Semester project report

Fast Prototyping of Secure Applications :
Modeling and creating secure chat applications

Student: Vikram KATURI
Supervisor: Ludovic APVRILLE

EURECOM
Sophia Antipolis

Contents

Introduction

Foreword

For engineers working on designing and verifying complex systems, modeling plays a huge role. **TTool** (pronounced "tea-tool") is a toolkit dedicated to the **edition of UML and SysML diagrams**, and to the **simulation and formal verification (safety, security, performance)** of those diagrams. Hence, it can be used to model complex systems.
Ttool, with the help of its C code generator, enables us to generate an executable out of the designed model.

This project aims to demonstrate the use of the modeling and C code generator in Ttool to quickly prototype and realize a secure chat application. Although there was some work done along these lines earlier, the work was mainly concentrated around building the secure protocols, but a real time chat application eco-system was not realized.

Building a secure chat application would require developing the security protocols, programming the server to facilitate client to client communication, programming the client apps to make use of the security protocols and to establish a secure channel with remote peers.

Our goal is to modularize the aspects of these applications and provide the flexibility needed to make use of them in a plug and play manner to generate the applications as desired.
So the TTool users will be able to create their own secure chat application prototypes easily, like using a lego set.

Our application prototype requires a central server, and remote clients which connect to it individually, over tcp sockets and communicate only with it. The server is responsible for the end to end communication between the clients, like a message relay agent. I utilize the secure and simpler symmetric encryption protocol to secure messages between clients.

Previous Work

In the previous project, an implementation of the secure symmetric encryption protocol has been proposed between two entities, using Diffie-Hellman key-exchange and AES encryption. My work will make use of this previous work for implementation of secure encryption between clients communicating through a central server.

While the previous work concentrates on implementing the encryption protocol between two blocks communicating in UDP with predefined message content, my work is going to focus on how to realize a client to client real-time chat application in a TCP client-server architecture using the c code generator from the Ttool models.

Secure Symmetric Encryption

Protocol Overview

The Secure Symmetric Encryption involves two steps:

Step 1: To generate and exchange keys.
Step 2: To use the generated keys and encrypt and decrypt messages.

To perform a secured key exchange, it uses Diffie-Hellman protocol, and to encrypt a message, it uses the AES algorithm.

Step 1: **Diffie–Hellman key exchange** is a method of securely exchanging cryptographic keys over a public channel

To better understand how it works, let's take an example where you have two parties A and B who wish to establish a secure communication channel.

They both have access to a public variable g.

A and B each generate a random number, respectively a and b and keep them secret.

Then A sends g^a to B and B sends g^b to A.

A computes $(g^a)^b$ and B computes $(g^b)^a$, g^{ab} is the key which has been securely exchanged.

Calculating x, given g and g^x is hard - discrete logarithm problem. So, The key exchange is safe from evesdroppers.

Step 2: **Advanced Encryption Scheme(AES)** uses the same key for encrypting and decrypting the messages.

To encrypt the message, A first starts to compute a message key by using a hash function over the message and the key. Once A has the message key, they use the message
key and the key to compute the AES key and the AES iv that will be used to perform the AES encryption algorithm. The message key generation is designed not to leak any information about neither the key nor the AES key and the AES iv .

Then A sends the message key and the encrypted message to B. While B knows the key from previous DH key-exchange, it
can compute the AES key and the AES iv thanks to the message key and the key. Then B can

decipher the message.

Functions used from previous work

Here I briefly mention the functions I use from the previous work for achieving Diffie-Hellman and AES, without going deeply into their internal implementations.

(the function signatures are representational)

Diffie Hellman:
random_a() - generates a random number of 256 bytes. (for example : a)
DH_client(a) - takes in the random number generated by random_a() and calculates g^a mod p
DH_clientthird(b , g^amod p) - takes g^a mod p and b and calculate g^{ab}mod p , which is the key.

