Project 2:

TCP and UDP Performance Measurement

CS158A-02

Project 2

Aishwarya Venketeswaran, Luke Sieben, and Chin Tsai

Submitted to Layla Pezeshkmehr

**Table of Contents**

[Introduction](#h.uibobdqshndt)

[Group Information](#h.r59px9nk0wzq)

[Description of Socket Programming in C](#h.er1fwvikw0nm)

[Project Description](#h.hggyjj793fam)

[UDP only](#h.e8ln0vj6i4io)

[Sections i, ii and iii](#h.ngw28acqwjkm)

[Section iv.](#h.3gkgo39o15ox)

[Section v.](#h.x190c3w4hvh)

[TCP only](#h.l5r18fg8afjs)

[Section i,ii,iii](#h.l4y2zmfl5c35)

[Section iv](#h.426uswcrx856)

[Section v](#h.m5d1x29sutxd)

[Section e](#h.8urddj1rmk7y)

[UDP and TCP simultaneously](#h.aa28rkga5ew5)

[Section i, ii and iii](#h.pkl2vraxdpw6)

[Section iv](#h.b82985xtqjzt)

[Section v](#h.9xlisv6ibv5c)

[References](#h.yqlgvi4oc1v5)

### 

### 

## Introduction

### Group Information

Luke Sieben: siebenluke@gmail.com

Chin Tsai: chin.tsai@sjsu.edu

Aishwarya Venketeswaran [**Coordinator**]: aishwarya.venketeswaran@sjsu.edu

### Description of Socket Programming in C

In C we have a lot of different functions at our disposal to do socket programming.

To start us off we have the **socket** method which creates a socket for us to use.

For *servers* we will then use the **bind** method to attach the server to an IP address and port.

Then if we are building a TCP server we will use the **listen** method to listen to wait for a connection followed by the **accept** method which means that we have accepted the client’s connection request. Then we use the **read** and **write** methods to send data back and forth between ourselves and the clients. Finally we use the **close** method to close the socket.

On the other hand, if we are building a UDP server we will simply start off by using the **recvfrom** method to wait until a client sends us some data. We will then that command and the **sendto** method to respond back and forth with the client. Once we are done communicating we will use the **close** method to close the socket.

After creating a socket in a *client* we will not have to use the bind method.

Then if we are building a TCP client we will then have to use the **connect** method to try and get a connection with the TCP server. After that we will use the same **read** and **write** method the server uses to send data back and forth. Finally we will use the **close** method to close the socket.

On the other hand, if we are building a UDP client we will need to use the **gethostbyname** method to get the address of the UDP server we are trying to communicate with. After that we will use the same **sendto** and **recvfrom** methods to send data back and forth between ourselves and the server. We will use the **close** method we when are done communicating to close the socket.

The following contains more in-depth info about the bolded methods used in socket programming.

|  |  |
| --- | --- |
| int **socket**(int domain, int type, int protocol)  This creates an endpoint for us to use to connect servers and clients and will return a negative value on failure. | domain = domain for PF\_INET  type = the type of server or client we want to make: SOCK\_STREAM - TCP and SOCK\_DGRAM - UDP, protocol = 0 |
| int **bind**(int sockfd, struct sockaddr \*my\_address, socklen\_t addrlen)  This binds a server to an IP address and port and returns a negative value on failure. | sockfd = return value from the **socket** method  my\_address = a structure that holds the all of the address info about the server  addrlen = the size of my\_address |
| int **listen**(int sock, int backlog)  This listens for a connection request from clients and returns a negative value on failure. | sock = the return value from the **socket** method  backlog = the maximum number of connections that can be waited on |
| int **accept**(int socket, (stuct sockaddr\*)&client, socklen\_t \*client\_len)  This accepts a connection from a client and returns a negative value on failure. | socket = value returned by the **listen** method  client = structure to hold the client’s info  client\_len = the size of the client structure |
| int **connect**(int sock, (struct sockaddr\*)&server\_addr, socklen\_t len)  This method connects a client to a server and returns a negative value on failure. | sock = the return value of the **socket** method  server\_addr = the structure to hold the server’s information  len = the size of the server structure |
| int **read**(int sock, void \*mesg, size\_t len, int flags)  This method reads data from a socket and returns a negative value on failure. | sock = the return value of the **socket** method  mesg = the mesg to be read  len = the buffer size of mesg  flags = 0 |
| int **writes**(int sock, void \*mesg, size\_t len, int flags)  This method writes data to a socket and returns a negative value on failure. | sock = the return value of the **socket** method  mesg = the mesg to be sent  len = the buffer size of mesg,flags = 0 |
| int **recvfrom**(int s, char \*buf, int len, int flags, struct sockaddr \*from, int fromlen)  This method receives data from a socket and returns a negative value on failure. | s = the socket  buf = the buffer to store the received data  len = the buffer size, flags = 0  from = the structure the data is coming from  fromlen = the size of the from structure |
| int **sendto**(int s, char \*msg, int len, int flags, struct sockaddr \*to, int tolen)  This method sends data to a socket and returns a negative value on failure. | s = the socket  msg = the message to be sent  len = the buffer size, flags = 0  to = the structure the data is being sent  tolen = the size of the to structure |
| int **close**(int sock)  This method closes the socket and returns a negative value on failure. | sock = the socket to be closed |

### Project Description

The goal of this project is to use sockets to send data of various sizes through the network. We use sockets for communication. In the following sections, we implement TCP and UDP and compare their performance.

## 

## 

## UDP only

The goal of this project is to create a UDP server capable of sending data of various sizes through the network. We start off by sending 1B of data to the server 1000 times from a client, and then various amount ranging from 1KB to 64KB of data 100 times.

