

CS4150 Computer Networks Laboratory – Assignment 7

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

Aim

This lab aims to simulate the hamming codes along with other processes such as padding, pre-coding and framing in the data-link layer.

Finding the port X of h2 and Y of h3

Figure 1: We can see that port 8567 has an open UDP process running. (X=8567)

```
PowerShell
tc@h1:~$ sudo nmap -sU -p8560-8570 192.168.1.2

Starting Nmap 6.40 ( http://nmap.org ) at 2021-10-19 05:28 UTC
Nmap scan report for 192.168.1.2
Host is up (0.00043s latency).
PORT      STATE      SERVICE
8560/udp  closed
8561/udp  closed
8562/udp  closed
8563/udp  closed
8564/udp  closed
8565/udp  closed
8566/udp  closed
8567/udp  open|filtered
8568/udp  closed
8569/udp  closed
8570/udp  closed
MAC Address: 08:00:27:FB:88:E4 (Cadmus Computer Systems)
```

Figure 2: By using tcpdump on h3 and connecting to h2 from h1 using netcat, we can monitor the incoming UDP packets from h2 to know the port on which the server has to run. (Y = 9567)

```
PowerShell
tc@h1:~$ nc -u 192.168.1.2 8567
10
10001001
1001010
10101
101110100
█

tc@h3:~$ sudo tcpdump -nn src 192.168.1.2 -i any and udp
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
20:55:50.322908 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 3
20:56:22.567354 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 9
20:56:24.967239 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 8
20:56:26.203602 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 6
20:56:28.667185 IP 192.168.1.2.8567 > 192.168.1.3.9567: UDP, length 10
```

In Figure 2,

On the left side, we connect the client, through UDP, to h2 (192.168.1.2) on port 8567 that was obtained in Figure 1.

On the right side, the server starts listening to packets from h2 (*src 192.168.1.2*) on any interface (*-i any*) and it listens to udp packets only.

As we send a message from the client, the message is received by the server (h3) and by looking at the logs, we can see that h3 receives the packets on port **9567**.

Client

File: *client.cpp*

From Figure 1, we know that the port on which h2 is listening is 8567 i.e **X = 8567**.

For framing, I have used the “Flag bytes with byte stuffing” method which adds special frames namely *FLAG* and *ESC* at appropriate locations. I have also used post-padding to maintain the length of all the frames to a constant length. This post-padding can be easily removed on the server-side.

Terminologies

1. *FLAG*: The special flag data as a binary string. (Value is 1)
2. *ESC*: The special esc data as a binary string. (Value is 0)
3. *m*: The length of the message bits.
4. *r*: The number of redundant check bits based on *m*.
5. *n*: The total frame size that is passed along (*m*+*r*).

Requirements

1. Pad the frame (including *FLAG* and *ESC*) to the input message size.
2. Frame the bits using byte-stuffing to a length of *m* + *r*.

Functions

1. `get_hamming_code`: Returns the hamming code of an input binary string.
 - a. Initializes a variable *hamming_code* with zeros.
 - b. First, fill all the data bits to their respective positions.
 - c. Now for each parity bit position,
 - i. Check all the positions in which this check-bit can occur when that position is expressed as the sum of powers of 2.
 - ii. If the modulo 2 sum of all those positions results in 1, set the current position in hamming code to 1, else set it to 0.
 - d. Return the resultant hamming code.

Algorithm

1. The initial set of lines create a socket, store it in the socket file descriptor and inform h1 the IP address and port on which h2 is listening.
2. Now the number of redundant bits are calculated using the input *m* value and the *FLAG* and *ESC* frames are encoded using hamming codes.
3. The string is converted to its 8-bit binary form as required in the question.
4. The client first sends the encoded *FLAG* frame to indicate the start of a message.
5. For each of the consecutive *m* data bits, I do the following:
 - a. Pad the binary string with zeroes at the end to match the data size *m*.
 - b. If the resulting string is equal to either the *FLAG* or *ESC* data, then we stuff the data bits with *ESC* data bits before it. So if the data bits consist of [*FLAG*], then it becomes [*ESC*][*FLAG*] and if it is [*ESC*], it becomes [*ESC*][*ESC*]. This additional encoded *ESC* frame is sent to the server.
 - c. Now the data-bits that were obtained in 5.a are encoded using Hamming codes using the function `get_hamming_codes` and sent over to the server through h2.
6. Once all the data bits are sent to h2, I send another encoded *FLAG* frame indicating the end of a message.