AES:
aes_ige_encryption(msg) - takes in the input buffer msg and returns a message key followed by the encrypted message.
aes_ige_decryption(encrypted_msg) - takes in the encrypted msg buffer and outputs the decrypted message.

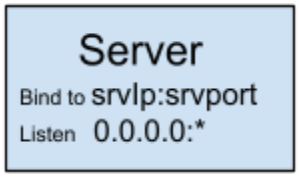
Chat Application

Architecture and design

Server assisted client to client communication

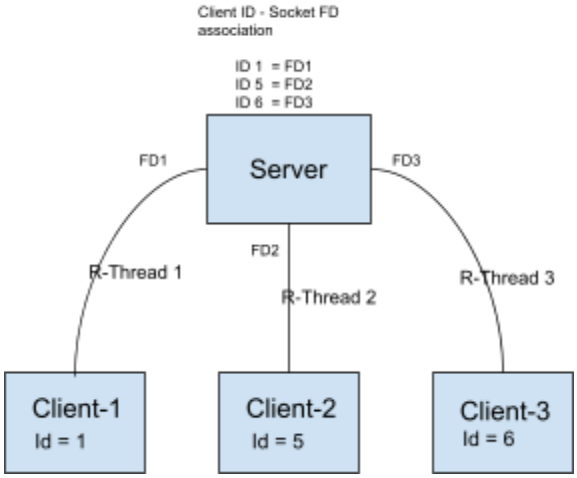
The design chosen for the chat application is a client-server architecture, with multiple clients and a single central server. This solution enables easy scaling and ease of implementation with generic and light weight server, which only does basic functionality of forwarding/relaying.

The server binds to the given ip and port and listens for any incoming connection requests from clients to accept.



The clients will then connect to the server ip and port and advertise their client-id.

The server upon receiving a client connection and client-id advertisement, will create a dedicated thread to constantly read(Read thread) the incoming traffic from the client connection on the client connection's file descriptor(FD).

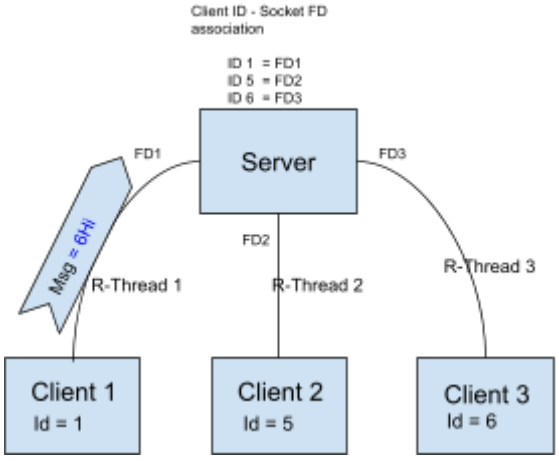


The server also maintains an association between the client ID and its connection FD.

The clients only know of the connection details of the server and not of the other clients. The clients, however, will know the client_id of their peers with whom they want to communicate with, like a user-id.

The clients will send a message to the server, prepended by the target_client_id.
For example: client 1, to send a message "Hi" to client 3, whose client_id is 6, will send "6Hi" to the server.

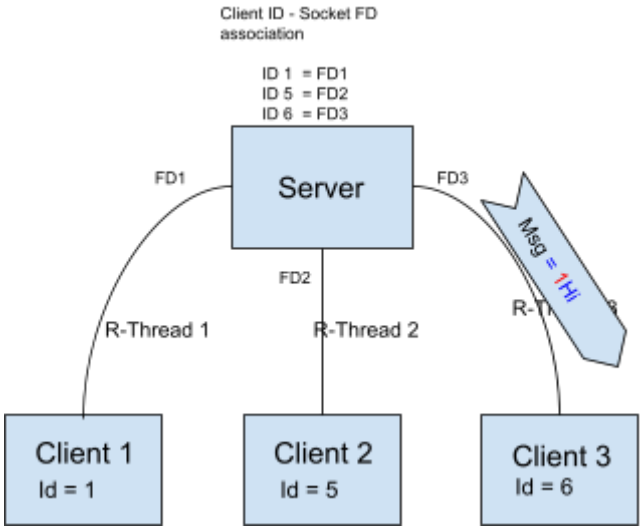
Client 1 (client ID 1), sending "Hi" to client 3(client ID 6) = Client 1 sending "6Hi" to Server.



