# Chat Program with Error Correctness by Hamming Code and CRC

Contents

### System Description

1. A high-level data flow diagram for the system

Server

Client2

Client5

Client4

Client3

Client1

Client6

Server

Hamming

CRC

Receive Message

Send Message

CRC

Hamming

Client-n

Client-m

1. A list of routines and their brief descriptions

- In sender(client):

● **message:** sender choose a receiver and type a message

● **protocol:** message is formatted in a given protocol

● **hamming:** message is encoded by hamming code

● **CRC:** calculate CRC of the encoded message and then send to server

- In server(transfer):

● **CRC:** from received message calculating CRC and compare if true

**● hamming**: decode message

● **protocol**: get a receiver’s address from decoded message

● **find:** check if there is such an address in connected clients from list

● **message:** sender choose a receiver and type a message

● **protocol:** message is formatted in a given protocol

● **hamming:** message is encoded by hamming code

● **CRC:** calculate CRC of the encoded message and then send to client(receiver)

-In receiver:

● **CRC:** from received message calculating CRC and compare if true

**● hamming**: decode message

● **protocol**: get a sender’s address and message contents from decoded message

1. Implementation details

● In server, the connected client’s list are implemented by linked list(source file: client\_list.h, client\_list.c).

● messages are encoded by hamming code, which is implemented in a function hamming\_encode ; messages are decoded by hamming code, which is is implemented in a function hamming\_decode (source file: hamming.h, hamming.c).

● error detection is implemented in a function detectAndCorrectError (source file: hamming.h, hamming.c).

// Function to detect errors and correct 1 bit error using Hamming code

unsigned char detectAndCorrectError(unsigned char hammingCode)

{

// Calculate number of parity bits needed

int m = 4; // Number of data bits

int r = 3; // Number of parity bits

// Calculate parity for each received parity bit

int errorPos = 0;

int errorFound = 0;

for (int i = 0; i < r; i++)

{

int count = 0;

for (int j = 0; j < m + r; j++)

{

if (((j + 1) & j) != 0 && (j & (1 << i)) != 0) {

if ((hammingCode >> j) & 1)

count++;

}

}

if (count % 2 != ((hammingCode >> ((1 << i)-1)) & 1)) {

errorPos += (1 << i); // Set corresponding bit in error position

errorFound = 1;

}

}

// Correct error if found

if (errorFound)

{

hammingCode ^= 1 << (errorPos-1); // Flip erroneous bit

printf("Error detected and corrected at bit position %d\n", errorPos);

}

// recover nibble

unsigned char nibble = 0;

int j = 0;

for (int i = 0; i < m + r; i++)

{

if (i == 0 || i == 1 || i == 3 || i == 7)

{

// hammingCode[i] = 0; // Initialize parity bits to 0

}

else

{

nibble |= ((hammingCode >> i) & 1) << j;

j++;

}

}

return nibble;

}

● CRC values are calculated by function calculateCRC32 (source file: crc.h, crc.c).

uint32\_t calculateCRC32(const uint8\_t \*data, size\_t length)

{

uint32\_t crc = 0xFFFFFFFF; // Initialize register with all bits set to 1

const uint32\_t polynomial = 0xEDB88320; // Pre-defined constant

for (size\_t i = 0; i < length; i++)

{

uint8\_t byte = data[i];

crc ^= byte; // XOR byte with least significant byte of register

for (size\_t j = 0; j < 8; j++)

{

// Process each bit of byte

if (crc & 1) // Check if least significant bit of register is 1

crc = (crc >> 1) ^ polynomial;

else

crc = crc >> 1; // Shift register one bit to the right

}

}

return ~crc; // Final value of register is CRC32 value (with all bits inverted)

}

● Protocol is implemented by parseXml

1. How to run your program

./root/my\_chat/

make clean

make all

./server

./client

### Source Code

-client.c

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include "define.h"

#include "tag.h"

#include "crc.h"

#include "encDec.h"

void \*receive\_messages(void \*arg)