Server

File: server.cpp

From Figure 2, we know that the port to which h2 sends(to h3) is 9567 i.e **Y = 9567**.

Terminologies

1. FLAG: The special flag data as a binary string. (Value is "1")
2. ESC: The special esc data as a binary string. (Value is 0)

Requirement

Decode the original message by removing errors and pad bits from the received frames.

Functions

1. `correct_hamming_code`: Given an input binary string, it finds the 1-bit error and flips it.
 - a. For each parity bit position,
 - i. Check all the positions in which this check-bit can occur when that position is expressed as the sum of powers of 2.
 - ii. If the modulo 2 sum of all those positions results in 1, append 1 to the syndrome else append 0 to it.
 - b. Here it reverses the syndrome, converts the binary string to integer, fixes the error and return the corrected hamming code.
2. `retrieve_message`: Given a binary string, it corrects the error and returns the data bits.
 - a. Correct all the single-bit errors using the above function.
 - b. Iterate over all data bit positions, add them to a string and return the string.
3. `binary_to_string`: Converts a binary string to the actual string input by the user by iterating over every 8 bits of the string.

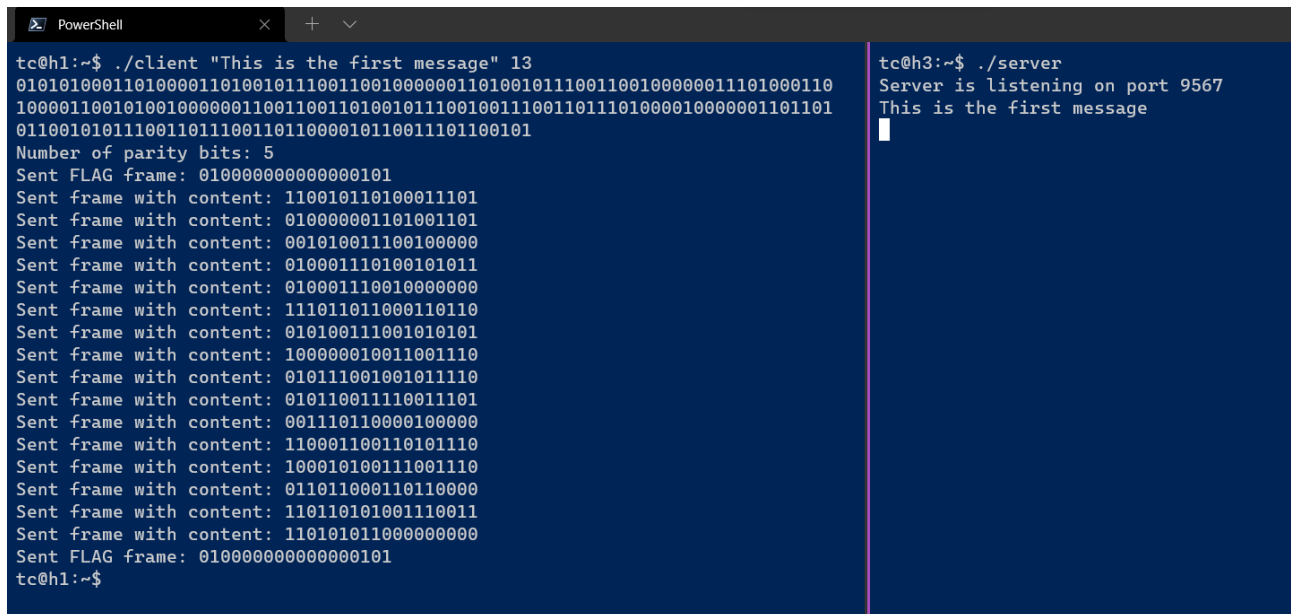
Algorithm

1. The initial set of lines create a socket, store it in the socket file descriptor and inform h3 the IP address and port on which h2 will send the frames to it.
2. The server at h3 has to bind to a socket and listen to incoming frames from h2, this is done in the next few lines using the *bind* method.
3. The *prev_byte* variable stores the previous frame that was seen by the server and the *isMessage* variable helps the server in identifying the start and end of the message.
4. The server in a busy waiting manner does the following:
 - a. Try to receive a frame from h2.
 - b. If it has received a frame, it corrects the errors, removes the padded bits and returns the original data bits that were sent to h2 by h1.
