Computer Networks - II

Programming assignment - I

Saurabh (CS14BTECH11031)

Part - I

Client-side

The client side program invokes two system calls, the first, "**sendto()**", where the client sends a packet to the server, and the second, "**recvfrom()**", where, the same packet is echoed by the server back to the client. The following is the structure of the packet :

```
struct timestamp {
  struct timeval current_time;
  long elapsed;
  char buffer[1024];
};
```

The first field is the timestamp, the second stores the time elapsed for a one way trip, and the last argument is the character buffer, which stores the time at which the packet was deployed. The socket is created by the client as follows :

```
if((socket_id = socket(AF_INET, SOCK_DGRAM, 0)) == -1){
  std::cerr << "Socket couldn't be created.\n";
  return 1;
}
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
addr_size = sizeof(struct sockaddr);
```

We can see that the port is set to **5000** and the IP is localhost (which can be changed in case of remote server), and the server protocol is set to UDP using **SOCK_DGRAM**.

Now, coming to the system calls involved in sending and receiving the packets, the sendto() system call is used to transmit a message to another socket.
The send() call may be used only when the socket is in a connected state (so that the intended recipient is known). On success, this call return the number of bytes sent. On error, -1 is returned, and errno is set appropriately.

The **recvfrom()** call is used to receive messages from a socket. It may be used to receive data on both connectionless and connection-oriented sockets. If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking. **recvfrom()** places the received message into the buffer buf. The caller needs to specify the size of the buffer in len argument. On success, returns number of bytes received, on error, -1 is returned and errno is set appropriately.
The following is the structure of the invocation of the above two calls in the client program :

```
for(int i = 0;i < num_messages;i++){
  struct timestamp rtt_struct, new_rtt_struct;
  gettimeofday(&rtt_struct.current_time, 0);
  rtt_struct.elapsed = 0;
  currentTime(rtt_struct.buffer);
  int bytes_sent = sendto(socket_id, &rtt_struct, sizeof(rtt_struct), 0, (struct sockaddr *) &server_addr, addr_size);
  if(setsockopt(socket_id, SOL_SOCKET, SO_SNDTIMEO, (char *) &tv, sizeof(struct timeval)) < 0){
    perror("Error in setting socket options\n");
  }
  int bytes_rcvd = recvfrom(socket_id, &new_rtt_struct, sizeof(new_rtt_struct), 0, NULL, NULL);
```

The above program runs a loop and sends/receives the packets, and timeout is set appropriately in the setsockopt() method to indicate that the packet has been lost.

Server-side

The server does the opposite sequence as done by the client. The server first waits for a message to be sent from the client, and does a ‘blocking’ **recvfrom()** unless a message from the client is sent to the server, thus, confirming a message/packet sending entity exists, and then upon successful receival, the server “echoes” back the packet to the client and this continues until the client keeps on sending messages/packets. The prerequisites like the setting up of socket, getting a socket-id, setting fields like **sin_family**, the IP and port is similar and the fields used are the same as in the client. The following is the additional job needed to be done in the server :

```
addr_size = sizeof(struct sockaddr);
if(bind(socket_id, (struct sockaddr *)&server_addr, addr_size) == -1){
    std::cerr << "Error in binding the socket as socket already in use.\n";
    return 1;
}
```

The server needs to do this because when a socket is created with the socket system call as described in the client section, it exists in a name space (address family) but has no address assigned to it. **bind()** assigns the address specified by **server_addr** to the socket referred to by the file descriptor **socket_id**. **addrlen** (sizeof server_addr) specifies the size, in bytes, of the address structure pointed to by server_addr. More informally, this operation can be termed as “assigning a name to a socket”.

After the **bind()** operation is complete, a for loop runs which keeps receiving and echoing back the messages to the client as shown in the code :

```
for(int i = 0;i < num_messages;i++){
  struct timestamp rtt_struct;
  int bytes_rcvd = recvfrom(socket_id, &rtt_struct, sizeof(rtt_struct), 0, (struct sockaddr *) &client_addr, &addr_size);
  get_time = rtt_struct.current_time;
  cout << "Received at server : ";
  cout << rtt_struct.buffer << endl;
  gettimeofday(&rtt_struct.current_time, 0);
  rtt_struct.elapsed = (rtt_struct.current_time.tv_sec - get_time.tv_sec) * 1000000
   + rtt_struct.current_time.tv_usec - get_time.tv_usec;
  int bytes_sent = sendto(socket_id, &rtt_struct, sizeof(rtt_struct), 0, (struct sockaddr *) &client_addr, addr_size);
}
```

