Ex. No. 6 **Hamming Code** **Karthik D**

6 September 2021 **UCS1511 - Networks Lab** **195001047**

Aim

To implement Hamming Code for Single Error Correction using C socket program with the help of TCP-based message transmission server

Question

Simulate Hamming Code error-correction using TCP, wherein:

The sender/server should perform the following:

- Read the input from a user (zero's and one's)
- Encoding a message by Hamming Code
    - Calculate the number of redundant bits
    - Position the redundant bits
    - Calculate the values of each redundant bit
- .Introduce error (single bit error or no error)
- Send the data to receiver

The receiver/client should do the following:

- Receive the data from the sender
- Check for any error by performing the following operations
    - Calculation of the number of redundant bits
    - Positioning the redundant bits
    - Parity checking
    - If any error, correct the error and display the original message.

Algorithms

(a) Server-side

Step 1: Create a network socket with parameters suitable for an end-point of TCP based communication

Step 2: Bind the socket to INADDR_ANY which is defined as a *zero* address, allowing the socket to be reachable by all active interfaces on the device. Set the port to a preset value, known to the targeted clients as well

Step 3: Set the socket status to passive i.e initiate listening on the socket to allow it to accept incoming connection requests

Step 4: Wait for a connection request from a client and accept the first such request. Save the file descriptor of the connection-socket. This will be used to communicate with the client

Step 5: Prepare a memory buffer to read and store messages from the connection.

Step 6: Start an infinite loop to perform the following operations,

i: Accept a bit-string from the user.

ii: If the input message is a connection termination string, terminate the loop.

iii: Calculate the number of redundant bits required for the input bit string size using the relation $2r \geq m + r + 1$, where m is the bit-string size and r is the minimum number of redundant bits

iv: Position the redundant bits at appropriate positions i.e at positions from the right, which are exact powers of 2 i.e 1, 2, 4, 8, … These are the 1st, 2nd, 3rd, 4th … redundant bits respectively

v: Set the value of redundant bits such that the ith redundant bit along with all bits in the string at positions having a 1 at the ith position from right in their binary representations exhibit an even parity

vi: Ask the user if an error has to be added to the message. If so, randomly XOR one of the bits in the encoded message with 1 to change the bit. Otherwise, keep the encoded message unchanged

vii: Use the *write()* system call to send this hamming encoded message to the client by writing to the client's socket stream

(Repeat till user chooses to terminate in step-(i)

Step 7: Close the created sockets using the *close()* system call and terminate the process

(b) **Client-side**

Step 1: Create a network socket with parameters suitable for an end-point of TCP based communication

Step 2: Accept the target server address as input from the user

Step 3: Using the accepted address and a preset port number agreed upon between the server and client, send a connection request to the server using the *connect()* system call

Step 4: Prepare a memory buffer to read and store messages from the connection.

Step 5: Start an infinite loop to perform the following operations,

i: Use the *read()* system call to block and read a message sent from the server by reading into the own socket, and store the message in the buffer

ii: If the server has already exited (indicated by a received message size of 0), terminate the loop

i: Calculate the number of redundant bits present in the received in the bit-string size using the relation $2r \geq m + r + 1$, where m is the bit-string size and r is the minimum number of redundant bits. The size of the received bit-string is hence (m+r)

ii: Read the redundant bits from the bit-string. Their positions are exact powers of 2 i.e 1, 2, 4, 8, … from the right side. These are the 1st, 2nd, 3rd, 4th … redundant bits respectively

iii: Convert the binary number obtained by the combination - ... r4-r3-r2-r1 to the decimal form, where ri represents the ith redundant bit. This decimal number represents the position of error in the received hamming encoded message, from the right.

iv: If the position value is 0, there is no error. Skip to step-(vi)

v: XOR the value at the position from right in the string with 1 to correct the error

vi: Remove the redundant bits from this corrected message

vii: Display the message to the user

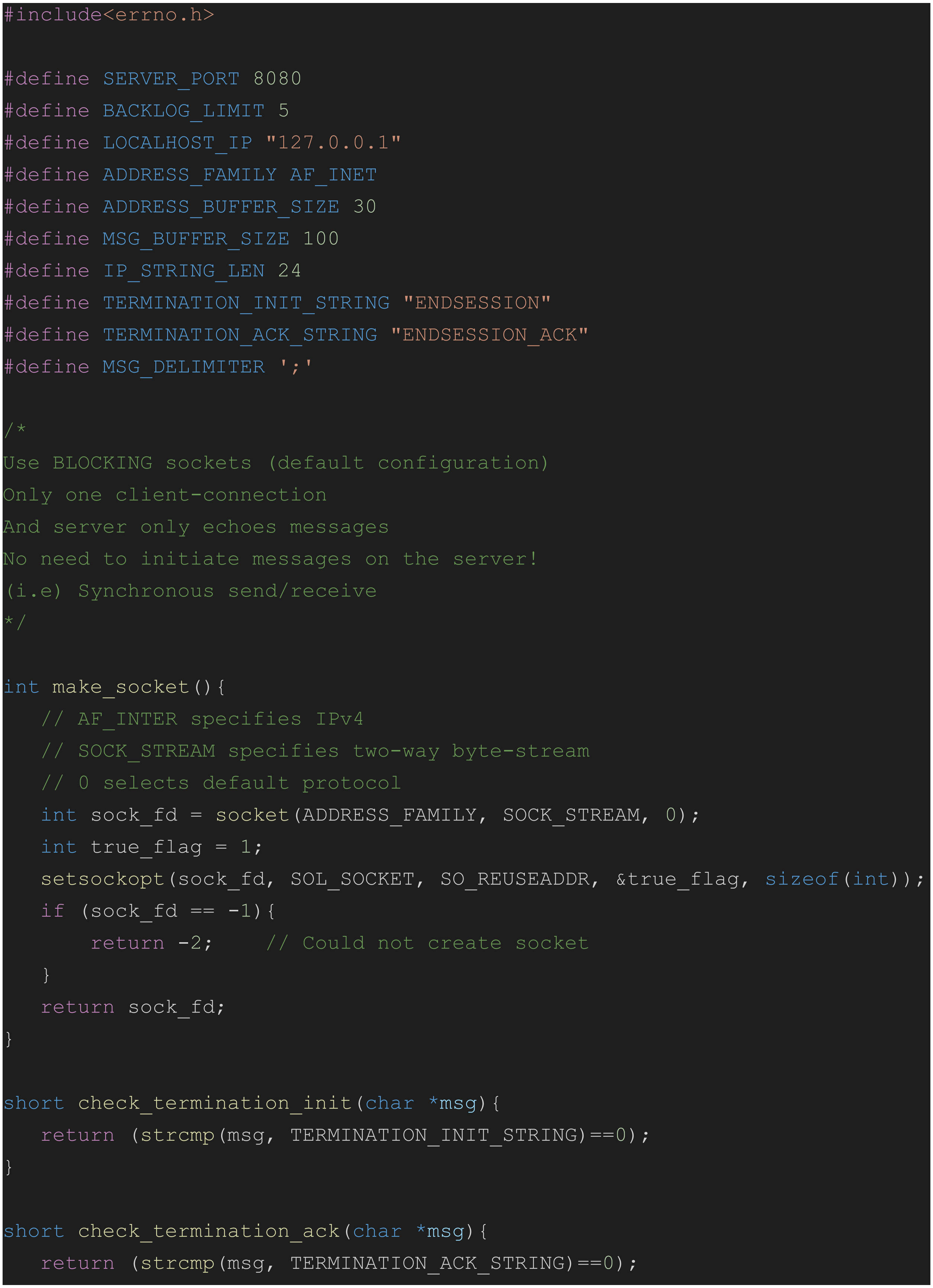
(Repeat till server termination is detected in step-(ii))

