaddr_in s_server, s_client;
  int si_len = sizeof(s_client);

  // Creating the socket
  sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if(sock < 0)
    {
      printf("Socket creation failed.\n");
    }
  else
    {
      printf("Socket created with descriptor %d\n", sock);
    }

  // Initializing and binding the socket
  s_server.sin_family = AF_INET;
  s_server.sin_port = htons(PORT);
  s_server.sin_addr.s_addr = htonl(INADDR_ANY);

  bind_status = bind(sock, (struct sockaddr*)&s_server, \
          sizeof(s_server));
  if(bind_status < 0)
    {
      printf("Binding socket failed.\n");
    }
  else
    {
      printf("Binding socket successful.\n");
    }




  // Server wait for a packet
  bytes_received = recvfrom(sock, recv_buf, sizeof(recv_buf), \
          0, (struct sockaddr*)&s_client, &si_len);

  printf("Data received from %s:%d\n", inet_ntoa(s_client.sin_addr)
   , \
   s_client.sin_port);
  printf("%d bytes of data received : %s\n", bytes_received,
   recv_buf);

  // Sending the received packet back to the client
  strcpy(send_buf, recv_buf);
  bytes_sent = sendto(sock, send_buf, sizeof(send_buf), 0, \
          (struct sockaddr*)&s_client, si_len);
  printf("%d bytes of data sent : %s\n", bytes_sent, send_buf);

  return 0;
}
```

2. The following C program in written in VS code for creating client to request the server at localhost:5000 :

```c
#include<sys/socket.h>
#include<arpa/inet.h>
#include<stdio.h>
#include<string.h>

#define BUFFERSIZE 1024
#define SERVERADDR "127.0.0.1"

int main()
{
  struct sockaddr_in s_server;
  int sock, bytes_received, si_len = sizeof(s_server), bytes_sent;
  char recv_buf[BUFFERSIZE], send_buf[BUFFERSIZE];

  // Creating the client socket
  sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if(sock < 0)
    {
      printf("Socket creation failed.\n");
      return 1;
    }
  else
    {
      printf("Socket creation successful at %d\n", sock);
    }


  // Assigning server IP and port for sending
  s_server.sin_family = AF_INET;
  s_server.sin_port = htons(5000);

  if(!inet_aton(SERVERADDR, &s_server.sin_addr))
    {
      printf("IP network format conversion failed.\n");
      return 1;
    }
  else
    {
      printf("IP network format conversion successful.\n");
    }


  printf("Message to send: ");
  fgets(send_buf, BUFFERSIZE, stdin);

  // Sending the message to the server
  bytes_sent = sendto(sock, send_buf, sizeof(send_buf), 0, \
          (struct sockaddr*)&s_server, si_len);
  printf("%d bytes sent: %s\n", bytes_sent, send_buf);

  // Wait for the echo message
  bytes_received = recvfrom(sock, recv_buf, sizeof(recv_buf), \
          0, (struct sockaddr*)&s_server, &si_len);
  printf("%d bytes received: %s\n", bytes_received, recv_buf);

  return 0;
}
```

3. For phase2, the following C program in written in VS code for creating server at port 5000 that listens to multiple requests:

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <sys/time.h>

#define BUFFERSIZE 1024
#define PORT 5000
#define NUM_MESSAGES 10

int main() {
    int sock;
    int bind_status, bytes_received, bytes_sent;
    char send_buf[BUFFERSIZE], recv_buf[BUFFERSIZE];
    struct sockaddr_in s_server, s_client;
    int si_len = sizeof(s_client);
    struct timeval recv_time, send_time;
    int received_seq[NUM_MESSAGES] = {0};  // Track received
   sequence numbers

    // Creating the socket
    sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (sock < 0) {
        printf("Socket creation failed.\n");
        return 1;
    }
    printf("Socket created with descriptor %d\n", sock);

    // Initialising and binding the socket
    s_server.sin_family = AF_INET;
    s_server.sin_port = htons(PORT);
    s_server.sin_addr.s_addr = htonl(INADDR_ANY);

    bind_status = bind(sock, (struct sockaddr*)&s_server, sizeof(
   s_server));
    if (bind_status < 0) {
        printf("Binding socket failed.\n");
        return 1;
    }
    printf("Binding socket successful.\n");

    // Receive all messages
    for (int i = 0; i < NUM_MESSAGES; i++) {
        bytes_received = recvfrom(sock, recv_buf, sizeof(recv_buf),
    0, (struct sockaddr*)&s_client, &si_len);
        if (bytes_received < 0) {
            printf("Error receiving data\n");
            return 1;
        }

        // Get the current timestamp when data is received
        gettimeofday(&recv_time, NULL);

        // Extract sequence number
        int seq;
        sscanf(recv_buf, "%02d:", &seq);
        received_seq[seq] = 1;  // Mark sequence as received

        printf("Received from %s:%d | Seq: %d | Data: %s\n",
               inet_ntoa(s_client.sin_addr), ntohs(s_client.
    sin_port), seq, recv_buf);

        // Echo the message back
        sprintf(send_buf, "%02d:%s", seq, recv_buf + 3);
        bytes_sent = sendto(sock, send_buf, sizeof(send_buf), 0, (
    struct sockaddr*)&s_client, si_len);
        if (bytes_sent < 0) {
            printf("Error sending data\n");
            return 1;
        }

        gettimeofday(&send_time, NULL);

        printf("Echoed Seq %d back to client\n", seq);
    }

    // Check if all messages were received in order
    printf("\nOrder Check: ");
    int in_order = 1;
    for (int i = 0; i < NUM_MESSAGES; i++) {
        if (received_seq[i] == 0) {
            printf("MISSING %d ", i);
            in_order = 0;
        }
    }
    printf("\n");

    if (in_order)
        printf("All messages received in order!\n");
    else
        printf("Some messages were lost or out of order!\n");

    return 0;
}
```

