CS378 Lab-5 Report

Saksham Rathi (22B1003)
Department of Computer Science, IIT Bombay

Nandan Manjunath (22B0920)
Department of Computer Science, IIT Bombay

Part 1: (Theory)

Let us prove this by induction.

Case1 : number of links(k) = 1

Packet 1 is ahead and reaches $t_1 = P/C_1$ where C_1 is the link's bandwidth. Packet 2 waits until Packet 1 is completely sent and then is sent. It takes time $P/C_1 + P/C_1$, which is $t_2 = 2 * P/C_1$.

In this case, $P/(t_2 - t_1)$ is C_1, which is the correct bottleneck bandwidth. Therefore, the base case is true.

Case2 : True for number of links(k) then true for links(k+1)

Until it covers k links, let the time they take be t_1 and t_2, and bottleneck bandwidth be C which is $P/(t_2 - t_1)$ (induction assumption). This bandwidth C is the minimum among all links k. Let $k + 1^{th}$ link bandwidth be D.

Case2.1: If D is greater than or equal to C

Now new t_1' will become $t_1 + P/D$ and as $D >= C$, t_1' will be less than t_2 therefore, by the time complete Packet 2 crosses link k, complete Packet 1 will cross $k + 1^{th}$ link also. So, Packet 2 doesn't wait, and the new t_2' will be $t_2 + P/D$. In this case, $P/(t_2' - t_1')$ is C, which matches the bottleneck bandwidth.

Case2.2: Id D is less than C

Then, the bottleneck bandwidth will become D. The new t_1' is $t_1 + P/D$. and as $D < C$, t_1' will be more than t_2 therefore Packet2 has to wait until Packet 1 is completely sent across $k + 1^{th}$ link. This waiting time is $t_1' - t_2$, and then it is sent. The new t_2' is t_2+waiting time $+P/D$ which is $t_2 + t_1' - t_2 + P/D$. In this case, $P/(t_2' - t_1')$ is D, which matches the bottleneck bandwidth.

By induction, we proved that bottleneck bandwidth can be calculated as $P/(t_2 - t_1)$

Part 2: (Implementation)

(a) Here is the code for creating datagram sockets at the sender side:

```
int sockfd;
struct sockaddr_in dest_addr;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);  // UDP socket
if (sockfd == -1) {
    // Socket creation failed
    perror("Socket creation failed");
    return -1;
}
// Set up the destination address structure
dest_addr.sin_family = AF_INET;
dest_addr.sin_port = htons(8080);  // Set an appropriate port
dest_addr.sin_addr.s_addr = inet_addr(dest_ip);
```

Listing 1: Sender Datagram Socket Creation

The first line declares an integer variable sockfd which will store the file descriptor for the socket once it is created. The second line declares a variable *dest_addr* of type struct *sockaddr_in*. This structure is used to specify the destination address for the socket communication, including details like the IP address and port number. The third line creates a new socket by calling the socket() function. This function takes three arguments.

- The first argument specifies the address family, which is AF_INET in this case. This indicates that we are using the IPv4 protocol.
- The second argument specifies the type of socket, which is SOCK_DGRAM. This indicates that we are creating a datagram socket, which is used for connectionless communication.
- The third argument is the protocol, which is 0 in this case. This allows the system to choose the appropriate protocol based on the type of socket and the address family.

If the socket creation fails, the function prints an error message using perror() and returns -1. The tenth line sets the address family of the destination to AF_INET, which means it is using the IPv4 protocol. The second last line sets the destination port number for the socket. The htons() function is used to convert the port number (8080 in this case) from host byte order (which varies by machine architecture) to network byte order (which is big-endian). The last line sets the IP address of the destination in the *sin_addr.s_addr* field. The *inet_addr*() function converts a human-readable IPv4 address (passed as *dest_ip*) into a format suitable for storing in the *sin_addr.s_addr* field.