Step 6: Close the created sockets using the *close()* system call and terminate the process

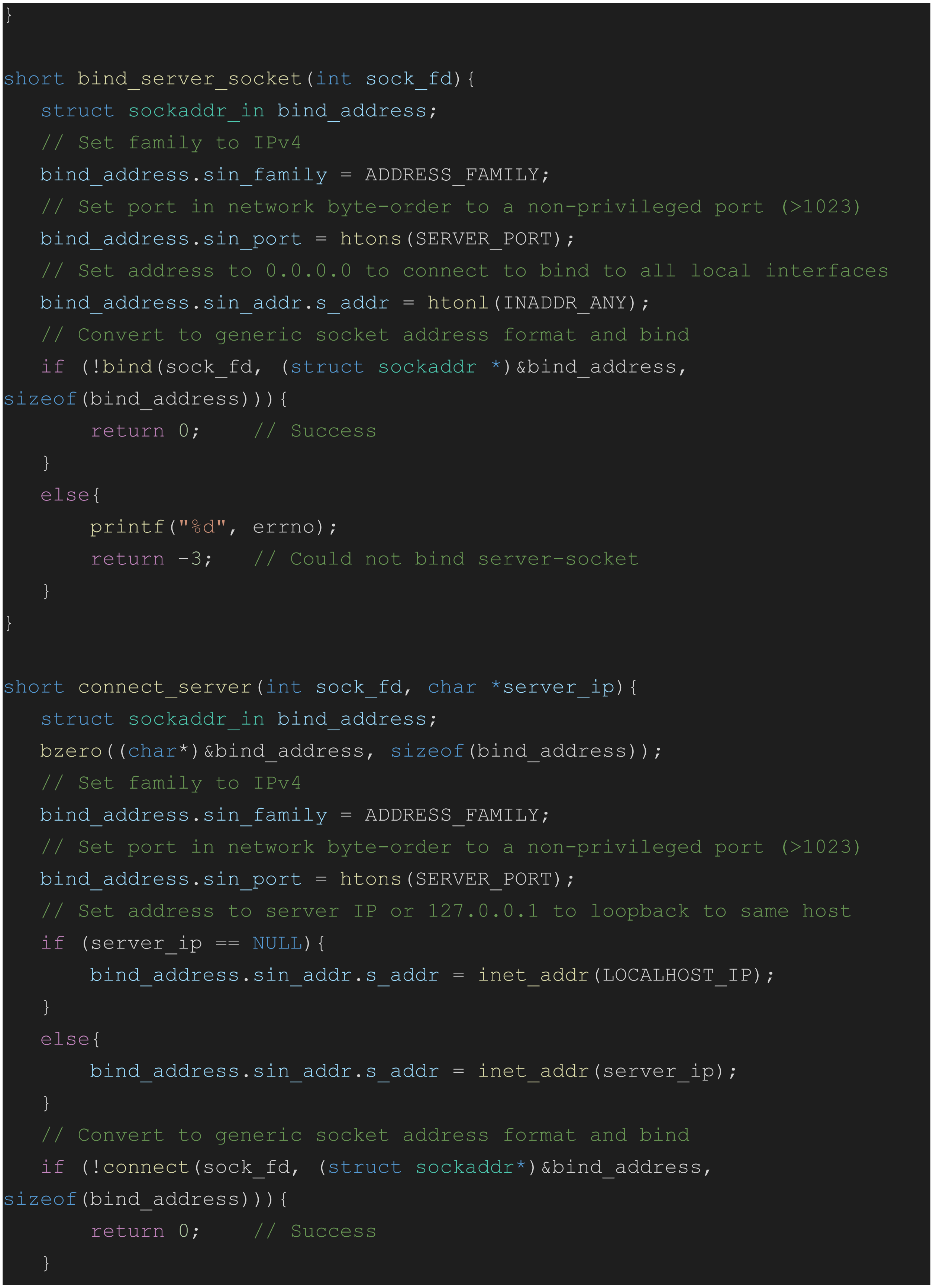
C Program Code

1. tcp_socket.h - TCP connection helper functions

```
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<string.h>
```