4. The following C program in written in VS code for creating client to request multiple queries to the server at localhost:5000 :

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <unistd.h>

#define BUFFERSIZE 1024
#define SERVERADDR "127.0.0.1"
#define PORT 5000
#define NUM_MESSAGES 10

int main() {
    struct sockaddr_in s_server;
```

```
 int sock, bytes_received, si_len = sizeof(s_server), bytes_sent
;
 char messages[NUM_MESSAGES][BUFFERSIZE];  // To store user
input messages
 char send_buf[BUFFERSIZE], recv_buf[BUFFERSIZE];
 struct timeval send_times[NUM_MESSAGES], recv_times[
NUM_MESSAGES];
 long total_rtt = 0;

 // Creating the client socket
 sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
 if (sock < 0) {
     printf("Socket creation failed.\n");
     return 1;
 }
 printf("Socket creation successful at %d\n", sock);

 // Assigning server IP and port
 s_server.sin_family = AF_INET;
 s_server.sin_port = htons(PORT);
 if (!inet_aton(SERVERADDR, &s_server.sin_addr)) {
     printf("IP network format conversion failed.\n");
     return 1;
 }
 printf("IP network format conversion successful.\n");

 // Prompts for messages to send
 for (int i = 0; i < NUM_MESSAGES; i++) {
     printf("Enter message %d: ", i + 1);
     fgets(messages[i], BUFFERSIZE - 10, stdin);
     messages[i][strcspn(messages[i], "\n")] = 0;
 }

 // Send the messages to server
 for (int i = 0; i < NUM_MESSAGES; i++) {
     snprintf(send_buf, BUFFERSIZE, "%02d:ClientX:%.1000s", i,
messages[i]);  // Add header
     gettimeofday(&send_times[i], NULL);  // Timestamp before
sending

     bytes_sent = sendto(sock, send_buf, strlen(send_buf) + 1,
0, (struct sockaddr*)&s_server, si_len);
     if (bytes_sent < 0) {
         printf("Error sending data\n");
         return 1;
     }
     printf("Sent (%d): %s\n", i, send_buf);
 }

 // Receive responses and compute RTT
 for (int i = 0; i < NUM_MESSAGES; i++) {
     bytes_received = recvfrom(sock, recv_buf, sizeof(recv_buf),
 0, (struct sockaddr*)&s_server, &si_len);
     if (bytes_received < 0) {
         printf("Error receiving data\n");
         return 1;
     }
     gettimeofday(&recv_times[i], NULL);  // Timestamp after
```

```
    receiving

        // Extract sequence number
        int seq;
        sscanf(recv_buf, "%02d:", &seq);
        printf("Received (%d): %s\n", seq, recv_buf);

        // Calculate RTT
        long rtt_sec = recv_times[i].tv_sec - send_times[seq].
    tv_sec;
        long rtt_usec = recv_times[i].tv_usec - send_times[seq].
    tv_usec;
        if (rtt_usec < 0) {
            rtt_sec--;
            rtt_usec += 1000000;
        }
        long rtt = rtt_sec * 1000000 + rtt_usec;
        total_rtt += rtt;

        printf("RTT for message %d: %ld microseconds\n", seq, rtt);
    }

    printf("Average RTT: %ld microseconds\n", total_rtt /
    NUM_MESSAGES);