### 

### Sections i, ii and iii

**Program (Code):**

|  |  |
| --- | --- |
| udpServer.c  #include <stdio.h>  #include <string.h>  #include <unistd.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <sys/types.h>    #define BUFFER\_SIZE 64000    int main(int argc, char \*argv[]) {  if (argc < 2) {  printf("Error, port argument is missing\n");  return 0;  }    struct sockaddr\_in server, client;  int sock, serverLength, result;  socklen\_t clientLength;  char buffer[BUFFER\_SIZE];    // socket - a connection endpoint  sock = socket(PF\_INET, SOCK\_DGRAM, 0); // SOCK\_DGRAM is for UDP, SOCK\_STREAM is for TCP  if(sock < 0) {  printf("Error on socket\n");  return 0;  }    // setup server  serverLength = sizeof(server);  bzero(&server, serverLength);  server.sin\_family = PF\_INET;  server.sin\_port = htons(atoi(argv[1])); // a good port is 8090  server.sin\_addr.s\_addr = INADDR\_ANY;    // bind - attaching to an IP and Port  if(bind(sock, (struct sockaddr \*) &server, sizeof(server)) < 0) {  printf("Error on bind\n");  return 0;  }    // wait for clients to connect  printf("Server is waiting...");  clientLength = sizeof(client);  while(1) {  // recvfrom - receive message from client  result = recvfrom(sock, buffer, BUFFER\_SIZE, 0, (struct sockaddr \*)&client, &clientLength);  if(result < 0) {  printf("Error on recvfrom\n");  }  else {  printf("Received %d bytes\n", strlen(buffer) + 1);  //printf("Received %d bytes: %s\n", strlen(buffer) + 1, buffer); // slowdown for concurrent connections    // sendto - send an acknowledgement  result = sendto(sock, buffer, strlen(buffer), 0,(struct sockaddr \*)&client, clientLength);  if(result < 0) {  printf("Error on sendto\n");  }  }  }    // close - end the connection  close(sock);    return 0;  } | udpClient.c  #include <netdb.h>  #include <stdio.h>  #include <string.h>  #include <time.h>  #include <unistd.h>  #include <netinet/in.h>  #include <sys/socket.h>  #include <sys/types.h>    #define BUFFER\_SIZE 64000    void sendBytes(const char \*port, const char \*hostname, const int BYTE\_SIZE, const int LOOP\_AMOUNT);    int main(int argc, char \*argv[]) {  if (argc < 3) {  printf("Error, port/ip address argument is missing\n");  return 0;  }    // a good port is 8090  sendBytes(argv[1], argv[2], 1, 1);  sendBytes(argv[1], argv[2], 1, 1000);  sendBytes(argv[1], argv[2], 1024, 100);  sendBytes(argv[1], argv[2], 1024 \* 4, 100);  sendBytes(argv[1], argv[2], 1024 \* 8, 100);  sendBytes(argv[1], argv[2], 1024 \* 16, 100);  sendBytes(argv[1], argv[2], 1024 \* 32, 100);  sendBytes(argv[1], argv[2], 64000, 100);  return 0;  }    void sendBytes(const char \*port, const char \*hostname, const int BYTE\_SIZE, const int LOOP\_AMOUNT) {  clock\_t startClock, endClock;  struct sockaddr\_in server, client;  int i, sock, serverLength, result, lostPackets;  double elapsedTime, totalRTT;  socklen\_t clientLength;  struct hostent \*host;  char outBuffer[BYTE\_SIZE], inBuffer[BUFFER\_SIZE];    // socket - a connection endpoint  sock = socket(PF\_INET, SOCK\_DGRAM, 0); // SOCK\_DGRAM is for UDP, SOCK\_STREAM is for TCP  if(sock < 0) {  printf("Error on socket\n");  return;  }    // gethostbyname - gets the server's address  host = gethostbyname(hostname); // "localhost" refers to ourself  if(host == 0) {  printf("Error on gethostbyname\n");  return;  }  bcopy((char \*)host->h\_addr, (char \*)&server.sin\_addr, host->h\_length);    // setup server  serverLength = sizeof(server);  bzero(&server, serverLength);  server.sin\_family = PF\_INET;  server.sin\_port = htons(atoi(port));  server.sin\_addr.s\_addr = INADDR\_ANY;    // setup buffers  for(i = 0; i < BYTE\_SIZE - 1; i++) {  outBuffer[i] = 'a';  }  outBuffer[BYTE\_SIZE - 1] = '\0';    bzero(inBuffer, BUFFER\_SIZE);    // send LOOP\_AMOUNT messages of BYTE\_SIZE  lostPackets = 0;  totalRTT = 0;  clientLength = sizeof(client);  for(i = 0; i < LOOP\_AMOUNT; i++) {  // setup startClock  startClock = clock();    // sendto - send a message  result = sendto(sock, outBuffer, BYTE\_SIZE, 0, (struct sockaddr \*)&server, serverLength);  if(result < 0) {  printf("Error on sendto\n");  lostPackets++;  }  else {  // recvfrom - receive acknowledgement from server  result = recvfrom(sock, inBuffer, BUFFER\_SIZE - 1, 0, (struct sockaddr \*)&client, &clientLength);  if(result < 0) {  printf("Error on recvfrom\n");  }    // calculate totalRTT  endClock = clock();  elapsedTime = (double) (endClock - startClock) / CLOCKS\_PER\_SEC;  totalRTT += elapsedTime;  //printf("%2d: %fs, %fs\n", i, elapsedTime, totalRTT);  }  }  // close - end the connection  close(sock);    // calculate throughput, average RTT, etc  printf("Stats on sending %d bytes %d times\n", BYTE\_SIZE, LOOP\_AMOUNT);  double delay = totalRTT / (LOOP\_AMOUNT + 0.0f);  printf("Average RTT (delay): %fs\n", delay);  double delayWithoutLostPackets = totalRTT / (LOOP\_AMOUNT - lostPackets + 0.0f);  printf("Number of lost packets: %ds\n", lostPackets);  printf("Average delay without lost packets: %fs\n", delayWithoutLostPackets);  double throughput = BYTE\_SIZE / delayWithoutLostPackets;  printf("Average throughput: %fBps\n", throughput);  printf("\n");  return;  } |