Summarizing what happens at the client-side and the server-side, we have :

At client-side :

```
0.gethostbyname()
1.socket()
2.memset(server_addr.sin_zero, '\0', sizeof(server_addr.sin_zero))
4.sendto()
5.recvfrom()
```

And at server-side :

```
0.Variable initialization
```

1.socket()
2.bind()
3.recvfrom()
4.sendto()

Part - II

Client-side

The only change implemented in this code, is that a while loop is run until the interval of sleep method is positive, we keep sending packets, and inside the while loop, we further run a for loop, with maximum limit set to a variable which increases by 1 in every while loop iteration. At the end of a while loop, the sleep interval decreases by 1 second. So basically, in the first iteration, 1 packet is sent, in the second iteration, 2 packets are sent, 3 in the 3rd, and so until the interval between two consecutive batch of packets to be sent is positive. This is implemented as following :

```
while(interval > 0){
  for(int i = 0;i < no_of_packets;i++){
    currentTime(buffer_rcvd);
    gettimeofday(&t0, 0);
    bytes_sent = sendto(socket_id, buffer_rcvd, sizeof(buffer_rcvd), 0, (struct sockaddr *) &server_addr, addr_size);
    buffer_rcvd[bytes_sent] = '\0';
    bytes_rcvd = recvfrom(socket_id, buffer_rcvd, sizeof(buffer_rcvd), 0, (struct sockaddr *) &server_addr, &addr_size);
    gettimeofday(&t1, 0);
    cout << "Received message on trip " << trip << " : " << buffer_rcvd << endl;
    total_size += bytes_sent;
    total_time += ((t1.tv_sec - t0.tv_sec) * 1000000
     + t1.tv_usec - t0.tv_usec);
    total_time /= 1000000.0;
    throughput += (total_size)/total_time;
    average_delay = total_time/count++;
    for(int j = 0;j < interval;j++){
      fprintf(fp, "%lf\n", throughput);
      fprintf(fp_, "%lf\n", average_delay);
    }
  }
  no_of_packets++, trip++;
  sleep(interval--);
}
```

The appropriate **throughput** (bytes/second) and **average_delay** (seconds) calculation is done at the end of loop.

Server-side

The server-side code in 2nd part is mostly same as that in the 1st part. Same sendto() and recvfrom() system calls are invoked to echo all the messages sent by the client back to it. The following is the main code doing this :

```
if(bind(socket_id, (struct sockaddr *)&server_addr, addr_size) == -1){
    std::cerr << "Error in binding the socket as socket already in use.\n";
    return 1;
}
while(1){
  bytes_rcvd = recvfrom(socket_id, buffer_rcvd, sizeof(buffer_rcvd), 0, (struct sockaddr *) &client_addr, &addr_size);
  buffer_rcvd[bytes_rcvd] = 0;
  cout << "Message received at server : " << buffer_rcvd << endl;
  bytes_sent = sendto(socket_id, buffer_rcvd, sizeof(buffer_rcvd), 0, (struct sockaddr *) &client_addr, addr_size);
}
```

The following is the screenshot of the packet capture using wireshark, where my IP is **172.16.2.23** and the remote host's IP is **172.16.2.41**. The only configuration needed to run a client-server echo program is set the IP address to be the IP address of the host computer in the client-side program :

```
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr.s_addr = inet_addr("172.16.2.41");
addr_size = sizeof(struct sockaddr);
```

The following is the screenshot showing my machine's IP address, which is the IP address of the client :

```
wlp6s0    Link encap:Ethernet  HWaddr 4c:bb:58:13:f2:c8
          inet addr:172.16.2.23  Bcast:172.16.7.255  Mask:255.255.248.0
```

The following screenshot clearly shows the echo between the client and the server, as the packets travel back-and-forth :

78	4.389011272	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
79	4.389095776	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
112	6.402650544	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
113	6.402790881	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
145	8.418413976	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
146	8.418523169	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
170	10.434118548	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
171	10.434274367	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
191	12.450125165	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
192	12.450255408	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
207	14.463931856	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
208	14.464081709	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048
217	16.479060065	172.16.2.23	172.16.2.41	UDP	1090 56010 → 5000 Len=1048
218	16.479176827	172.16.2.41	172.16.2.23	UDP	1090 5000 → 56010 Len=1048

The graph for **throughput** (bytes/sec) and **average_delay** (sec) is saved separately in the assignment directory along with the screenshots of the working of the code.