 - c. If this frame turns out to be a *FLAG* frame and the *isMessage* variable is false, it indicates the start of a message and if the *isMessage* is true, it indicates the end of a message. At the end of the message, the original message is printed.
 - d. If the received frame is an *ESC* frame, I ignore this frame and read the next *FLAG* or *ESC* frame into the message field. This is because an *ESC* frame was stuffed by the receiver to avoid ambiguities.
 - e. If the received message is not a special frame, it is appended to the end of the message.

The server can identify the start and end of the messages through the *FLAG* frames. If a *FLAG* frame is preceded by an *ESC* frame, then it belongs to the message, otherwise, it is either the start or end of a message which can be decided based on a local flag.

Results

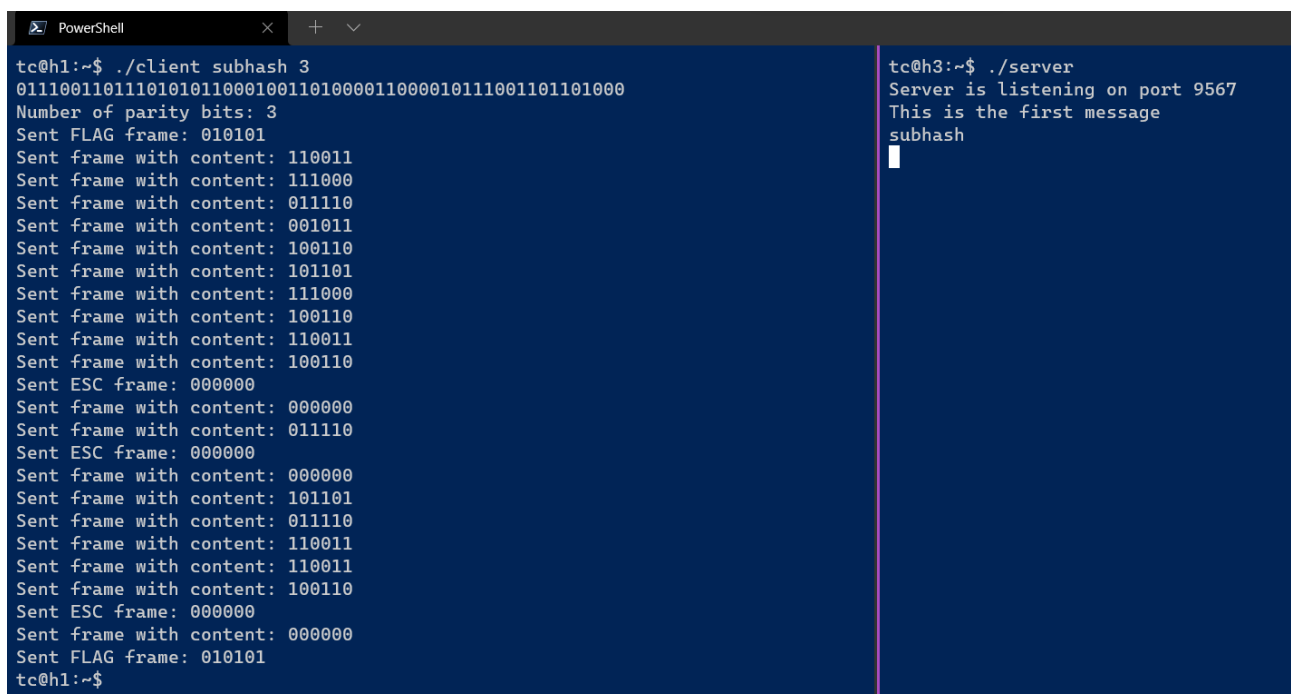
Figure 3: Client sends a message through h2 which is received by h3 and decoded



```
tc@h1:~$ ./client "This is the first message" 13
010101000110100001101001011100110010000001101001011100110010000001101000110
100001100101001000000110011001101001011100100111001101101000010000001101101
011001010111001101110011011000010110011101100101
Number of parity bits: 5
Sent FLAG frame: 010000000000000101
Sent frame with content: 110010110100011101
Sent frame with content: 010000001101001101
Sent frame with content: 001010011100100000
Sent frame with content: 010001110100101011
Sent frame with content: 010001110010000000
Sent frame with content: 111011011000110110
Sent frame with content: 010100111001010101
Sent frame with content: 100000100110011110
Sent frame with content: 010111001001011110
Sent frame with content: 010110011110011101
Sent frame with content: 001110110000100000
Sent frame with content: 110001100110101110
Sent frame with content: 100010100111001110
Sent frame with content: 011011000110110000
Sent frame with content: 110110101001110011
Sent frame with content: 110101011000000000
Sent FLAG frame: 010000000000000101
tc@h1:~$

tc@h3:~$ ./server
Server is listening on port 9567
This is the first message
```