**Performance Measurements:**

|  |  |
| --- | --- |
| **Sample Output # 1**  Stats on sending 1 bytes 1000 times  Average RTT (delay): 0.000140s  Number of lost packets: 0s  Average delay without lost packets: 0.000140s  Average throughput: 7142.857143Bps  Stats on sending 1024 bytes 100 times  Average RTT (delay): 0.000100s  Number of lost packets: 0s  Average delay without lost packets: 0.000100s  Average throughput: 10240000.000000Bps  Stats on sending 4096 bytes 100 times  Average RTT (delay): 0.000200s  Number of lost packets: 0s  Average delay without lost packets: 0.000200s  Average throughput: 20480000.000000Bps  Stats on sending 8192 bytes 100 times  Average RTT (delay): 0.000100s  Number of lost packets: 0s  Average delay without lost packets: 0.000100s  Average throughput: 81920000.000000Bps  Stats on sending 16384 bytes 100 times  Average RTT (delay): 0.000200s  Number of lost packets: 0s  Average delay without lost packets: 0.000200s  Average throughput: 81920000.000000Bps  Stats on sending 32768 bytes 100 times  Average RTT (delay): 0.000300s  Number of lost packets: 0s  Average delay without lost packets: 0.000300s  Average throughput: 109226666.666667Bps  Stats on sending 64000 bytes 100 times  Average RTT (delay): 0.000500s  Number of lost packets: 0s  Average delay without lost packets: 0.000500s  Average throughput: 128000000.000000Bps | **Sample Output #2**  Stats on sending 1 bytes 1000 times  Average RTT (delay): 0.000140s  Number of lost packets: 0s  Average delay without lost packets: 0.000140s  Average throughput: 7142.857143Bps  Stats on sending 1024 bytes 100 times  Average RTT (delay): 0.000200s  Number of lost packets: 0s  Average delay without lost packets: 0.000200s  Average throughput: 5120000.000000Bps  Stats on sending 4096 bytes 100 times  Average RTT (delay): 0.000100s  Number of lost packets: 0s  Average delay without lost packets: 0.000100s  Average throughput: 40960000.000000Bps  Stats on sending 8192 bytes 100 times  Average RTT (delay): 0.000100s  Number of lost packets: 0s  Average delay without lost packets: 0.000100s  Average throughput: 81920000.000000Bps  Stats on sending 16384 bytes 100 times  Average RTT (delay): 0.000200s  Number of lost packets: 0s  Average delay without lost packets: 0.000200s  Average throughput: 81920000.000000Bps  Stats on sending 32768 bytes 100 times  Average RTT (delay): 0.000300s  Number of lost packets: 0s  Average delay without lost packets: 0.000300s  Average throughput: 109226666.666667Bps  Stats on sending 64000 bytes 100 times  Average RTT (delay): 0.000500s  Number of lost packets: 0s  Average delay without lost packets: 0.000500s  Average throughput: 128000000.000000Bps |

### 

### Section iv.

The server and client are on the localhost network. The “tracert localhost” command gave:

*Tracing route to y510p [::1]*

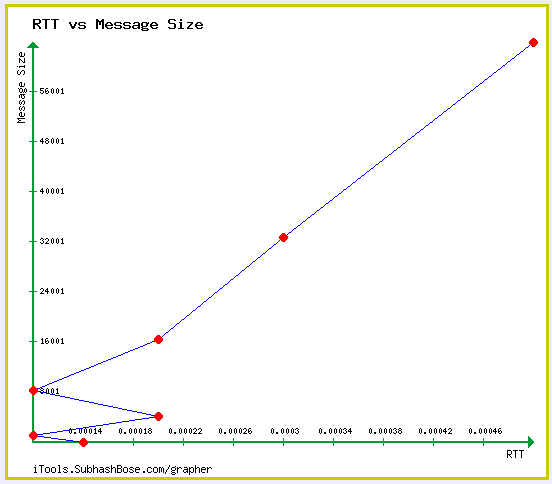
*over a maximum of 30 hops:*

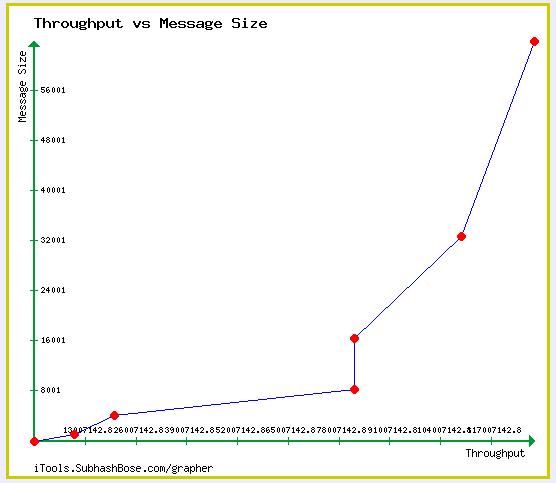
*1 <1 ms <1 ms <1 ms y510p [::1]*

*Trace complete.*

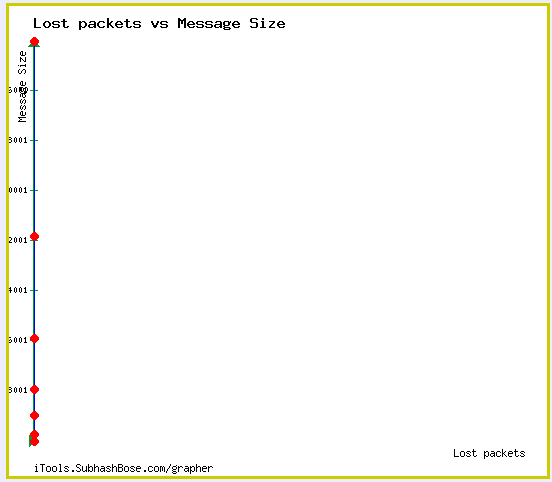
The IP address of the localhost is **127.0.01**

Since they are on the localhost network the data transfers that are taking place should be extremely fast since we are sending messages to ourselves.