    return 0;
}
```

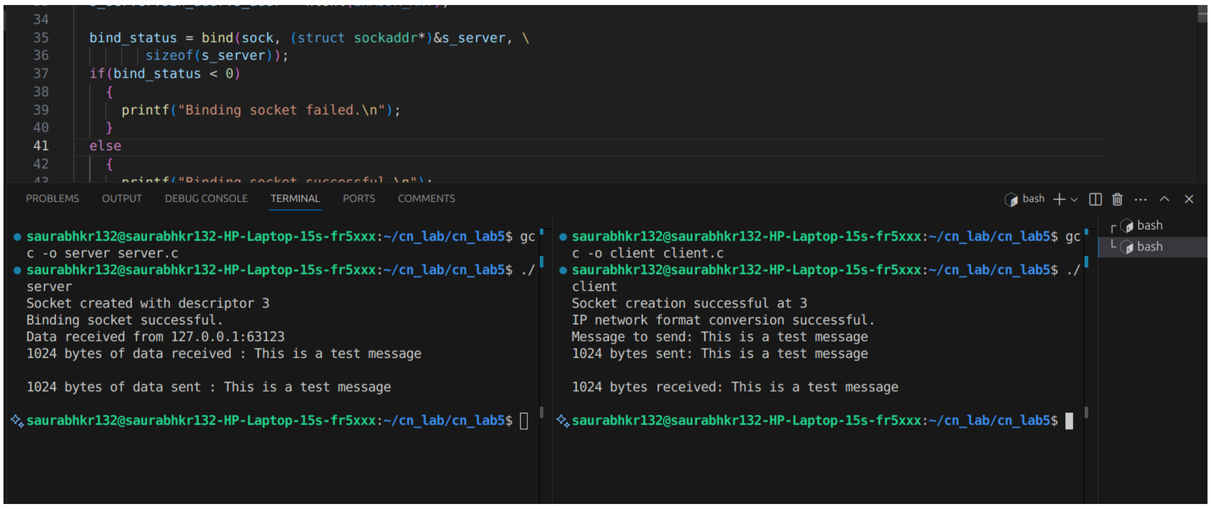
5. Compiling of code is done by the command (file must be in the current working directory):

   ```
   gcc -o output_file_name file_to_compile.c
   ```

6. The compiled binary file can be run be the command:

   ```
   ./output_file_name
   ```

Observations

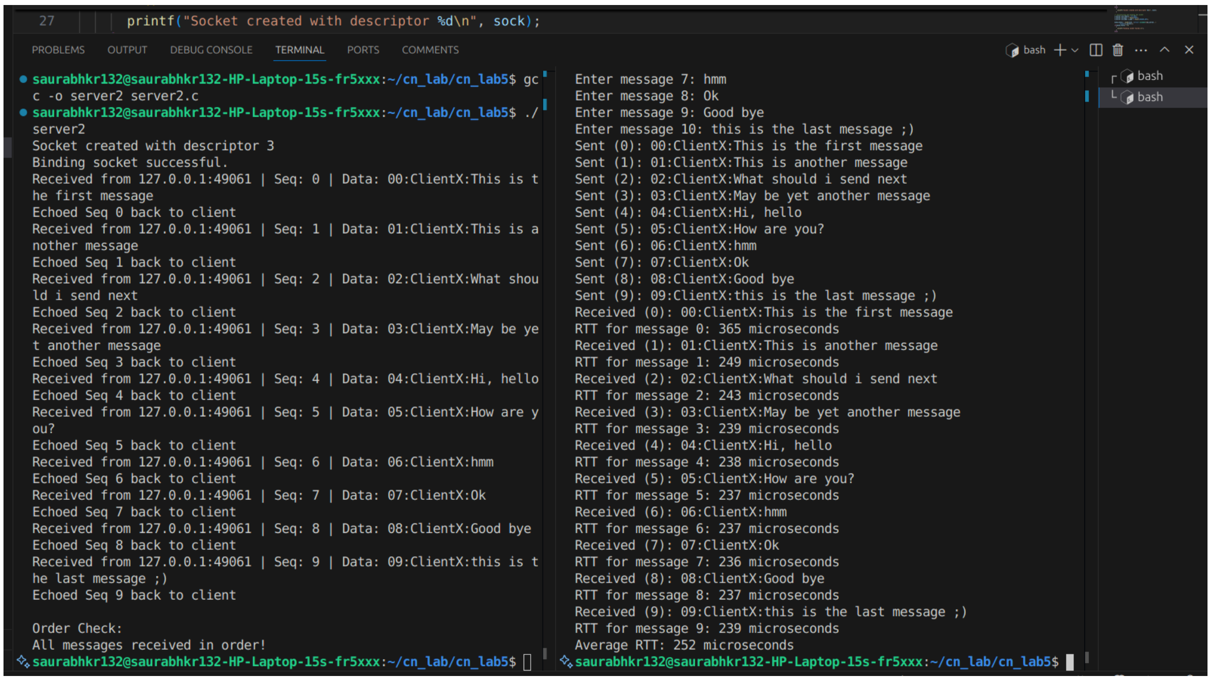


