컴퓨터 통신 경진대회

FINAL REPORT

과 제 명	보안성이 강화된 1:N 채팅 서버 및 클라이언트 구축
제 출 일 자	2022.11.23
연구 참여자	정보통신공학과 20190895 김찬영 정보통신공학과 20190954 허진환
수 업 명	컴퓨터 통신(7-9)

목차

1. 설계과제 소개

1) 과제명 : 보안성이 강화된 1:N 채팅 서버 및 클라이언트 구축

2) 설계 추진 배경

광범위해진 네트워크에 따라 해킹 공격 또한 다양한 방법으로 늘어가고 있다. 특히, 유출 및 사이버공격에 민감한 정보를 다루는 군 관련 및 공공기관, 은행 등의 기관들은 이런 데이터 유통에 있어 신뢰성과 보안성, 폐쇄성이 갖추어져야 한다.

위 배경에 따라 폐쇄성을 보장하기 위해 인가된 사용자만이 서버에 접속할 수 있도록 하며, 데이터 유통 시 보안성 향상을 위해 송·수신되는 데이터에 대해 고급 암호화 표준(AES)을 사용하고, 신뢰성이 보장되는 TCP 프로토콜을 사용하여 1:N 채팅 서버 및 클라이언트를 설계하였다.

3) 설계 목표

(1) C/C++ 소켓 프로그래밍을 통한 서버-클라이언트 간 소켓 파이프 생성의 이해

(2) C/C++ 소켓 함수들에 대한 이해 및 응용

(3) 채팅서버 구축을 통한 서버-클라이언트 간 data 송수신 원리 이해

(4) 멀티쓰레드 개념 및 소켓 입출력 모델 중 WSAEventSelect모델에 대한 이해 및 응용

(5) AES암호화 방식에 대한 이해 및 암호화된 data 송수신 구현

(6) Encoding, Decoding 함수 이해 및 개념

(7) 채팅 프로그램 UI의 구현

4) 연구 참여자

정보통신공학과 20190895 김찬영

WSAEventSelect모델에 대한 이해 및 채팅 서버 구현

UI 기반의 클라이언트 구현

정보통신공학과 20190954 허진환

AES암호화에 대한 자료 조사 및 분석, 프로그래밍으로 구현

Encoding, Decoding에 대한 자료 조사 및 분석, 프로그래밍으로 구현

2. 설계 과제 내용

1) TCP 서버 및 클라이언트 통신

뼈대가 되는 TCP 서버 및 클라이언트의 통신에는 수업 주 교재인 TCP/IP 윈도우 소켓 프로그래밍(김선우 저, 한빛아카데미)의 10장 04절 소켓 입출력 모델 중 WSAEventSelect 모델을 참고하여 프로그래밍하였다. WSAEventSelect모델은 소켓과 관련된 네트워크 이벤트를 이벤트 객체를 통해 감지하며, 이벤트 객체를 소켓당 하나씩 생성하고 이벤트 객체들을 관찰하면서 여러 소켓을 처리하게 된다.

동작 원리는 각 소켓마다 이벤트 객체를 하나씩 생성해서 짝을 지어주면, 네트워크 이벤트가 발생할 때 이벤트 객체는 신호 상태가 된다. 따라서 이벤트 객체의 신호 상태를 통해 네트워크 이벤트 발생을 감지하게 되고, 발생한 이벤트에 맞게 적절한 동작이 수행하도록 프로그래밍했다.

우선 서버 소스파일(Secured_server.c)에 대해서 설명하겠다. 위 WSAEventSelect모델을 구현하기 위해 다음과 같이 사용자 정의 함수들을 선언하였다.

```
//TCP서버 동작에 관련된 사용자 정의 함수
void server_setup();//인가된 IP만 접속을 가능하게 하기 위해 서버 시작 전 SETUP하는 함수
int server_init();//서버 시작 시 서버를 초기화하는 단계. winsock 초기화, server socket 생성, bind, listen 단계를 포함한다
int server_close();//서버 종료 시 sock_array에 저장된 모든 socket을 종료하고 이벤트 객체 핸들을 닫음
unsigned int WINAPI chat(void* param);//서버에서 통신 및 채팅을 위한 사용자 정의 쓰레드 함수
unsigned int WINAPI recv_and_forward(void* param);//클라이언트로부터 메세지를 받은 후 모든 클라이언트들에게 유니캐스팅 하는 함수
int add_client(int index);//FD_ACCEPT 이벤트 발생 시 클라이언트를 추가
int read_client(int index);//FD_READ 이벤트 발생 시 읽어들이는 함수
void remove_client(int index);//FD_CLOSE 이벤트 발생 시 클라이언트를 제거하는 함수
int notify_client(char* message);//모든 클라이언트들에게 메세지를 보내는 (유니캐스팅)함수(암호화+인코딩 후 전송)
char* get_client_ip(int index);//클라이언트의 IP를 획득하는 함수
```

또한 모든 소켓은 이벤트 객체와 함께 저장되어 관리되도록 다음과 같이 구조체를 정의하여 사용하였다.

```
//소켓에 HANDLE event를 포함하여 구조체 선언
typedef struct sock_info {
    SOCKET s;
    HANDLE ev;
} SOCK_INFO;
```

(1) server_setup() 함수는 인가된 IP만 접속을 가능하게 하기 위해 서버 시작 전 SETUP하는 함수이다. 서버 접근 허용 IP주소는 stack에 저장되어 관리된다(stack함수는 stack.c에 저장). 사용자 정의 함수 search_num()과 search_stack() 함수를 정의하여 스택이지만 pop 이외의 원하는 데이터의 삭제 연산을 추가하였다.

(2) server_init() 함수는 서버 시작 시 서버를 초기화하는 단계로, winsock초기화, server socket 생성, bind, listen 단계를 포함한다.

(3) server_close() 함수는 서버 종료 시 sock_array에 저장된 모든 socket을 종료하고 이벤트 객체 핸들을 닫는다.

```
HANDLE event = WSACreateEvent();//네트워크 이벤트 개체 생성
sock_array[total_socket_count].ev = event;
sock_array[total_socket_count].s = server_socket;

WSAEventSelect(server_socket, event, FD_ACCEPT);
total_socket_count++;

while (1)
{
    memset(&handle_array, 0, sizeof(handle_array));
    for (int i = 0; i < total_socket_count; i++)
        handle_array[i] = sock_array[i].ev;

    index = WSAWaitForMultipleEvents(total_socket_count,//이벤트 객체가 신호 상태가 되기를 기다린다
        handle_array, FALSE, INFINITE, FALSE);
    if ((index != WSA_WAIT_FAILED) && (index != WSA_WAIT_TIMEOUT))
    {
        WSAEnumNetworkEvents(sock_array[index].s, sock_array[index].ev, &ev);//발생한 네트워크 이벤트를 알아내고, 적절한 소켓 함수 호출 후 처리
        if (ev.lNetworkEvents == FD_ACCEPT)//클라이언트 접속 이벤트 감지시 add_client
            add_client(index);
        else if (ev.lNetworkEvents == FD_READ)//클라이언트가 메세지 보냈을 경우 read_client
            read_client(index);
        else if (ev.lNetworkEvents == FD_CLOSE)//클라이언트 접속 종료시 remove_client
            remove_client(index);
    }
}
closesocket(server_socket);
```

(chat함수의 핵심 내용)

(4) chat() 함수는 서버에서 이벤트를 감지하고 채팅 기능을 수행하기 위한 사용자 정의 쓰레드 함수이다. 서버는 2개의 쓰레드를 생성 후 병렬적으로 기능을 수행하게 된다. WSAWaitForMultipleEvents()함수와 WSAEnumNetworkEvents()함수를 통해 서버 소켓 조기화 후 지속적으로 연결된 클라이언트로부터 수신할 이벤트가 있는지 이벤트를 모니터링한다. 이벤트에는 FD_ACCEPT(클라이언트 접속), FD_READ(메세지 수신), FD_CLOSE(클라이언트 접속 종료)가 존재한다.

```
accept_sock = accept(sock_array[0].s, (SOCKADDR*)&addr, &len);

if (search_stack(inet_ntoa(addr.sin_addr)) == NULL) {
    textcolor(4);
    printf("\n[TCP server] 인가되지 않은 IP 접속 시도 (IP주소: %s, port: %d)\n",
        inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    textcolor(7);
    sprintf(msg, "\n[TCP server] 인가되지 않은 IP 접속 시도 (IP주소: %s, port: %d)\n",
        inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    notify_client(msg);
    closesocket(accept_sock);   //우선 모든 클라이언트에 대해서 accept한 다음, IP주소를 읽어와 stack안의 IP와 비교하여 없을 경우 소켓 닫으며 접속을 차단함
    return 1;
}
else {
    HANDLE event = WSACreateEvent();//인가된 IP가 접속했을 경우 이벤트 생성
    sock_array[total_socket_count].ev = event;
    sock_array[total_socket_count].s = accept_sock;

    WSAEventSelect(accept_sock, event, FD_READ | FD_CLOSE);//소켓 이벤트를 모니터하기 위해 FD_READ및 FD_CLOSE 이벤트를 등록

    total_socket_count++;
    printf("[TCP server] 새로운 사용자 접속 (IP주소: %s)\n", inet_ntoa(addr.sin_addr));


    sprintf(msg, "[TCP server] 새로운 사용자 접속\n");
    notify_client(msg);
}
```

(add_client함수의 핵심 내용)