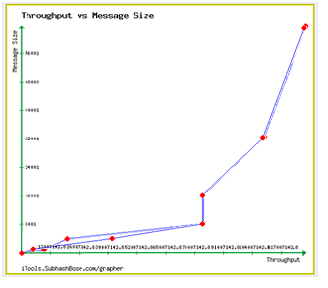
The RTT vs Message Size graph shows us that the data didn’t take much time at all to be transferred.

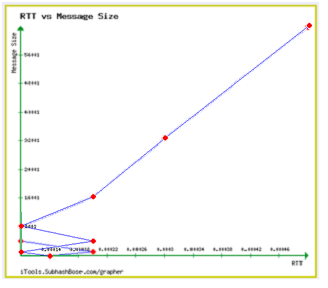


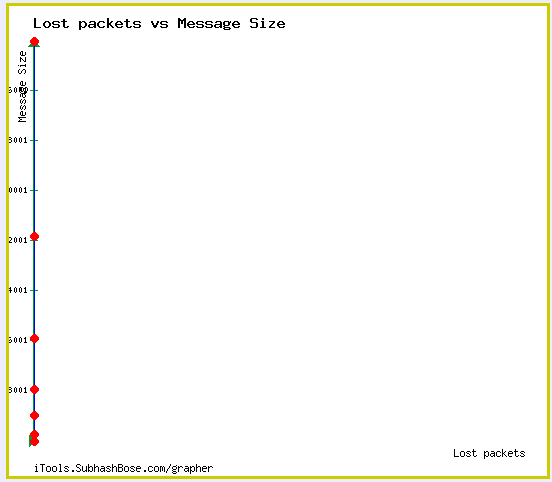
The Lost packets vs Message Size graph shows us that the throughput went up as we were sending more data in chunks.

The Throughput vs Message Size (bottom) graph shows us that none of the packets were lost.

**Comparison of Sample Output #1 and #2**

Both clients sent data to the server at the same time, however, the one that sent data first stopped the other one from connecting and as a result the second one to go didn’t start until the first one finished. And that’s why the results are very similar. The following graphs were made from both data sets.





**Other notes**

Using 64 \* 1024 for 64KB of data to be sent across the network did not work, so 64000 was used in its place. Transferring data on the same network using ip addresses didn’t work, so localhost was used in its place.

### Section v.

Code

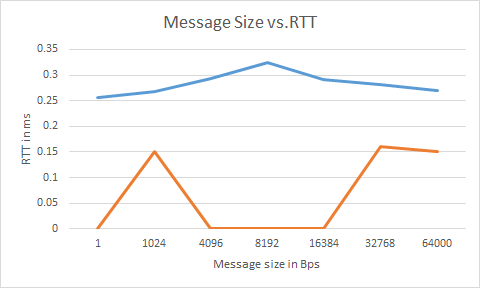
The code for doing this is the same as above (Section i,ii, and iii).

Performance Measurements

The performance measurements for while having two clients send messages to a server is as follows:

|  |  |
| --- | --- |
| Client 1  Stats on sending 1 bytes 1000 times  Average RTT (delay): 0.000256s  Number of lost packets: 0s  Average delay without lost packets:0.000256s  Average throughput: 3905.105926Bps  Stats on sending 1024 bytes 100 times  Average RTT (delay): 0.000268s  Number of lost packets: 0s  Average delay without lost packets:0.000268s  Average throughput: 3826177.932220Bps  Stats on sending 4096 bytes 100 times  Average RTT (delay): 0.000294s  Number of lost packets: 0s  Average delay without lost packets:0.000294s  Average throughput: 13951904.080659Bps  Stats on sending 8192 bytes 100 times  Average RTT (delay): 0.000325s  Number of lost packets: 0s  Average delay without lost packets:0.000325s  Average throughput: 25219345.503802Bps  Stats on sending 16384 bytes 100 times  Average RTT (delay): 0.000292s  Number of lost packets: 0s  Average delay without lost packets:0.000292s  Average throughput: 56044331.942259Bps  Stats on sending 32768 bytes 100 times  Average RTT (delay): 0.000281s  Number of lost packets: 0s  Average delay without lost packets:0.000281s  Average throughput: 116616249.688601Bps  Stats on sending 64000 bytes 100 times  Average RTT (delay): 0.000270s  Number of lost packets: 0s  Average delay without lost packets:0.000270s  Average throughput: 237256719.184430Bps | Client 2 |

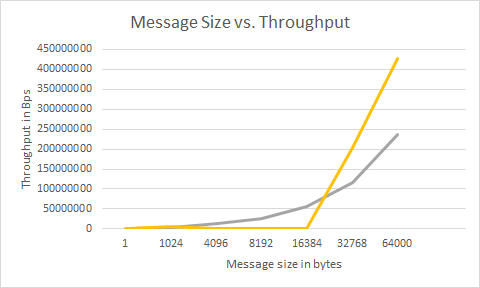
(i) round-trip time vs. message size



We can conclude that the RTT provided by one client is much higher than the other.

