

CS4150: Computer Networks Lab

Lab7

111901030

Mayank Singla

Q1. Virtual network for this lab has 3 VMs namely **h1** (192.168.1.1), **h2** (192.168.1.2), and **h3** (192.168.1.3). On machine **h1** write a program called a **client** that takes two arguments S and m , the first argument is a string, and the second argument is an integer. The program should output the binary (8-bit per character) of the string S . The program should then frame and code this data using an $(m + r, m)$ Hamming code (r should be the least possible for the given number of message bits m). Each Hamming code should be sent as a UDP packet to port X of **h2**, where $X \in [8560, 8570]$. **h2** will introduce a random 1-bit error in each Hamming code it receives and send the modified code to port Y of **h3**. On **h3**, write a program called a **server** that accepts codes with a 1-bit error sent by **h2**, corrects the error, assembles all the corrected codes to obtain string S , prints S with a new line, and waits for the following string.

```
cs4150@aha-acd9fl-0581:~/Downloads/lab7_network$ ./connect.sh h1
spawn ssh -p 14501 -o StrictHostKeyChecking=no tc@localhost
tc@localhost's password:
( '>')
/) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
(/-__-__-\)      www.tinycorelinux.net

tc@h1:~$ sudo nmap -p 8560-8570 -sU 192.168.1.2

Starting Nmap 6.40 ( http://nmap.org ) at 2022-10-19 08:11 UTC
Nmap scan report for 192.168.1.2
Host is up (0.00064s latency).
PORT      STATE      SERVICE
8560/udp   closed     unknown
8561/udp   closed     unknown
8562/udp   closed     unknown
8563/udp   closed     unknown
8564/udp   closed     unknown
8565/udp   closed     unknown
8566/udp   closed     unknown
8567/udp   open|filtered unknown
8568/udp   closed     unknown
8569/udp   closed     unknown
8570/udp   closed     unknown
MAC Address: 08:00:27:FB:88:E4 (Cadmus Computer Systems)

Nmap done: 1 IP address (1 host up) scanned in 17.61 seconds
tc@h1:~$
```

Running **nmap** with the UDP protocols in the given protocol range (as in the above screenshot) says the open port on machine **h2** is **8567**

Similarly, I tried to find the open UDP port on the machine **h3** to find port **Y**, but **nmap** was not able to find any and said that all the ports were closed.

```
tc@h1:~$ sudo nmap -sU 192.168.1.3

Starting Nmap 6.40 ( http://nmap.org ) at 2022-10-19 08:14 UTC
Nmap scan report for 192.168.1.3
Host is up (0.00095s latency).
All 1000 scanned ports on 192.168.1.3 are closed
MAC Address: 08:00:27:47:0D:B8 (Cadmus Computer Systems)

Nmap done: 1 IP address (1 host up) scanned in 1097.67 seconds
tc@h1:~$
```

Therefore, I set up only the client code on machine **h1** to send any random data (not the actual data we want to send in the question) to the machine **h2** on the found port **X**, which will then send that UDP packet to the port on machine **h3**

Then, I ran the **tcpdump** command on machine **h3** to check for all the interfaces using the filter for the source host by providing the IP address of machine **h2** and also disabled the DNS conversion to see the actual IP address and the port.

This way on sending the packet from the client from **h1**, I will be able to see live packets coming on **h3** and will be able to find the port **Y** on which the machine **h2** forwards the UDP packet

```
cs4150@aha-acdgfl-0581:~/Downloads/lab7_network$ ./connect.sh h3
spawn ssh -p 14503 -o StrictHostKeyChecking=no tc@localhost
tc@localhost's password:
( '>')
/) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
(/-__-__-\)      www.tinycorelinux.net

tc@h3:~$ sudo tcpdump src 192.168.1.2 -i any -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
```

```
tc@h1:~$ ./client "Thank You!" 5
tc@h1:~$
```

We can see in the below screenshot the packet coming from the machine **h2** depicted by the output message: **192.168.1.2.8567 > 192.168.1.3.9567**

This tells that the packet was sent from the found port **X** of machine **h2** to port **9567** of **h3**