(5) add_client()함수는 FD_ACCEPT 이벤트 발생 시 실행되는 함수이다. 우선 클라이언트를 accept한 다음 IP를 스택에서 비교해 접속을 허용할지, 접속을 차단할지 결정한다.

```
//쓰레드를 추가적으로 생성해서 클라이언트가 송신한 메세지를 수신하고 수신한 메세지를 각 클라이언트들에게 전송한다
int read_client(int index)
{
    unsigned int tid;
    HANDLE mainthread = (HANDLE)_beginthreadex(NULL, 0, recv_and_forward, (void*)index, 0, &tid);
    WaitForSingleObject(mainthread, INFINITE);

    CloseHandle(mainthread);

    return 0;
}
```

(read_client 함수의 핵심 내용)

(6) read_client() 함수는 FD_READ 이벤트 발생 시 쓰레드를 추가적으로 생성해서 클라이언트가 송신한 메세지를 수신하고 수신한 메세지를 각 클라이언트들에게 전송하게 한다. 쓰레드 함수에는 recv_and_forward()함수가 사용되었는데, 아래에서 설명하도록 하겠다.

```
//CLIENT 메세지 입력->CLIENT 암호화->CLIENT 인코딩->CLIENT에서 전송->
//SERVER 수신->SERVER 디코딩->SERVER 복호화
//SERVER 암호화->SERVER 인코딩->SERVER에서 전송->모든 CLIENT에게
//클라이언트로부터 받은 데이터는 암호화된 후 인코딩되었음
recv_len = recv(sock_array[index].s, recv_msg, MAXBYTE, 0);
if (recv_len > 0)
{
    addr_len = sizeof(client_address);
    getpeername(sock_array[index].s, (SOCKADDR*)&client_address, &addr_len);

    //클라이언트로부터 받은 메세지를 디코딩(base64_decode) 해서 buf에 저장
    strcpy(buf, base64_decode(recv_msg, strlen(recv_msg), &decoding_num));

    printf("──────────────────────────────────────────────────────────────\n");
    printf(" * Client recv()로 받은 Encoded+Encrypted data입니다\n");
    textcolor(6);
    printf("%s\n", recv_msg);
    textcolor(7);
    printf(" * 이후 Decoding하면, Encrypted data가 출력됩니다\n");
    textcolor(10);
    for (int i = 0; i < strlen(buf); i++) {
        printf("%02X ", buf[i]);
    }//hexadecimal 형태로 출력
    textcolor(7);
    printf("\n");


    //암호화된 상태의 buf를 복호화
    data_decryption(buf, strlen(buf));

    //디코딩->복호화된 내용을 서버에서 출력
    strcpy(recv_msg, DEC);
    printf(" * 아래는 Decoding-> Decryption 이후의 plain data입니다\n");
    textcolor(9);
    printf("%s\n", recv_msg);
    textcolor(7);
    printf("──────────────────────────────────────────────────────────────\n");

    //sock_array에 저장된 모든 소켓에 유니캐스팅. notify_client함수에 암호화 및 인코딩 기능 포함됨
    notify_client(recv_msg);
}

_endthreadex(0);
return 0;
```

(recv_and_forward()함수의 핵심 내용)

(7) recv_and_forward()함수는 사용자 정의 쓰레드 함수로, 메시지를 수신받은 뒤 메시지를 Decoding 후 Decryption해서 서버에 메시지를 출력해준 다음, 다시 Encryption 후 Encode해서 연결된 모든 클라이언트들에게 유니캐스팅하는 함수이다.

(8) remove_client() 함수는 FD_CLOSE 이벤트 발생 시 클라이언트를 제거하는 함수로, closesocket()및 WSACloseEvent()과정을 포함한다.

```
//서버에서 모든 클라이언트에게 메세지를 보내주는 함수(유니캐스팅)
int notify_client(BYTE message[])
{
    BYTE server_msg[MAXBYTE] = "\0";
    memset(server_msg, '\0', MAXBYTE);
    data_encryption(message, strlen(message));
    strcpy(server_msg, base64_encode(ENC, strlen(ENC), &encoding_num));
    for (int i = 1; i < total_socket_count; i++)
        send(sock_array[i].s, server_msg, MAXBYTE, 0);
    return 0;
}
//서버에서 클라이언트에게 메세지를 보낼 때 또한 암호화 및 인코딩 과정을 거쳐서 보낸다.
```

(9) notify_client() 함수의 경우, 서버와 연결된 모든 클라이언트들에게 반복문을 통해 유니캐스팅으로 메세지를 보내주는 함수이다. (모든 데이터는 전송 시 암호화 및 인코딩 과정을 포함한다.)

```
//메인 함수 시작지점
int main()
{
    textcolor(14);
    server_setup();
    textcolor(7);

    unsigned int thread_id;
    BYTE server_msg[MAXBYTE] = "\0";
    BYTE input[MAXBYTE] = "\0";
    HANDLE mainthread;

    mainthread = (HANDLE)_beginthreadex(NULL, 0, chat, (void*)0, 0, &thread_id);//주 스레드 생성
    if (mainthread)
    {
        while (1)
        {
            memset(server_msg, '\0', MAXBYTE);
            memset(input, '\0', MAXBYTE);

            strcpy(server_msg, "[TCP server] ");
            gets(input);//서버에서 클라이언트에게 공지할 메세지 입력받음
            if (strcmp(input, "/quit") == 0)
                break;
            strcat(server_msg, input);//공지 메세지의 형식으로 만들어줌
            notify_client(server_msg);//암호화+인코딩 후 유니캐스팅
        }
        server_close();
        WSACleanup();
        CloseHandle(mainthread);
    }

    return 0;
}
```

(서버 소스파일의 메인함수)

(10) mainthread에는 chat 쓰레드 함수를 실행하도록 _beginthreadex()함수를 사용해 주 쓰레드를 생성하였으며, 이는 클라이언트로부터 메시지를 수신 및 유니캐스팅해주는 과정을 포함한다. 또한 서버 콘솔창에서도 메시지를 입력하면 클라이언트들에게 공지 형식으로 유니캐스팅 할 수 있도록 프로그래밍 하였다.

다음은 클라이언트 소스파일(Secured_client.c)에 대해서 설명하도록 하겠다.

먼저 클라이언트를 실행하면 기본 저장된 서버로 접속할 것인지, 아니면 다른 서버로 접속할 것인지 결정한다. 기본 접속 서버는 LOCALHOST(127.0.0.1, port=9000)으로 설정하였다. 그 후 대화명을 입력받은 다음 client_init() 함수에서 서버에 접속하기 위한 소켓을 초기화하고 connect()를 시도한다. 이후 서버로부터의 메시지 수신과 클라이언트로부터의 메시지 송신을 동시에 수행하기 위해 쓰레드를 생성한다. 이 때 사용되는 쓰레드 함수는 chat()함수로 정의하였으며, 쓰레드로 실행되면서 지속적으로 서버로부터 수신할 메시지가 있는지 이벤트를 감시한다. 그리고 FD_READ 이벤트 발생 시, 서버로부터 수신한 메시지를 출력하며, FD_CLOSE 이벤트 발생 시 서버가 종료되었음을 알려주는 구조이다.

클라이언트 소스파일의 경우 사용자의 편의성을 위해서 UI를 구현했기 때문에, 서버-클라이언트 통신 간 인터페이스의 일관성을 맞추기 위한 알고리즘을 만들어 프로그래밍하였다. 이는 뒤의 UI 설계과제 내용에서 설명하도록 하겠다.

```
//wsastartup, socket, connect함수 포함
int client_init(char* ip, int port)
{
    SOCKET server_socket;
    WSADATA wsadata;
    SOCKADDR_IN server_address = { 0 };

    if (WSAStartup(MAKEWORD(2, 2), &wsadata) != 0)//윈속초기화
    {
        printf("WSAStartup 에러\n");
        return -1;
    }

    if ((server_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)//소켓생성
    {
        puts("socket 에러.");
        return -1;
    }

    memset(&server_address, 0, sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = inet_addr(ip);
    server_address.sin_port = htons(port);

    if ((connect(server_socket, (struct sockaddr*)&server_address, sizeof(server_address))) < 0)//서버소켓에 연결시도
    {
        puts("connect 에러.");
        return -1;
    }

    return server_socket;
}
```

(client_init()함수 내용)

```
WSAEventSelect(s, event, FD_READ | FD_CLOSE);
while (1)
{

    index = WSAWaitForMultipleEvents(1, &event, FALSE, INFINITE, FALSE);
    if ((index != WSA_WAIT_FAILED) && (index != WSA_WAIT_TIMEOUT))
    {
        WSAEnumNetworkEvents(s, event, &ev);
        if (ev.lNetworkEvents == FD_READ)//read이벤트 감지
        {
            memset(recv_msg, '\0', MAXBYTE);

            int len = recv(s, recv_msg, MAXBYTE, 0);//메세지를 수신받는 과정
            if (len > 0) {
                if ((*msgcntp) == 40 - 7) {     //클라이언트의 채팅창이 꽉 찼을 경우 채팅 내용을 초기화하고 출력해주기 위해 사용
                    chatclr();
                }
                chat_ui();//콘솔창에 채팅 ui를 다시 출력한다
                gotoxy(0, (*msgcntp + 7));
                strcpy(buf, base64_decode(recv_msg, strlen(recv_msg), &decoding_num));  //메세지 수신받았을 경우 Decoding 시행
                data_decryption(buf, strlen(buf));                                      //Decoding 이후 Decryption하면 plain data 획득
                strcpy(recv_msg, DEC);
                plain_len = strlen(recv_msg);       //UI의 일관성을 맞추기 위해 plain data가 채팅창 가로 size인 60자 이상을 넘길 경우
                row = (int)(plain_len / 60 + 1);    //이게 몇줄짜리 데이터인지 row에 저장
                (*msgcntp) += row;
                if (((*msgcntp) + row > 34)) {
                    chatclr();
                }
                time_t curr = time(NULL);
                struct tm* d = localtime(&curr);

                printf("[%02d:%02d]%s", d->tm_hour, d->tm_min, recv_msg);   //time함수를 사용해 서버로부터 메세지를 전송받았을 때의 시간 출력
                row = 0;//row를 0으로 다시 초기화
                chat_ui();
            }

        }
        else if (ev.lNetworkEvents == FD_CLOSE)//close이벤트 감지되면 클라이언트 소켓 끄고 서버 연결 종료
        {
            system("cls");
            printf("[TCP server] 서버와의 연결이 끊어짐\n");
            closesocket(s);
            break;
        }
```

