CS378 Lab 5 Report

Ekansh Ravi Shankar, Ved Danait

October 2024

Part 1

We need to prove that $\frac{P}{t_2-t_1} = C$, where P is the size of the packet in bits, t_1 is the time of arrival of the first bit at D, and t_2 is the time of arrival of the second bit at D, and C is the bottleneck link speed.

This can be proved by induction.

For the base case, there is only one link, with speed C_1 bits/sec, which is also the bottleneck speed, that is $C_1 = C$. The second packet will start being sent once the first one has completely reached the destination, and the time taken for the second packet to reach is $\frac{P}{C_1} = \frac{P}{C} = t_2 - t_1$. Hence, $\frac{P}{t_2-t_1} = C$.

Now, we assume that this is true for the first k links. That is, at router R_k, $\frac{P}{t_2-t_1} = C$, where C is the bottleneck link speed of the first k links.

We now need to prove that this is also true for $k + 1$ links.

1. The $(k+1)$th link is not the bottleneck link. In this case, at R_k, $t_2-t_1 = \frac{P}{C}$, and since the $(k+1)$th link is not a bottleneck, the first packet will reach R_{k+1}, before the second packet reaches R_k, since $\frac{P}{C_{k+1}} <= t_2 - t_1$ at R_k. Thus, $t_2 - t_1 = \frac{P}{C}$ at R_{k+1}, and hence, $\frac{P}{t_2-t_1} = C$, where C is the bottleneck link speed of the first $k + 1$ links.

2. The $(k+1)$th link is a bottleneck link. In this case, the second packet will arrive at R_k before the first packet reaches R_{k+1}, since $\frac{P}{C_{k+1}} > t_2 - t_1$, at R_k. Hence, the second packet starts getting transmitted only after the first packet reaches R_{k+1}. Now, time taken for the second packet to reach R_{k+1} is $\frac{P}{C_{k+1}} = t_2 - t_1$ at R_{k+1}, and now $C_{k+1} = C$ that is, the bottleneck link speed. Thus, $\frac{P}{t_2-t_1} = C$.

Part 2

(a) The following lines create datagram sockets

```
sockfd = socket(AF_INET, SOCK_DGRAM, 0); // AF_INET = IPv4, SOCK_DGRAM = UDP
if (sockfd < 0) {
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}

```

In this part, the socket function takes parameters **AF_INET**, which indicates that IPv4 has to be used, **SOCK_DGRAM** initiates a datagram socket for UDP, and **0** is the UDP protocol number.

```
if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Bind failed");
    return 1;
}

```

This code in the receiver is used to bind the socket to the appropriate port. **sockfd** is the socket file descriptor, and **server_addr** is used to store the address of the sender (which has been set to accept anything), type of IP, and port to connect to.

(b) The following lines send/write data to the socket

```
send_time = get_current_time();
if (sendto(sockfd, packet, packet_size, 0, (struct sockaddr *)&receiver_addr, sizeof(receiver_addr)) < 0) {
    perror("sendto failed");
    exit(EXIT_FAILURE);
}

```

Here, **sockfd** is the file descriptor for the socket from part a), **packet** is a pointer to where the data to be sent has been allocated, **packet_size** is the size of the packet sent in bytes, **0** indicates that no special flags are being used, **receiver_addr** is the address of the receiver which contains information such as destination IP address, port number, and type of IP used, and **sizeof(receiver_addr)** is the size of the address, to specify how many bytes to read. If the function returns < 0, then it has failed to send the data.

(c) The following lines read data from the socket

```
int n = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&sender_addr, &addr_len);
if (n < 0) {
    perror("recvfrom failed");
    break;
}

```

Here, **sockfd** is the file descriptor for the socket from part a), **buffer** is a pointer to where the data should be received, **sizeof(buffer)** is the maximum size allocated which can be received, **0** indicates that no special flags are being used, **sender_addr** is the address of the sender which contains information such as port number, and type of IP used, and **addr_len** is the size of the sender address, which will be updated to the actual size. If the function returns < 0, then it has failed to receive the data.

(d) The following lines measure the time at which the packets have arrived

```
    long get_current_time() {
        struct timeval time_now;
        gettimeofday(&time_now, NULL);
        return (time_now.tv_sec * 1000000) + time_now.tv_usec;
    }
    int n = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&sender_addr, &addr_len);
    if (n < 0) {
        perror("recvfrom failed");
        break;
    }

    recv_time_1 = get_current_time();
    packet_count++;

    // Receive the second packet of the pair
    n = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr *)&sender_addr, &addr_len);
    if (n < 0) {
        perror("recvfrom failed");
        break;
    }

    recv_time_2 = get_current_time();
    packet_count++;

```

get_current_time is a function, which initialises a timeval struct, which gets the time of the day, using gettimeofday, and **tv_sec** gives the current second and **tv_usec** gives the current microsecond. This has been used to measure the current time before the 2nd packet has been received, and after the 2nd packet has been received.