Here is the code for creating datagram sockets at the receiver side:

```
int sockfd;
struct sockaddr_in server_addr, client_addr;
char buffer[BUFFER_SIZE];
socklen_t addr_len = sizeof(client_addr);
sockfd = socket(AF_INET, SOCK_DGRAM, 0);  // UDP socket
if (sockfd == -1) {
    // If the socket creation fails, print an error message and
        return -1
    perror("Socket creation failed");
    fclose(fp);
    return -1;
```

```
    }
    // Set up the server address structure
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(8080); // Listening on port number
        8080
    server_addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sockfd, (struct sockaddr*)&server_addr, sizeof(
       server_addr)) == -1) {
        // If the bind fails, print an error message, close the
           socket, close the file, and return -1
        perror("Bind failed");
        close(sockfd);
        fclose(fp);
        return -1;
    }
```

Listing 2: Sender Datagram Socket Creation

The third line is the buffer, where we will receive inputs fron the sender. The fourth line is the length of the client address structure. The fifth line is similar to the sender side's code. The tenth line sets the IP address of the server to INADDR_ANY, which means the server will listen on all available network interfaces. The eleventh line binds the socket to the server address. The bind() function associates the socket with the server address so that it can receive packets sent to that address. If the bind fails, the function prints an error message using perror(), closes the socket using close(), closes the file using fclose(), and returns -1. The bind() function takes three arguments.

- The first argument is the socket file descriptor.
- The second argument is a pointer to the server address structure cast to a struct sockaddr pointer.
- The third argument is the size of the server address structure.

(b) Here is the code for sending data to the socket:

```
sendto(sockfd, packet1, sizeof(packet1), 0, (struct sockaddr*)&
   dest_addr, sizeof(dest_addr));
```

Listing 3: Sending Data to the Socket

The sendto() function is used to send data to a socket. It takes six arguments.

- The first argument is the socket file descriptor.
- The second argument is a pointer to the data to be sent.
- The third argument is the size of the data to be sent.
- The fourth argument is a set of flags, which is 0 in this case.
- The fifth argument is a pointer to the destination address structure cast to a struct sockaddr pointer.
- The sixth argument is the size of the destination address structure.

We are sending an initial packet which contains the size of the packets (P) which is used in bandwidth calculation. And an ending packet helps in the termination of the receiver side code.

(c) Here is the code for receiving data from the socket:

```
recvfrom(sockfd, buffer, BUFFER_SIZE, 0, (struct sockaddr*)&
    client_addr, &addr_len);
```

Listing 4: Receiving Data from the Socket

The recvfrom() function is used to receive data from a socket. It takes six arguments.

- The first argument is the socket file descriptor.
- The second argument is a pointer to a buffer where the received data will be stored.
- The third argument is the size of the buffer.
- The fourth argument is a set of flags, which is 0 in this case.
- The fifth argument is a pointer to the client address structure cast to a struct sockaddr pointer.
- The sixth argument is a pointer to the length of the client address structure.

(d) Measuring the time interval of packets:

```
gettimeofday(&t2, NULL); // Get time in microseconds
double delta_t = time_diff(t1, t2);  // Time difference in
    microseconds
double C = (packet_size*8) / delta_t;  // packet_size in bytes,
     so converting to bits
```

Listing 5: Measuring the time interval of packets

The function gettimeofday() is used to get the current time in microseconds. It takes two arguments.

- The first argument is a pointer to a timeval structure where the time will be stored.
- The second argument is NULL, which means the time will be based on the system clock.

The function time_diff() calculates the difference between two timeval structures in microseconds. The variable *delta_t* stores the time difference between the two packets in microseconds. The variable *C* calculates the bandwidth in bits per second by dividing the packet size (in bytes) by the time difference (in seconds).

(e) Sending two packets one after the other:

```
char packet1[P], packet2[P];
// Sending the first packet
sendto(sockfd, packet1, sizeof(packet1), 0, (struct sockaddr*)&
    dest_addr, sizeof(dest_addr));
// Sending the second packet
sendto(sockfd, packet2, sizeof(packet2), 0, (struct sockaddr*)&
    dest_addr, sizeof(dest_addr));
// Sleep for the given spacing time
usleep(spacing_ms * 1000);
```

Listing 6: Sending two packets one after the other

In sender.c, we are sending the packets one after the other (there is no delay in between). So, as soon as the first packet is sent, the code prepares the second packet and sends it immediately. After this pair is sent, a usleep statement is added before the next pair is sent.