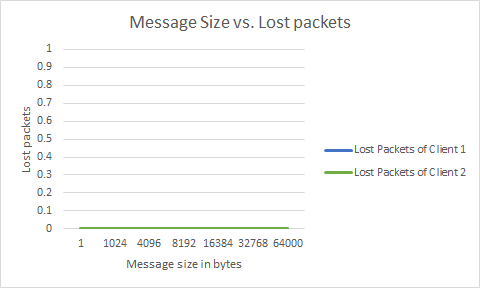
But, both the clients were in one network itself.

(ii) throughput vs. message size



Unlike the message size vs. throughput size, this graph shows that the performance of clients is almost comparable.

(iii) number of lost packets vs. message size



Although RTT and throughput were different significantly, the number of lost packets in both are the same all the time.

We must note that there is significant difference in the performance of each client although both of their performance are proportional to the payload. Both the clients were in the same network. However, one client seems to be much more faster than the other.

As far as we can see, there has not been much difference in the performance even when two clients are communicating with one server simultaneously. We must note that UDP is a connectionless type protocol. Hence, it does not set a status while establishing a connection.This might be the reason for the results we are seeing here.

**Notes**: We must note that the other measurements in this section were done in a team member’s computer different than the computer in which this section’s measurements have been calculated. Hence, the difference in performance could be affected by the networks involved.

## 

## TCP only

The purpose of TCP experiments to investigate the relationship between payload size, RTT, and throughput in the TCP protocol. This is achieved by sending payloads of various sizes from one a client to a server.The first experiment to find out how the size of a payload will affect throughput and RTT. We sent payloads of 1KB, 4KB, 8 KB, 16KB, 32KB, and 64KB from a client to a server (both behind same router). The results are presented below.The second experiment is a repetition of the first experiment. Except this time we simultaneously send packets to a server from two different clients. The experiment is conducted so that both clients and the server are behind same router.

### Section i,ii,iii

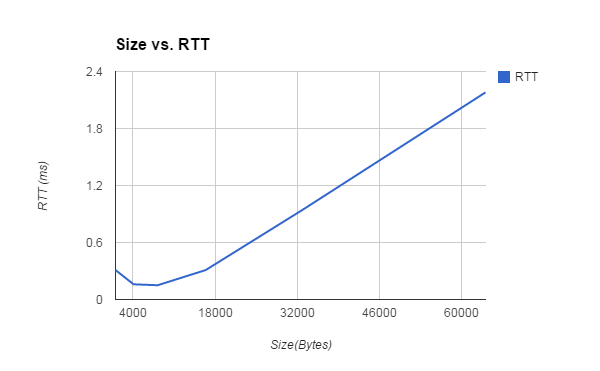
**Code**

|  |  |
| --- | --- |
| **Server Code**  #include <sys/socket.h>  #include <netinet/in.h>  #include <arpa/inet.h>  #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <errno.h>  #include <string.h>  #include <sys/types.h>  #include <time.h>  #define MSize 65000  int main(int argc, char \*argv[])  {    if (argc < 2) {  printf("Error, port argument is missing\n");  return 0;  }      int server = 0, client = 0;  struct sockaddr\_in server\_addr;  struct sockaddr\_in client\_addr;  int len,sin\_size ;  char buf[MSize];  //create TCP socket  server = socket(AF\_INET, SOCK\_STREAM, 0);  server\_addr.sin\_family = AF\_INET;  server\_addr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  server\_addr.sin\_port = htons(atoi(argv[1]));  sin\_size = sizeof(struct sockaddr\_in);    //bind socket  bind(server, (struct sockaddr\*)&server\_addr, sizeof(server\_addr));    //listening for connection  listen(server, 65000);  puts("accepting connection\n");    while(1){    //accept connection  client = accept(server, (struct sockaddr\*)& client\_addr, &sin\_size);    if (client < 0) {  perror("error");  exit(1);  }    //keep communicating with client  while(len=recv(client, buf, sizeof(buf) , 0)>0){  //buf[len]='\0';  //printf("%s\n", buf);  char \*msg = "Hello!";  int size;  size = strlen(msg);  send(client, msg, sizeof(msg), 0);  //c++;  //printf("%d\n", c);  //bzero(buf, MSize);  }    close(client);  }  } | **Client Code**  #include<stdio.h> //printf  #include<string.h> //strlen  #include<sys/socket.h> //socket  #include<arpa/inet.h> //inet\_addr  #include <sys/time.h>  #define MSize 64000  void sendBytes(const char \*port, const char \*hostname, const int BYTE\_SIZE, const int LOOP\_AMOUNT);  int main(int argc , char \*argv[])  {  if (argc < 3) {  printf("Error, port/ip address argument is missing\n");  return 0;  }  sendBytes(argv[1], argv[2], 1, 1);  sendBytes(argv[1], argv[2], 1, 1000);  sendBytes(argv[1], argv[2], 1024, 1000);  sendBytes(argv[1], argv[2], 1024 \* 4, 1000);  sendBytes(argv[1], argv[2], 1024 \* 8, 1000);  sendBytes(argv[1], argv[2], 1024 \* 16, 1000);  sendBytes(argv[1], argv[2], 1024 \* 32, 1000);  sendBytes(argv[1], argv[2], 64000, 1000);  return 0;  }  void sendBytes(const char \*port, const char \*hostname, const int BYTE\_SIZE, const int LOOP\_AMOUNT) {    clock\_t startClock, endClock;  double elapsed=0.0f ;  double totalRTT=0.0f;    int sock;  struct sockaddr\_in server;  char server\_reply[MSize];  char Msg\_size[BYTE\_SIZE];      int x=0;  for(x = 0; x < BYTE\_SIZE; x++) {  Msg\_size[x] = 'a';  }  //Create socket  sock = socket(AF\_INET , SOCK\_STREAM , 0);  if (sock == -1)  {  printf("Could not create socket");  }  puts("Socket created");  //get server information  server.sin\_addr.s\_addr = inet\_addr(hostname);  server.sin\_family = AF\_INET;  server.sin\_port = htons( atoi(port) );  //Connect to server  if (connect(sock , (struct sockaddr \*)&server , sizeof(server)) < 0)  {  perror("connect failed. Error");  return;  }  puts("Connected\n");  //keep communicating with server  int y=0;  for(y=0;y<LOOP\_AMOUNT;y++){    startClock = clock();  //Send some data  if( send(sock , Msg\_size , BYTE\_SIZE, 0) < 0)  {  puts("Send failed");  return;  }  //Receive a reply from the server  int len=0;  if(len = recv(sock , server\_reply , sizeof(server\_reply) , 0) < 0)  {  puts("recv failed");  }  endClock = clock();  double x=(double)CLOCKS\_PER\_SEC;  elapsed = ((double)endClock - (double)startClock)/x;  totalRTT = totalRTT+elapsed;  }  printf("\n");  printf("Stats on sending %d bytes %d times\n", BYTE\_SIZE, LOOP\_AMOUNT);  double delay = totalRTT / (LOOP\_AMOUNT + 0.0f);  printf("Average RTT (delay): %fms\n", delay\*1000);  double throughput = BYTE\_SIZE\*8 / delay;  printf("Average throughput: %fbps\n", throughput);  printf("\n");  close(sock);  return ;  } |

