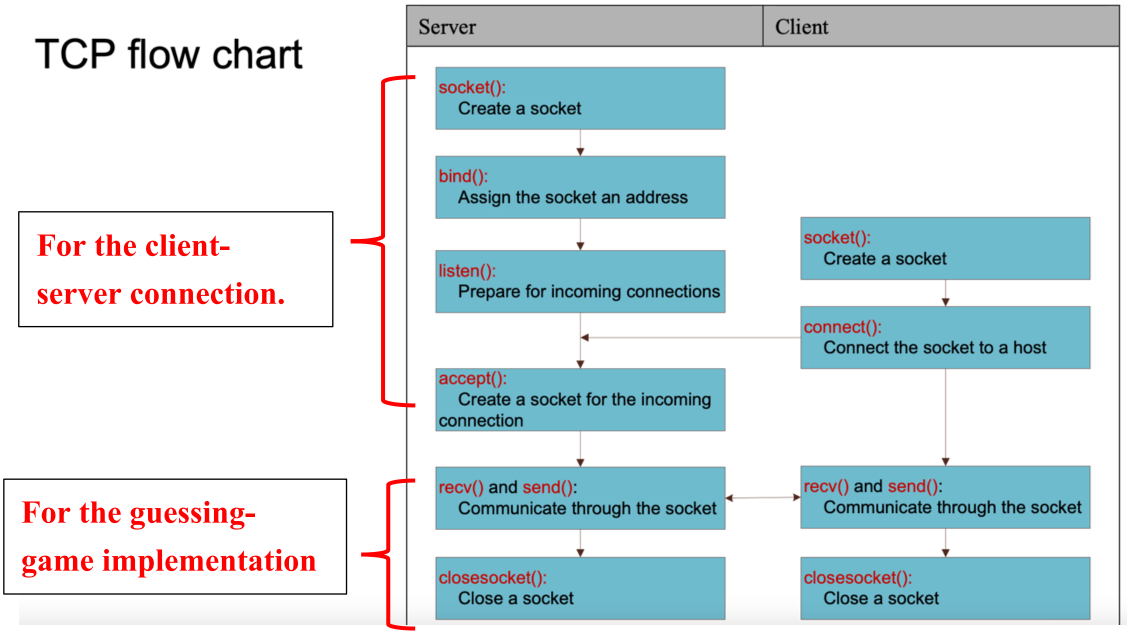
Computer Network Final project 2021
The number guessing game

106041023 肇綺筠 科管 21

- Details of your implementation, including server-side and client-side.
- Step-by-step screenshots and explanations of the execution of each function.
- The answers to the Wireshark observation.
- Descriptions of difficulties you encountered and your solutions.

1. Socket programming

The client and server connect each other through the flow below:

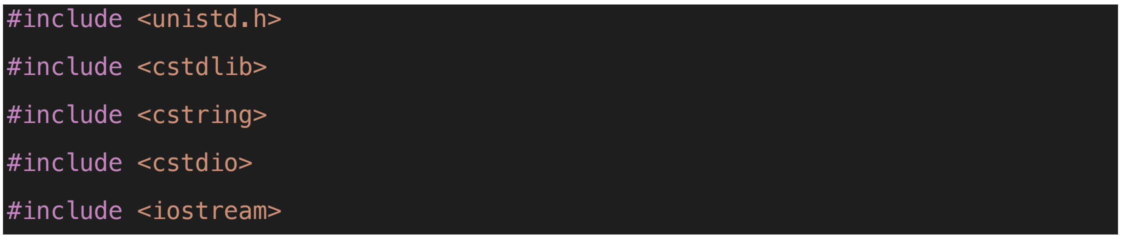

TCP flow chart
Server
Client
socket(): Create a socket
bind(): Assign the socket an address
listen(): Prepare for incoming connections
accept(): Create a socket for the incoming connection
recv() and send(): Communicate through the socket
closesocket(): Close a socket
socket(): Create a socket
connect(): Connect the socket to a host
recv() and send(): Communicate through the socket
closesocket(): Close a socket

For the client-server connection.