The Server will now check the first byte of the incoming message on R-Thread1 from client 1 and figure out it is for client_id == 6.
It now looks up the “client_id : FD” association table and gets the socket FD of the target client.
ID-6 : FD-3

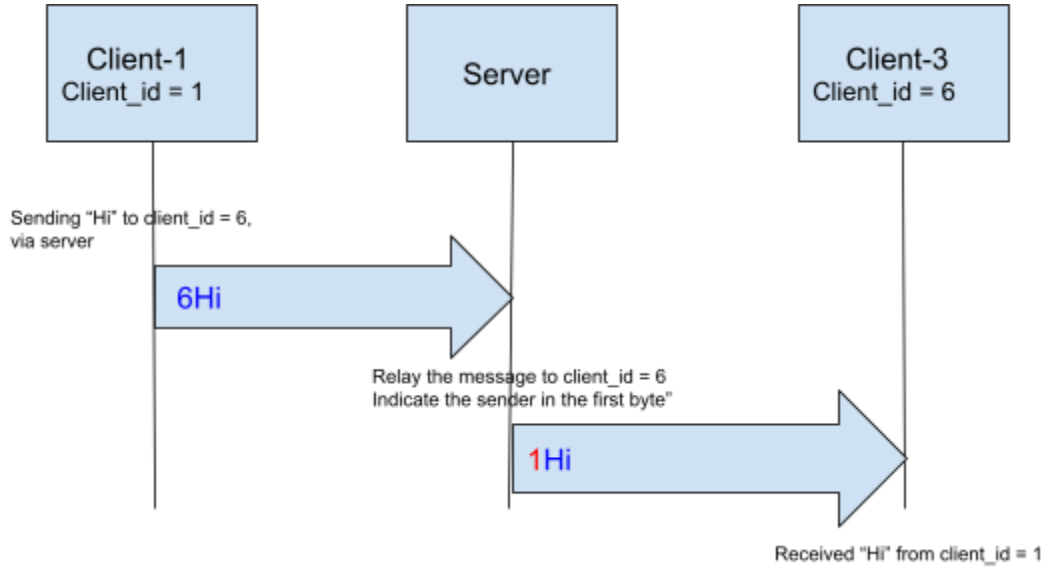
Now, the server rewrites the first byte of the message with the client_id of the sender, to indicate the sender of the message and sends it to the target_client’s FD.

Client 1 (client ID 1), sending “Hi” to client 3(client ID 6) = Client 1 sending “6Hi” to Server = Server sending “1Hi” to client 3(client ID 6)



Now, client-3’s reader function will see a message “1Hi” from server.
This indicates to client-3 that it is from a client with client_id = 1.

Thus, the server assists the client-to-client communication.



Establishing secure communication(similar to telegram)

