CS378: Computer Networks Laboratory

Lab 5: Building a Bottleneck Bandwidth estimation tool using Datagram Sockets

Ananya Rao (22b0980), Hari Bhavasar (22b0934)

October 2024

Part 1: Theory

To Prove

Given S,R_1,R_2,..R_n,D and (n+1) links between them such that any packet enters router R_i on link i and exits it at link i+1, with the bitrate at i^{th} link being C_i bits/second.
S sends two packets of size P bits each with t_1 being the time of arrival of the last bit of the first packet at D and t_2 being the time of arrival of the last bit of the second packet at D.
We have to prove that $\frac{P}{t_2-t_1} = C$ where C is the bottleneck link speed, that is, C = min(C_i) over all i

Base Case

For n = 0, there is only one link from S to D, with bitrate being C. Let us say

- t_1 = time of arrival of last bit of first packet at D
- t_2 = time of arrival of last bit of second packet at D

Since t_1 is the time of arrival of the the last bit of the first packet at D, and the first bit of the second packet will arrive at D immediately after the last bit of the first packet does, we can say t_1 is the arrival of the first bit of the second packet at D. Thus, the first bit of the second packet arrives at D at time t_1 and the last bit of the second packet arrives at D at time t_2, thus the second packet (of size P bits), took (t_2 - t_1) seconds to travel the link. (The number of bits in packet)/(time taken by packet to travel the link) = bitrate of link and thus $\frac{P}{t_2-t_1} = C$

Induction

Assume for a sub-link of length k or less, if t_1 is the time of arrival of the last bit of the first packet at R_k and t_2 is the time of arrival of the last bit of the second packet at R_k then the bottleneck link speed is given by $\frac{P}{t_2-t_1}$
Now assume that $t_1^{'}$ is the time of arrival of the last bit of the first packet at R_{k+1} and $t_2^{'}$ is the time of arrival of the last bit of arrival of the last bit of the second packet at R_{k+1}
There are two cases:

$C_{k+1} > C$

In this case the bottleneck in the (k+1) links will be C (as $C < C_{k+1}$). At time t_1, the last bit of the first packet has reached R_k, thus, $t_1^{'} = t_1 + \frac{P}{C_{k+1}}$
Similarly, $t_2^{'} = t_2 + \frac{P}{C_{k+1}}$ Now, $\frac{P}{t_2^{'}-t_1^{'}} = \frac{P}{(t_2+\frac{P}{C_{k+1}})-(t_1+\frac{P}{C_{k+1}})} = \frac{P}{t_2-t_1} = C = bottleneck$ Thus, induction holds

$C_{k+1} \leq C$

In this case the bottleneck of the (k+1) links will be C_{k+1}. At time t_1 the last bit of the first packet would have reached R_k, at time t_2 the last bit of the second packet would have reached R_k. Furthermore, $t_2 = t_1 + \frac{P}{C}$. Furthermore, at this point, the last bit of the second packet would not have reached R_{k+1} as $C_{k+1} \leq C$. The last bit of the first packet reaches R_{k+1} at time $t_1 + \frac{P}{C_{k+1}}$, the last bit of the second packet will reach R_{k+1} at time time $t_1^{'} + \frac{P}{C_{k+1}} = t_1 + 2 * \frac{P}{C_{k+1}}$ (as it won't start sending the second packet until the first packet is fully sent! The first packet is fully sent at $t_1^{'} = t_1 + \frac{P}{C_{k+1}}$, and then it takes $\frac{P}{C_{k+1}}$ time more to send the packet). Thus, $\frac{P}{t_2^{'} - t_1^{'}} = \frac{P}{(t_1 + 2*\frac{P}{C_{k+1}}) - (t_1 + \frac{P}{C_{k+1}})} = \frac{P}{\frac{P}{C_{k+1}}} = C_{k+1}$ Thus, induction holds in all cases

Part 2: Implementation

(a) Create Datagram Sockets

sender.c

```c
int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd < 0)
{
    perror("Socket creation failed");
    exit(EXIT_FAILURE);
}
struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));

server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
```

receiver.c

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

// Bind the socket to the given port
struct sockaddr_in server_addr, client_addr;
memset(&server_addr, 0, sizeof(server_addr));
memset(&client_addr, 0, sizeof(client_addr));

server_addr.sin_family = AF_INET;
server_addr.sin_addr.s_addr = INADDR_ANY; // Listen on any available interface
server_addr.sin_port = htons(port);

if (bind(sockfd, (const struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Bind failed");
    close(sockfd);
    exit(EXIT_FAILURE);
}
```

(b) Send/write data to the socket