### 

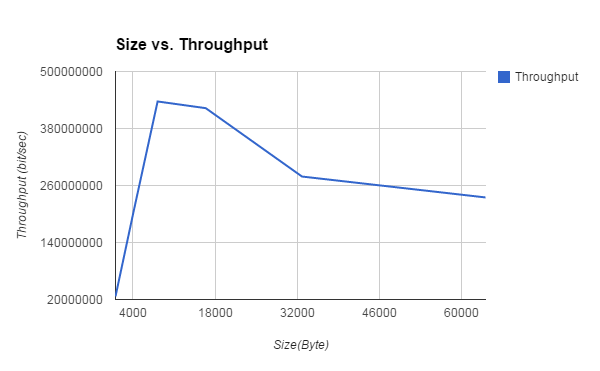
### 

### Section iv

**Various performance measurements and their graphs:**  

|  |  |
| --- | --- |
| Size(byte) | RTT(ms) |
| 1024 | 0.31 |
| 4096 | 0.16 |
| 8192 | 0.15 |
| 16384 | 0.31 |
| 32768 | 0.94 |
| 64000 | 2.18 |

From the graph, it is clearly that RTT is proportionally to payload size. This is because the MTU of the Ethernet is 1500 bytes. Any package that is greater than 1500 bytes have to be broken into segments. Thus, more packets have to be sent, and the overall RTT increases.



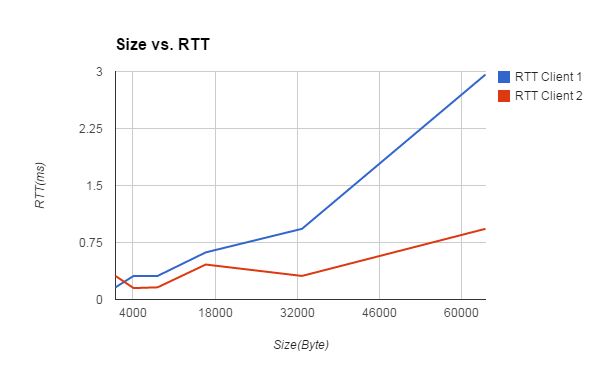
|  |  |
| --- | --- |
| Size(byte) | Throughput (bit/sec) |
| 1024 | 26425806 |
| 4096 | 204800000 |
| 8192 | 436906666 |
| 16384 | 422812903 |
| 32768 | 278876595 |
| 64000 | 234862385 |

Throughput also increases with payload size .However, if the packet become way too big and have to be broken down into segments, and the overall throughput will decrease due to more packets have to be sent in order to transmit a complete message. The graph shows that throughput only increases for a certain range and fall after payloads become too big.

### 

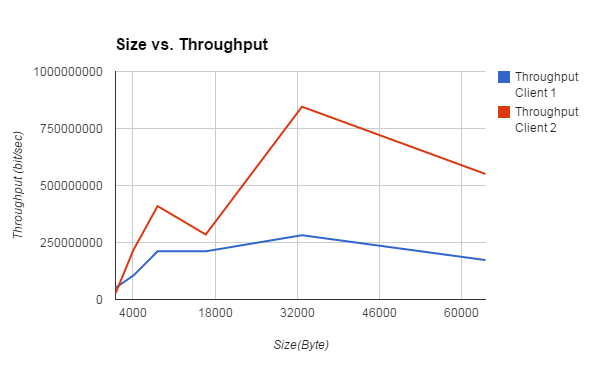
### 

### Section v



|  |  |  |
| --- | --- | --- |
| Size  (Byte) | RTT Client 1 (ms) | RTT Client 2 (ms) |
| 1024 | 0.16 | 0.31 |
| 4096 | 0.31 | 0.15 |
| 8192 | 0.31 | 0.16 |
| 16384 | 0.62 | 0.46 |
| 32768 | 0.93 | 0.31 |
| 64000 | 2.96 | 0.93 |

From the graph, both clients’ RTT are proportional to payload size as expected. While the result of client1 (same machine as the first experiment) doesn’t deviate too much from the first experiment, the result of client2 have overall lower RRT. One explanation is the router gives client2 better QoS. According to Wireshark, there are no interweaving requests . This means the server serves the clients in FCFS basis, and no clients can interrupt the server while the server is in the middle of sending a complete message (i.e message that is bigger than MTU).