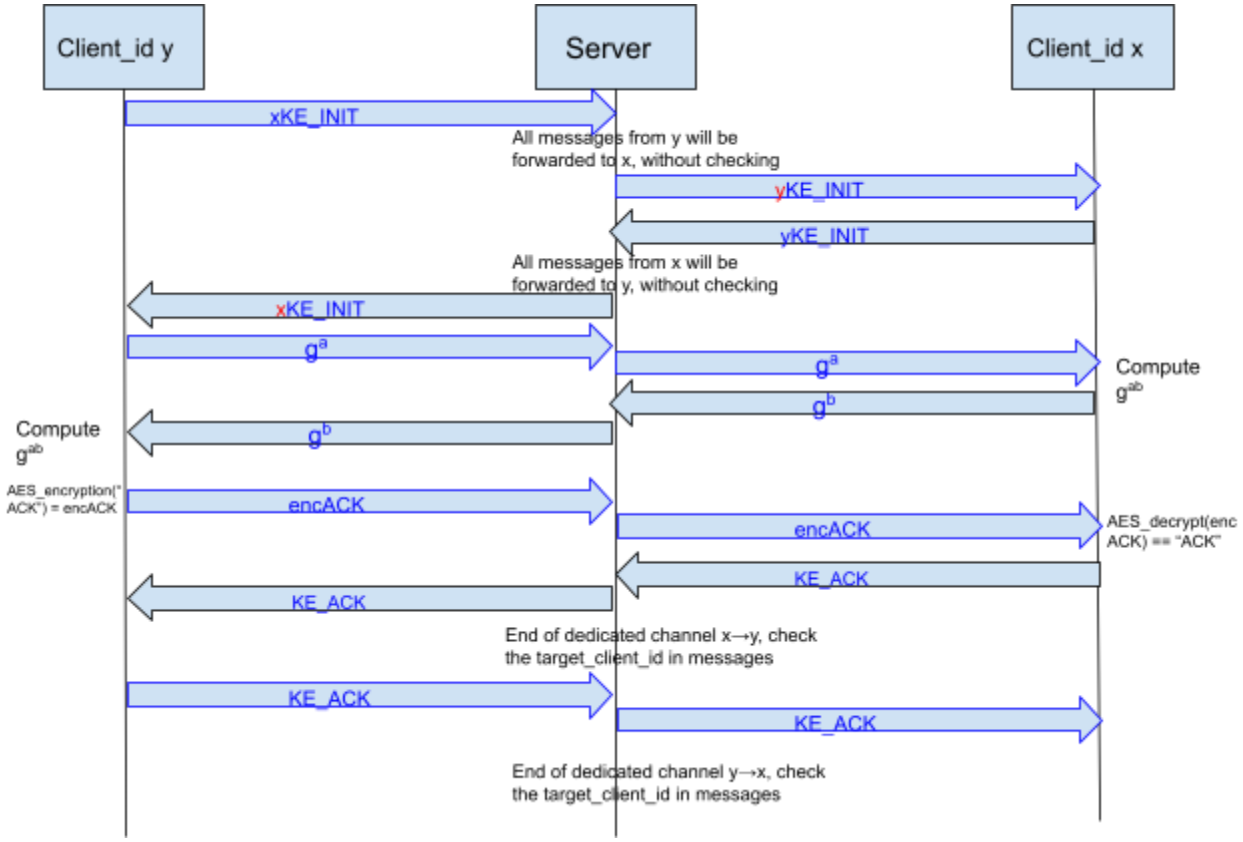
Diffie-Hellman Key Exchange

One of the participating clients(id=y) initiates DH key exchange with a target_client(id=x) by sending a “xKE_INIT” to the server. The server forwards “yKE_INIT” to client id=x.

The server notices that a key exchange is going to take place between client y and client x, and it treats every message coming from client y as if it is destined to client x(without checking the first byte for target_client_id), until it receives a “KE_FAIL” indicating Key Exchange Failure, or a “KE_ACK” indicating a key exchange success.