```c
for (int i = 0; i < numPairs; i++){
    // first message
```

```
    sprintf(message,"%d",2*i+1); // adding packet identifier to the message
    char tm[4];
    sprintf(tm,"%d",2*i+1);
    message[strlen(tm)] = ' ';
    memset(message+1+strlen(tm),'a',packetSize); // adding text message to fill the packet

    // Send message to server
    sent_bytes = sendto(sockfd, message, strlen(message), 0,
                        (struct sockaddr *)&server_addr, sizeof(server_addr));

    /* Error handling */

    // second message
    for (int i = 0; i < packetSize; i++)
    {
        msg[i] = 'a';
    }
    sprintf(message,"%d",2*i+2); // adding packet identifier to the message
    sprintf(tm,"%d",2*i+2);
    message[strlen(tm)] = ' ';
    memset(message+1+strlen(tm),'a',packetSize); // filling the packet

    // Send to the server
    sent_bytes = sendto(sockfd, message, strlen(message), 0,
                        (struct sockaddr *)&server_addr, sizeof(server_addr));

    /* Error handling */
    usleep(1000*timeDelay);
}
```

(c) Read data from the socket

```
while(1){
    // Read continuously to receive the stream of packets from client
    fflush(stdout);

    char buffer[size+4];
    socklen_t client_len = sizeof(client_addr);

    // Receive from the Client (server.c)
    ssize_t recv_len = recvfrom(sockfd, buffer, 5+size, 0, (struct sockaddr *)&client_addr,
                                                           &client_len);
    clock_gettime(CLOCK_MONOTONIC_RAW, &end);

    /* Error Handling and printing message*/

    /* Find packet number */

    /* Find C using the time difference between two adjacent packets*/

    start = end;
    prev_pkt = curr_pkt;
    }
```

(d) Measure the time of arrival of packets

```
struct timespec start, end;
double total_time;
int prev_pkt = -1,curr_pkt;
```

```
4
5  /* Some other decalartions */
6
7  while(1){
8
9          fflush(stdout);
10
11         ssize_t recv_len = recvfrom(sockfd, buffer, 5+size, 0,
12              (struct sockaddr *)&client_addr, &client_len);
13
14         // The following function is used to measure time
15         // It avoids the extra time gained because of context switching
16         // (which is significant on this scale)
17         clock_gettime(CLOCK_MONOTONIC_RAW, &end);
18
19         // start stores the time struct of prev packet (see line 42)
20
21         buffer[recv_len] = '\0';
22         printf("Message received:%s\n",buffer);
23
24         if (buffer[0] == '1'){
25             printf("Process completed");
26             break;
27         }
28
29         char str[4];
30         for(int i=0;i<recv_len-size-1;i++) str[i] = buffer[i];
31         curr_pkt = atoi(str);
32
33         // If the current packet has an even identifier
34         // (each pair's identifiers are (2j+1,2j+2))
35         // and the prev packet is indeed the first packet of this pair, we calculate
36         // the time using the two structs (start and end)
37         if ((prev_pkt!= -1) && (curr_pkt == prev_pkt + 1) && (prev_pkt%2 == 1)) {
38             total_time = end.tv_nsec - start.tv_nsec + (end.tv_sec - start.tv_sec)*1e9;
39             double C = recv_len*1e3*8/total_time;
40             fprintf(file,"%d %f\n",prev_pkt,C); // store it in the file given in argument
41         }
42         start = end;
43         prev_pkt = curr_pkt;
44     }
```

(e) Correctness of `sender.c`

In the code block given for **part (b)**, in every iteration over the number of pairs, the client writes two times in the socket, back to back, followed by a sleep statement, ensuring the adjacency of the packets of a pair even in the case of a context switch.

Part 3: Experimentation

Experiment 1: Receiver and Sender on same device

Histogram

Bin size: 0.1 *Mbps*
Packet size: 1250 *bytes*
Number of pairs: 200

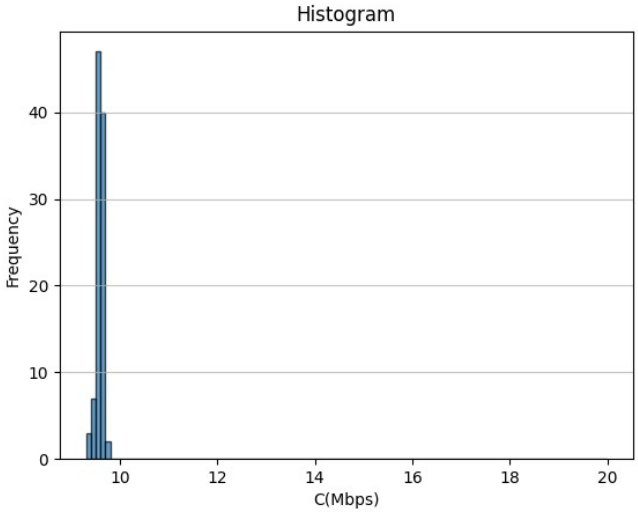

Figure 1: Caption

Observation

The histogram is clustered around the bin ≈ 9.5. Values lower than C bin at the peak can be attributed to network congestion, latency, and variability in measurements. Conversely, some values exceed the expected C due to factors like load balancing and the use of multiple paths, where some packets traveled faster routes, leading to higher C values. Although the variance is not significant, which implies that there aren't many alternative routes with higher C values. In other words, most of the packets traverse the same path indicating less traffic, which is justified because the server and the client, both were being executed on the same device.

Experiment 2: Receiver and Sender on different devices

Path 1 (temp: h15-h6)

Traceroute:

```
traceroute to 10.61.83.48 (10.61.83.48), 30 hops max, 60 byte packets
 1  DESKTOP-84B39O5.mshome.net (172.30.208.1)  0.478 ms  0.452 ms  0.439 ms
 2  XiaoQiang (192.168.31.1)  1.783 ms  1.765 ms  1.751 ms
 3  10.6.10.250 (10.6.10.250)  2.640 ms  2.623 ms  2.610 ms
 4  10.250.6.1 (10.250.6.1)  3.913 ms  4.013 ms  3.803 ms
 5  172.16.2.1 (172.16.2.1)  3.872 ms  3.773 ms  3.750 ms
 6  172.16.12.1 (172.16.12.1)  3.819 ms  5.777 ms  5.758 ms
 7  10.61.83.48 (10.61.83.48)  50.893 ms  20.540 ms  21.659 ms
```

Histogram

- *P(packet size):* 1250 *bytes*

- *Number of Pairs:* 1500
- *Bin size:* 10 *Mbps*

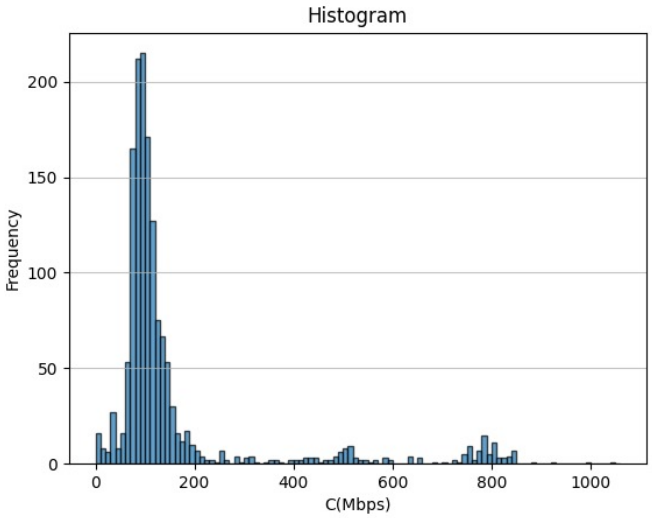

Figure 2: Caption

Observation
On carefully observing the plot, we can estimate a peak at $C \approx 100$. High variance in value of C can be attributed to high traffic and multiple available paths. The congested network forces packets to choose different routes based on load balancing and path availability which results in such a wide range of C values. It is not feasible to pin point a C value because of high variance.

Path 2 (temp: mars-h6)

Traceroute:

```
traceroute to 10.129.3.5 (10.129.3.5), 30 hops max, 60 byte packets
 1  DESKTOP-84B39O5.mshome.net (172.30.208.1)  0.472 ms  0.444 ms  0.429 ms
 2  XiaoQiang (192.168.31.1)  2.349 ms  2.315 ms  4.128 ms
 3  10.6.10.250 (10.6.10.250)  3.424 ms  4.984 ms  3.390 ms
 4  10.250.6.1 (10.250.6.1)  3.657 ms  4.380 ms  5.006 ms
 5  172.16.2.1 (172.16.2.1)  3.523 ms  3.489 ms  4.830 ms
 6  10.250.129.2 (10.250.129.2)  4.971 ms  6.858 ms  6.840 ms
 7  10.129.3.5 (10.129.3.5)  6.757 ms  2.566 ms  2.547 ms
```

Histogram

- *P(packet size):* 1250 *bytes*
- *Number of Pairs:* 1500
- *Bin size:* 10 *Mbps*

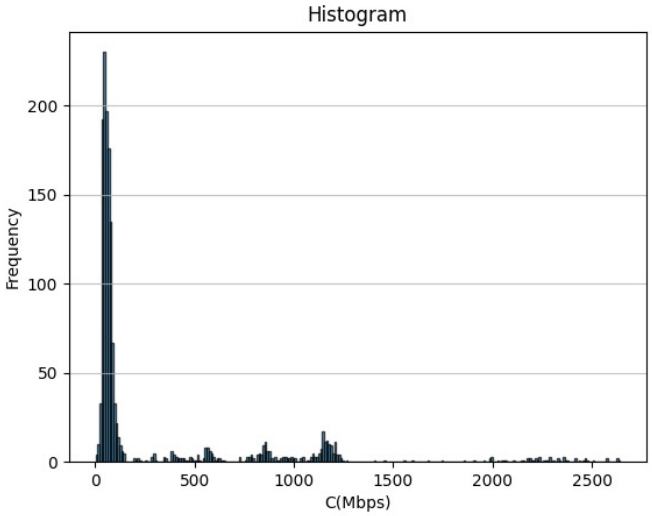

Figure 3: Caption

Observation
There are many routes with different C which have comparable amount of packet traffic as we can observe from the plot based on the frequency. Non-zero frequency at high values show availability of some very high speed routes across networks. It is not feasible to easily estimate the bottle-neck link speed (C) from the graph because of the scale. Although, an estimated C can be pin-pointed, corresponding to the peak value.

Path 3 (temp: sl2-h6)

Traceroute:

```
traceroute to 10.130.154.69 (10.130.154.69), 30 hops max, 60 byte packets
 1  DESKTOP-84B39O5.mshome.net (172.30.208.1)  0.844 ms  0.716 ms  0.697 ms
 2  XiaoQiang (192.168.31.1)  3.403 ms  3.389 ms  3.104 ms
 3  10.6.10.250 (10.6.10.250)  3.083 ms  3.067 ms  2.937 ms
 4  10.250.6.1 (10.250.6.1)  3.355 ms  3.331 ms  3.540 ms
 5  172.16.2.1 (172.16.2.1)  3.287 ms  3.245 ms  3.392 ms
 6  10.250.130.2 (10.250.130.2)  3.687 ms  5.464 ms  5.445 ms
 7  10.130.154.69 (10.130.154.69)  5.321 ms  4.286 ms  4.249 ms
```

Histogram

- *P(packet size):* 1250 *bytes*
- *Number of Pairs:* 1500
- *Bin size:* 10 *Mbps*

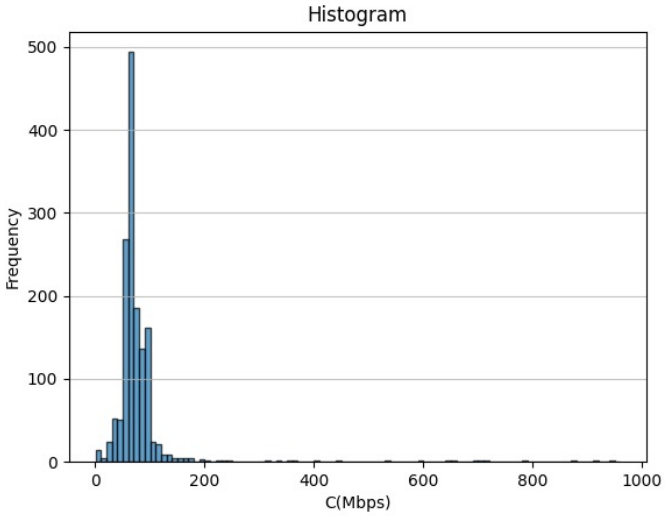

Figure 4: Caption

Observation
A distinctive peak can be observed at $C \approx 70$ and therefore, it is possible to pin-point an estimate of C. The spread of values indicates moderate level traffic. Most of the routes taken by the packets have C values concentrated around 70. Yes, the bottleneck speed can be easily estimated by looking at the plot.