Hence, the value of **Y** is **9567**

```
cs4150@aha-acdgfl-0581:~/Downloads/lab7_network$ ./connect.sh h3
spawn ssh -p 14503 -o StrictHostKeyChecking=no tc@localhost
tc@localhost's password:
( '>')
/) TC (\   Core is distributed with ABSOLUTELY NO WARRANTY.
(/-_-_-_)   www.tinycorelinux.net

tc@h3:~$ sudo tcpdump src 192.168.1.2 -i any -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
09:08:20.052640 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 2
09:08:25.078305 ARP, Request who-has 192.168.1.3 tell 192.168.1.2, length 46
09:08:25.162939 ARP, Reply 192.168.1.2 is-at 08:00:27:fb:88:e4, length 46
^C
3 packets captured
3 packets received by filter
0 packets dropped by kernel
tc@h3:~$
```

The division of the lab can be tested by the final working implementation of my client and server code which satisfies all the requirements.

Explanation of the Client Code

I am using the **4B/5B** encoding to do the precoding of the message and also using an unused 5B code as the frame delimiter. So, I created the map **b4to5** having the 4B to 5B code mappings (This mapping is the same as used in the slides for this lecture).

I created the function **gcd()** to find the gcd of two numbers, which I can then use to calculate the lcm of two numbers.

I then created the function **getMinRedundantBits()** to get the minimum value of **r** that satisfies the inequality: $(m + r + 1) \leq 2^r$ which can be obtained using a simple while loop.

I then created the function **getBinaryMsg()** which returns the binary (8-bit per character) representation of a given string. I am using the built-in **bitset** library to get the binary representation of each ASCII value of the characters and appending them all to get the final string.

I then created the function **getMinPadBits()** to get the minimum number of bits that needs to be padded to the binary string of a given length and a given number of message bits in the frame. The logic for this function is as follows:

Let's say the input length of the binary string is **s** and we want to pad **x** bits

- We want $(s + x)$ to be divisible by 4 so that they can nicely be grouped for encoding
- We want $(s + x)$ to be divisible by 8 so that they can nicely be decoded at the server
- We want $((((s + x) / 4) * 5) + 5 + 5)$ (which is the length of the string obtained after padding, 4B/5B encoding, and framing) to be divisible by m (input message bits) so that they can nicely be grouped for sending the packets in the size of $n (= m + r)$

Padding in this way makes it a lot easier for the server to reverse all the steps to obtain the original message correctly.

I then created the function **padBinaryString()** which pads the given binary string with 0's as many times as given in the input.

I then created the function **preCodeMsg()** which encodes the given string using 4B/5B encoding. The logic is simple in that it takes a group of 4 bits as a key for the map, and appends all the values to obtain the encoded string.

I then created the function **frameMsg()** which frames the message with the **frame delimiter** at the start and the end. The choice of the frame delimiter was made in such a way that it does not overlap with the 5B encodings at the start and the end, so I chose it as **00001**.

I then created the function **getHammingCode()** which returns the hamming code for the given binary string of length m . It uses the standard algorithm learned in lectures to compute the parity bits to construct the final hamming code.

Then, in my **main()** function, I am reading the inputs from the command line. I am restricting the size of the frame to a maximum length, as the server will read the data up to a maximum length. Then I am reading the server info using the **getaddrinfo()** function by providing it the IP and port of the machine **h2**. To the hints of this function, I am using the domain as **AF_INET** for IPv4, socket type as **SOCK_DGRAM** for UDP packets, and **AI_PASSIVE** to use the client's IP address.

Then, in the linked list of results obtained, I am trying to create a socket from the first result I can, which has the same domain, type, and protocol as that of the server using the standard **socket()** system call. Then using all the above functions, I am constructing my **framedMsg** first which is padded, encoded using 4B/5B encoding, and framed using frame delimiter.

Then, in a for loop, I am taking a group of m bits at a time from this framed message and sending the hamming code of size $n (= m + r)$ to the server using the **sendto()** system call used for UDP packets.

Finally, I am freeing up the resources used by the structure of the server info and closing the socket file descriptor for the created socket.