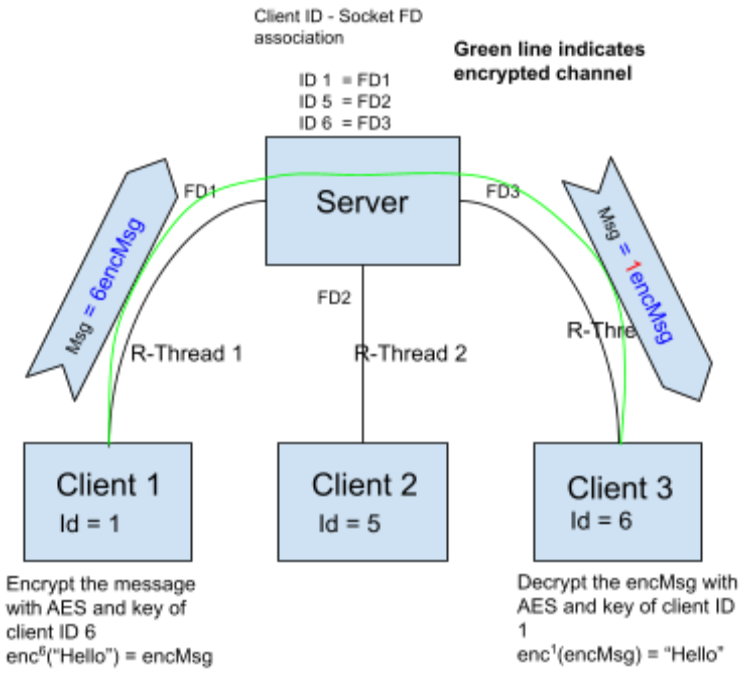
Upon receiving a KE_INIT message from a client, the remote client sends a KE_INIT from its end(client x sends yKE_INIT to the server), indicating its readiness and willingness for a key exchange(with client y) and the same procedure of dedicated channel is followed at the server.

Once both the parties successfully exchange the DH keys, the KE initiator sends an encrypted “ACK” message, the responder upon successful decryption of this “ACK” message will send an unencrypted “KE_ACK” message, to indicate the server of the end of KE and to free the dedicated channel, and to indicate to the initiator of KE success. The initiator sends back a “KE_ACK” which serves the same purpose as the former.

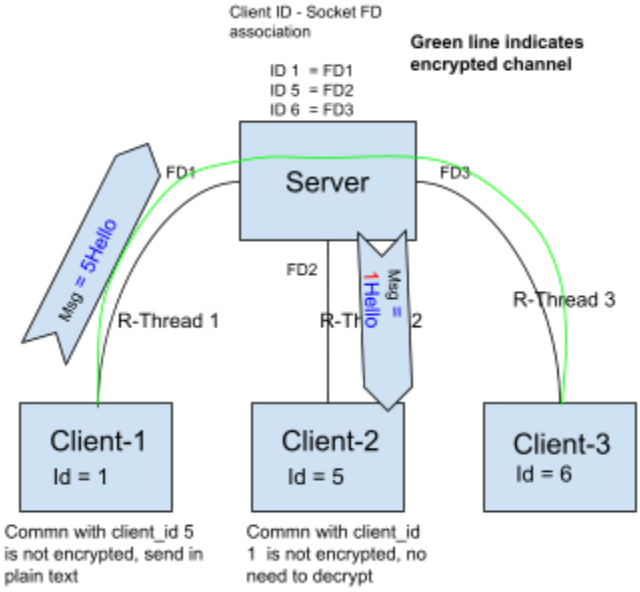


Client to client encrypted communication

Each client after successful DH key-exchange with a remote peer, will mark the remote peer as “encryption enabled”, thus, any further communication with the remote peer will thus be AES encrypted, which only the sender and receiver can decipher.
However, the first byte will still be the unencrypted client_id of the target, to enable the server to handle the relay functionality, without divulging any information about the message.



If the communication between two clients has not been encrypted yet, then the clients will only talk in plain text, which is unsecure.(as shown below)



Implementation in c and and modeling in ttool

Server:

The **server** code consists of a main server function, which binds to the ip and port given(by the user in prototyping) and listens for any incoming connection requests from clients to accept.

```c
/* Input: Expects a pointer to the structure object sockaddr_in(with details of
 *  ip, port, proto.
 * Example: &servaddr is passed in below example
 *      struct sockaddr_in servaddr;
 *      servaddr.sin_family = AF_INET;
 *      servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
 *      servaddr.sin_port = htons(2222);
 * Description: This is the code to start a server, which keeps
 *      listening for new client connections and creates individual client threads.
 */
void * server(struct sockaddr_in *servarg) { //Create server
    int MAX=256;
    int sockfd, len;
    int i=0;
    int tmpconnfd;
    int client_id;
    int number_of_clients_allowed = 10; //it is 10 for now, can be updated according to the usecase

    char buff[MAX];

    struct sockaddr_in servaddr, cli;

    pthread_t thread[10];

    // socket creation and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created..\n");
    bzero(&servaddr, sizeof(servaddr));

    // Binding newly created socket to given IP and verification
    if ((bind(sockfd, (struct sockaddr*)servarg, sizeof(servaddr))) != 0) {
        printf("socket bind failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully binded..\n");

    // Now server is ready to listen and verification
    if ((listen(sockfd, 5)) != 0) {
        printf("Listen failed...\n");
        exit(0);
    }
    else
        printf("Server listening..\n");

    len = sizeof(cli);
    while(i<10) {

        // Accept the data packet from client and verification
        tmpconnfd = accept(sockfd, (struct sockaddr*)&cli, &len);
        if (connfd[i] < 0) {
            printf("server accept failed...\n");
            exit(0);
        }
        else
            printf("server accept the client(not id) %d...\n", i);
```

As seen in the above snippet, currently, the upper limit on the number of client connections accepted is set to 10, but this can be updated according to the needs and use case.

Note: Please do update the number of bytes in the message, dedicated to the client_id field in case it is updated. It is currently the first byte.

The Server, after accepting a client connection, will expect the client to advertise its client_id, so that it can maintain a mapping of client_id : connection FD.
The mapping is maintained in an int array int connfd[10], with the array index being the client_id and the value stored being the connection FD.

connfd[client_id] = connectionFD

Then the server spawns a reader thread(read_from_client) dedicated to the specific client_id.

```
        //Get the clientID advertised.(a must)
        bzero(buff, sizeof(buff) );
        if(read(tmpconnfd, buff, sizeof(buff)) <= 0) {
                printf("An error in read, waiting for the clientID, ignoring the client\n");
                continue;
        }
        if(strlen(buff)!= 1) {
                printf("Got a clientId bigger than 1 byte : %ld bytes, ignoring this client\n", strlen(buff) );
                continue;
        }

        client_id = atoi(buff);
        printf("client ID is %d\n", client_id);

        //Save the connfd in the respective slot of the array.
        connfd[atoi(buff)]=tmpconnfd;
        pthread_create( &thread[atoi(buff)], NULL, read_from_client, (void*)&client_id);
        i++;
    }

    for(i=0; i<10; i++) {
            pthread_join(thread[i], NULL);
    }

        // After chatting close the socket
    close(sockfd);

}
```

Given below is the code of the **read_from_client** function which will be constantly listening/reading from a particular client's connection FD.
Unless the key_change is in progress, the function derives the target_client_id from the message(the first byte of the message)
target_client = atoi(&buff[0]);