(쓰레드 함수 chat()의 핵심 내용)

이벤트에 FD_READ와 FD_CLOSE를 추가해 각각의 이벤트가 발생했을 경우의 처리를 구분하여 프로그래밍 하였다. 통신의 핵심 기능을 살펴보면, FD_READ이벤트 발생 시 recv()함수로 수신받은 메시지를 디코딩 함수 및 복호화 함수를 사용해 메시지를 plain data로 만든 다음, 이것을 클라이언트의 콘솔 화면에 출력해주는 간단한 방식이다.

FD_CLOSE 이벤트가 감지되었을 경우 서버와의 연결이 끊어짐을 클라이언트에게 알리고 소켓을 닫는다.

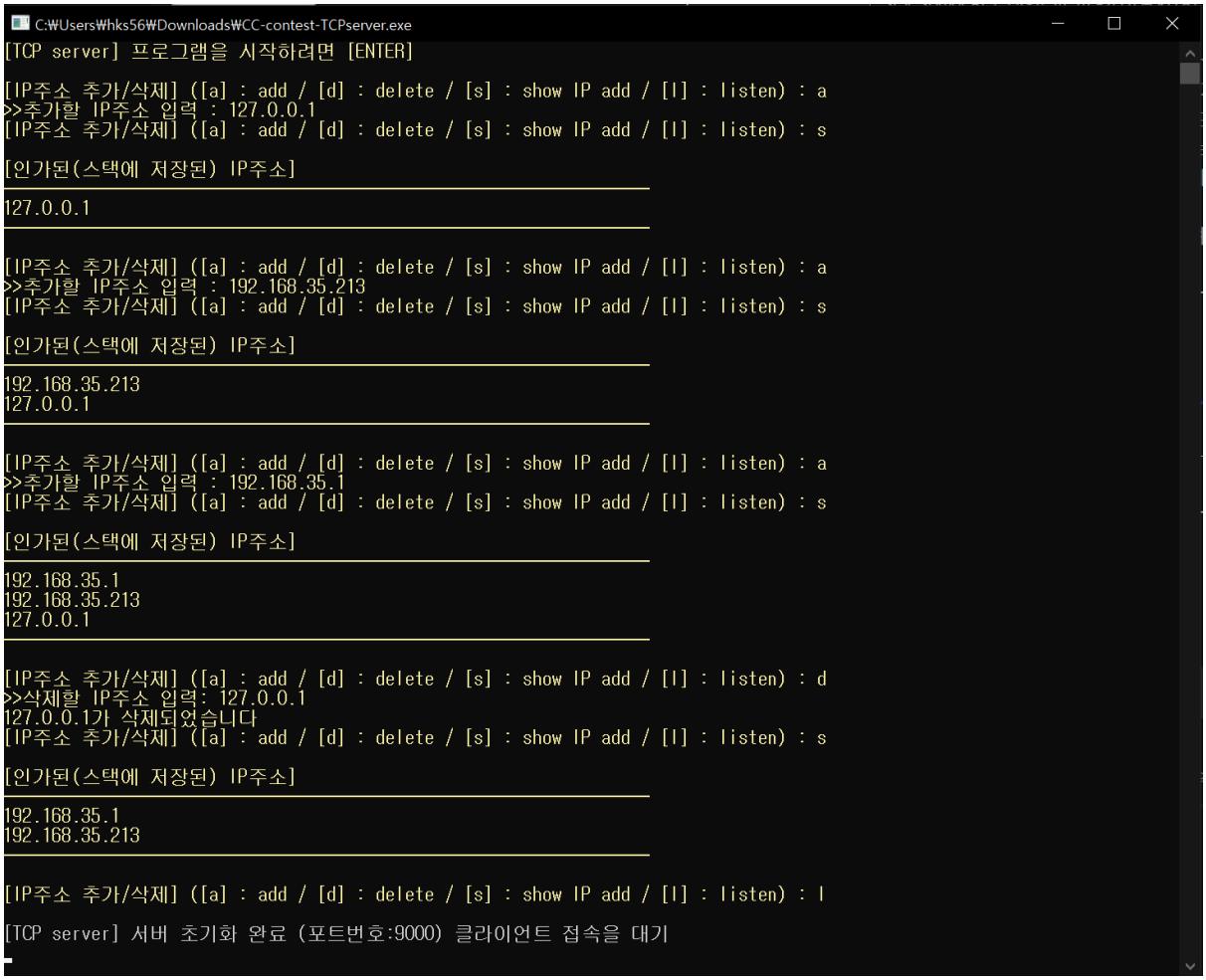
```
mainthread = (HANDLE)_beginthreadex(NULL, 0, chat, (void*)sock, 0, &tid);
if (mainthread)
{
    chatclr();
    while (1)
    {
        memset(input_msg, '\0', MAXBYTE);
        memset(send_msg, '\0', MAXBYTE);
        memset(buf, '\0', MAXBYTE);


        chat_ui();
        gets_s(input_msg, MAXBYTE);

        if (!strcmp(input_msg, "/quit")) {      // quit명령어 입력시 채팅 종료
            break;
        }
        else if (!strcmp(input_msg, "/cls")) {  // cls명령어 입력시 채팅 내용이 표시되는 화면 초기화
            chatclr();
            continue;
        }
        else if (((*msgcntp) > 34)) {           // 채팅창 화면이 꽉 찼을 경우, 자동으로 초기화
            chatclr();
        }
        else if (strlen(input_msg) == 0) {      // 사용자가 입력한 내용이 없을 경우, 전송하지 않고 채팅 단계를 반복
            continue;
        }

        sprintf(send_msg, "[%s] %s", nickname, input_msg);  //보낼 메세지에 닉네임과 보낼 내용을 형식지정을 통해서 저장


        //전송을 위해 암호화 및 인코딩
        data_encryption(send_msg, strlen(send_msg));
        strcpy(send_msg, base64_encode(ENC, strlen(ENC), &encoding_num));



        //메세지 전송하는 단계
        send(sock, send_msg, MAXBYTE, 0);
        gotoxy(0, (*msgcntp + 7));

    }

    closesocket(sock);
    WSACleanup();
    CloseHandle(mainthread);
}

return 0;
```

(클라이언트 소스파일의 메인함수 핵심 내용)

서버와 마찬가지로 _beginthreadex()함수를 사용해 chat()함수를 쓰레드로 실행시킨다. 지속적으로 메시지를 수신받음과 동시에 전송할 수 있도록 gets_s()함수를 사용해 메시지를 입력받은 다음, 이것을 암호화 후 인코딩하여 전송 규격에 맞게 만들어 서버로 전송하게 한다.

2) IP주소에 기반한 Network Access Control

서버의 폐쇄성을 위해 인가된 사용자(Client)만이 채팅 서버에 접속할 수 있도록 서버관리자가 서버 최초 실행 시 인가할 IP를 스택 안에 추가하고 Listening상태로 돌입한다.



서버 최초 실행 시 위 그림과 같이 a, d, s, l의 4가지 옵션이 주어지며, 서버 관리자에 의해 a 입력 시 인가할 IP를 문자열의 형식으로 추가할 수 있도록 구현하였고, 이는 사용자 정의 함수들로 구현된 연결리스트 스택에 저장되도록 하였다.

```
typedef char Element; // 형변환을 용이하게 하기 위해 Element 정의

typedef struct LinkedNode { // 노드 구조체(네트워크 대역이 저장됨)
    Element SECURED_IPADD[15]; // IP가 문자열로서 저장됨
    struct LinkedNode* link; // 노드의 link로서 다른 노드들과 연결됨
} Node;
```

서버관리자의 실수로 IP가 잘못 입력되었을 경우에 대응하기 위해 s, d 입력옵션을 추가하여 s입력 시 위 실행 예시와 같이 a입력을 통해 추가된 IP가 print_stack() 함수를 통해 출력될 수 있도록 하였고,

```
// 스택 안의 데이터 요소 출력하는 함수
void print_stack()
{
    Node* p;
    for (p = top; p != NULL; p = p->link)
        printf("%sWn", p->SECURED_IPADD);
    // p가 top에서부터 링크를 따라가 끝까지 각 노드의 데이터값(문자열)을 출력
}
```