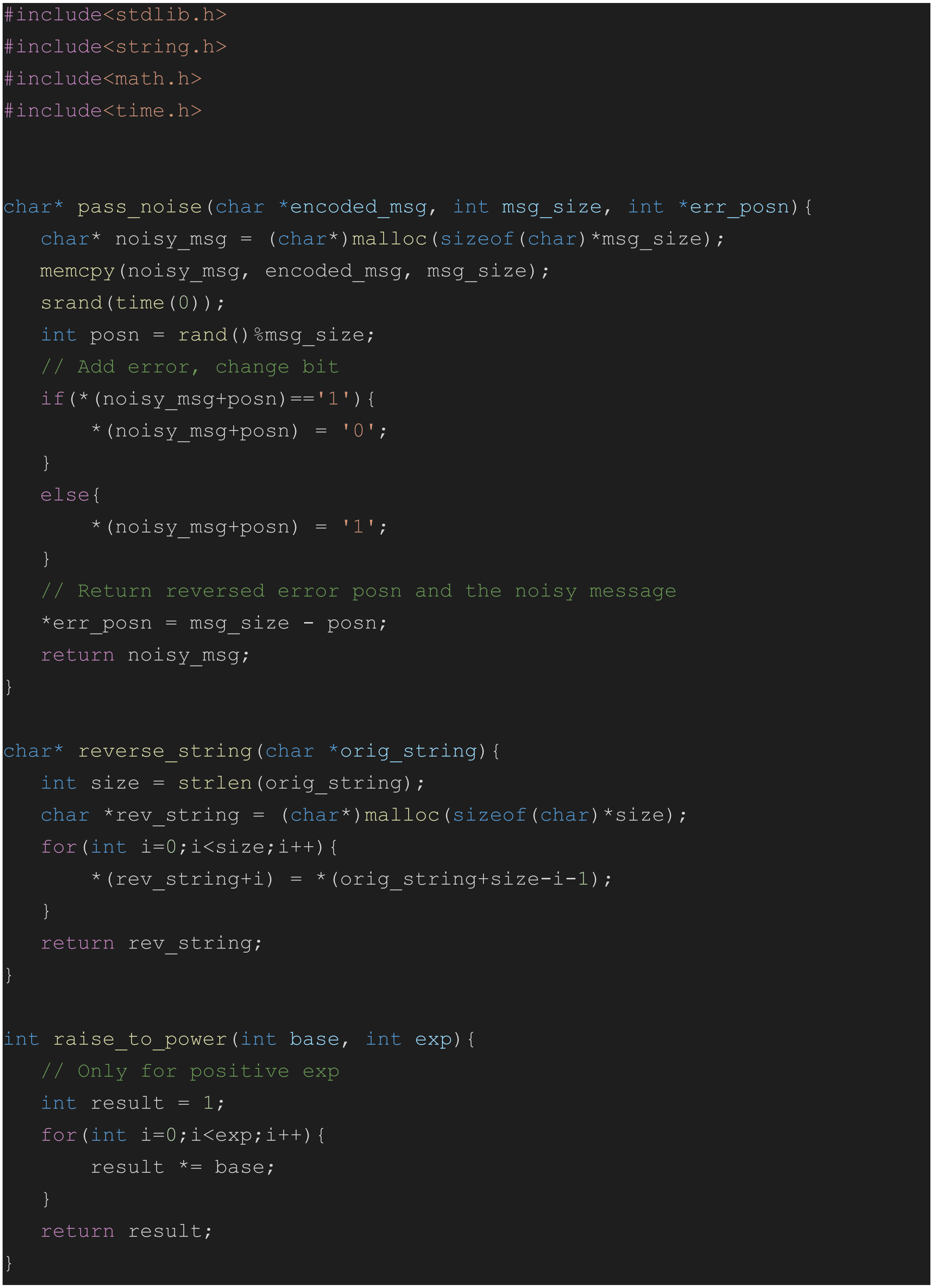
```c
#include<errno.h>

#define SERVER_PORT 8080
#define BACKLOG_LIMIT 5
#define LOCALHOST_IP "127.0.0.1"
#define ADDRESS_FAMILY AF_INET
#define ADDRESS_BUFFER_SIZE 30
#define MSG_BUFFER_SIZE 100
#define IP_STRING_LEN 24
#define TERMINATION_INIT_STRING "ENDSESSION"
#define TERMINATION_ACK_STRING "ENDSESSION_ACK"
#define MSG_DELIMITER ';'

/*
Use BLOCKING sockets (default configuration)
Only one client-connection
And server only echoes messages
No need to initiate messages on the server!
(i.e) Synchronous send/receive
*/

int make_socket(){
   // AF_INTER specifies IPv4
   // SOCK_STREAM specifies two-way byte-stream
   // 0 selects default protocol
   int sock_fd = socket(ADDRESS_FAMILY, SOCK_STREAM, 0);
   int true_flag = 1;
   setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, &true_flag, sizeof(int));
   if (sock_fd == -1){
       return -2;     // Could not create socket
   }
   return sock_fd;
}

short check_termination_init(char *msg){
   return (strcmp(msg, TERMINATION_INIT_STRING)==0);
}

short check_termination_ack(char *msg){
   return (strcmp(msg, TERMINATION_ACK_STRING)==0);
```



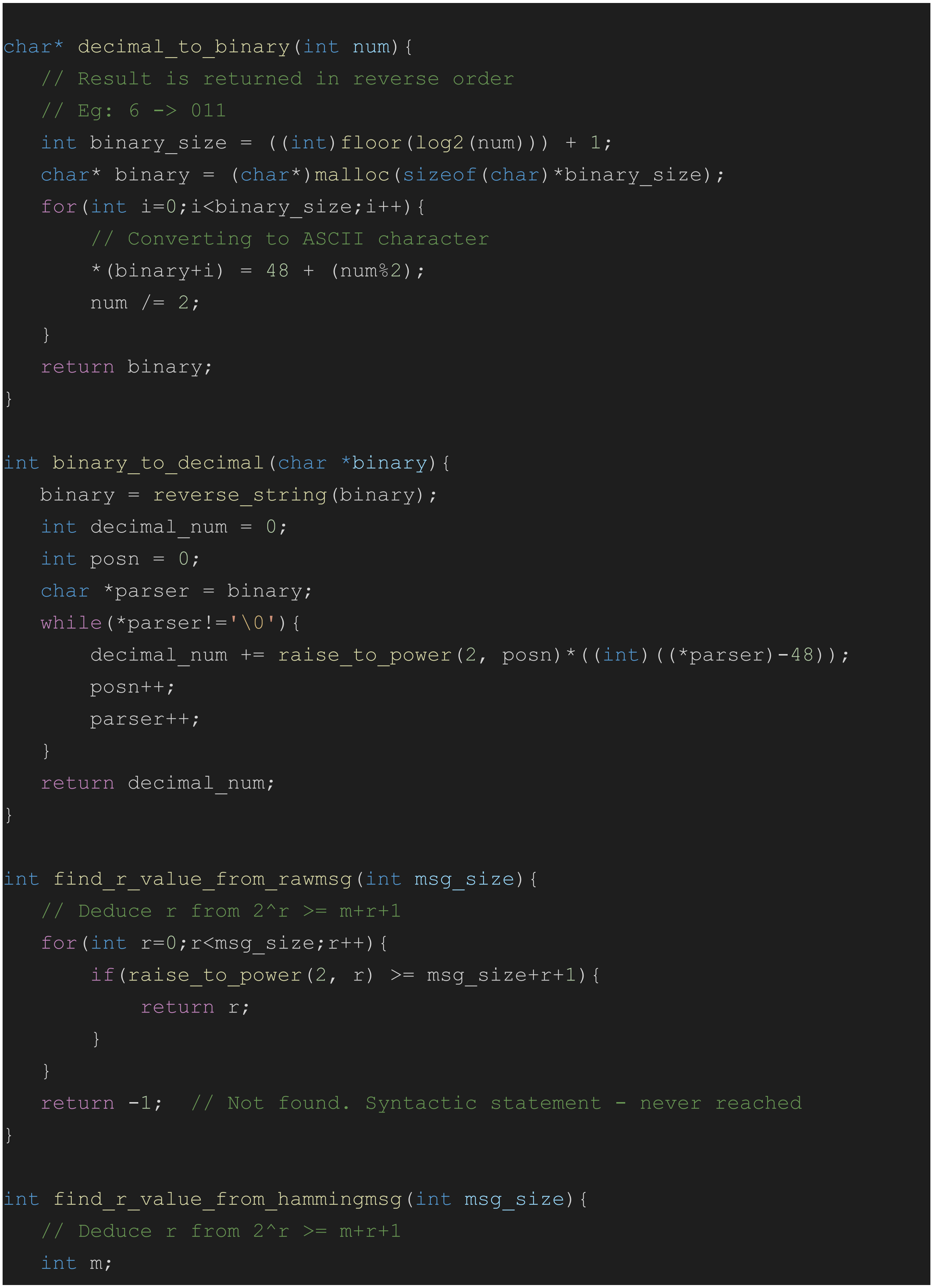
```
}

short bind_server_socket(int sock_fd){
    struct sockaddr_in bind_address;
    // Set family to IPv4
    bind_address.sin_family = ADDRESS_FAMILY;
    // Set port in network byte-order to a non-privileged port (>1023)
    bind_address.sin_port = htons(SERVER_PORT);
    // Set address to 0.0.0.0 to connect to bind to all local interfaces
    bind_address.sin_addr.s_addr = htonl(INADDR_ANY);
    // Convert to generic socket address format and bind
    if (!bind(sock_fd, (struct sockaddr *)&bind_address,
sizeof(bind_address))){
        return 0;       // Success
    }
    else{
        printf("%d", errno);
        return -3;      // Could not bind server-socket
    }
}

short connect_server(int sock_fd, char *server_ip){
    struct sockaddr_in bind_address;
    bzero((char*)&bind_address, sizeof(bind_address));
    // Set family to IPv4
    bind_address.sin_family = ADDRESS_FAMILY;
    // Set port in network byte-order to a non-privileged port (>1023)
    bind_address.sin_port = htons(SERVER_PORT);
    // Set address to server IP or 127.0.0.1 to loopback to same host
    if (server_ip == NULL){
        bind_address.sin_addr.s_addr = inet_addr(LOCALHOST_IP);
    }
    else{
        bind_address.sin_addr.s_addr = inet_addr(server_ip);
    }
    // Convert to generic socket address format and bind
    if (!connect(sock_fd, (struct sockaddr*)&bind_address,
sizeof(bind_address))){
        return 0;       // Success
    }
    else{
        return -5;    // Could not connect to the local server
    }
}

short initiate_listen(int sock_fd){
    if (!listen(sock_fd, BACKLOG_LIMIT)){
        return 0;     // Success
    }
    else{
        return -4;    // Could not initiate listening
    }
}

// BLOCKING FUNCTION
int accept_client(int sock_fd, struct sockaddr_in *client_addr, int
*client_addr_len){
    // Address of client is registered
    int client_sock_fd = accept(sock_fd, (struct sockaddr*)client_addr,
client_addr_len);
    if (client_sock_fd == -1){
        return -5;    // Did not connect to a client
    }
    else if (*client_addr_len > sizeof(struct sockaddr_in)){
        *client_addr_len = -1;     // Non-Fatal Warning: Client address was
truncated to fit in buffer
    }
    return client_sock_fd;
}

void destroy_socket(int sock_fd){
    close(sock_fd);
}
```