my_id = *(int *)client_id; is used to remember the client_id corresponding to a reader thread.

```c
/* Input: *Client_id (char pointer to the client_id
 *
 * Description :
 * Called by server
 * Simple reader loop/thread to keep reading on the connfd of the given client_id
 * stored in array connfd[] with client_id as index
*/
void *read_from_client(void *client_id)
{
        int my_id = *(int *)client_id;
        int MAX=256;
        char buff[MAX], buff2[MAX];
        int n;
        int target_client;
        bool key_exchange = false;
        for(;;) {
                bzero(buff, MAX);
                // read the message from client and copy it in buffer
                if(read(connfd[my_id], buff, sizeof(buff)) <= 0 ) {
                        printf("client tr %d encountered an error , exiting",my_id);
                        return NULL;
                }
             //   printf("client tr %d Received: %s\n", my_id, buff);

                /*The keyexchange has been finished, remove the flag and indicate the client.
                 * All the messages from now will be treated as normal messages , and will
                 * be checked for the target_client ID in the first byte
                 */
                if( (strcmp(buff, "KE_ACK") == 0) || (strcmp(buff, "KE_FAIL") == 0) ){
                                key_exchange = false;
                                write_to_target_client(target_client, buff);
                                continue;

                }
                if(key_exchange == true) {
                        write_to_target_client(target_client, buff);
                        continue;
                }

                //Derive the target_client from the first byte of the message.
                //Example: 2hello
                //Meaning, a message "hello" destined to clientID "2".
                target_client = atoi(&buff[0]);

                /*Received a message of format "xKE_INIT", where x is the target client
                 * this indicates a start of key exchange protocol, until we receive
                 *  KE_ACK or KE_FAIL in clear text treat all messages coming from
                 *  this client as if they are destined towards the target alone.
                 *  i.e; dont expect target_client id during these transactions.
                 */
                if(strcmp(&buff[1], "KE_INIT") == 0) {
                                key_exchange = true;
                                //sprintf(buff, "%d%s", my_id, buff+1);
                } else {
                        printf("client tr %d Received: %s\n", my_id, buff);
                }

                /*Update the message by placing the sourceClient in the first byte,
                 * followed by the message.
                 * Received: <dstClientID> + <message>, from sourceClient
                 * Sent: <srcClientID> + <message>, to the dstClientID
                 */
                sprintf(buff, "%d%s", my_id, buff+1);
                printf("target: %d source: %d\n", target_client, my_id);

                //Write to the connfd of the target_client
                write_to_target_client(target_client, buff);
        }
```

Before relaying the message to the target_client, the first byte of the message is overwritten with the sender client's id.

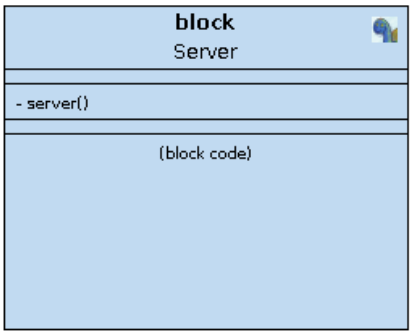
```
/*Update the message by placing the sourceClient in the first byte,
 * followed by the message.
 * Received: <dstClientID> + <message>, from sourceClient
 * Sent: <srcClientID> + <message>, to the dstClientID
 */
sprintf(buff, "%d%s", my_id, buff+1);
printf("target: %d source: %d\n", target_client, my_id);

//Write to the connfd of the target_client
write_to_target_client(target_client, buff);
```

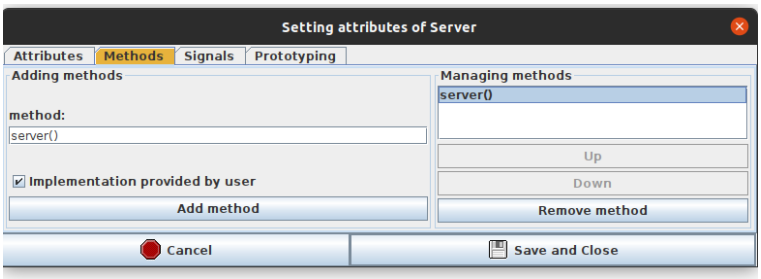
Write_to_target_client , is a simple writer program to write the given buffer to the connection FD of the given client_id

```
/* Input: client_id : To derive the connection fd to write to.
 *        *buff : message to be sent.
 * Description : Simple writer to write to a particular client_id
 * (the connfd is looked up in the
 * stored array connfd[] with client_id as index)
 */
void write_to_target_client(int client_id, char* buff) {
        int MAX=256;
        printf("sending %s to %d\n", buff, client_id);
        if(write(connfd[client_id], buff, MAX) == -1) {
                printf("%s",strerror(errno));
        }
}
```

Ttool modeling

The server block's model in ttool is very basic, and has only one method(please check "Implementation provided by user")