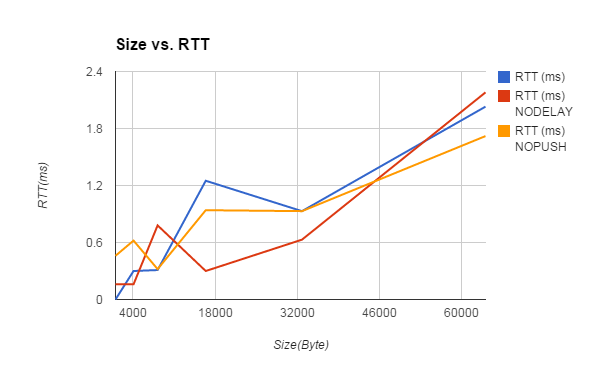
|  |  |  |
| --- | --- | --- |
| Size | Throughput Client 1(bit/sec) | Throughput Client 2 (bit/sec) |
| 1024 | 51200000 | 26425806 |
| 4096 | 105703225 | 218453333 |
| 8192 | 211406451 | 409600000 |
| 16384 | 211406451 | 284939130 |
| 32768 | 281875268 | 845625806 |
| 64000 | 172972972 | 550537634 |

The throughput also behaves like the first experiment. It increases proportionally at first but decreases if payload size becomes too big. Client2 has overall higher throughput than client1 due to lower RTT.

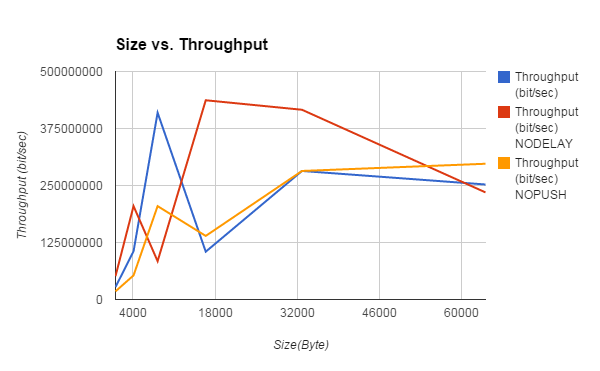
We also send 1 byte of data from a client to a server 1000 times. The outcome doesn’t seem to have any significance except it has very low RTT and low throughputs

### Section e

We tested TCP\_NODELAY and TCP\_NOPUSH options that are available in setsockopt() method.TCP\_NODELAY disable Nagle’s algorithm that buffer sent data. This means that packet will be sent immediately without delay without waiting for its buffer to fill. TCP\_NOPUSH does exact opposite of TCP\_NODELAY. It delays TCP data until its buffer is full. Results:



We expected that TCP\_NODELAY option will give us much lower RTT and high throughput, and TCP\_NOPUSH will give us higher RTT and low throughput. However, both option’s data doesn’t seem to have any significant change from data taken without any options.



## 

## 

## UDP and TCP simultaneously

In this section, we are comparing TCP and UDP by running them concurrently. In Section i-iv, we are running them in the same client. In Section v, they are done in different clients.

### Section i, ii and iii

**Code**

In this section, we ran both TCP and UDP concurrently in a machine. So, the code is same as given above.

**Performance Comparison**

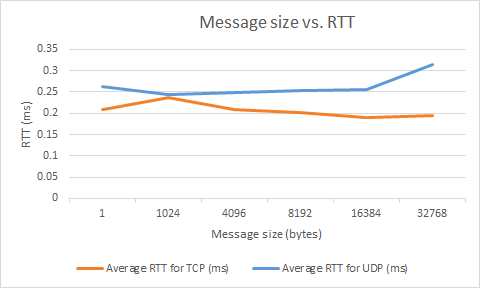
|  |  |
| --- | --- |
| TCP  Stats on sending 1 bytes 1000 times  Average RTT (delay): 0.203563ms  Average throughput: 39299.872767bps  Stats on sending 1024 bytes 1000 times  Average RTT (delay): 0.209234ms  Average throughput: 39152336.618332bps  Stats on sending 4096 bytes 1000 times  Average RTT (delay): 0.236739ms  Average throughput: 138414034.020588bps  Stats on sending 8192 bytes 1000 times  Average RTT (delay): 0.209707ms  Average throughput: 312512219.429966bps  Stats on sending 16384 bytes 1000 times  Average RTT (delay): 0.201678ms  Average throughput: 649907277.938098bps  Stats on sending 32768 bytes 1000 times  Average RTT (delay): 0.189713ms  Average throughput: 1381792497.087697bps  Stats on sending 64000 bytes 1000 times  Average RTT (delay): 0.195744ms  Average throughput: 2615661271.865274bps | UDP  Stats on sending 1 bytes 1000 times  Average RTT (delay): 0.000256s  Number of lost packets: 0s  Average delay without lost packets:0.000256s  Average throughput: 3911.704995Bps  Stats on sending 1024 bytes 100 times  Average RTT (delay): 0.000262s  Number of lost packets: 0s  Average delay without lost packets:0.000262s  Average throughput: 3911083.950806Bps  Stats on sending 4096 bytes 100 times  Average RTT (delay): 0.000244s  Number of lost packets: 0s  Average delay without lost packets:0.000244s  Average throughput: 16777258.949783Bps  Stats on sending 8192 bytes 100 times  Average RTT (delay): 0.000248s  Number of lost packets: 0s  Average delay without lost packets:0.000248s  Average throughput: 33064255.731353Bps  Stats on sending 16384 bytes 100 times  Average RTT (delay): 0.000254s  Number of lost packets: 0s  Average delay without lost packets:0.000254s  Average throughput: 64600583.550193Bps  Stats on sending 32768 bytes 100 times  Average RTT (delay): 0.000256s  Number of lost packets: 0s  Average delay without lost packets:0.000256s  Average throughput: 128005000.195320Bps  Stats on sending 64000 bytes 100 times  Average RTT (delay): 0.000314s  Number of lost packets: 0s  Average delay without lost packets:0.000314s  Average throughput: 203562340.966921Bps |