Explanation of the Server Code

This time I am having the map **b5tob4** to perform the reverse **4B/5B** encoding having the same mappings as present on the server side.

I created a function **get_in_addr()** which converts a **sockaddr** struct type to either **sockaddr_in** type or **sockaddr_in6** type based on whether it is using IPv4 (**AF_INET** family) or IPv6 (**AF_INET6** family).

I then created two helper functions **startsWith()** and **endsWith()** which checks if a given string starts/ends with another given string or not.

I then created the function **binToDec()** which converts a binary string to its decimal values. I used the built-in **bitset** library to do so.

I then created the function **correctHammingCode()** which corrects the input hamming code having a 1-bit error. The algorithm is standard which is to compute the parity bits again and build the syndrome to get the faulty bit.

I then created the function **extractMsg()** which extracts the message bits from the input hamming code. The message bits are present at the index positions which are not powers of 2.

I then created the function **deFrameMsg()** which de-frames the input message by simply removing the frame delimiter from the start and the end of the input string

I then created the function **decodeMsg()** which decodes the input message using the reverse **4B/5B** mappings by simply taking a group of 5 bits at a time and using the map **b5tob4** to get the 4B values and appending them all.

I then created the function **printMsg()** which prints the final received message by converting the input binary (8-bit per character) to its ASCII representation. It prints the message sent from the client with a new line.

In my **main()** function, I am first setting up the standard code to set up the server socket. I am calling the **getaddrinfo()** function with the port of the server to get its information and the hints argument is created in the same way as that for the client. Then, in the same way as that on the client, I am creating a socket and this time also binding the socket to the address and port of the server so that the server is ready to accept packets on this socket. Finally, I am freeing up the resources used by the structure of the server info as it is no longer needed.

Then, I am creating the **clientAddr** struct of type **socket_storage** to populate the client information received when the packet arrives. I am also creating a buffer for reading the input data and a buffer to print the client's IP address.

Then, I am having a variable **msg** which will be the final binary message received from the client assembled, and a variable **startFrame** which depicts the start of a frame initiated.

Then, in an infinite while loop, I am reading the packets from the client using the **recvfrom()** system call for UDP packets which is a blocking call until the message arrives. After the message is read, it returns the number of bytes read. So, at the end of the buffer after these many bytes, I am putting null character '\0' to mark it as the end of the string (since my buffer created is of size up to the maximum capacity the server can read).

Then, using the above-created functions, I am correcting the hamming code, extracting the message bits out of the corrected code, and appending the message bits to the **msg** variable.

I am checking that as soon as the length of the total message received is greater than the length of the frame delimiter, I am checking if the message starts with the frame delimiter or not, which should be the ideal case and should always occur and then marking the **startFrame** variable as true.

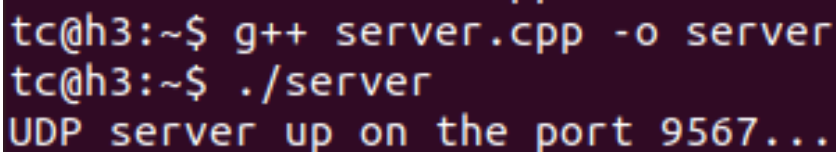
I am also checking that if the frame is started and the total message length is greater than twice the length of the frame delimiter and if it also ends with the frame delimiter, then I have successfully obtained one complete message from the client. So, I am de-framing and decoding the message and printing its ASCII representation. Also, when this happens, I am marking the **startFrame** variable as false and the **msg** variable as an empty string to read the next message.

Finally, I am closing the socket file descriptor for the created socket.

Here are the screenshots attached for the working demo.

Example 1.