{

int client\_socket = \*(int \*)arg;

char buffer[BUFFER\_SIZE \* 2];

while (1)

{

memset(buffer, 0, sizeof(buffer));

if (recv\_with\_crc(client\_socket, buffer, sizeof(buffer), 0) <= 0)

break;

char startTag[128];

char from[128];

char content[128];

sprintf(startTag, "<%s>", TAG\_LOGIN\_LIST);

if (strncmp(buffer, startTag, strlen(startTag)) == 0)

{

extract\_content(buffer, TAG\_LOGIN\_LIST, content);

printf("\nClient List");

printf("\n-----------------------------------------------------\n");

printf("%s", content);

printf("-----------------------------------------------------\n");

continue;

}

sprintf(startTag, "<%s>", TAG\_MSG);

if (strncmp(buffer, startTag, strlen(startTag)) == 0)

{

extract\_content(buffer, TAG\_FROM, from);

extract\_content(buffer, TAG\_BODY, content);

// printf("content: %s\n", content);

char msg[BUFFER\_SIZE];

hamming\_decode(content, msg, strlen(content));

printf("msg from %s: %s\n", from, msg);

continue;

}

else

printf("%s\n", buffer);

}

pthread\_exit(NULL);

}

int main(int argc, char\* argv[])

{

int client\_socket;

struct sockaddr\_in server\_addr;

pthread\_t tid;

char ip[20] = "";

int port;

if (argc < 2)

strcpy(ip, "127.0.0.1");

else

strcpy(ip, argv[1]);

if (argc < 3)

port = PORT;

else

port = atoi(argv[2]);

// Create client socket

client\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (client\_socket == -1)

{

perror("Socket creation failed");

return 1;

}

// Set up server address

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip, &(server\_addr.sin\_addr)) <= 0)

{

perror("Invalid address");

return 1;

}

// Connect to the server

if (connect(client\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

{

perror("Connection failed");

return 1;

}

printf("Connected to the server\n");

// Create a thread to receive messages from the server

if (pthread\_create(&tid, NULL, receive\_messages, &client\_socket) != 0)

{

perror("Thread creation failed");

return 1;

}

// Send messages to the server

char message[BUFFER\_SIZE];

int login = 0;

char answer;

char username[64];

char password[64];

while (1)

{

if (login == 0)

{

printf("LOGIN? [y/n] ");

answer = getchar();

getchar();

if (answer == 'y')

{

int login\_succeed = 0;

while (login\_succeed == 0)

{

printf("You choose LOGIN option...\n");

printf("username: ");

if (fgets(username, sizeof(username), stdin))

{

username[strcspn(username, "\n")] = '\0';

printf("password: ");

if (fgets(password, sizeof(password), stdin))

{

password[strcspn(password, "\n")] = '\0';

sprintf(message, "<%s><%s><%s>%s</%s><%s>%s</%s></%s></%s>\n",

TAG\_REQUEST, TAG\_LOGIN, TAG\_USERNAME, username, TAG\_USERNAME, TAG\_PASSWORD, password, TAG\_PASSWORD, TAG\_LOGIN, TAG\_REQUEST);

send\_with\_crc(client\_socket, message, strlen(message), 0);

login\_succeed = 1;

}

}

}

login = 1;

}

else if (answer == 'n')

continue;

else

{

printf("input correct option character\n");

break;

}

}

else

{

char option;

printf("Please select one of the following options\n");

printf("1: MSG\n");

printf("2: DISCONNECT\n");

answer = getchar();

getchar();

if (answer == '1')

{

char to\_name[100];

char temp\_msg[BUFFER\_SIZE];

uint8\_t enc\_msg[BUFFER\_SIZE \* 2];

char send\_msg[BUFFER\_SIZE \* 2 + 1000];

printf("to: ");

if (fgets(to\_name, sizeof(to\_name), stdin) != NULL)

{

to\_name[strcspn(to\_name, "\n")] = '\0';

printf("msg: ");

if (fgets(temp\_msg, sizeof(temp\_msg), stdin) != NULL)