2. hamming_code.h - Hamming code encoding and decoding helper functions

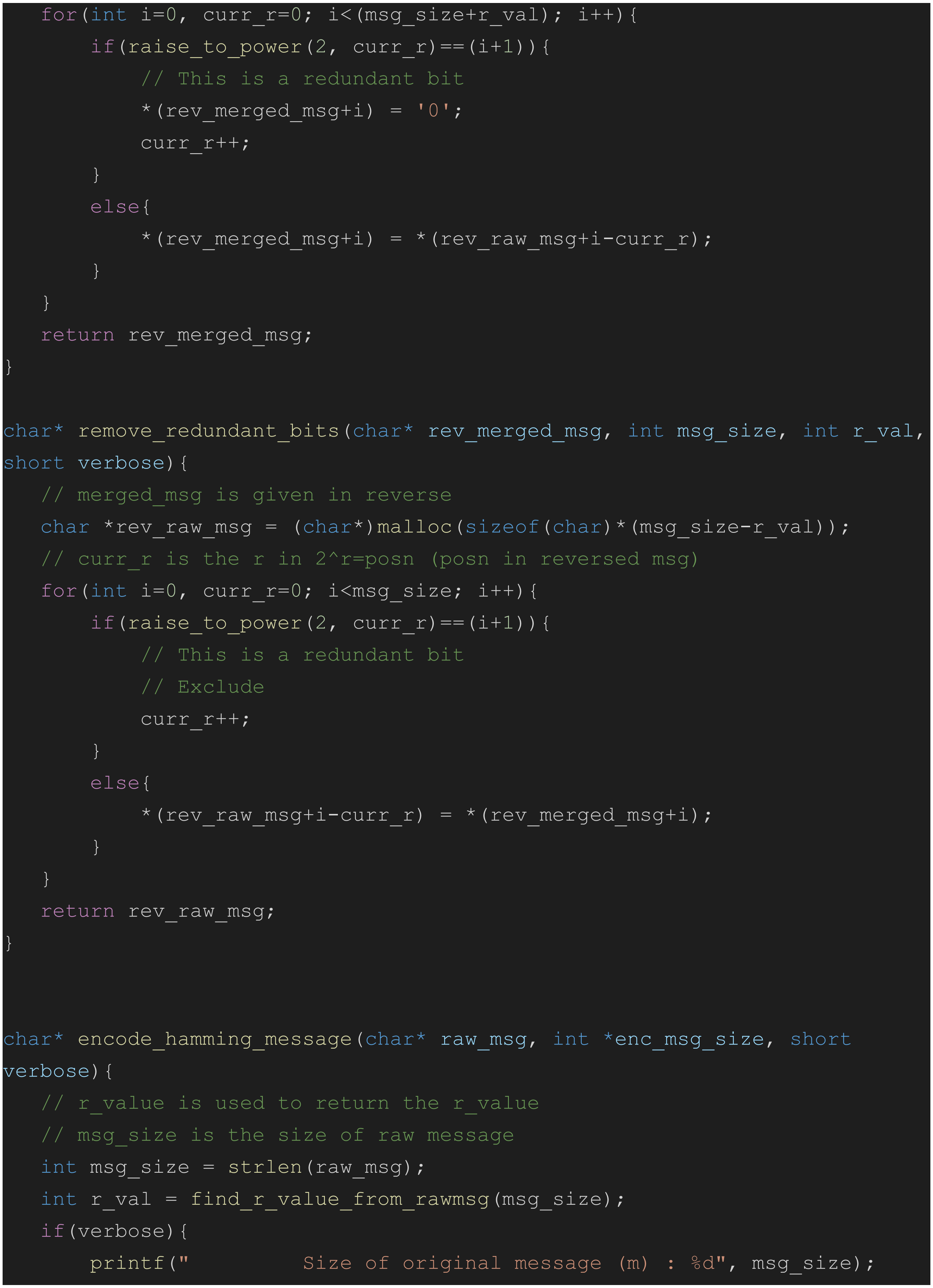
```
#ifndef hamming_code_h
#define hamming_code_h
```



```
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include<time.h>


char* pass_noise(char *encoded_msg, int msg_size, int *err_posn){
   char* noisy_msg = (char*)malloc(sizeof(char)*msg_size);
   memcpy(noisy_msg, encoded_msg, msg_size);
   srand(time(0));
   int posn = rand()%msg_size;
   // Add error, change bit
   if(*(noisy_msg+posn)=='1'){
       *(noisy_msg+posn) = '0';
   }
   else{
       *(noisy_msg+posn) = '1';
   }
   // Return reversed error posn and the noisy message
   *err_posn = msg_size - posn;
   return noisy_msg;
}

char* reverse_string(char *orig_string){
   int size = strlen(orig_string);
   char *rev_string = (char*)malloc(sizeof(char)*size);
   for(int i=0;i<size;i++){
       *(rev_string+i) = *(orig_string+size-i-1);
   }
   return rev_string;
}

int raise_to_power(int base, int exp){
   // Only for positive exp
   int result = 1;
   for(int i=0;i<exp;i++){
       result *= base;
   }
   return result;
}
```



```c
char* decimal_to_binary(int num){
    // Result is returned in reverse order
    // Eg: 6 -> 011
    int binary_size = ((int)floor(log2(num))) + 1;
    char* binary = (char*)malloc(sizeof(char)*binary_size);
    for(int i=0;i<binary_size;i++){
        // Converting to ASCII character
        *(binary+i) = 48 + (num%2);
        num /= 2;
    }
    return binary;
}

int binary_to_decimal(char *binary){
    binary = reverse_string(binary);
    int decimal_num = 0;
    int posn = 0;
    char *parser = binary;
    while(*parser!='\0'){
        decimal_num += raise_to_power(2, posn)*((int)((*parser)-48));
        posn++;
        parser++;
    }
    return decimal_num;
}

int find_r_value_from_rawmsg(int msg_size){
    // Deduce r from 2^r >= m+r+1
    for(int r=0;r<msg_size;r++){
        if(raise_to_power(2, r) >= msg_size+r+1){
            return r;
        }
    }
    return -1;  // Not found. Syntactic statement - never reached
}

int find_r_value_from_hammingmsg(int msg_size){
    // Deduce r from 2^r >= m+r+1
    int m;
    for(int r=0;r<msg_size;r++){
        m = msg_size - r;
        if(raise_to_power(2, r) >= m+r+1){
            return r;
        }
    }
    return -1;  // Not found. Syntactic statement - never reached
}

short find_even_parity(char *rev_merged_msg, int msg_size, int rbit_num, short exclude_rbit, short verbose){
    // Find's the even parity for posns relevant to rbit_num
    // msg_size is the entire size of merged message
    // assessed_bits returns the set of bits that were examined
    int start_at = (exclude_rbit ? raise_to_power(2, rbit_num) : raise_to_power(2, rbit_num)-1);
    // Exclude the r-bit itself if exclude is set to 1. Otherwise, start from rbit
    // Eg: for r2 (rbit_num=1), exclude upto position-2 (index-1 in the reversed msg)
    int count_ones = 0;
    for(int i=start_at;i<msg_size;i++){
        // i is the index. posn=i+1
        if(decimal_to_binary(i+1)[rbit_num]=='1'){
            if(*(rev_merged_msg+i)=='1'){
                count_ones++;
            }
            if(verbose){
                printf(" %d,", (i+1));
            }
        }
    }
    // Return 1 if odd number of 1s. 0, otherwise
    return count_ones%2;
}

char* position_redundant_bits(char* rev_raw_msg, int msg_size, int r_val){
    // raw_msg is given in reverse
    char *rev_merged_msg = malloc(sizeof(char)*(msg_size+r_val));
    // curr_r is the r in 2^r=posn (posn in reversed msg)
```