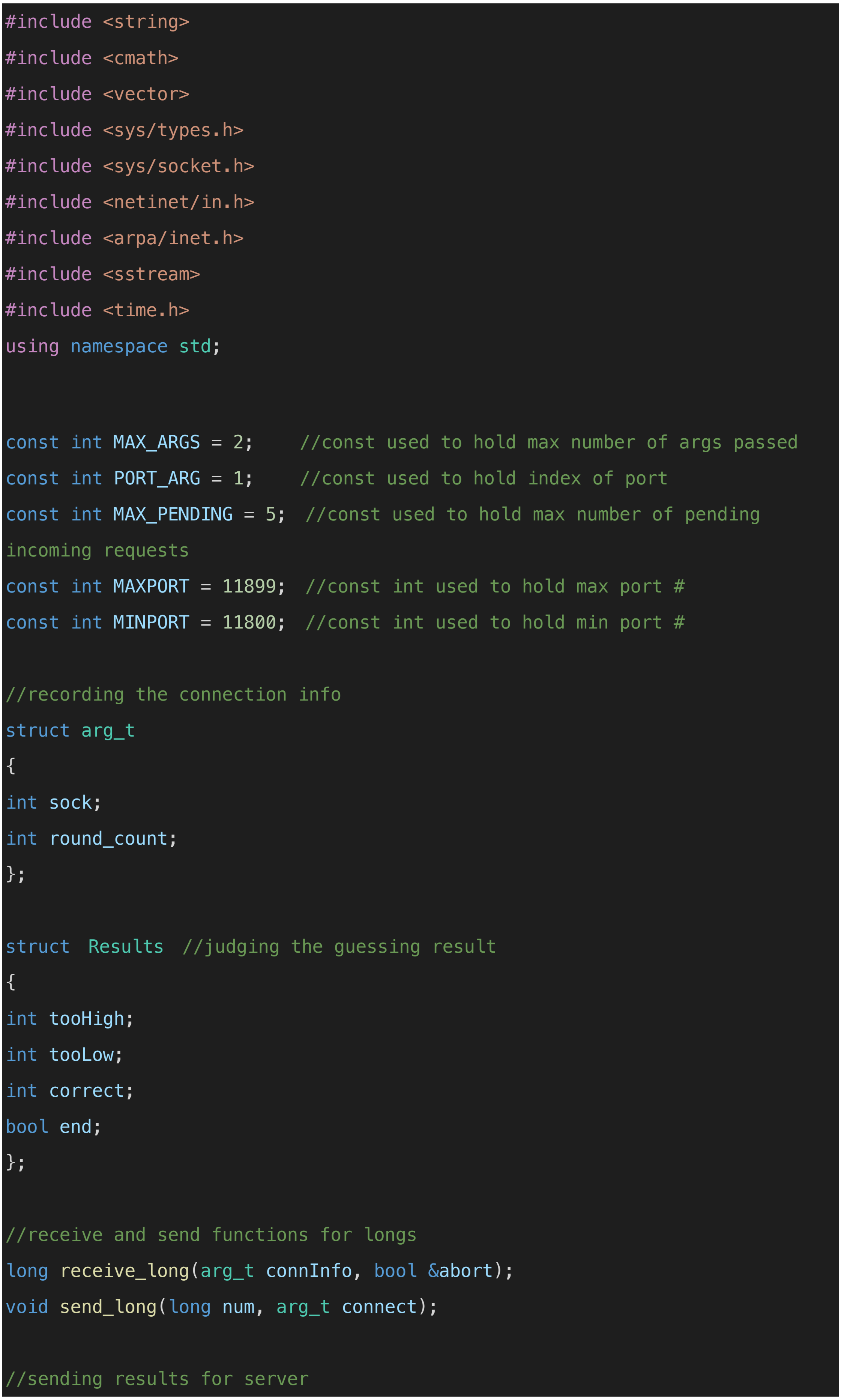
For the guessing-game implementation

➢ Server program

i. Modules, variables, and functions at the server side:



```
#include <unistd.h>
#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <iostream>
```



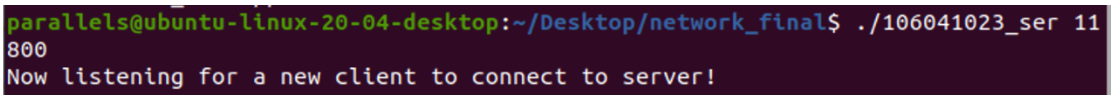
```
#include <string>
#include <cmath>
#include <vector>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sstream>
#include <time.h>
using namespace std;


const int MAX_ARGS = 2;     //const used to hold max number of args passed
const int PORT_ARG = 1;     //const used to hold index of port
const int MAX_PENDING = 5;  //const used to hold max number of pending incoming requests
const int MAXPORT = 11899;  //const int used to hold max port #
const int MINPORT = 11800;  //const int used to hold min port #

//recording the connection info
struct arg_t
{
int sock;
int round_count;
};

struct  Results  //judging the guessing result
{
int tooHigh;
int tooLow;
int correct;
bool end;
};

//receive and send functions for longs
long receive_long(arg_t connInfo, bool &abort);
void send_long(long num, arg_t connect);

//sending results for server
```

```
void send_result(Results result, arg_t connect);
//conversion function for Results(so the data can become readable by the
network)

Results notNet(Results toConv);
Results toNet(Results toConvert);
//the below function is for generating random guessing number for client.
long gen_ran()
{
return (rand() %1000);
}
```

ii. Part1: Client-server connection in the main() function

Keys: Create socket, binding, listening and then accept request from client.

► While compiling server under Ubuntu, we should input:



```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/network_final$ ./106041023_ser 11
800
Now listening for a new client to connect to server!
```

Where the port number goes to 11800 (decided by the user)

► **(1.)** According to the TCP flow chart, we need to construct a socket, if sock <0. Then there's an error happening while creating socket for connection.

(2.) After we successfully created a socket, we then set up the port imported by users.

```
int status;       //status int used to check TCP functions
int clientSock;     //socket used to hold client
(1.)
//creating socket
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (sock < 0) {
cerr << "Error with socket. Now exiting program. " << endl;
close(sock);
exit (-1);
}
(2.)//setting the port
struct sockaddr_in servAddr;
servAddr.sin_family = AF_INET; // always AF_INET
servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
servAddr.sin_port = htons(portNum);
```

(3.) Now is time for **binding** the address to the socket, if **status** < 0, it implies that there’s an error while binding.

(4.) After binding, we need to set up a **listen()** function to set the server socket to “listening status.”

(5.) Passing the “listening socket” into **accept()** function, then accept the request from client including IP address.

```
(3.)
//binding the sockAdress & checking if it worked
status = bind(sock, (struct sockaddr *) &servAddr,
sizeof(servAddr));
if (status < 0) {
cerr << "Error with bind. Now exiting program. " << endl;
close(sock);
exit (-1);
}
(4.)
//setting the server to listen for a client
status = listen(sock, MAX_PENDING); //client:connect function
cerr << "Now listening for a new client to connect to server!" << endl;
if (status < 0) {
cerr << "Error with listen. Now exiting program. " << endl;
close(sock);
exit (-1);
}

while(true){
(5.)
//accepting the next client & testing if there are errors
struct sockaddr_in clientAddr;
socklen_t addrLen = sizeof(clientAddr);
clientSock = accept(sock,(struct sockaddr *) &clientAddr, &addrLen);
if (clientSock < 0) {
cerr << "Error with accept. Now exiting program. " << endl;
close(clientSock);
exit(-1);
}
//setting the connectionInfo in args_p, passing it into “func” for game
implementation
arg_t *args_p = new arg_t;
args_p->sock = clientSock;
func((void*)args_p);
}
}
```