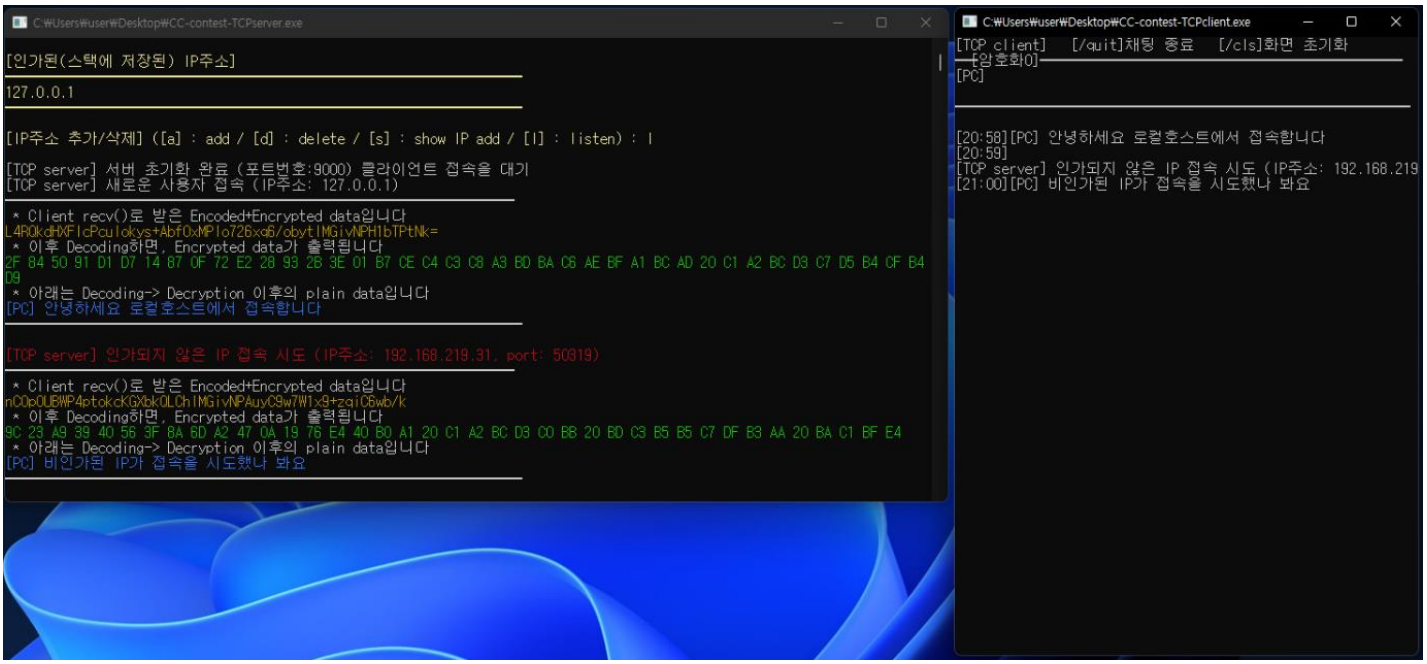
d입력을 통해 잘못 입력되어진 IP를 삭제할 수 있도록 하였다. 후입선출로 동작하는 스택이므로 pop()을 통해 삭제되어지지 않고, search_pop()함수를 통해 원하는 IP를 삭제할 수 있도록 하였다.

```
// 원하는 IP만 POP하도록 하는 함수
void search_pop(int count)
{
    if ((count - 1) != 0)
    {
        Node* p = top;
        Node* a;
        if (is_empty()) // 공백상태인지 검사
            error("스택 공백 에러Wn");

        for (int i = 1; i < count - 1; i++)
            p = p->link;
        a = p;
        for (int i = 0; i < 2; i++)
            a = a->link;
        p->link = a;
    }
    else if (count == 1)//top에 있는 걸 삭제하고싶으면?
        pop();
}
```

만약, 인가되지 않은 Client IP가 서버로의 접속을 시도할 경우, accept() 이후 데이터 송신/수신 (send()및 recv()) 이전에 closesocket() 함으로써 접속을 제한한다.

```
accept_sock = accept(sock_array[0].s, (SOCKADDR*)&addr, &len); // 우선 모든 클라이언트에 대해서 accept한 다음,

    /*스택 안에 저장된(인가된) IP주소가 아닐 경우*/
if (search_stack(inet_ntoa(addr.sin_addr)) == NULL) {
    textcolor(4);
    printf("Wn[TCP server] 인가되지 않은 IP 접속 시도 (IP주소: %s, port: %d)Wn",
        inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    textcolor(7);
    sprintf(msg, "Wn[TCP server] 인가되지 않은 IP 접속 시도 (IP주소: %s, port: %d)Wn",
        inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
    notify_client(msg);
    closesocket(accept_sock); // IP주소를 읽어와 stack안의 IP와 비교하여 없을 경우 소켓 닫으며 접속을 차단함
    return 1;
}
```

(↑ Secured_Server.c 소스파일 내 add_client() 함수 中 일부 발췌)



(↑ 인가되지 않은 IP Client가 서버에 접속을 시도할 경우 서버 및 기존 클라이언트 화면)

위 사진과 같이 인가되지 않은 IP의 접속 시도가 서버에 기록됨으로써, 서버 관리자는 해당 IP에 대해 인식할 수 있으며, 채팅 서버에 대한 네트워크 접근 제어(NAC)가 가능해진다.

3) 암호화 및 인코딩

본 채팅 프로그램은 스니핑 공격(네트워크 상 데이터 도청)에 대응할 수 있도록 높은 보안성이 요구되어지므로, 데이터 송·수신(send(), recv()) 간 고급 암호화 표준(AES)방식에 따라 실제로 암호화된 데이터가 송·수신되어야 한다. n명의 클라이언트가 접속하는 멀티스레드 채팅 서버 특성 상 병렬적으로 코드가 실행되었을 때 컴퓨터의 성능을 저하시킬 우려가 있으므로, AES_128_ECB 알고리즘을 사용해 암호화 과정을 최대한 단순화(ECB Mode)하고, 128bit(=16Bytes)의 키 길이를 가지도록 하여(AES 128) 최대한 줄이고자 하였다. (Round 0 포함한 총 Round 수 = 11 Rounds)

기본적으로, 보안 채팅 서버와 보안 클라이언트 프로그램 n개를 구동하면, 현재 별다른 네트워크 디바이스가 없는 환경에서 통신하므로 링크 암호화로서 구현되어 중계 노드인 채팅 서버에서도 암·복호화가 이루어진다. 그에 반해 일반 채팅 서버와 보안 클라이언트 프로그램 n개를 구동하면, 중계 노드인 채팅 서버에서는 암·복호화가 이루어지지 않으므로 서버에서는 (암호키가 없으므로) 데이터가 무엇인지 구별하지 못하는 종단간 암호화로서 구현된다.

우리 조가 사용한 AES 128 (ECB Mode)는 128bit(=16Bytes)의 크기를 가진 Block단위로 데이터가 암호화된다. AES 암호키는 대칭키 방식으로 암·복호화에 동일한 키가 사용되어 Client 프로젝트와 Server 프로젝트 각각에서 encryption-decryption.c 소스파일 내에 동일한 키를 선언해주었다.

```
BYTE Key[] = { 0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c };
```

(↑ 암호키 길이 = 16Bytes = 128bit // AES 128)

해당 암호키를 사용한 암호화 알고리즘은 다음과 같이 구현되었다.

```
// ECB -> 김찬영, 허진환 팀 사용 AES모드
// (블록 단위로 순차적 암호화, 블록=16Byte이므로 입력되는 데이터 Input을 16Byte로 맞춰주는 Padding 과정이 필요)
#if ECB
// AES_ECB 암호화 함수
VOID WINAPI AES_ECB_Encrypt(LPCBYTE Input, LPCBYTE Key, LPBYTE Output, int Length) {
    BYTE ExpKey[KEYEXPSIZE]; // 총 Key 사이즈(주암호키+각 라운드키)를 받아 키값이 저장될 공간(변수) 선언

    memcpy(Output, Input, Length);
    KeyExpansion(ExpKey, Key); // 초기 Key Expansion을 통해 각 라운드 라운드키 생성
    Cipher((BYTE(*)[4])Output, ExpKey); // 주암호키와 라운드키로 최종 암호화
}
```

```
// 최종 암호화 과정 -> 김찬영, 허진환 팀 기준 AES_128(bits)는 총 Round가 11번 실행됨(초기 Round 0 포함).
LOCAL(VOID) Cipher(BYTE State[4][4], LPBYTE ExpKey) {
    AddRoundKey(State, ExpKey, 0); // Round00서의 AddRoundKey() 실행

    for (int Round = 1; Round < Nr; Round++) { // AES_128이므로 Nr(초기 라운드(Round0) 제외 라운드 수) == 10이고, Round0을 제외한 Round1~9까지의 밑 1~4과정 9번 반복
        SubBytes(State); // 1. SubBytes()
        ShiftRows(State); // 2. ShiftRows()
        MixColumns(State); // 3. MixColumns()
        AddRoundKey(State, ExpKey, Round); // 4. AddRoundKey()
    } // 여기까지 마쳤을 시 초기 Round 포함 총 Round 수 = 10

    // 마지막 Round10과정
    SubBytes(State);
    ShiftRows(State);
    // Round10에서 MixCollum()과정 생략
    AddRoundKey(State, ExpKey, Nr); // 모든 과정 마치면 초기 Round 포함 총 Round 수 = 11 -> AES_128기준 총 Round 수 = 11
}
```

초기 KeyExpansion() 과정 이후 Cipher()함수를 통해 Round 시작

① Round 0 : AddRoundKey()

② Round 1~9 : SubBytes() → ShiftRows() - MixColumns() - AddRoundKey() 과정 반복

③ Round 10 : SubBytes() - ShiftRows() - AddRoundKey()