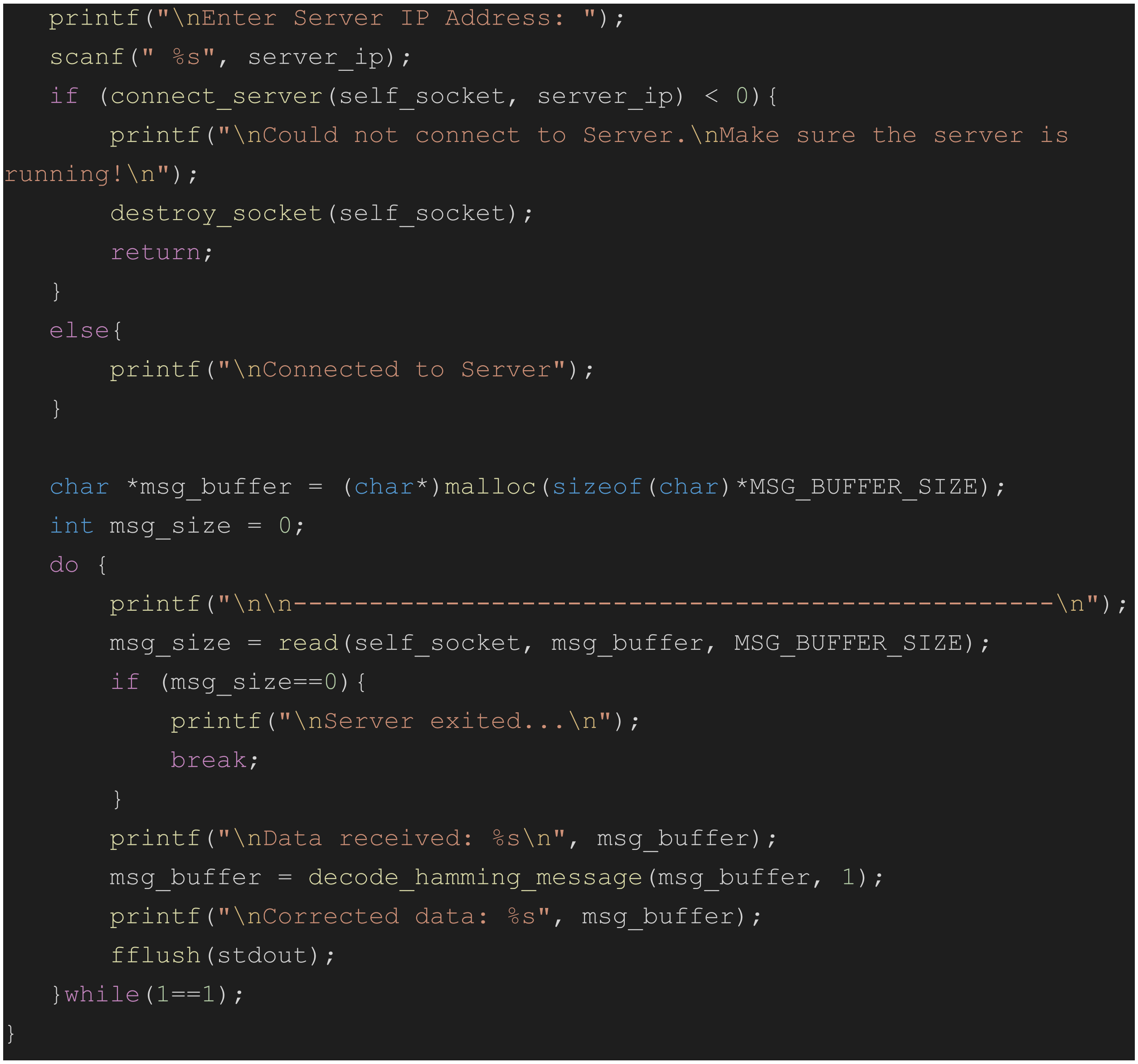
```
    for(int i=0, curr_r=0; i<(msg_size+r_val); i++){
        if(raise_to_power(2, curr_r)==(i+1)){
            // This is a redundant bit
            *(rev_merged_msg+i) = '0';
            curr_r++;
        }
        else{
            *(rev_merged_msg+i) = *(rev_raw_msg+i-curr_r);
        }
    }
    return rev_merged_msg;
}

char* remove_redundant_bits(char* rev_merged_msg, int msg_size, int r_val,
short verbose){
    // merged_msg is given in reverse
    char *rev_raw_msg = (char*)malloc(sizeof(char)*(msg_size-r_val));
    // curr_r is the r in 2^r=posn (posn in reversed msg)
    for(int i=0, curr_r=0; i<msg_size; i++){
        if(raise_to_power(2, curr_r)==(i+1)){
            // This is a redundant bit
            // Exclude
            curr_r++;
        }
        else{
            *(rev_raw_msg+i-curr_r) = *(rev_merged_msg+i);
        }
    }
    return rev_raw_msg;
}


char* encode_hamming_message(char* raw_msg, int *enc_msg_size, short
verbose){
    // r_value is used to return the r_value
    // msg_size is the size of raw message
    int msg_size = strlen(raw_msg);
    int r_val = find_r_value_from_rawmsg(msg_size);
    if(verbose){
        printf("         Size of original message (m) : %d", msg_size);
        printf("\nMin. r computed using `2^r >= (m+r+1)`: %d", r_val);
    }
    // Reverse the raw message for convenience
    char *rev_raw_msg = reverse_string(raw_msg);
    // Insert the redundant bit
    char *rev_merged_msg = position_redundant_bits(rev_raw_msg, msg_size,
r_val);
    // Set parity values to the redundant bits
    char *redundant_bits = (char*)malloc(sizeof(char)*r_val);
    int parity_val;
    for(int r=0;r<r_val;r++){
        if(verbose){
            printf("\n\nFinding Parity Bit at R%d\nComputed at positions:
", (r+1));
        }
        parity_val = find_even_parity(rev_merged_msg, msg_size+r_val, r, 1,
1);
        *(rev_merged_msg+(raise_to_power(2, r)-1)) = 48 + parity_val;
        *(redundant_bits+r_val-r-1) = 48 + parity_val;
        if(verbose){
            printf("\nParity Bit Value: %d", parity_val);
        }
    }
    if(verbose){
        printf("\n\n Redundant bits: %s", redundant_bits);
    }
    // Return the full message size
    *enc_msg_size = (r_val + msg_size);
    // Reverse-back the merged string
    return reverse_string(rev_merged_msg);
}

char* decode_hamming_message(char *merged_msg, short verbose){
    // msg_size is the size of merged message
    int msg_size = strlen(merged_msg);
    int r_val = find_r_value_from_hammingmsg(msg_size);
    if(verbose){
        printf("        Size of encoded message (m+r) : %d", msg_size);
        printf("\nMin. r computed using `2^r >= (m+r+1)`: %d", r_val);
    }
    // Reverse the hamming message
    char *rev_merged_msg = reverse_string(merged_msg);
    // Compute parities
    char *error_posn_binary = (char*)malloc(sizeof(char)*r_val);
    int parity_val;
    for(int r=0;r<r_val;r++){
        if(verbose){
            printf("\n\nFinding Parity Bit at R%d\nComputed at positions: 
", (r+1));
        }
        parity_val = find_even_parity(rev_merged_msg, msg_size, r, 0, 1);
        *(error_posn_binary+r_val-r-1) = 48 + parity_val;
        if(verbose){
            printf("\nParity Bit Value: %d", parity_val);
        }
    }
    int correction_posn = binary_to_decimal(error_posn_binary);
    if(correction_posn!=0){
        if((*(rev_merged_msg+correction_posn-1))=='0'){
            *(rev_merged_msg+correction_posn-1) = '1';
        }
        else{
            *(rev_merged_msg+correction_posn-1) = '0';
        }
    }
    if(verbose){
        printf("\n\n Binary form of correction posn: %s", 
error_posn_binary);
        printf("\nDecimal form of correction posn: %d", correction_posn);
        printf("\nCorrected hamming-encoded message: %s", 
reverse_string(rev_merged_msg));
    }
    // Return the redundant bits
    return reverse_string(remove_redundant_bits(rev_merged_msg, msg_size, 
r_val, verbose));
}

#endif
```