{

temp\_msg[strcspn(temp\_msg, "\n")] = '\0';

int enc\_len = hamming\_encode(temp\_msg, enc\_msg, strlen(temp\_msg));

printf("temp\_msg = %s, enc\_len = %d, enc\_msg = %s\n", temp\_msg, enc\_len, enc\_msg);

sprintf(send\_msg, "<%s><%s>%s</%s><%s>%s</%s><%s>%s</%s></%s>",

TAG\_MSG,

TAG\_FROM, username, TAG\_FROM,

TAG\_TO, to\_name, TAG\_TO,

TAG\_BODY, /\* temp\_msg \*/ enc\_msg, TAG\_BODY,

TAG\_MSG);

send\_with\_crc(client\_socket, send\_msg, strlen(send\_msg), 0);

}

}

}

else if (answer == '2')

{

printf("Disconnect? [y/n] ");

answer = getchar();

getchar();

if (answer == 'y')

{

sprintf(message, "<%s>disconnect<%s>", TAG\_REQUEST, TAG\_REQUEST);

send\_with\_crc(client\_socket, message, strlen(message), 0);

login = 0;

}

}

else

{

printf("input correct option\n");

break;

}

}

}

close(client\_socket);

return 0;

}

-server.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <pthread.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include "define.h"

#include "tag.h"

#include "encDec.h"

#include "client\_list.h"

int clients[MAX\_CLIENTS];

pthread\_mutex\_t clients\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

void notifyAllClients()

{

struct node \* temp1 = head;

if (temp1 != NULL && temp1->next == NULL)

return;

while (temp1 != NULL)

{

char temp\_str[1000];

struct node \* temp2 = head;

sprintf(temp\_str, "<%s>", TAG\_LOGIN\_LIST);

while (temp2 != NULL)

{

if (temp2 != temp1) // Omit self

sprintf(temp\_str + strlen(temp\_str), "%s\n" , temp2->name);

temp2 = temp2->next;

}

sprintf(temp\_str + strlen(temp\_str), "</%s>", TAG\_LOGIN\_LIST);

send\_with\_crc(temp1->id, temp\_str, strlen(temp\_str), 0);

temp1 = temp1->next;

}

}

void \*handle\_client(void \*arg)

{

int client\_socket = \*(int \*)arg;

int to\_client\_socket = -1;

char buffer[BUFFER\_SIZE];

// Receive and broadcast messages

while (1)

{

memset(buffer, 0, sizeof(buffer));

if (recv\_with\_crc(client\_socket, buffer, sizeof(buffer), 0) <= 0)

break;

xmlNode\* root = parseXML(buffer);

// Print the XML tree

if (root != NULL)

{

if (strcmp(root->tag, TAG\_REQUEST) == 0)

{

xmlNode\* currentNode = root->children;

if (currentNode != NULL)

{

if (strcmp(currentNode->tag, TAG\_LOGIN) == 0)

{

currentNode = currentNode->children;

char tempUsername[128];

char tempPassword[128];

char svrMsg[512];

while (currentNode != NULL)

{

if (strcmp(currentNode->tag, TAG\_USERNAME) == 0)

strcpy(tempUsername, currentNode->content);

else if(strcmp(currentNode->tag, TAG\_PASSWORD) == 0)

strcpy(tempPassword, currentNode->content);

else

{

sprintf(svrMsg, "<%s>Undefined LogIn Type.</%s>", TAG\_INFO, TAG\_INFO);

send\_with\_crc(client\_socket, svrMsg, strlen(svrMsg), 0);

}

currentNode = currentNode->next;

}

pthread\_mutex\_lock(&clients\_mutex);

insertAtEnd(client\_socket, tempUsername, tempPassword);

notifyAllClients();

pthread\_mutex\_unlock(&clients\_mutex);

// printList();

}

else if(strcmp(currentNode->content, "disconnect") == 0)

{

close(client\_socket);

pthread\_mutex\_lock(&clients\_mutex);

deleteNode(client\_socket);

notifyAllClients();

pthread\_mutex\_unlock(&clients\_mutex);