(각 함수들에 대한 상세설명 및 AES알고리즘 과정은 코드 내 주석에 포함됨.)*

복호화 알고리즘은 암호화 알고리즘의 역순으로 구현하였다.

```
// 최종 복호화 과정 // 암호화의 역순
LOCAL(VOID) InvCipher(BYTE State[4][4], LPBYTE ExpKey) {
    AddRoundKey(State, ExpKey, Nr);

    for (int Round = Nr - 1; Round > 0; Round--) {
        InvShiftRows(State);
        InvSubBytes(State);
        AddRoundKey(State, ExpKey, Round);
        InvMixColumns(State);
    }

    InvShiftRows(State);
    InvSubBytes(State);
    AddRoundKey(State, ExpKey, 0);
}
```

실제로 채팅 프로그램 작동 시 데이터가 암·복호화될 수 있도록 위 AES 알고리즘을 aes.c 소스파일에 담아두고,

```
BYTE ENC[MAXBYTE]; // 암호화되어 담길
BYTE DEC[MAXBYTE]; // 복호화되어 담길

/* 통신 중 send()되는 순수 data를 aes_128로 암호화하는 함수 */
void data_encryption(unsigned char* plain_data, int len) {
    memset(ENC, '₩0', MAXBYTE);//암호화될 데이터가 담길 문자열 각 요소를 NULL로 초기화
    while (strlen(plain_data) % 16 != 0)// aes_128은 블록 단위로 순차적 암호화, 블록 = 16Byte이므로 입력되는 데이터 Input을 16Byte로 맞춰주는 Padding 과정
        plain_data[strlen(plain_data)] = ' '; // data길이가 16으로 쪼개지지 않으면, 계속 ' '을 찍어 문자열 길이를 16의 배수로 맞춤

    AES_ECB_Encrypt(plain_data, Key, ENC, len);// 암호화 진행 (aes.c에 정의)
}

/* 통신 중 recv()되는 암호화된 data를 복호화하는 함수 */
void data_decryption(unsigned char* cipher_data, int len) {
    memset(DEC, '₩0', MAXBYTE);// 복호화될 데이터가 담길 문자열 각 요소 NULL로 초기화
    AES_ECB_Decrypt(cipher_data, Key, DEC, len);// 복호화 진행 (aes.c에 정의)
}
```

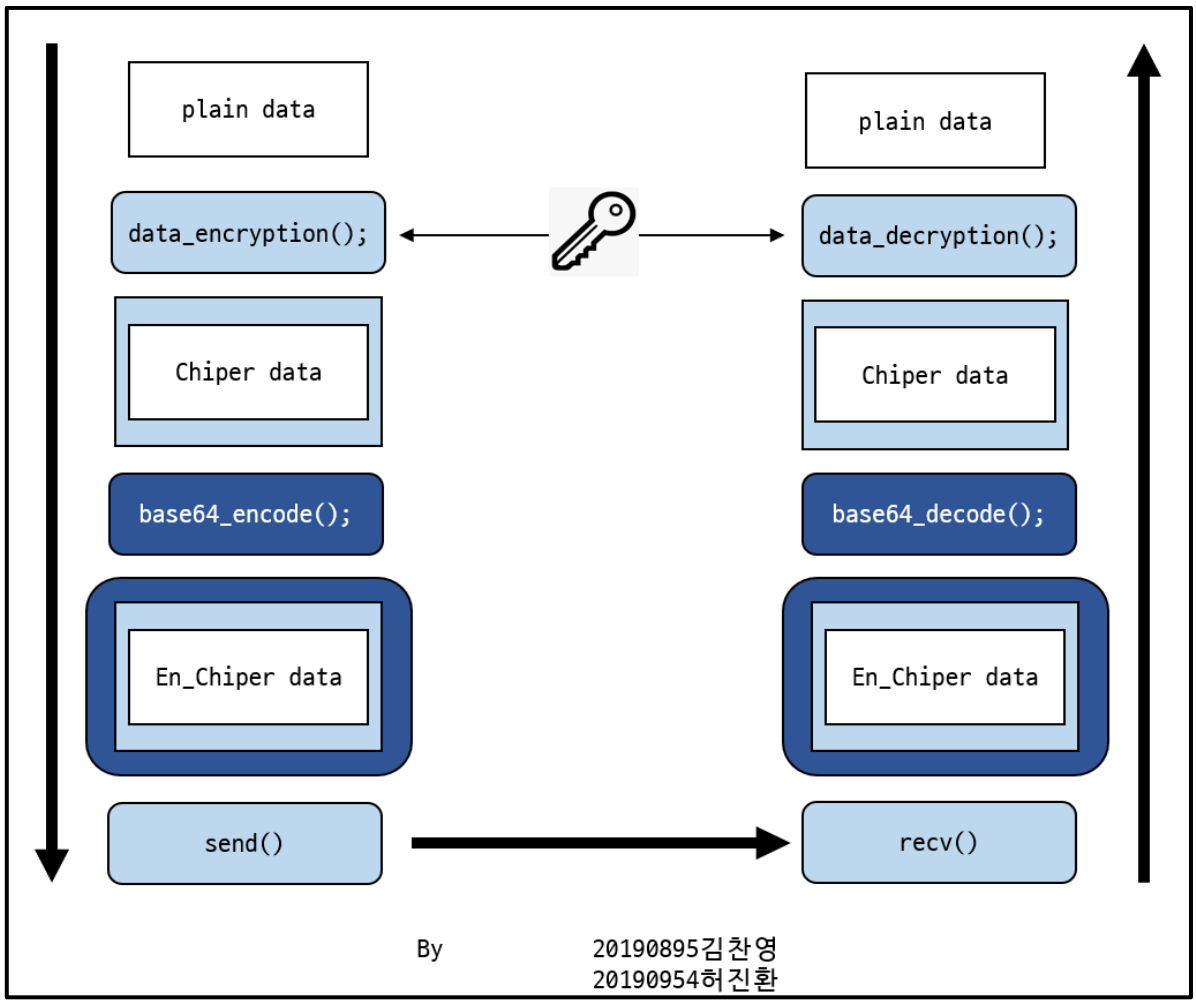
다음과 같은 전역변수와 암·복호화 함수를 encryption-decryption.c 소스파일 내에 정의하여 실제 데이터 암·복호화가 수행되도록 하였으며, 전역 변수인 암호화된 Cipher Text가 들어갈 Unsigned Char 배열 ENC와 복호화된 Plain Text가 들어갈 Unsigned Char 배열 DEC는 extern을 통해 링크 과정에서 소스파일 간 해당 전역 변수들이 공유되도록 하였다.

```
//암호화, 복호화에 필요한 변수
extern int length;          //암호화할 문자열의 길이
extern BYTE ENC[MAXBYTE];   //암호화 함수 사용시 ENC에 암호화된 내용 저장
extern BYTE DEC[MAXBYTE];   //복호화 함수 사용시 DEC에 복호화된 내용 저장
```

(↑ Secured_Server.c에서 해당 전역변수가 공유되도록 함 / Client의 경우도 동일)

2.2) 단락에서 잠깐 언급되었지만, AES에 의해 암호화된 Data가 송·수신될 때 각각 인코딩·디코딩 과정이 수행된다. 이 이유는 기본적으로 AES에 의해 암호화된 Data는 이진 데이터이므로 데이터 교환 시 수신지에서 텍스트 형식으로 해당 데이터를 읽으려고 할 때 format이 안 맞는 경우가 있기 때문이다. 본 채팅 프로그램에서는 Unsigned Char 배열 자료형으로 Cipher Text와 Plain Text를 처리하여 Base64 인코딩/디코딩을 하지 않아도 데이터 송·수신 간 사용자가 데이터를 읽는 데 문제는 없지만, 프로그램의 이식성을 높이고, 혹시 모를 버그를 방지하기 위해 Base64 인코딩/디코딩을 추가하였다.

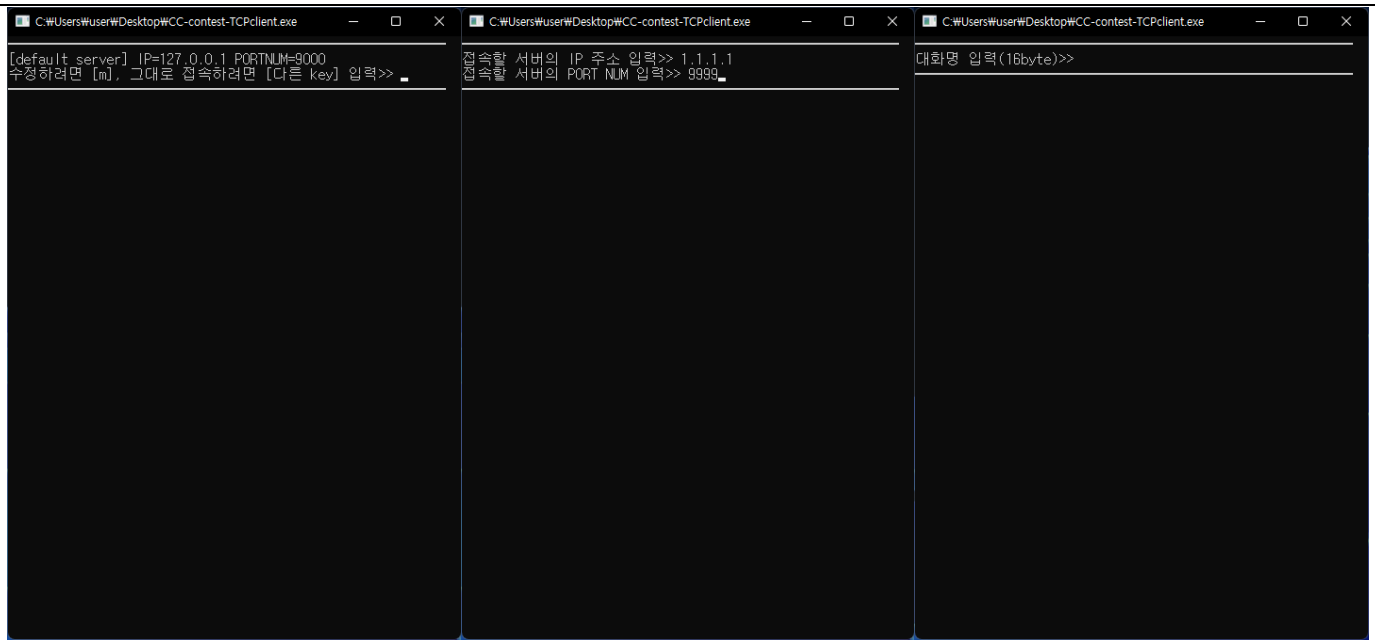
따라서, 데이터가 송신될 때(send())는 원래의 데이터를 (1)암호화하고 (2)인코딩하여 송신하고, 수신될 때(recv())는 이렇게 암호화+인코딩된 데이터를 plain_data로 만들기 위해 송신 시의 역순으로 (1)디코딩하고 (2)복호화하여 해석하게 된다. 간단히 그림으로 표현하면 아래와 같다.