3. server.c - Server-side script

```
#include<stdio.h>
#include<stdlib.h>

#include "tcp_socket.h"

#ifndef hamming_code_h
   #include "hamming_code.h"
#endif

void main(){
   int self_socket = make_socket();
   if(self_socket<0){
       printf("\nCould not create socket. Retry!\n");
       return;
   }

   if (bind_server_socket(self_socket)<0){
       printf("\nCould not bind server socket. Retry!\n");
       destroy_socket(self_socket);
       return;
   }

   if (initiate_listen(self_socket)<0){
       printf("\nCould not listen on server socket. Retry!\n");
       destroy_socket(self_socket);
       return;
   }
   else{
       printf("\nServer listening for connections from all local
interfaces...\n");
   }

   struct sockaddr_in *client_addr = malloc(sizeof(struct sockaddr_in));
   int client_addr_len = sizeof(struct sockaddr_in);
   // BLOCKING routine to accept a client
   int client_socket = accept_client(self_socket, client_addr,
&client_addr_len);
   if (client_socket<0){
       printf("\nError when connecting to client. Retry!\n");
       destroy_socket(self_socket);
       return;
   }
   else if(client_addr_len == -1){
       printf("Client connected.\nCould not read address\n");
   }
   else{
       char *client_addr_ip_str =
(char*)malloc(sizeof(char)*ADDRESS_BUFFER_SIZE);
       // Alternatively, use inet_ntoa
       inet_ntop(ADDRESS_FAMILY, (void*)&client_addr->sin_addr,
client_addr_ip_str, ADDRESS_BUFFER_SIZE);
       int client_addr_port = (int)ntohs(client_addr->sin_port);
       if (client_addr_ip_str == NULL) {
           printf("Client connected.\nCould not read address\n");
       }
       else{
           printf("Connected to Client (%s:%d)\n", client_addr_ip_str,
client_addr_port);
       }
   }

   char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
   int msg_size = 0;
   int error_posn;
   char ch;
   printf("\n\nEnter 'ENDSESSION' to terminate connection\n");
   do{
       printf("\n---------------------------------------------------\n");
       printf("\nEnter Data: ");
       scanf(" %s", msg_buffer);
       if (check_termination_init(msg_buffer)){
           printf("\nTerminating server...\n");
           destroy_socket(client_socket);
           break;
       }
       msg_buffer = encode_hamming_message(msg_buffer, &msg_size, 1);
       // Add error if user specifies
       printf("\n\nDo you want to add error? (y/n): ");
       scanf(" %c", &ch);
       if(ch=='y'){
           printf("Hamming encoded message (no error): %s", msg_buffer);
           msg_buffer = pass_noise(msg_buffer, msg_size, &error_posn);
           printf("\n Message after adding random noise: %s", msg_buffer);
           printf("\nError added at position: %d", error_posn);
       }
       else{
           // No error is added
           printf("Hamming encoded message (no error): %s", msg_buffer);
       }
       // Send the data bits
       msg_size = write(client_socket, msg_buffer, msg_size);
       printf("\n(Data transmitted)\n");
   }while(1==1);

   destroy_socket(self_socket);
   return;
}
```