iii. Part2: Game implementation in the func() function

Keys: recv() the number from client and send() the judging result

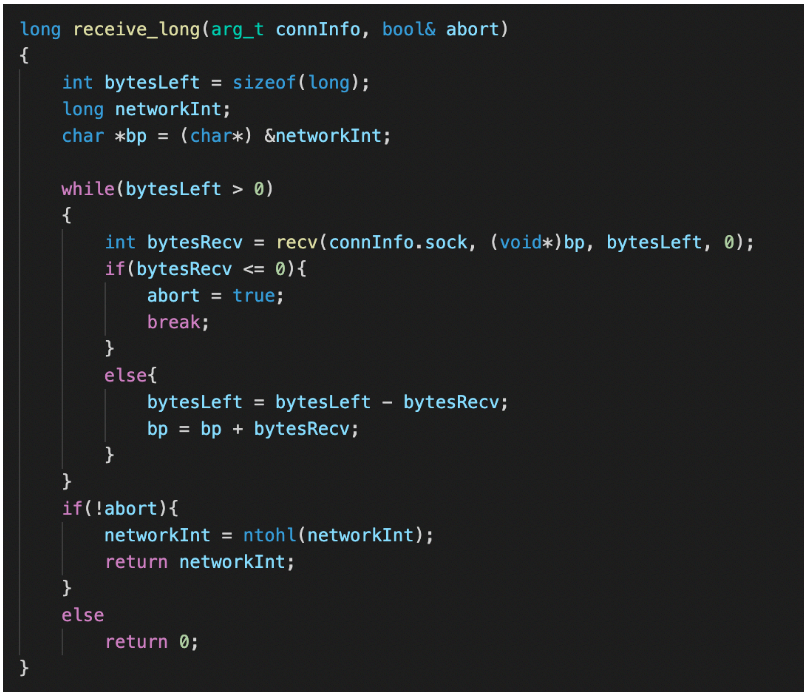
► Initial variables in func()

```

void func(void* pass)
{
    //reclaiming variables from args_pa
    arg_t *args_p;
    args_p = (arg_t*)pass;

    //setting initial variables
    srand(time(NULL));  //seeding random variable
    args_p->round_count = 0;    //setting roundCount to 0
    long roundCount = 0;    //long used to keepTrack of rounds
    long actualNums;    //long arr used to hold random #'s generated
    long numsGuess; //long arr used to hold user's guess
    long numOn;
    bool won = false;      //bool used to test if the client has won
    Results result;     //uninitialized result
    Results *rPointer;  //result Pointer
    bool exit = false;
    bool exit_ = false; // to check if the connection ends
```

► The game implementation

(1.) Generate a **random number** for client to guess

(2.) In the **receive_long()** function, we accept and translate the number given by the client. The implementation of **receive_long():**

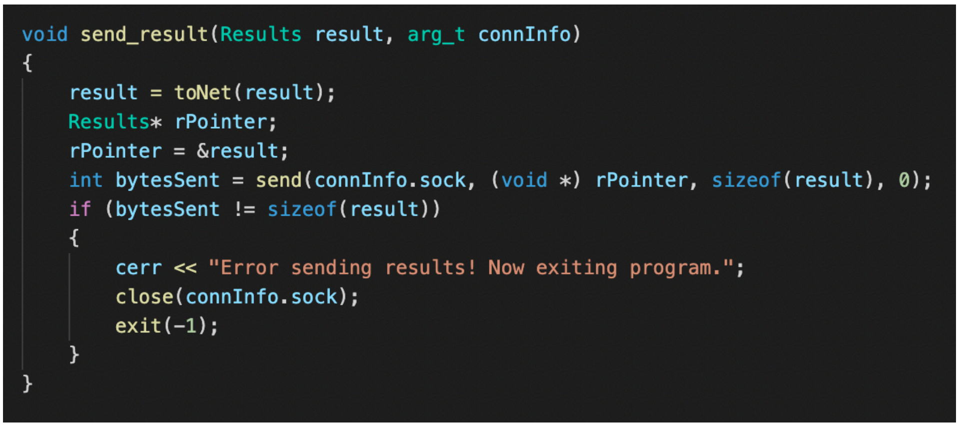


```
long receive_long(arg_t connInfo, bool& abort)
{
    int bytesLeft = sizeof(long);
    long networkInt;
    char *bp = (char*) &networkInt;

    while(bytesLeft > 0)
    {
        int bytesRecv = recv(connInfo.sock, (void*)bp, bytesLeft, 0);
        if(bytesRecv <= 0){
            abort = true;
            break;
        }
        else{
            bytesLeft = bytesLeft - bytesRecv;
            bp = bp + bytesRecv;
        }
    }
    if(!abort){
        networkInt = ntohl(networkInt);
        return networkInt;
    }
    else
        return 0;
}
```

▲ Here we uses recv() to get the packet input by the client, also using ntohl to convert the guessing number into readable status and then return

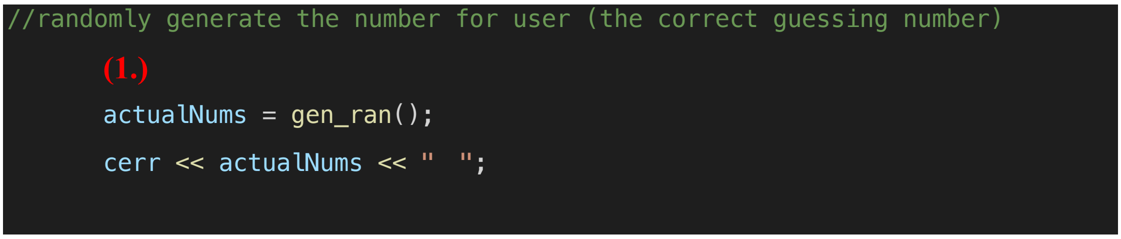
(3.) Judging the guessing number given by the client, if the number matches, this round ends, and the next round will start again.