Part 3: Experimentation - Experiment-1

We restricted the throughput to 10Mbps through the following command:

```
sudo tc qdisc add dev lo root tbf rate 10mbit burst 9kbit latency
   50ms
```

Listing 7: Restricting the throughput to 10Mbps

The sender was run with the following paramters:

- Packet size (P) = 1000 bytes = 8000 bits
- Destination IP Address = 127.0.0.1
- Spacing time = 100ms
- Total number of pairs of packets = 100

The receiver was set to produce the bandwidth values of the 100 packets in the ”part3-e1-buffer” output file.

Here is the histogram which we have obtained:

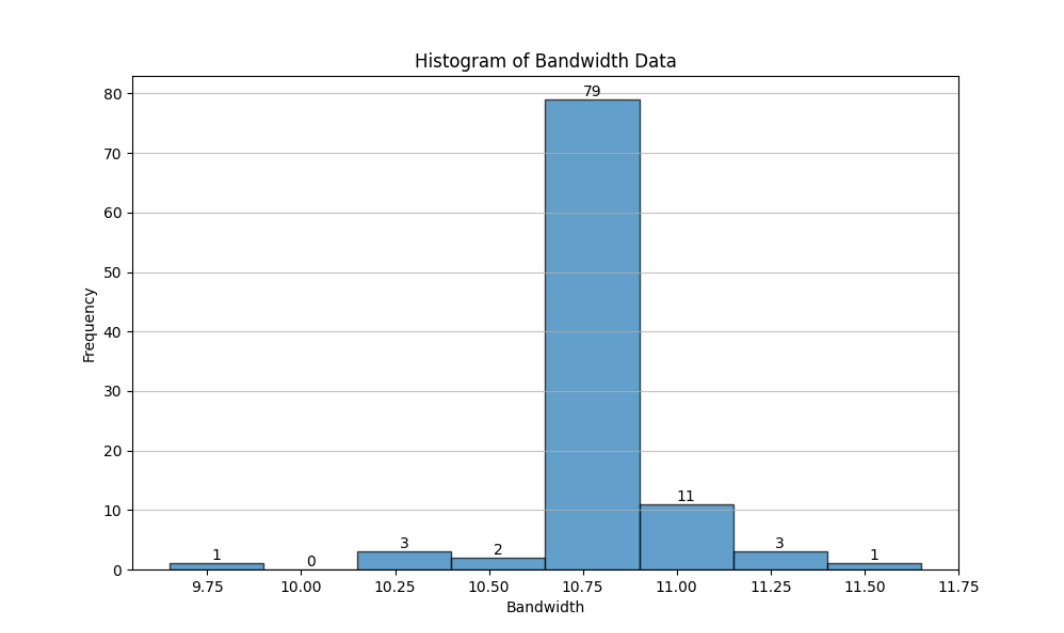

Figure 1: Histogram of Bandwidth values

The bin size was set to be 0.25 Mbps. Yes, someone looking at the histogram can easily say that the bottleneck was set to be 10 Mbps because all the values are quite close to 10 Mbps. If we would have removed this protocol constraint, then histogram would have been spread out.

A peak of the histogram occurs around 10.75 Mbps (which is a little higher than the limit set by us). We have some values to the left and right of this peak too.

Part 3: Experimentation - Experiment-2

This time we didn't impose any restrictions on the throughput for any interface on either machine. Here are the three paths and the corresponding histograms:

- Path 1: Hostel-6 (sender) and Hostel-3 (receiver)

  Traceoutput:

```
traceroute 10.64.17.94
traceroute to 10.64.17.94 (10.64.17.94), 30 hops max, 60 byte packets
1  Nandan.mshome.net (172.25.80.1)  0.754 ms  0.706 ms  0.695 ms
2  dlinkrouter (192.168.0.1)  6.189 ms  5.637 ms  5.108 ms
3  10.6.160.250 (10.6.160.250)  20.349 ms  20.276 ms  20.259 ms
4  10.250.6.1 (10.250.6.1)  20.251 ms  20.466 ms  20.236 ms
5  172.16.2.1 (172.16.2.1)  20.219 ms  20.209 ms  20.154 ms
6  172.16.12.1 (172.16.12.1)  20.181 ms  18.069 ms  17.572 ms
7  10.64.17.94 (10.64.17.94)  45.818 ms  45.210 ms  45.464 ms
```