(e) The following lines show how the code sends two packets back to back.

```
    for (int i = 1; i <= total_pairs * 2; i++) {
        memset(packet, 0, packet_size);
        sprintf(packet, "Packet %d", i%(2*total_pairs));  // Include packet number in the data

        // (b) Send/write data to the socket
        send_time = get_current_time();
        if (sendto(sockfd, packet, packet_size, 0, (struct sockaddr *)&receiver_addr, sizeof(receiver_addr)) < 0) {
```

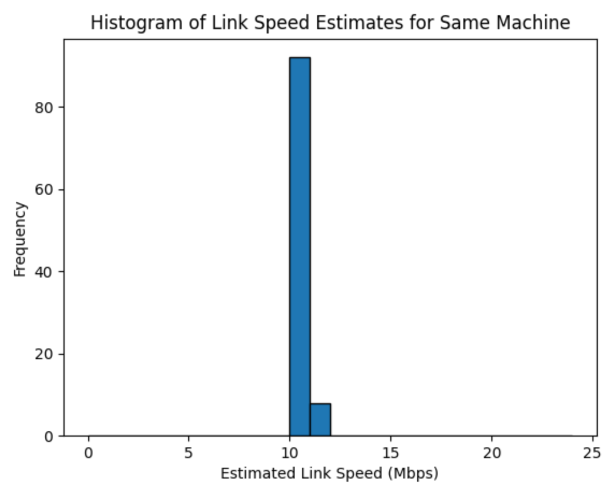
```
            perror("sendto failed");
            exit(EXIT_FAILURE);
        }

        // For odd packets (first in a pair), send the next packet immediately
        if (i % 2 == 0) {
            usleep(spacing_ms * 1000); // Wait between pairs
        }
    }

```

Here, we can see that the program sleeps every alternate packet sent, which means that two packets have been sent one after another.

Part 3

Experiment-1

The histogram of the links speed estimates after running the code on the same machine is as follows

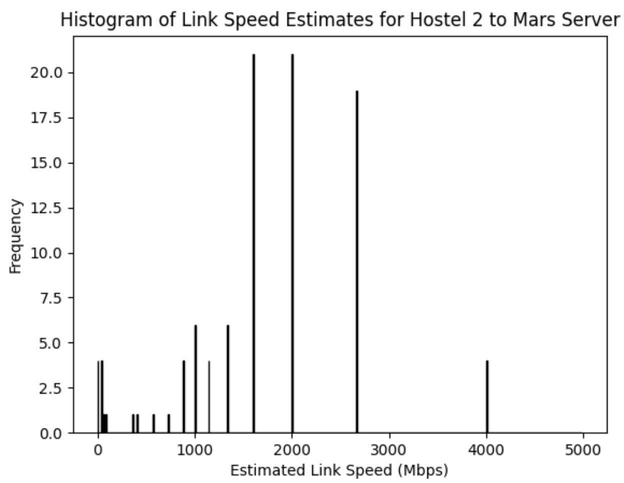


We see a clear spike in regions 10-11 Mbps and 11-12 Mbps. This allows us to infer that the bottleneck link speed will be close to 10Mbps, which had been set.

Experiment-2

Path 1 - Hostel 2 to Mars Server

The first path chosen was personal laptop at Hostel 2 to the Mars Server at CC. The traceroute output is given below
traceroute to 10.129.3.5 (10.129.3.5), 30 hops max, 60 byte packets
1 LAPTOP-ONUMCLUI.mshome.net (172.18.128.1) 0.536 ms 0.489 ms 0.472 ms
2 10.61.0.1 (10.61.0.1) 2.708 ms 5.408 ms 5.398 ms
3 172.16.12.2 (172.16.12.2) 5.383 ms 5.374 ms 5.365 ms
4 10.250.129.2 (10.250.129.2) 16.213 ms 14.894 ms 16.195 ms
5 mars.cse.iitb.ac.in (10.129.3.5) 5.331 ms 7.997 ms 7.988 ms
The histogram is as follows