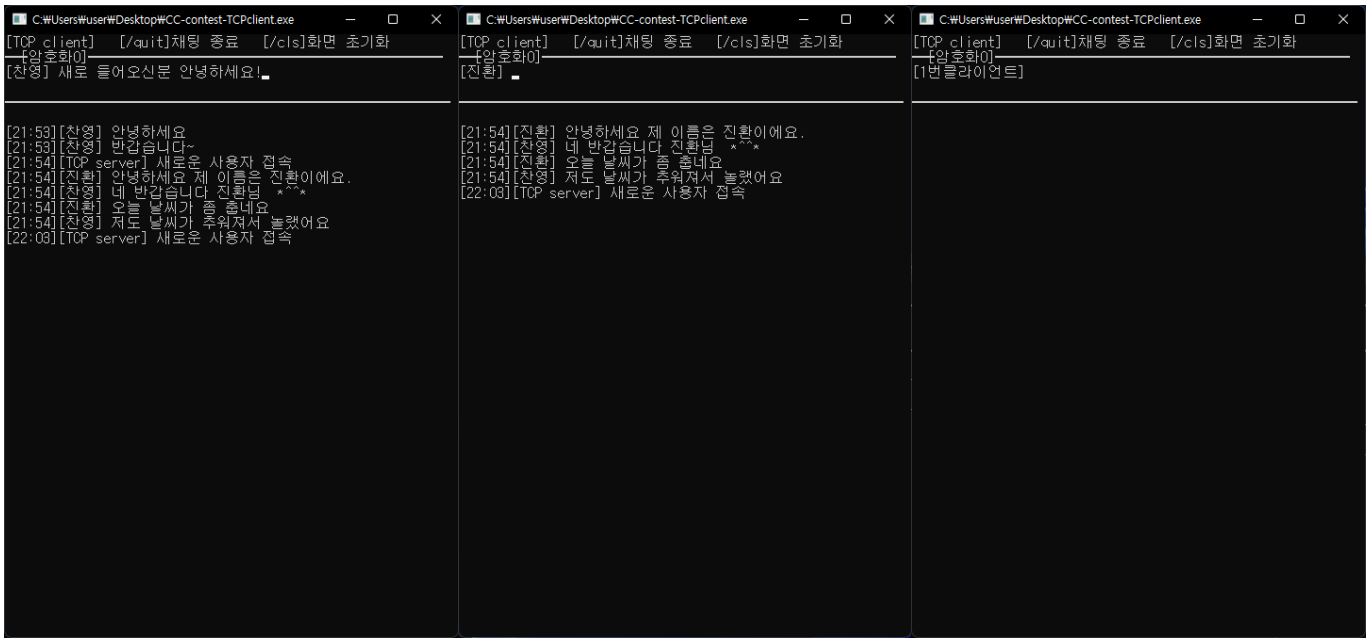
4) 클라이언트 프로그램 UI

이번 과제를 수행하면서 가장 중요하게 목표로 생각했던 것은 암호화 통신이 제대로 되는 것이며, 두번째는 직관적이고 한 눈에 들어오는 UI이다. 결국 프로그램은 사람이 사용하는 것이기 때문에, 사용자가 프로그램을 이용함에 있어서 불편한 요소를 최대한 제거하기 위해서 UI를 프로그래밍 하였다. UI를 구현하는데 있어서 사용자 정의 함수로 널리 사용하는 gotoxy()함수와 현재 시간에 대한 정보를 얻을 수 있는 time()함수를 사용하였다.

클라이언트 프로그램의 콘솔창 크기는 system("mode con:cols=60 lines=40"); 함수를 사용해 가로60줄, 세로 40줄로 선언하였다.

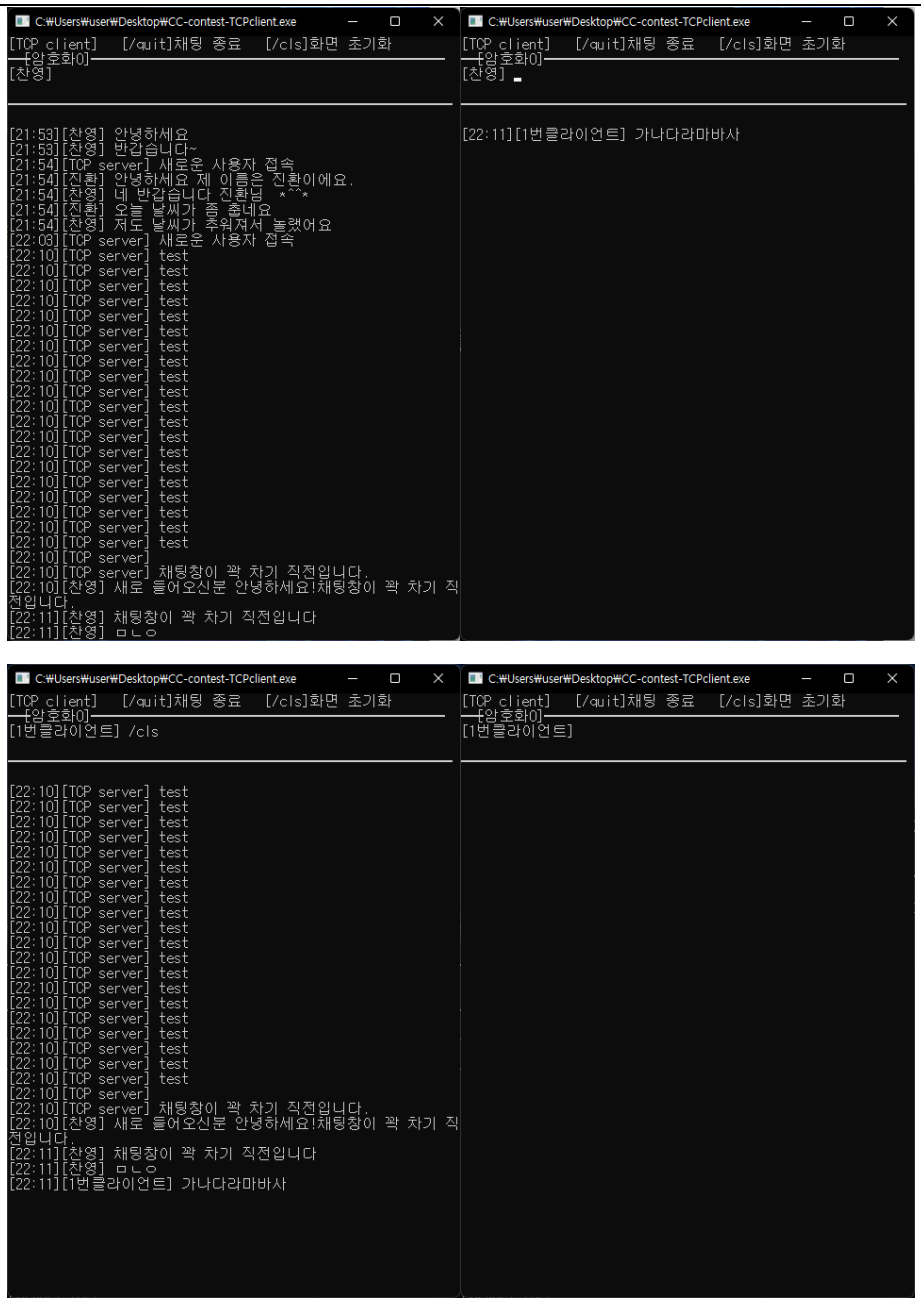


위 3개의 사진은 좌측부터 순서대로 (1)처음 실행 시 Default server에 접속할 것인지, 아니면 다른 서버에 접속할 것인지 결정하는 화면, (2)m을 눌러서 접속 서버를 수정하는 모습, (3)이후 대화명을 입력하는 화면이다.