  Number of hops = 7

  Here is the histogram for this path:

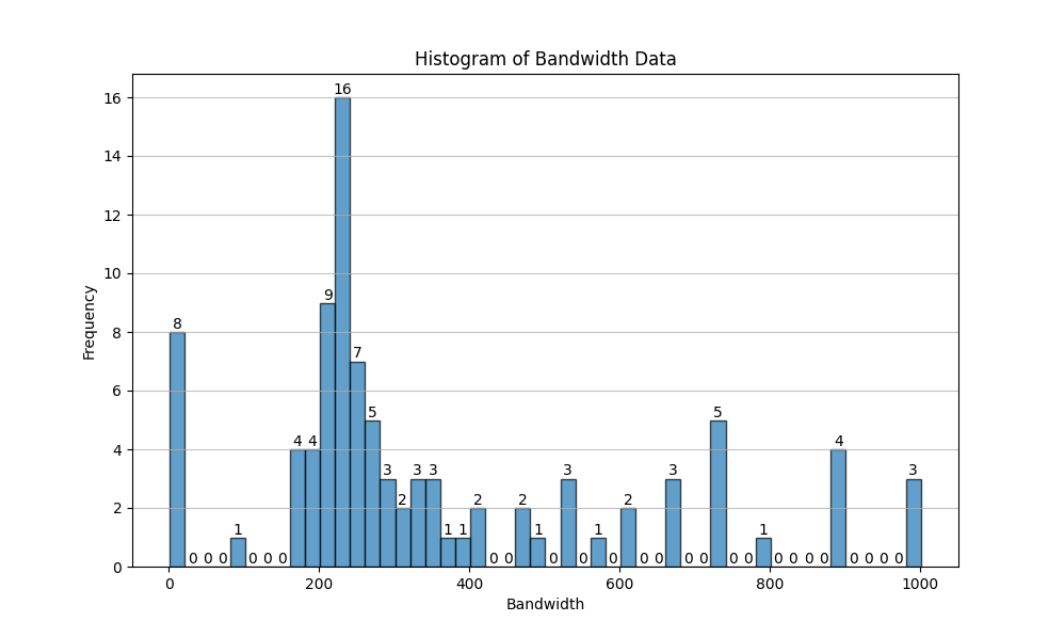

Figure 2: Histogram of Bandwidth values for Path 1

- Path 2: Hostel-5 Ethernet (sender) and SL-Machine (receiver)

  Traceoutput:

```
1  192.168.0.1 (192.168.0.1)  2.202 ms  1.924 ms  1.887 ms
2  10.5.10.250 (10.5.10.250)  2.133 ms  2.211 ms  2.287 ms
3  10.250.5.1 (10.250.5.1)  2.810 ms  3.001 ms  5.935 ms
4  10.250.18.1 (10.250.18.1)  8.537 ms  2.646 ms  3.284 ms
5  172.16.2.1 (172.16.2.1)  2.286 ms  2.286 ms  2.043 ms
```

```
6  10.250.130.2 (10.250.130.2)  2.627 ms  3.639 ms  3.504 ms
7  10.130.155.33 (10.130.155.33)  3.026 ms  3.279 ms  2.985 ms
```

Number of hops = 7

Here is the histogram for this path:

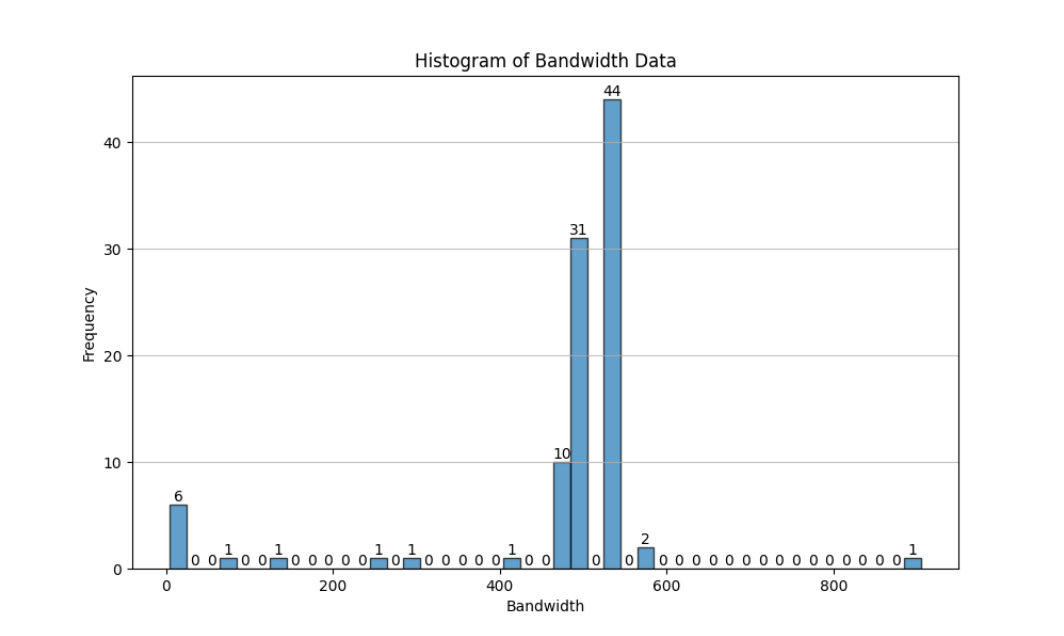

Figure 3: Histogram of Bandwidth values for Path 2

- Path 3: Hostel-5 (sender) and Hostel-5 (receiver) (different rooms in the same hostel) Traceoutput:

```
traceroute 10.64.10.66
traceroute to 10.64.10.66 (10.64.10.66), 30 hops max, 60 byte packets
1  Nandan.mshome.net (172.25.80.1)  0.576 ms  0.508 ms *
2  * 192.168.1.1 (192.168.1.1)  6.254 ms  6.244 ms
3  * * *
4  10.250.5.1 (10.250.5.1)  6.111 ms  6.105 ms  6.052 ms
5  10.250.18.1 (10.250.18.1)  6.644 ms  6.639 ms *
6  * 172.16.2.1 (172.16.2.1)  41.568 ms  37.670 ms
7  172.16.12.1 (172.16.12.1)  41.184 ms  41.168 ms  41.161 ms
8  10.64.10.66 (10.64.10.66)  134.785 ms  134.778 ms  134.770 ms
```

Number of hops = 8

Here is the histogram for this path:

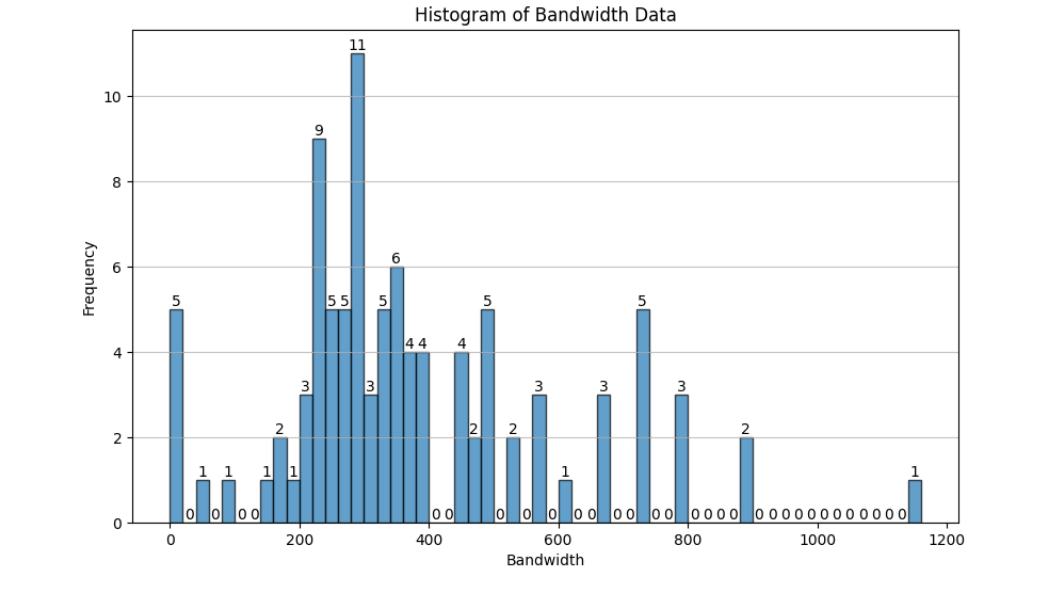

Figure 4: Histogram of Bandwidth values for Path 3

- Path 4: SL-Machine (New CC) (sender) and Hostel-5 (Eduroam) (receiver)
  Traceoutput:

```
traceroute to 10.64.10.66 (10.64.10.66), 30 hops max, 60 byte packets
1  _gateway (10.130.154.250)  1.023 ms  1.028 ms  1.285 ms
2  10.250.130.1 (10.250.130.1)  0.325 ms  0.303 ms  0.283 ms
3  172.16.12.1 (172.16.12.1)  0.435 ms  0.415 ms  0.395 ms
4  10.64.10.66 (10.64.10.66)  18.087 ms  18.065 ms  18.181 ms
```

  Number of hops = 4
  Here is the histogram for this path:

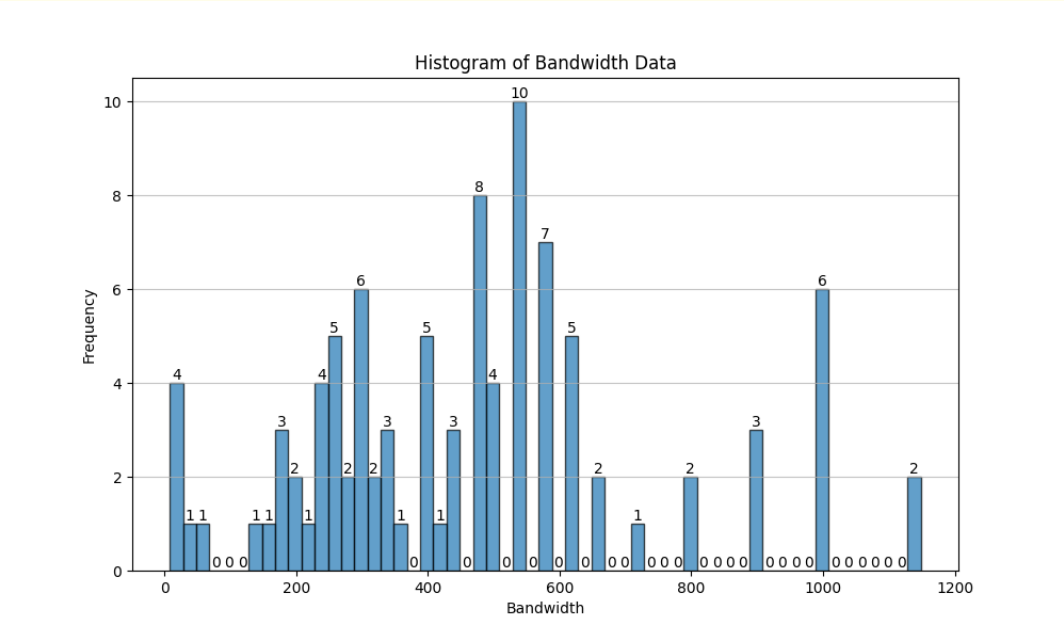

Figure 5: Histogram of Bandwidth values for Path 4

Observations:

- Based on the histogram, it is difficult to say what is the bottleneck bandwidth. The values are spread out and there is no clear peak.
- Path-2 has less number of hops, so its average bandwidth is higher than the other two paths, which is expected.