(4.) Now call send_result() function to send the judgement to the client, the detail of the function:

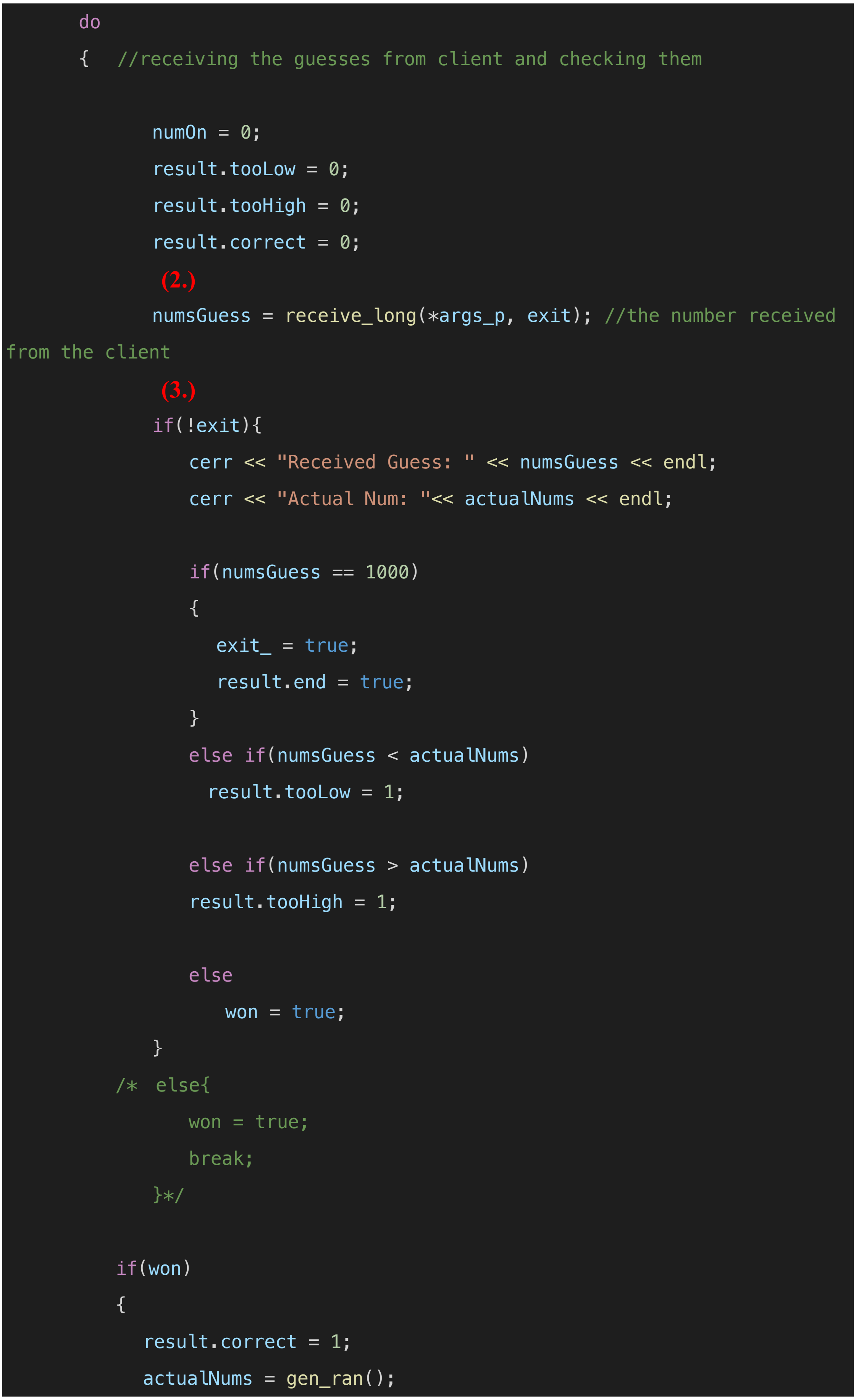


```
void send_result(Results result, arg_t connInfo)
{
    result = toNet(result);
    Results* rPointer;
    rPointer = &result;
    int bytesSent = send(connInfo.sock, (void *) rPointer, sizeof(result), 0);
    if (bytesSent != sizeof(result))
    {
        cerr << "Error sending results! Now exiting program.";
        close(connInfo.sock);
        exit(-1);
    }
}
```

▲ toNet function uses htonl() to convert the judgement into readable status for TCP/IP network, and the result was sent to the client using send()



```
//randomly generate the number for user (the correct guessing number)
        (1.)
        actualNums = gen_ran();
        cerr << actualNums << "  ";
```



```
       do
       {   //receiving the guesses from client and checking them

              numOn = 0;
              result.tooLow = 0;
              result.tooHigh = 0;
              result.correct = 0;
               (2.)
              numsGuess = receive_long(*args_p, exit); //the number received
from the client
               (3.)
              if(!exit){
                 cerr << "Received Guess: " << numsGuess << endl;
                 cerr << "Actual Num: "<< actualNums << endl;

                 if(numsGuess == 1000)
                 {
                   exit_ = true;
                   result.end = true;
                 }
                 else if(numsGuess < actualNums)
                  result.tooLow = 1;

                 else if(numsGuess > actualNums)
                 result.tooHigh = 1;

                 else
                    won = true;
              }
          /*  else{
                 won = true;
                 break;
              }*/

          if(won)
          {
             result.correct = 1;
             actualNums = gen_ran();
             won = false;
             cerr << "New Round Started, new actual number: "<<endl;
             cerr << actualNums << "  "<<endl;

          }
          (4.)
          send_result(result, *args_p);

       }while(!exit_);
```

(5.) Whenever the client asks to exit the game (I set inputting “1000” will end the entire game), then we close the server socket.

```
    (5.)
  if(exit_){
       cerr << endl << "User has left prematurely! " << endl;
    }

    cerr << endl << "Now awaiting a new client!" << endl;
    //closing sockets
    close(args_p->sock);
```