// printList();

}

}

}

else if (strcmp(root->tag, TAG\_MSG) == 0)

{

xmlNode\* currentNode = root->children;

char from[128];

char to[128];

char svrMsg[BUFFER\_SIZE];

while (currentNode != NULL)

{

if (strcmp(currentNode->tag, TAG\_FROM) == 0)

sprintf(from, "%s", currentNode->content);

else if (strcmp(currentNode->tag, TAG\_TO) == 0)

{

sprintf(to, "%s", currentNode->content);

// printf("%s\n", currentNode->content);

to\_client\_socket = getSockClientFromName(currentNode->content);

// printf("%d\n", to\_client\_socket);

}

else if (strcmp(currentNode->tag, TAG\_BODY) == 0)

sprintf(svrMsg, "<%s><%s>%s</%s><%s>%s</%s></%s>",

TAG\_MSG,

TAG\_FROM, from, TAG\_FROM,

TAG\_BODY, currentNode->content, TAG\_BODY,

TAG\_MSG);

currentNode = currentNode->next;

}

if (to\_client\_socket == -1)

{

sprintf(svrMsg, "<%s>'%s' does not exist now.</%s>", TAG\_INFO, to, TAG\_INFO);

send\_with\_crc(client\_socket, svrMsg, strlen(svrMsg), 0);

break;

}

else

send\_with\_crc(to\_client\_socket, svrMsg, strlen(svrMsg), 0);

printf("%s\n", svrMsg);

}

}

pthread\_mutex\_unlock(&clients\_mutex);

}

// Client disconnected

pthread\_mutex\_lock(&clients\_mutex);

for (int i = 0; i < MAX\_CLIENTS; i++)

{

if (clients[i] == client\_socket)

{

clients[i] = -1;

break;

}

}

pthread\_mutex\_unlock(&clients\_mutex);

close(client\_socket);

pthread\_exit(NULL);

}

int main(int argc, char\* argv[])

{

int server\_socket, client\_socket;

struct sockaddr\_in server\_addr, client\_addr;

int opt = 1;

pthread\_t tid;

int port;

if (argc == 1)

port = PORT;

else

port = atoi(argv[1]);

// Initialize clients array

for (int i = 0; i < MAX\_CLIENTS; i++)

clients[i] = -1;

// Create server socket

server\_socket = socket(AF\_INET, SOCK\_STREAM, 0);

if (server\_socket == -1)

{

perror("Socket creation failed");

return 1;

}

// Set socket options

if (setsockopt(server\_socket, SOL\_SOCKET, SO\_REUSEADDR | SO\_REUSEPORT, &opt, sizeof(opt)))

{

perror("setsockopt failed");

exit(EXIT\_FAILURE);

}

// Set up server address

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_addr.s\_addr = INADDR\_ANY;

server\_addr.sin\_port = htons(port);

// Bind the server socket to the specified address and port

if (bind(server\_socket, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

{

perror("Binding failed");

return 1;

}

// Start listening for incoming connections

if (listen(server\_socket, MAX\_CLIENTS) < 0)

{

perror("Listening failed");

return 1;

}

printf("Server started. Listening for connections...\n");

while (1)

{

socklen\_t client\_len = sizeof(client\_addr);

// Accept new client connection

client\_socket = accept(server\_socket, (struct sockaddr \*)&client\_addr, &client\_len);

if (client\_socket < 0)

{

perror("Accepting connection failed");

return 1;

}

// Add client to the array

pthread\_mutex\_lock(&clients\_mutex);

int i;

for (i = 0; i < MAX\_CLIENTS; i++)

{

if (clients[i] == -1)

{

clients[i] = client\_socket;

break;

}

}

pthread\_mutex\_unlock(&clients\_mutex);

// Create a thread to handle the client

if (pthread\_create(&tid, NULL, handle\_client, &clients[i]) != 0)

{

perror("Thread creation failed");

return 1;

}

}

close(server\_socket);

return 0;

}