Starting the server on machine **h3**



```
tc@h3:~$ g++ server.cpp -o server
tc@h3:~$ ./server
UDP server up on the port 9567...
```

Sending the message from the client. The client prints the binary, padded, pre-coded and framed representation of the input message. It also prints the sent frames containing the corresponding hamming codes.

```
tc@hl:~$ g++ client.cpp -o client
tc@hl:~$ ./client "Hello, World!" 10
Input Message: Hello, World!
Length: 13

Binary representation (8-bit per character): 01001000011001010110110001101100011011110010110000100000010101110110111101110010011011000110010000100001
Length: 104

Number of padding bits required: 0
Padded message: 010010000110010101101100011011100010110000100000010101110110111101110010011011000110010000100001
Length: 104

Pre-coded String: 01010100100111001011011101101001110111011010011010100111100101101111011101110111101000111011010011100101010001001
Length: 130

Framed String: 000010101001001110010110111011010011101110110011010100111100101101110111011101111010001110110100111001010100010010001
Length: 140

No. of message bits: 10
No. of check bits : 4
No. of frame bits : 14

Sent frame to "192.168.1.2" on Port 8567 with content: 11010001101010 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 01110011001110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 10001010101110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11101011001110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11101011001110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 1111101110100 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11111010010100 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 1111111001011 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11011110101110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 10111101101111 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11110101001110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 11101011001110 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 00011010010100 of length: 14
Sent frame to "192.168.1.2" on Port 8567 with content: 01001000100001 of length: 14
tc@hl:~$
```

This is the actual output of the submitted code which prints the client and packet information on receiving the packet. It also prints the corrected hamming code for each packet received. At the end, when the message is fully arrived, it prints the message with a new line.

```
tc@h3:~$ ./server
UDP server up on the port 9567...
Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11011001101010"
Corrected Packet "11010001101010"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "01110011011110"
Corrected Packet "01110011001110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "10001110101110"
Corrected Packet "10001010101110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "10101011001110"
Corrected Packet "11101011001110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11101111001110"
Corrected Packet "11101011001110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11111001110100"
Corrected Packet "11111101110100"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "10111010010100"
Corrected Packet "11111010010100"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11011111001011"
Corrected Packet "11111111001011"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "01011110101110"
Corrected Packet "11011110101110"
```



```
Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "10111100101111"
Corrected Packet "10111101101111"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11110101101110"
Corrected Packet "11110101001110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "11111011001110"
Corrected Packet "11101011001110"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "00011011010100"
Corrected Packet "00011010010100"

Got packet from "192.168.1.2"
Packet is 14 bytes long
Packet contains "01001100100001"
Corrected Packet "01001000100001"

Message Received: Hello, World!
```

We can see that the server is able to receive the message successfully.

Example 2.

I commented the lines printing the client and packet information on the server code for demo purpose.

Sending the first packet.

```
tc@h1:~$ ./client "Hi\n" 5
Input Message: Hi\n
Length: 4

Binary representation (8-bit per character): 01001000011010010101110001101110
Length: 32

Number of padding bits required: 0
Padded message: 01001000011010010101110001101110
Length: 32

Pre-coded String: 0101010010011101001101011110100111011100
Length: 40

Framed String: 00001010101001001110100110101111010011101110000001
Length: 50

No. of message bits: 5
No. of check bits : 4
No. of frame bits : 9

Sent frame to "192.168.1.2" on Port 8567 with content: 1000000011 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 010010100 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 001100100 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 000111100 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 101100111 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 110010111 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 101010100 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 000111100 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 001011000 of length: 9
Sent frame to "192.168.1.2" on Port 8567 with content: 1000000011 of length: 9
```

Sending the second packet

```
tc@hl:~$ ./client "Bye\n" 100
Input Message: Bye\n
Length: 5

Binary representation (8-bit per character): 0100001001111001011001010101110001101110
Length: 40

Number of padding bits required: 32
Padded message: 00000000000000000000000000000000100011001110010110010101110001101110
Length: 72

Pre-coded String: 111101111011110111101111011110111100101010100011110011011100101101011110100111011100
Length: 90

Framed String: 000011110111101111011110111101111011110010101010001111001101110010110101111010011101110000001
Length: 100

No. of message bits: 100
No. of check bits : 7
No. of frame bits : 107

Sent frame to "192.168.1.2" on Port 8567 with content: 0000000011110101110111101111011111011110111100101010100011111001101110010110101111010011101110000001 of length: 107
```

We can see that the server is able to receive both the messages successfully.

```
tc@h3:~$ ./server
UDP server up on the port 9567...
Message Received: Hi\n
Message Received: Bye\n
█
```