➢ Client program

i. Modules, variables, and functions at the client side:

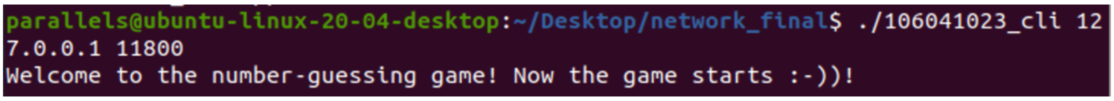
```
#include <unistd.h>
#include <cstdlib>
#include <cstring>
#include <cstdio>
#include <iostream>
#include <string>
#include <cmath>
#include <sys/types.h> // size_t, ssize_t
#include <sys/socket.h> // socket funcs
#include <netinet/in.h> // sockaddr_in
#include <arpa/inet.h> // htons, inet_pton
```

```
#include <unistd.h>
#include <vector>
using namespace std;

const int MAX_ARGS = 3; //max args
const int PORT_ARG = 2;
const int IP_ARG = 1; //port index recording the IP address
const int MAX_NUM = 1; //the number of guesses
const int MAXPORT = 11899; //max port
const int MINPORT = 11800; //min port


struct Results{  //the guessing results returned by server
    int tooHigh;
    int tooLow;
    int correct;
    bool end;
};

 //conversion function for Results
Results toNet(Results toConvert);
Results notNet(Results toConv);
 //receive function for Results
Results rec_result(int sock);

//recv & send function for long
long receive_long(int sock);
void send_long(long num, int sock);
```

ii. Part1: Client-server connection in the main() function

Keys: converting the input port number, initializing the socket and then connect to the server.

- ► While compiling client under Ubuntu, we should input:



```
parallels@ubuntu-linux-20-04-desktop:~/Desktop/network_final$ ./106041023_cli 12
7.0.0.1 11800
Welcome to the number-guessing game! Now the game starts :-))!
```

Where the port number goes to 11800, IP address goes to127.0.0.1 (decided by the user)

► **(1.)** According to the TCP flow chart, we need to construct a socket, if sock <0. Then there’s an error happening while creating socket for connection.

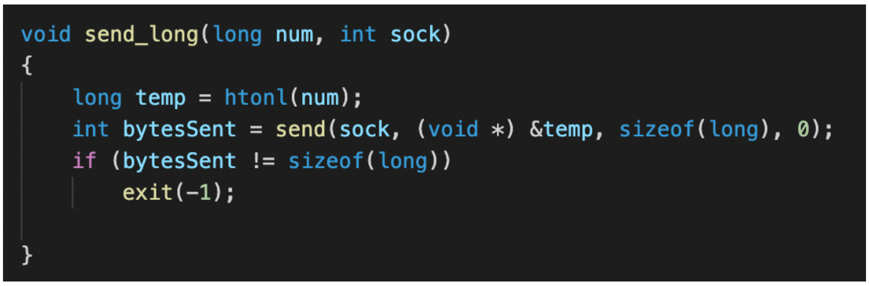
(2.) After we successfully created a socket, we can start to send connection request to server through “connect()”

```
unsigned short portNum = (unsigned short)strtoul(argv[PORT_ARG], NULL,
0);//converting port number into unsigned short

    unsigned long servIP;
    status = inet_pton(AF_INET, argv[IP_ARG], &servIP); //to check if the
server IP is valid
    if (status <= 0)
        exit(-1);
    struct sockaddr_in servAddr;
    servAddr.sin_family = AF_INET; // always AF_INET
    servAddr.sin_addr.s_addr = servIP; //this is for server
    servAddr.sin_port = htons(portNum); //converts to TCP/IP port number

    //initializing the socket
    (1.)
    int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP); //SOCK_STREAM goes
to TCP protocol
    if (sock < 0) {
        cerr << "Error with socket" << endl;
        exit (-1);
    }

    cerr << "Welcome to the number-guessing game! Now the game starts :-))! "
<< endl << endl;
    (2.)
    status = connect(sock, (struct sockaddr*)&servAddr,sizeof(servAddr));
//now connecting the socket to IP address
    if(status < 0)
    {
        cerr << "Error with connect" << endl;
        exit (-1);
    }
```

Part2: Game implementation in the main() function

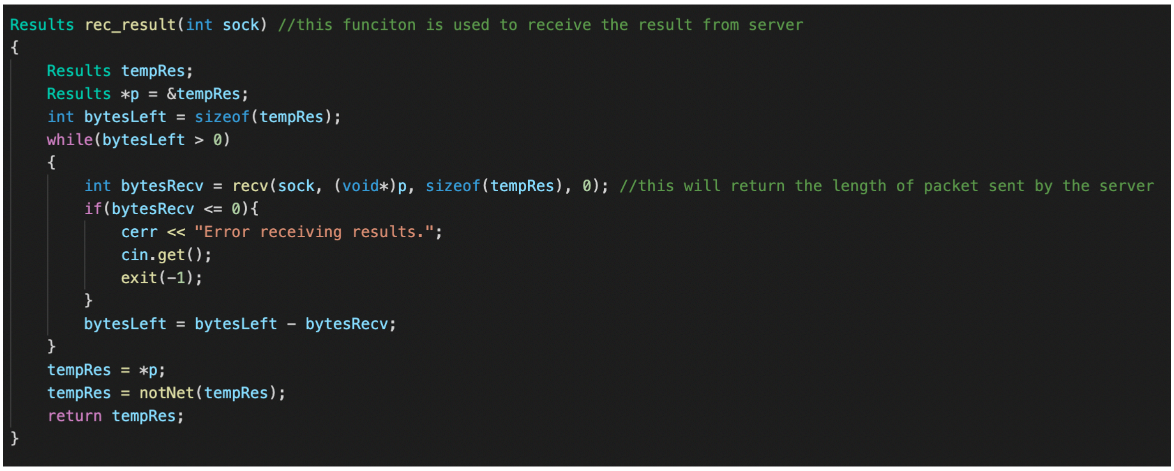
Keys: recv() the result from server / send() the user's answer