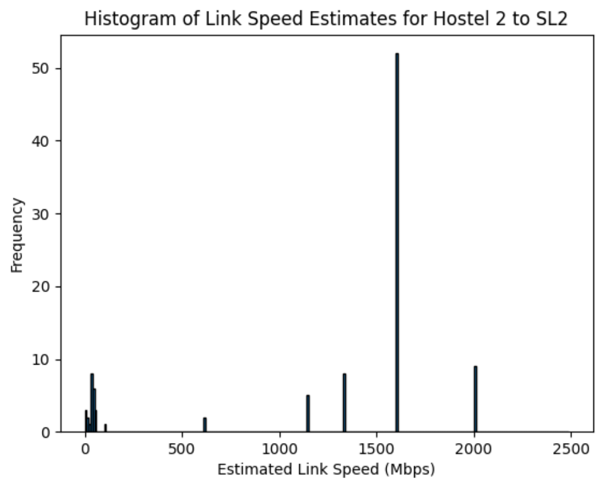


From this, we can see that the highest peaks are in the bins 1600-1610 and 2000-2010, and from the data, the actual modes are 1600Mbps and 2000Mbps.

Path 2 - Hostel 2 to SL2

The second path chosen was personal laptop at Hostel 2 to the SL2 lab at CC. The traceroute output is given below
traceroute to 10.130.154.9 (10.130.154.9), 30 hops max, 60 byte packets
1 LAPTOP-ONUMCLUI.mshome.net (172.18.128.1) 0.497 ms 0.469 ms 0.456 ms
2 10.61.0.1 (10.61.0.1) 27.201 ms 27.190 ms 27.179 ms
3 172.16.12.2 (172.16.12.2) 26.585 ms 26.575 ms 27.060 ms
4 10.250.130.2 (10.250.130.2) 27.625 ms 28.277 ms 27.605 ms

5 10.130.154.9 (10.130.154.9) 27.042 ms 27.031 ms 27.020 ms
The histogram is as follows



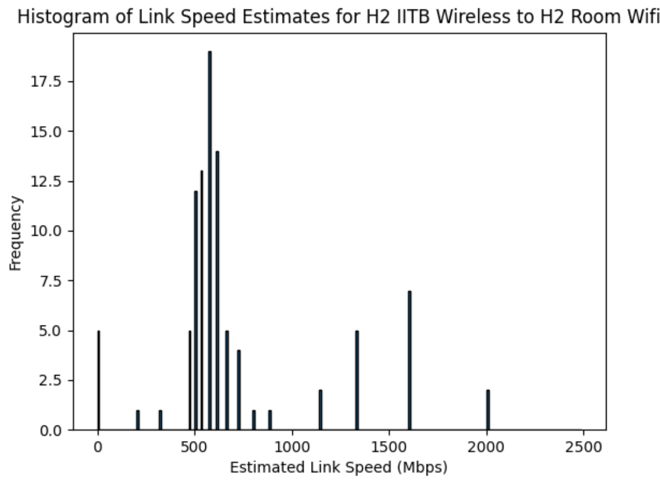
From this, we can see that the highest peak is in the bin 1600-1610 , and from the data, the actual mode is 1600 Mbps.

Path 3 - Hostel 2 IITB Wireless to Hostel 2 Room Wifi

The third path chosen was personal laptop at Hostel 2 connected to IITB Wireless and another personal laptop at Hostel 2 connected to room wifi. The traceroute output is given below
traceroute to 10.64.91.247 (10.64.91.247), 30 hops max, 60 byte packets
1 Ekansh.mshome.net (172.31.176.1) 0.907 ms 0.838 ms 0.818 ms
2 * 192.168.0.1 (192.168.0.1) 43.364 ms 43.352 ms
3 * 10.2.100.250 (10.2.100.250) 17.673 ms *
4 10.250.2.1 (10.250.2.1) 19.350 ms * *
5 172.16.6.2 (172.16.6.2) 208.569 ms 208.556 ms 208.540 ms
6 172.16.5.1 (172.16.5.1) 208.601 ms 204.341 ms 204.298 ms
7 172.16.12.1 (172.16.12.1) 204.280 ms * *
8 10.64.91.247 (10.64.91.247) 165.017 ms 164.975 ms 30.932 ms

The histogram is in the next page.

The histogram is as follows



From this, we can see that the highest peak is in the bin 570-580, and from the data, the actual mode is 571.428571 Mbps.

Thus, by looking at histograms, one can find clear peaks and try to estimate the value for the bottleneck link speed. This can especially be seen in path 1 and path 2, where the estimates are similar.