```c
    bind_status = bind(sock, (struct sockaddr*)&s_server, \
                sizeof(s_server));
    if(bind_status < 0)
      {
        printf("Binding socket failed.\n");
      }
    else
      {
```

```
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ gc
c -o server server.c
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ ./
server
Socket created with descriptor 3
Binding socket successful.
Data received from 127.0.0.1:63123
1024 bytes of data received : This is a test message

1024 bytes of data sent : This is a test message

saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$
```

```
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ gc
c -o client client.c
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ ./
client
Socket creation successful at 3
IP network format conversion successful.
Message to send: This is a test message
1024 bytes sent: This is a test message

1024 bytes received: This is a test message

saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$
```

Figure 1: Server-client UDP Communication



```c
        printf("Socket created with descriptor %d\n", sock);
```

```
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ gc
c -o server2 server2.c
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$ ./
server2
Socket created with descriptor 3
Binding socket successful.
Received from 127.0.0.1:49061 | Seq: 0 | Data: 00:ClientX:This is t
he first message
Echoed Seq 0 back to client
Received from 127.0.0.1:49061 | Seq: 1 | Data: 01:ClientX:This is a
nother message
Echoed Seq 1 back to client
Received from 127.0.0.1:49061 | Seq: 2 | Data: 02:ClientX:What shou
ld i send next
Echoed Seq 2 back to client
Received from 127.0.0.1:49061 | Seq: 3 | Data: 03:ClientX:May be ye
t another message
Echoed Seq 3 back to client
Received from 127.0.0.1:49061 | Seq: 4 | Data: 04:ClientX:Hi, hello
Echoed Seq 4 back to client
Received from 127.0.0.1:49061 | Seq: 5 | Data: 05:ClientX:How are y
ou?
Echoed Seq 5 back to client
Received from 127.0.0.1:49061 | Seq: 6 | Data: 06:ClientX:hmm
Echoed Seq 6 back to client
Received from 127.0.0.1:49061 | Seq: 7 | Data: 07:ClientX:Ok
Echoed Seq 7 back to client
Received from 127.0.0.1:49061 | Seq: 8 | Data: 08:ClientX:Good bye
Echoed Seq 8 back to client
Received from 127.0.0.1:49061 | Seq: 9 | Data: 09:ClientX:this is t
he last message ;)
Echoed Seq 9 back to client

Order Check:
All messages received in order!
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$
```

```
Enter message 7: hmm
Enter message 8: Ok
Enter message 9: Good bye
Enter message 10: this is the last message ;)
Sent (0): 00:ClientX:This is the first message
Sent (1): 01:ClientX:This is another message
Sent (2): 02:ClientX:What should i send next
Sent (3): 03:ClientX:May be yet another message
Sent (4): 04:ClientX:Hi, hello
Sent (5): 05:ClientX:How are you?
Sent (6): 06:ClientX:hmm
Sent (7): 07:ClientX:Ok
Sent (8): 08:ClientX:Good bye
Sent (9): 09:ClientX:this is the last message ;)
Received (0): 00:ClientX:This is the first message
RTT for message 0: 365 microseconds
Received (1): 01:ClientX:This is another message
RTT for message 1: 249 microseconds
Received (2): 02:ClientX:What should i send next
RTT for message 2: 243 microseconds
Received (3): 03:ClientX:May be yet another message
RTT for message 3: 239 microseconds
Received (4): 04:ClientX:Hi, hello
RTT for message 4: 238 microseconds
Received (5): 05:ClientX:How are you?
RTT for message 5: 237 microseconds
Received (6): 06:ClientX:hmm
RTT for message 6: 237 microseconds
Received (7): 07:ClientX:Ok
RTT for message 7: 236 microseconds
Received (8): 08:ClientX:Good bye
RTT for message 8: 237 microseconds
Received (9): 09:ClientX:this is the last message ;)
RTT for message 9: 239 microseconds
Average RTT: 252 microseconds
saurabhkr132@saurabhkr132-HP-Laptop-15s-fr5xxx:~/cn_lab/cn_lab5$
```

Figure 2: Server-client UDP Communication

Conclusions

- In the first phase, the a basic server and a client are created using libraries in which the client sends a request to the server and the server replies back with the same message.
- In the second phase, we implemented a ping-like utility to measure the Round-Trip Time (RTT) between a client and a server. The client sends 10 uniquely identified messages with a sequence number, sender name, and user data to the server, which echoes them back. By calculating the RTT for each packet and averaging the re-

sults, we analyzed network latency and packet reliability.

Since UDP is an unreliable transport protocol, messages may arrive out of order or get lost. However, in our controlled environment, packets were received in order, demonstrating stable network conditions. This experiment highlights the importance of sequence numbering and timestamping in UDP-based communications to ensure message tracking and performance evaluation.