► **(1.)**Suppose the input is all correct (0-999), then we use send_long() function to transfer the user's answer to the server. send_long() implementation:



```
void send_long(long num, int sock)
{
    long temp = htonl(num);
    int bytesSent = send(sock, (void *) &temp, sizeof(long), 0);
    if (bytesSent != sizeof(long))
        exit(-1);

}
```

(2.) After sending, client receive the judgement from server using rec_result(). Implementation:



```
Results rec_result(int sock) //this funciton is used to receive the result from server
{
    Results tempRes;
    Results *p = &tempRes;
    int bytesLeft = sizeof(tempRes);
    while(bytesLeft > 0)
    {
        int bytesRecv = recv(sock, (void*)p, sizeof(tempRes), 0); //this will return the length of packet sent by the server
        if(bytesRecv <= 0){
            cerr << "Error receiving results.";
            cin.get();
            exit(-1);
        }
        bytesLeft = bytesLeft - bytesRecv;
    }
    tempRes = *p;
    tempRes = notNet(tempRes);
    return tempRes;
}
```

(3.) Checking the info inside the packet sent by the server, output the result to the user (higher? Lower? Correct?)

```
        //if the input is valid, then send it to the server and check
whether the user win or not
         (1.)
        send_long(numGuess,sock);
         (2.)
        tmp = rec_result(sock); //receive results from server
         (3.)
       if(tmp.tooLow)
       {
          cerr << "lower than 999, " <<"higher than "<< numGuess << endl;

       }
       else if(tmp.tooHigh)
       {
           cerr << "lower than " << numGuess<<", higher than 0" << endl;
       }
       else if (tmp.correct) //to check if the user has won
       {
           win = true;
           cerr << "Answer Correct!"<<endl;
           // cerr << exit_;
           //victory = recv_string(sock);
       }
       roundCount++;
       if(win)
       {
           cerr << "New round started!"<<endl;
           cerr<<"==============="<<endl;
           roundCount = 0;
           win = false;
       }
    }
    while(!exit_);
```

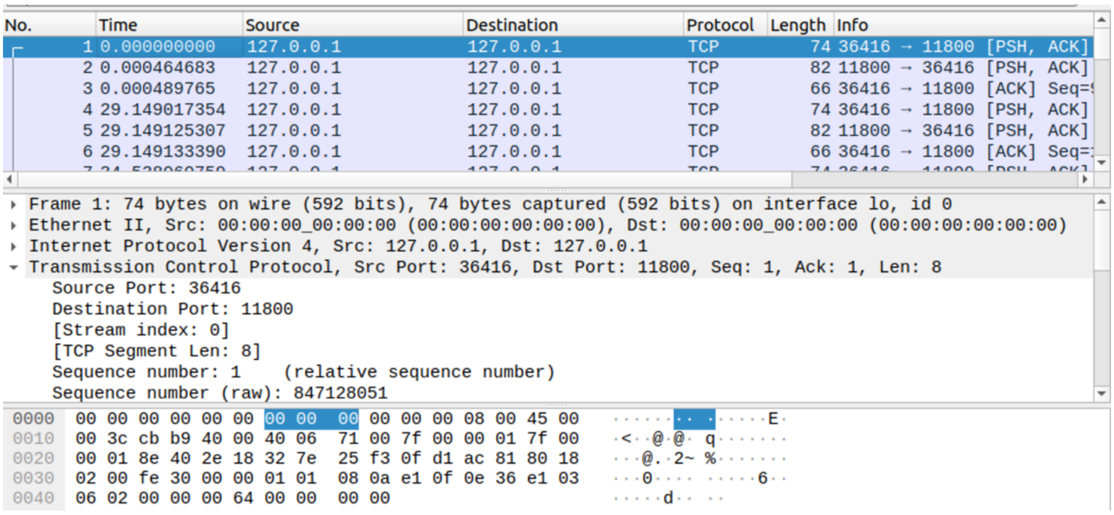
(4.) Now the game ends, close the socket

```
cerr << "The game ends";
    //now we close the socket
  (4.)
    status = close(sock);
    if (status < 0) {
        cerr << "Error with close" << endl;
        exit (-1);
    }
```

2. Wireshark observation

- **Capture the packets transmitted by the server and the client.**

Screenshots of capturing localhost packets:



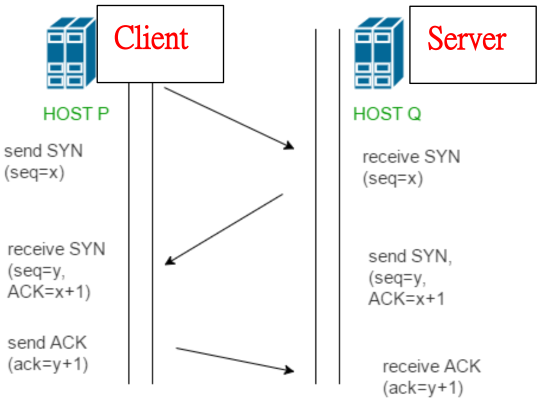
```
No.    Time          Source       Destination   Protocol Length Info
     1 0.000000000   127.0.0.1    127.0.0.1     TCP      74 36416 → 11800 [PSH, ACK]
     2 0.000464683   127.0.0.1    127.0.0.1     TCP      82 11800 → 36416 [PSH, ACK]
     3 0.000489765   127.0.0.1    127.0.0.1     TCP      66 36416 → 11800 [ACK] Seq=
     4 29.149017354  127.0.0.1    127.0.0.1     TCP      74 36416 → 11800 [PSH, ACK]
     5 29.149125307  127.0.0.1    127.0.0.1     TCP      82 11800 → 36416 [PSH, ACK]
     6 29.149133390  127.0.0.1    127.0.0.1     TCP      66 36416 → 11800 [ACK] Seq=
```

```
▸ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▾ Transmission Control Protocol, Src Port: 36416, Dst Port: 11800, Seq: 1, Ack: 1, Len: 8
    Source Port: 36416
    Destination Port: 11800
    [Stream index: 0]
    [TCP Segment Len: 8]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 847128051
```