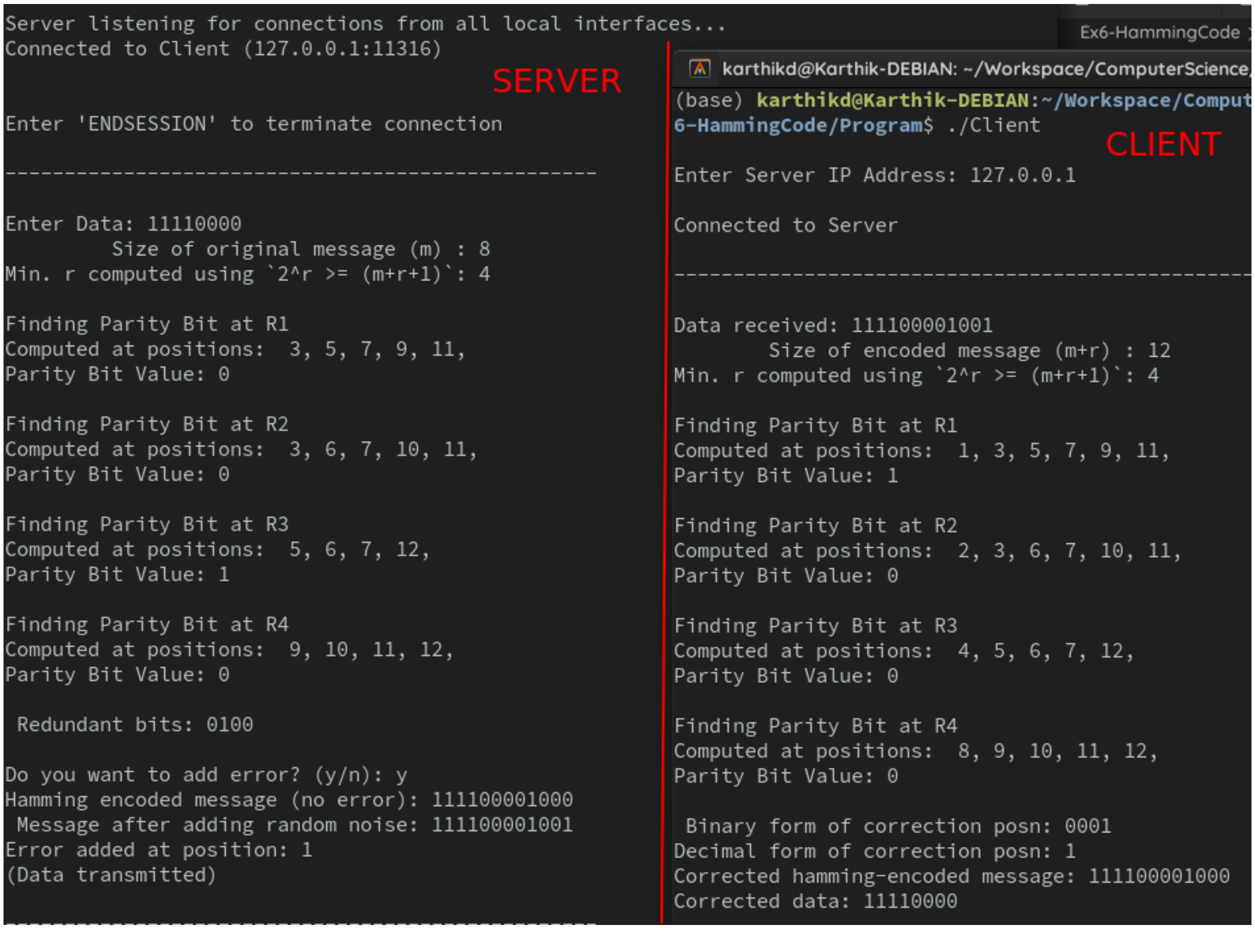
4. client.c - Client-side script

```
#include<stdio.h>
#include<stdlib.h>

#include "tcp_socket.h"

#ifndef hamming_code_h
    #include "hamming_code.h"
#endif

void main(){

    int self_socket = make_socket();
    if(self_socket<0){
        printf("\nCould not create socket. Retry!\n");
        return;
    }

    char *server_ip = (char*)malloc(sizeof(char)*IP_STRING_LEN);
```