Figure 4: Demonstrates framing, byte-stuffing and the server waiting for the next string.



```
tc@h1:~$ ./client subhash 3
011100110111010110001001101000011000010111001101101000
Number of parity bits: 3
Sent FLAG frame: 010101
Sent frame with content: 110011
Sent frame with content: 111000
Sent frame with content: 011110
Sent frame with content: 001011
Sent frame with content: 100110
Sent frame with content: 101101
Sent frame with content: 111000
Sent frame with content: 100110
Sent frame with content: 110011
Sent frame with content: 100110
Sent ESC frame: 000000
Sent frame with content: 000000
Sent frame with content: 011110
Sent ESC frame: 000000
Sent frame with content: 000000
Sent frame with content: 101101
Sent frame with content: 011110
Sent frame with content: 110011
Sent frame with content: 110011
Sent frame with content: 100110
Sent ESC frame: 000000
Sent frame with content: 000000
Sent FLAG frame: 010101
tc@h1:~$

tc@h3:~$ ./server
Server is listening on port 9567
This is the first message
subhash
```

Note: In the above figures, the client is on the left side and the server is on the right side.

Conclusions

Sending a binary string from h1 and receiving the 1-bit error binary string at h3

As can be seen from the above figures and the code, h1 sends a message to h2 as multiple frames, h2 introduces a 1-bit error in each of those frames and sends it to h3. h3 then decodes the message, flips the error bits and prints it.

Encoding and correcting errors using Hamming codes

Encoding the message is done by the client, correcting errors and decoding it is done by the server.

Padding, pre-coding and framing

As can be seen from Figure 4, the client also sends some *ESC* bytes which is the same as the next frame, this shows the framing technique. All the frames being the same size also shows the padding used.

Encoding *S* as a binary character array at **h1** and recovering it at **h3**, printing *S* along with a new line at **h3** and waiting for the next string.

We can see in the first line of client logs (Figure 3) that the entire string is converted to it's binary character array before being sent to h2. We can also see that the server did not stop running after one message (in Figure 3), instead, it decoded the message, printed it and waits for the next message to come and as soon as the next message comes, it follows the same steps and this can be seen in Figure 4 where the client waited and printed the second message.