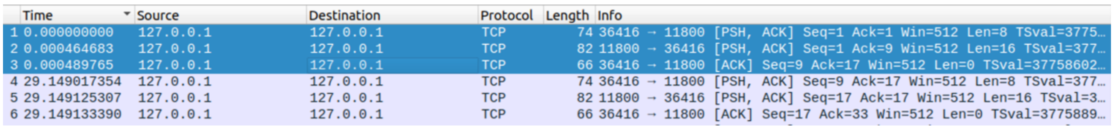
```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ········ ······E·
0010  00 3c cb b9 40 00 40 06  71 00 7f 00 00 01 7f 00   ·<··@·@· q·······
0020  00 01 8e 40 2e 18 32 7e  25 f3 0f d1 ac 81 80 18   ···@.·2~ %·······
0030  02 00 fe 30 00 00 01 01  08 0a e1 0f 0e 36 e1 03   ···0···· ·····6··
0040  06 02 00 00 00 64 00 00  00 00                     ·····d·· ··
```

- **Observations:**
    - **The packets used for TCP hand shaking:**

      For TCP 3-way hand shaking▼



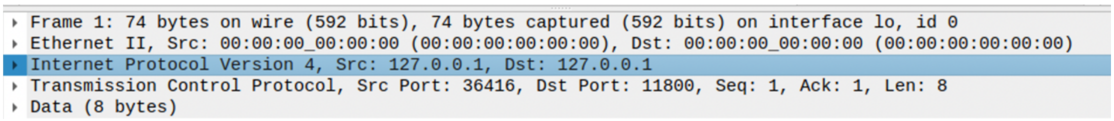
The packets used for hand shaking process▼



```
  Time           Source      Destination  Protocol Length Info
1 0.000000000    127.0.0.1   127.0.0.1    TCP      74 36416 → 11800 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=8 TSval=3775…
2 0.000464683    127.0.0.1   127.0.0.1    TCP      82 11800 → 36416 [PSH, ACK] Seq=1 Ack=9 Win=512 Len=16 TSval=377…
3 0.000489765    127.0.0.1   127.0.0.1    TCP      66 36416 → 11800 [ACK] Seq=9 Ack=17 Win=512 Len=0 TSval=37758602…
4 29.149017354   127.0.0.1   127.0.0.1    TCP      74 36416 → 11800 [PSH, ACK] Seq=9 Ack=17 Win=512 Len=8 TSval=377…
5 29.149125307   127.0.0.1   127.0.0.1    TCP      82 11800 → 36416 [PSH, ACK] Seq=17 Ack=17 Win=512 Len=16 TSval=3…
6 29.149133390   127.0.0.1   127.0.0.1    TCP      66 36416 → 11800 [ACK] Seq=17 Ack=33 Win=512 Len=0 TSval=3775889…
```

- **The server & client’s IP:**

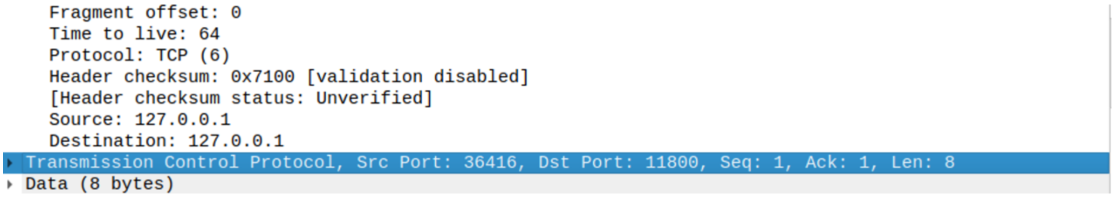
  On the same host, so the IPs are both 127.0.0.1(input by the user)



```
▸ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 36416, Dst Port: 11800, Seq: 1, Ack: 1, Len: 8
▸ Data (8 bytes)
```

- **The server & client’s port:**

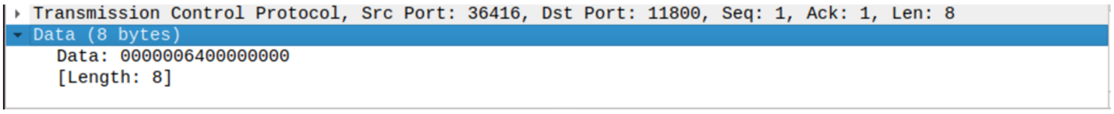
Client (Src) : 36416 Server (Dst) :11800



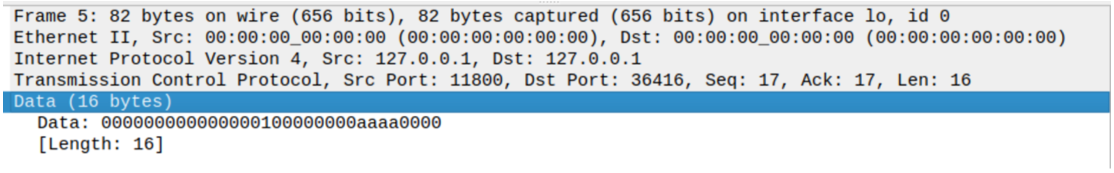
```
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x7100 [validation disabled]
    [Header checksum status: Unverified]
    Source: 127.0.0.1
    Destination: 127.0.0.1
▸ Transmission Control Protocol, Src Port: 36416, Dst Port: 11800, Seq: 1, Ack: 1, Len: 8
▸ Data (8 bytes)
```

- **The size of the packet transmitted by the client in bytes:**

8 or 16 bytes, depends on the guessing number input by the user



```
▸ Transmission Control Protocol, Src Port: 36416, Dst Port: 11800, Seq: 1, Ack: 1, Len: 8
▾ Data (8 bytes)
    Data: 0000006400000000
    [Length: 8]
```