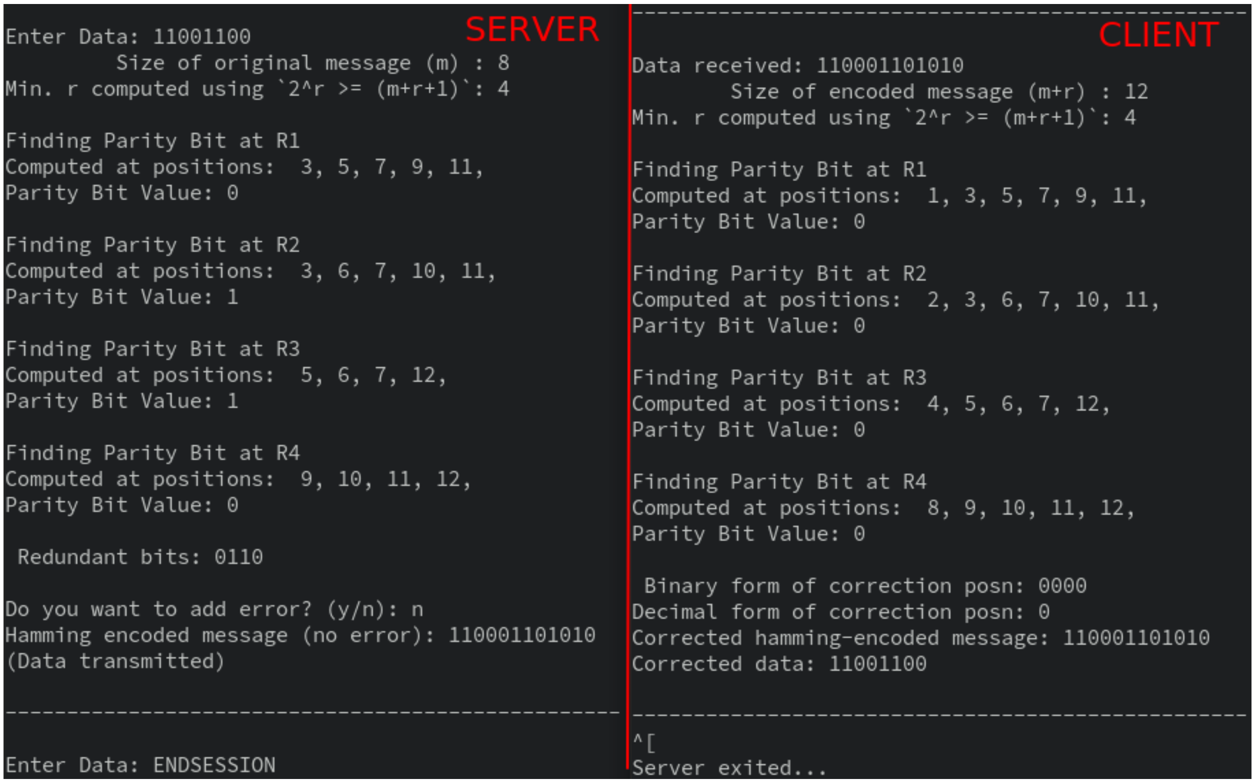
```c
    printf("\nEnter Server IP Address: ");
    scanf(" %s", server_ip);
    if (connect_server(self_socket, server_ip) < 0){
        printf("\nCould not connect to Server.\nMake sure the server is
running!\n");
        destroy_socket(self_socket);
        return;
    }
    else{
        printf("\nConnected to Server");
    }

    char *msg_buffer = (char*)malloc(sizeof(char)*MSG_BUFFER_SIZE);
    int msg_size = 0;
    do {
        printf("\n\n--------------------------------------------------\n");
        msg_size = read(self_socket, msg_buffer, MSG_BUFFER_SIZE);
        if (msg_size==0){
            printf("\nServer exited...\n");
            break;
        }
        printf("\nData received: %s\n", msg_buffer);
        msg_buffer = decode_hamming_message(msg_buffer, 1);
        printf("\nCorrected data: %s", msg_buffer);
        fflush(stdout);
    }while(1==1);
}
```

Sample Output



```
Server listening for connections from all local interfaces...
Connected to Client (127.0.0.1:11316)


Enter 'ENDSESSION' to terminate connection

--------------------------------------------------

Enter Data: 11110000
         Size of original message (m) : 8
Min. r computed using `2^r >= (m+r+1)`: 4

Finding Parity Bit at R1
Computed at positions:  3, 5, 7, 9, 11,
Parity Bit Value: 0

Finding Parity Bit at R2
Computed at positions:  3, 6, 7, 10, 11,
Parity Bit Value: 0

Finding Parity Bit at R3
Computed at positions:  5, 6, 7, 12,
Parity Bit Value: 1

Finding Parity Bit at R4
Computed at positions:  9, 10, 11, 12,
Parity Bit Value: 0

 Redundant bits: 0100

Do you want to add error? (y/n): y
Hamming encoded message (no error): 111100001000
 Message after adding random noise: 111100001001
Error added at position: 1
(Data transmitted)

--------------------------------------------------
```

```
(base) karthikd@Karthik-DEBIAN:~/Workspace/Comput
6-HammingCode/Program$ ./Client

Enter Server IP Address: 127.0.0.1

Connected to Server

--------------------------------------------------

Data received: 111100001001
        Size of encoded message (m+r) : 12
Min. r computed using `2^r >= (m+r+1)`: 4

Finding Parity Bit at R1
Computed at positions:  1, 3, 5, 7, 9, 11,
Parity Bit Value: 1

Finding Parity Bit at R2
Computed at positions:  2, 3, 6, 7, 10, 11,
Parity Bit Value: 0

Finding Parity Bit at R3
Computed at positions:  4, 5, 6, 7, 12,
Parity Bit Value: 0

Finding Parity Bit at R4
Computed at positions:  8, 9, 10, 11, 12,
Parity Bit Value: 0

 Binary form of correction posn: 0001
Decimal form of correction posn: 1
Corrected hamming-encoded message: 111100001000
Corrected data: 11110000
```



```
Enter Data: 11001100
         Size of original message (m) : 8
Min. r computed using `2^r >= (m+r+1)`: 4

Finding Parity Bit at R1
Computed at positions:  3, 5, 7, 9, 11,
Parity Bit Value: 0

Finding Parity Bit at R2
Computed at positions:  3, 6, 7, 10, 11,
Parity Bit Value: 1

Finding Parity Bit at R3
Computed at positions:  5, 6, 7, 12,
Parity Bit Value: 1

Finding Parity Bit at R4
Computed at positions:  9, 10, 11, 12,
Parity Bit Value: 0

 Redundant bits: 0110

Do you want to add error? (y/n): n
Hamming encoded message (no error): 110001101010
(Data transmitted)

--------------------------------------------------

Enter Data: ENDSESSION
```

```
--------------------------------------------------

Data received: 110001101010
        Size of encoded message (m+r) : 12
Min. r computed using `2^r >= (m+r+1)`: 4

Finding Parity Bit at R1
Computed at positions:  1, 3, 5, 7, 9, 11,
Parity Bit Value: 0

Finding Parity Bit at R2
Computed at positions:  2, 3, 6, 7, 10, 11,
Parity Bit Value: 0

Finding Parity Bit at R3
Computed at positions:  4, 5, 6, 7, 12,
Parity Bit Value: 0

Finding Parity Bit at R4
Computed at positions:  8, 9, 10, 11, 12,
Parity Bit Value: 0

 Binary form of correction posn: 0000
Decimal form of correction posn: 0
Corrected hamming-encoded message: 110001101010
Corrected data: 11001100

--------------------------------------------------
^[
Server exited...
```

Result

Implemented a socket program in C language using TCP to simulate hamming code for single correction in network communication. A sever-client message transmission system is developed, wherein a server sends a hamming encoded message to the client. The client in turn, decode this message and fixes the error if present. Through this implementation, the following aspects were understood:

1. Working procedure of hamming encoding and decoding for bit strings
2. A possible algorithmic implementation of hamming code based message transmission
3. Implementation details of socket programming using C language for TCP protocol