대화명을 입력하고 정상적으로 서버에 접속을 하면 다음과 같이 화면을 볼 수 있다. 본인이 설정한 닉네임이 출력되고 그 옆에 채팅 내용을 입력하는 형식으로, 내용을 입력하는 위치와 메시지가 출력되는 위치를 위-아래로 분리하

여 좀 더 직관적으로 채팅 내용을 읽고 채팅을 칠 수 있도록 프로그래밍 하였다. 그리고, 클라이언트에서 서버로 메시지가 전송된 후 다시 서버에서 모든 클라이언트들에게 유니캐스팅을 해 주면(notify_client()), 그 때의 시각을 같이 채팅창에 표시해줘 몇시에 어떠한 메시지를 전송받았는지 알 수 있다.



좌측의 사진처럼 채팅을 하던 도중 채팅창 화면이 꽉 차게 되었을 경우, 서버로부터 새로 채팅을 전송받을 경우 화면에 제대로 표시를 하기 위해 채팅창 화면을 한번 지워주고 새로 위에서부터 출력해준다. 또한, 아래 화면처럼 채팅창이 꽉 차지 않아도 /cls 명령어를 입력하면 채팅창을 사용자가 원하는 즉시 지울 수 있다.

다음으로 코드를 첨부하여 설명하겠다.

```
while (1)
{
    memset(input_msg, '\0', MAXBYTE);
    memset(send_msg, '\0', MAXBYTE);
    memset(buf, '\0', MAXBYTE);


    chat_ui();
    gets_s(input_msg, MAXBYTE);

    if (!strcmp(input_msg, "/quit")) {      // quit명령어 입력시 채팅 종료
        break;
    }
    else if (!strcmp(input_msg, "/cls")) {  // cls명령어 입력시 채팅 내용이 표시되는 화면 초기화
        chatclr();
        continue;
    }
    else if (((*msgcntp) > 34)) {           // 채팅창 화면이 꽉 찼을 경우, 자동으로 초기화
        chatclr();
    }
    else if (strlen(input_msg) == 0) {      // 사용자가 입력한 내용이 없을 경우, 전송하지 않고 채팅 단계를 반복
        continue;
    }

    sprintf(send_msg, "[%s] %s", nickname, input_msg);  //보낼 메세지에 닉네임과 보낼 내용을 형식지정을 통해서 저장


    //전송을 위해 암호화 및 인코딩
    data_encryption(send_msg, strlen(send_msg));
    strcpy(send_msg, base64_encode(ENC, strlen(ENC), &encoding_num));



    //메세지 전송하는 단계
    send(sock, send_msg, MAXBYTE, 0);
    gotoxy(0, (*msgcntp + 7));

}
```

위 사진은 main함수의 내용 중 클라이언트에서 메시지를 전송하는 부분이다. 여기서 gets_s(input_msg, MAXBYTE);를 통해 전송할 메시지를 입력받고, 이것을 ([닉네임] 내용~)형식에 맞도록 sprintf()함수를 사용해 send_msg에 저장하여 암호화+인코딩 후 전송하게 된다.

```
int len = recv(s, recv_msg, MAXBYTE, 0);//메세지를 수신받는 과정
if (len > 0) {
    if ((*msgcntp) == 40 - 7) {     //클라이언트의 채팅창이 꽉 찼을 경우 채팅 내용을 초기화하고 출력해주기 위해 사용
        chatclr();
    }
    chat_ui();//콘솔창에 채팅 ui를 다시 출력한다
    gotoxy(0, (*msgcntp + 7));
    strcpy(buf, base64_decode(recv_msg, strlen(recv_msg), &decoding_num));  //메세지 수신받았을 경우 Decoding 시행
    data_decryption(buf, strlen(buf));                                      //Decoding 이후 Decryption하면 plain data 획득
    strcpy(recv_msg, DEC);
    plain_len = strlen(recv_msg);       //UI의 일관성을 맞추기 위해 plain data가 채팅창 가로 size인 60자 이상을 넘길 경우
    row = (int)(plain_len / 60 + 1);    //이게 몇줄짜리 데이터인지 row에 저장
    (*msgcntp) += row;
    if (((*msgcntp) + row > 34)) {
        chatclr();
    }
    time_t curr = time(NULL);
    struct tm* d = localtime(&curr);

    printf("[%02d:%02d]%s", d->tm_hour, d->tm_min, recv_msg);   //time함수를 사용해 서버로부터 메세지를 전송받았을 때의 시간 출력
    row = 0;//row를 0으로 다시 초기화
    chat_ui();
}
```

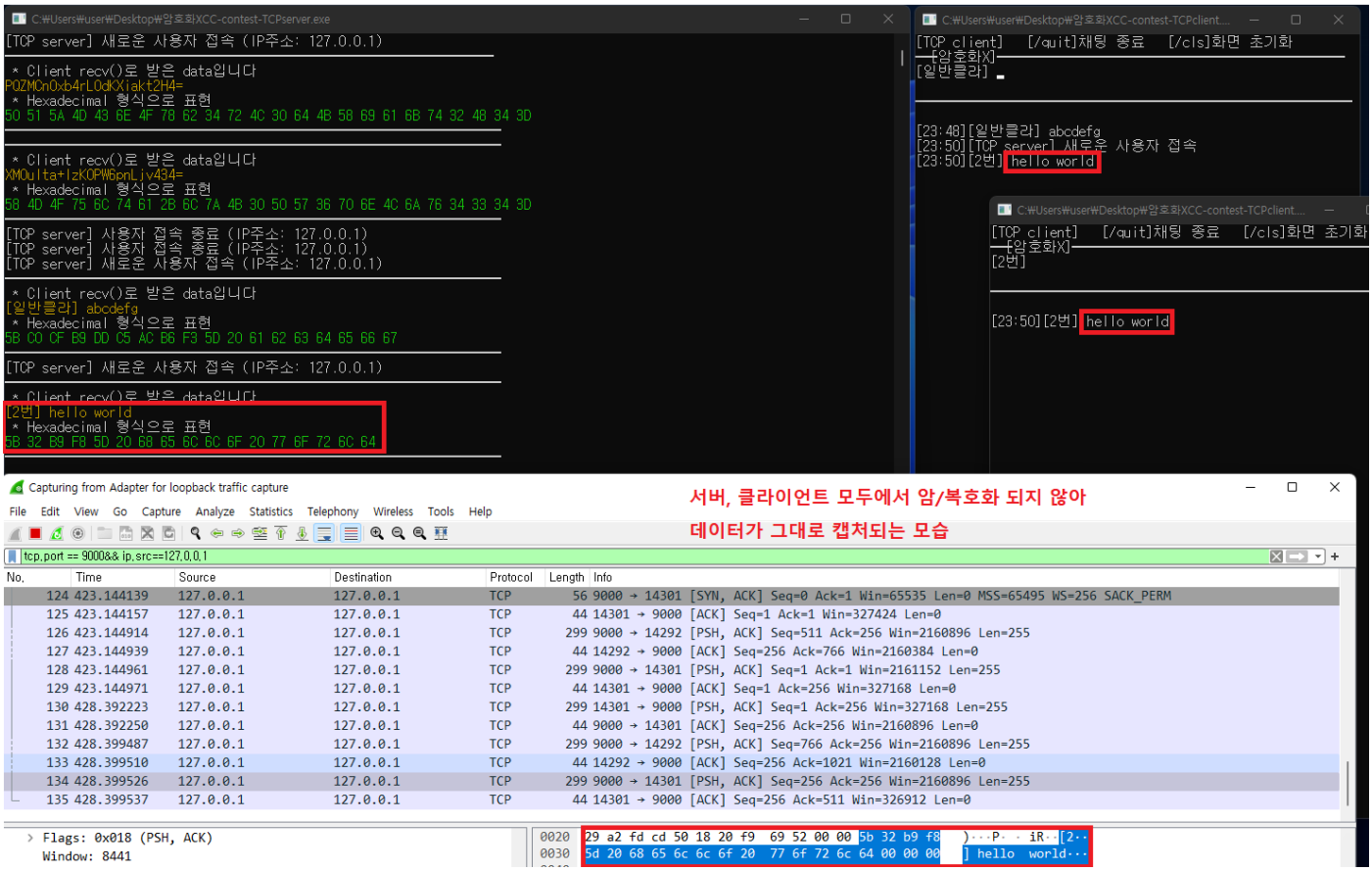
메시지를 클라이언트가(혹은 서버가) 전송해서 서버가 수신받고 다시 유니캐스팅을 해 준 다음, 클라이언트가 메시지를 수신받았을 때, msgcnt의 값이 33일 경우는 화면이 꽉 찬 상태임을 나타내기에, 이 경우에는 chatclr()사용자 정의 함수를 통해 채팅 내용을 초기화한다. Msgcnt는 현재 채팅창에 출력된 메시지가 총 몇줄인지 나타내는 정수형 변수이고, Msgcntp는 그것을 가리키는 포인터이다.

그 이후 수신받은 데이터가 가로 60자 기준 몇 줄짜리 데이터인지 구분하기 위해 int row 변수에 plain_len/60+1 값을 대입하여 구분한다. 그래서 row를 msgcnt와 더해서 다시 msgcnt에 그 값을 저장하고, 만약 그 값이 최대 줄 개수를 넘어갈 경우에 채팅 내용을 초기화하고 새로 출력해준다.