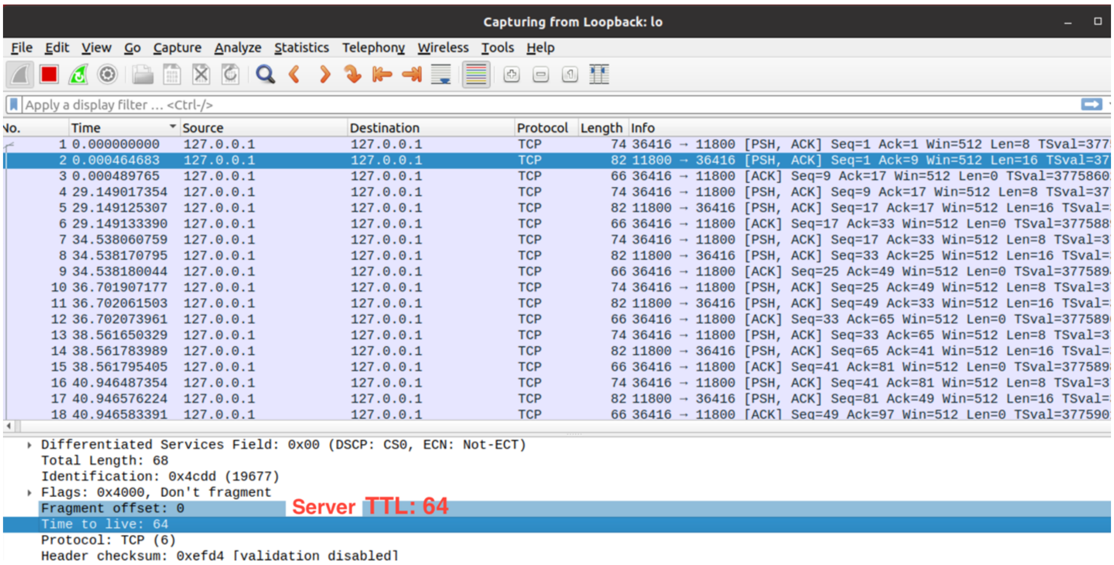
```
Frame 5: 82 bytes on wire (656 bits), 82 bytes captured (656 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 11800, Dst Port: 36416, Seq: 17, Ack: 17, Len: 16
Data (16 bytes)
  Data: 00000000000000001000000000aaaa0000
  [Length: 16]
```

- **The number of routers passed by the transmitted packet:**

Using the concept of Time-to-live (TTL): TTL refers to the amount of time or “hops” that a packet is set to exist inside a network before it is discarded by a router.

When the packet passes one router, the TTL will -1.

TTL of the server is 64, hence the number of routers goes to 64(default) – 64(current) = 0 (this also indicates that the hops are 0, TTL is unchanged)



Capturing from Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time          Source      Destination   Protocol  Length Info
    1 0.000000000   127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=1 Ack=1 Win=512 Len=8 TSval=377
    2 0.000464683   127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=1 Ack=9 Win=512 Len=16 TSval=37
    3 0.000489765   127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=9 Ack=17 Win=512 Len=0 TSval=3775860
    4 29.149017354  127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=9 Ack=17 Win=512 Len=8 TSval=37
    5 29.149125307  127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=17 Ack=17 Win=512 Len=16 TSval=
    6 29.149133390  127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=17 Ack=33 Win=512 Len=0 TSval=377588
    7 34.538060759  127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=17 Ack=33 Win=512 Len=8 TSval=3
    8 34.538170795  127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=33 Ack=25 Win=512 Len=16 TSval=
    9 34.538180044  127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=25 Ack=49 Win=512 Len=0 TSval=377589
   10 36.701907177  127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=25 Ack=49 Win=512 Len=8 TSval=3
   11 36.702061503  127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=49 Ack=33 Win=512 Len=16 TSval=
   12 36.702073961  127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=33 Ack=65 Win=512 Len=0 TSval=377589
   13 38.561650329  127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=33 Ack=65 Win=512 Len=8 TSval=3
   14 38.561783989  127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=65 Ack=41 Win=512 Len=16 TSval=
   15 38.561795405  127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=41 Ack=81 Win=512 Len=0 TSval=377589
   16 40.946487354  127.0.0.1   127.0.0.1     TCP        74 36416 → 11800 [PSH, ACK] Seq=41 Ack=81 Win=512 Len=8 TSval=3
   17 40.946576224  127.0.0.1   127.0.0.1     TCP        82 11800 → 36416 [PSH, ACK] Seq=81 Ack=49 Win=512 Len=16 TSval=
   18 40.946583391  127.0.0.1   127.0.0.1     TCP        66 36416 → 11800 [ACK] Seq=49 Ack=97 Win=512 Len=0 TSval=377590
```

```
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 68
    Identification: 0x4cdd (19677)
  ▸ Flags: 0x4000, Don't fragment
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0xefd4 [validation disabled]
```

Server TTL: 64

3. Challenges and solutions

a.

Challenge 1: Couldn't complete the client-server connection correctly, the port number couldn't be read by the server.

Solution 1: I used the function "strtoul()", skipping all chars in user's input string except numbers, and converting the port number into "unsigned short".

```
unsigned short portNum = (unsigned short)strtoul(argv[PORT_ARG], NULL, 0);
```

b.

Challenge 2: Couldn't send the judging (guessing) result to Client (Server) successfully

Solution 2: Like challenge 1, we need to convert the results into the type which is readable (by TCP Network)
I used "htonl()" to convert the result, and the problem was solved completely.

```
result = toNet(result);
```

```
Results toNet(Results toConvert)
{
    toConvert.tooHigh = htonl(toConvert.tooHigh);
    toConvert.tooLow = htonl(toConvert.tooLow);
    toConvert.correct = htonl(toConvert.correct);

    return toConvert;
}
```