### Section iv

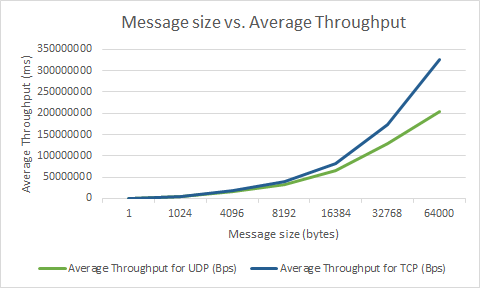
In this case, the server was in a different network than the client. Also, by using traceroute,I found that the path from the server to the client is not very long because it is in the same country.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Size (Bytes)** | **Average RTT for TCP (ms)** | **Average RTT for UDP (s)** | **Average Throughput for TCP (bps)** | **Average Throughput for UDP (Bps)** | **Average Lost packets for TCP** | **Average Lost packets for UDP** |
| 1 | 0.203563 | 0.000256 | 39299.872767 | 3911.704995 | N/A | 0 |
| 1024 | 0.209234 | 0.000262 | 39152336.618332 | 3911083.950806 | N/A | 0 |
| 4096 | 0.236739 | 0.000244 | 138414034.020588 | 16777258.949783 | N/A | 0 |
| 8192 | 0.209707 | 0.000248 | 312512219.429966 | 33064255.731353 | N/A | 0 |
| 16384 | 0.201678 | 0.000254 | 649907277.938098 | 64600583.550193 | N/A | 0 |
| 32768 | 0.189713 | 0.000256 | 1381792497.087697 | 128005000.195320 | N/A | 0 |
| 64000 | 0.195744 | 0.000314 | 2615661271.865274 | 203562340.966921 | N/A | 0 |

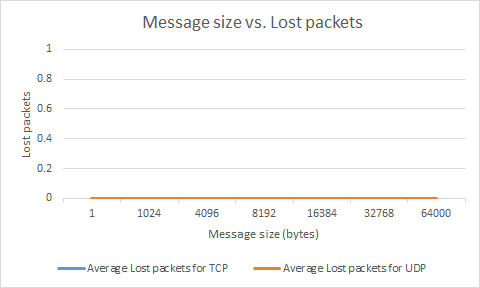
### 

(i) round-trip time vs. message size

From this graph, we can note that, in every case (except one), TCP has a lower RTT than UDP. Hence, TCP seems to be faster in transmitting messages than UDP.

(ii) throughput vs. message size

From this graph, we can note that, in most cases, TCP has a better throughput than UDP. Hence, TCP seems to be more efficient. We must consider the fact that in the case of UDP, we included lost packets while calculating eff. throughput. But, in the case of TCP this was not true.

(iii) number of lost packets vs. message size.

This graph makes us think that the number of lost packets in TCP is none. However, this value was not supposed to be calculated. Hence, it is not considered.

### 

### 

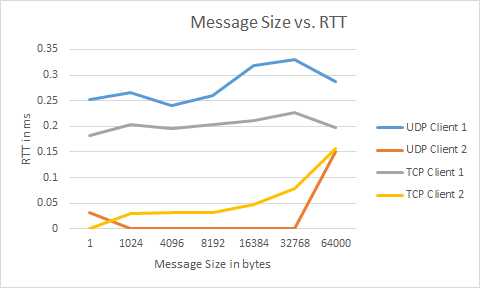
### Section v

In this section, TCP and UDP were run concurrently in two clients simultaneously.

**Code:** The code for doing this is the same as above (Section i,ii, and iii).

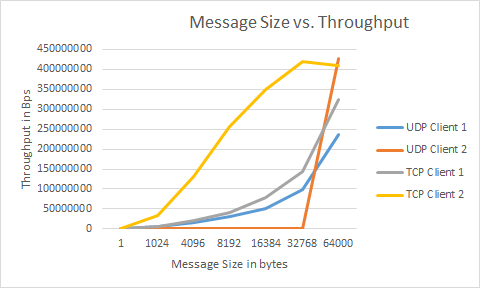
**Performance Measurements**

(i) round-trip time vs. message size



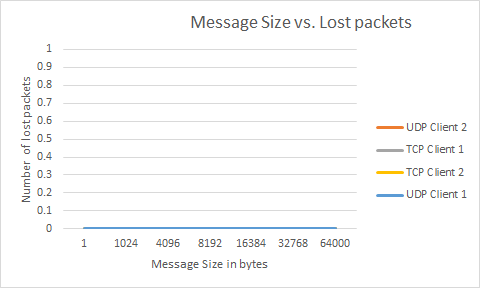
From this graph we can conclude that client 1 is better than the other. But, we can’t conclude anything regarding the protocols’ performance.

(ii) throughput vs. message size



From this graph, we can conclude that Client 2 has better throughput than Client 1. About the performance of protocols we can say that TCP seems to have better throughput than UDP in both cases.

(ii) number of lost packets vs. message size



This graph makes us think that the number of lost packets in TCP is none. However, this value was not supposed to be calculated. Hence, it is not considered.

## 

## References

1. 158A Socket Programming Slides
2. <http://www.linuxhowtos.org/C_C++/socket.htm>
3. <http://www.linuxhowtos.org/data/6/socket.txt>
4. <http://www.linuxhowtos.org/data/6/recvfrom.txt>
5. <http://www.linuxhowtos.org/data/6/sendto.txt>
6. <http://en.wikipedia.org/wiki/C_data_types>
7. <http://stackoverflow.com/questions/3800111/how-to-choose-which-port-to-use-for-a-service-windows-net>
8. <http://beej.us/guide/bgnet/>