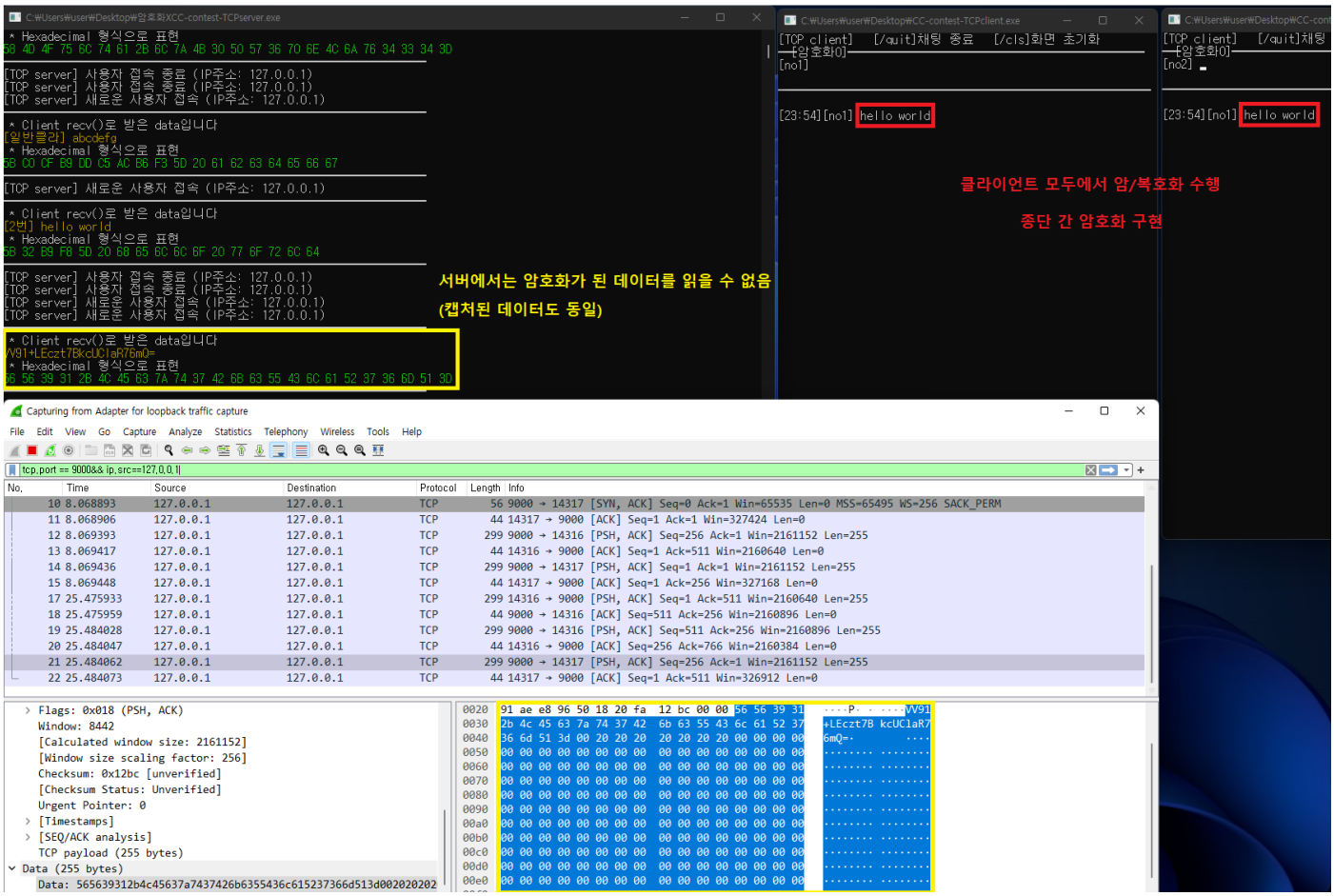
메시지를 출력할 때의 양식은 ([시:분] 전송받은 내용~) 으로 출력하게 된다. 애초에 클라이언트에서 보낼 때 닉네임과 내용이 합쳐져서 보내졌기 때문에, 다시 클라이언트에 출력할 때는 [시:분]만 추가적으로 출력하도록 프로그래밍 하였다.

이렇게 함으로써 위에 첨부한 스크린샷처럼 클라이언트가 동작함을 확인할 수 있다.

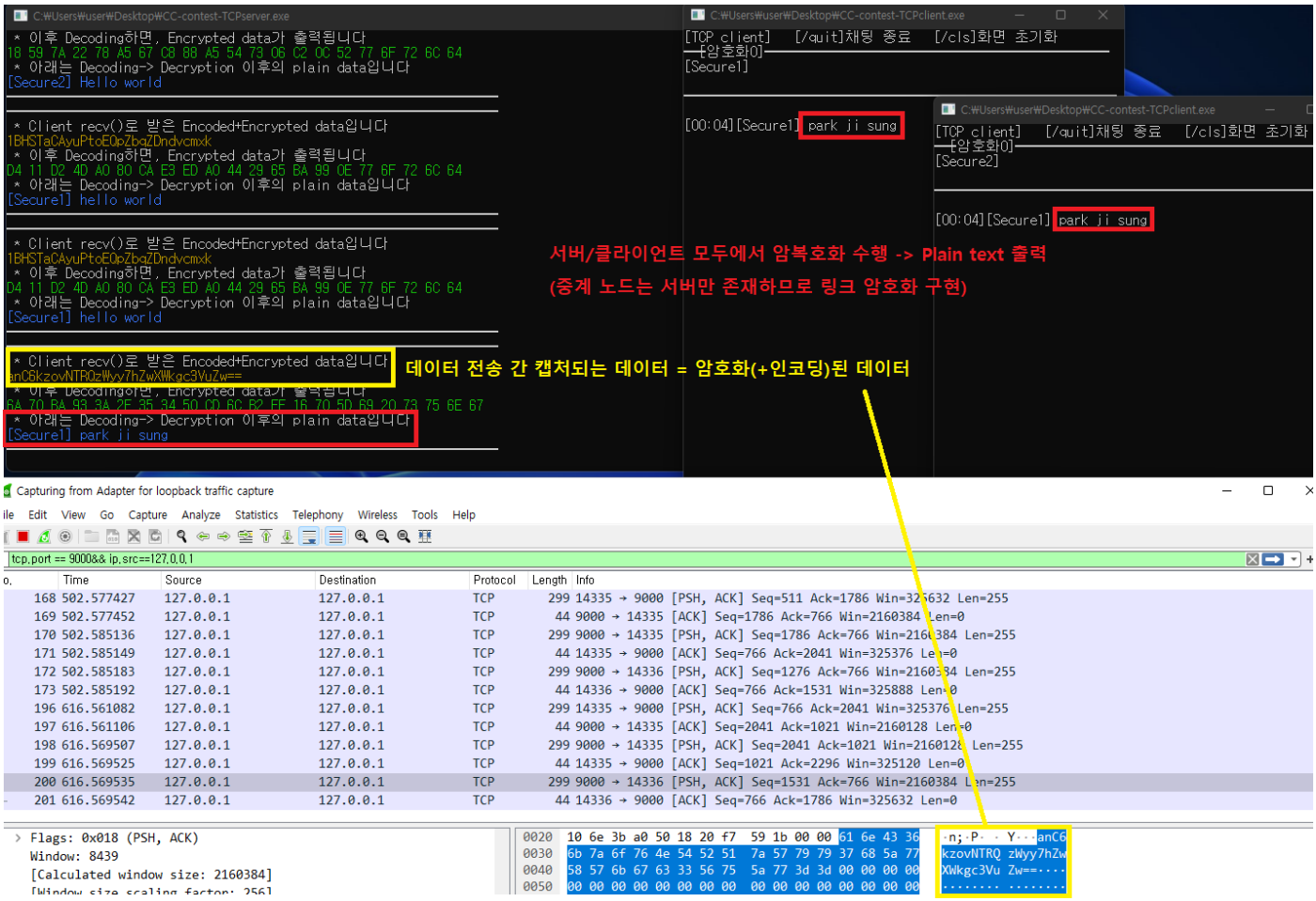
3. 디버깅 및 프로그램 동작



위 첨부화면은 암호화 및 복호화, 인코딩 및 디코딩을 거치지 않고 TCP 서버를 통하여 데이터가 전송되는 화면을 캡처한 것이다. WireShark 프로그램을 사용해서 캡처했을 경우, [일반클라] 사용자가 보낸 “hello world”라는 데이터가 그대로 캡처된다.



위 첨부화면은 암·복호화 및 인코딩,디코딩 기능이 구현되지 않은 일반 서버와 암·복호화 기능이 구현된 클라이언트 2개를 실행한 다음, “hello world”라는 데이터를 전송했을 시 WireShark에 캡처되는 화면이다. 클라이언트에서 암호화를 진행한 다음 데이터를 송신했기 때문에, WireShark와 일반 서버에서는 데이터가 암호화+인코딩된 상태로 캡처가 되고, 그 이후 일반 서버에서 연결된 클라이언트들에게 유니캐스트를 했을 경우 클라이언트는 복호화 기능이 존재하기 때문에 정상적으로 데이터를 복호화해서 plain data를 수신받을 수 있다. 결과적으로 이는 종단간 암호화를 구현하였음을 확인할 수 있다.



위 첨부화면은 암·복호화 및 인코딩, 디코딩 기능을 모두 포함하는 보안 서버와 보안 클라이언트 2개를 실행하여 “park ji sung”이라는 데이터를 전송하는 모습이다. WireShark로 데이터의 이동을 캡처했을 경우 암호화된 데이터가 인코딩된 상태로 전송되는 것을 확인할 수 있다. 현재 데이터는 Loopback Interface 상에서 송·수신되기 때문에 중간에 별다른 네트워크 디바이스가 없으므로, 보안 서버를 포함한 모든 클라이언트에서 암·복호화가 수행된다. 따라서 위 첨부화면상에서 링크 암호화가 구현되었다고 할 수 있다.