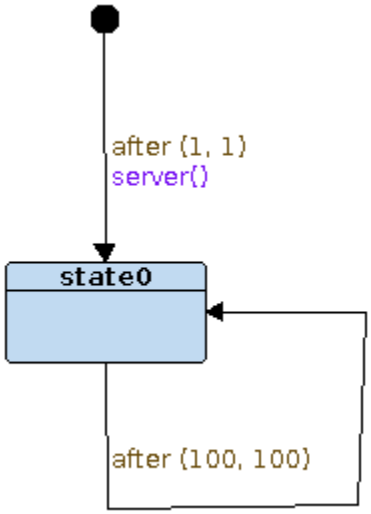




In the prototyping section, we have to pass the required input to the server function call. These ip and port numbers can be modified according to the user's needs.

```
void __userImplemented__Server__server(){
	struct sockaddr_in servaddr;
	servaddr.sin_family = AF_INET;
	servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
	servaddr.sin_port = htons(2222);

	pthread_t thread_id;
	pthread_create(&thread_id, NULL,server, (struct sockaddr_in*)&servaddr);
}
```

Given below is state machine of the server



Overall, as we can see, the server is very lightweight whose main functionality is to relay the messages between the clients.

Client:

The client app is more flexible compared to the server as there are a lot of functions that could be modularized at the client end.

The starting point of the client app is the function “**client**”, that connects to the given server ip and port and returns a socket FD.

```
/* Input: Expects a pointer to the structure object sockaddr_in(with details of server
 *  ip, port, proto.
 * Example: &servaddr is passed in below example
 *      struct sockaddr_in servaddr;
 *      servaddr.sin_family = AF_INET;
 *      servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
 *      servaddr.sin_port = htons(2222);
 * Description: This is the code to start a client, which connects to the given server
 *      ip and port and returns the socket FD of the connection.
 */
int client(struct sockaddr_in *servarg) {
    int MAX=256;
    int sockfd;
    struct sockaddr_in servaddr;
    char buff[MAX];
    int n;
    char buffer[256];

    // socket create and verification
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd == -1) {
        printf("socket creation failed...\n");
        exit(0);
    }
    else
        printf("Socket successfully created %d..\n", sockfd);

    mysockfd = sockfd;
    bzero(&servaddr, sizeof(servaddr));

    // connect the client socket to server
    if (connect(sockfd, (struct sockaddr*)servarg, sizeof(servaddr)) != 0) {
        printf("connection with the server failed...\n");
        exit(0);
    }
    else
        printf("connected to the server..\n");

    return sockfd;
}
```

Once the client has connected to the server, we now have to advertise our client-id to the server.

“advertise_client_id” is a simple function which informs the server of the client_id.

```
//Input args:
//      sockfd: socket to write to.
//      my_id: my client id that I want to advertise to the server.
//Output: void
//
//Description: The function that advertises client id to the server, so
//      that the server can keep an association of clients to their ids.
void advertise_client_id(int sockfd, int my_id) {
        int MAX=256;
        char buff[MAX];
        printf("Advertising my client Id to the server : %d\n", my_id);
        bzero(buff, sizeof(buff));
        sprintf(buff, "%d", my_id);
        if(-1 == write(sockfd, buff, MAX)) {
                printf("advertise_client_id failed, exiting. %s\n",strerror(errno));
                exit(1);
        }
}
```

Client also has two main functions for receiving and sending data. They are “**reader**” and “**writer**”

reader: Simple reader to read from the given socket fd(server connection).

```
/* Input:
 *      sockfd : connection fd to read from.
 * Description: Reads from the given socket and sets the given timeout for the read.
 */
void reader(int sockfd, int timeout_s) {
    int MAX=256;
    int first = 1;
    char buff[MAX];
    int n;
    int target_client;
    unsigned char *plaintext = malloc(MAX);
    struct timeval tv;
    tv.tv_sec = timeout_s;
    tv.tv_usec = 0;
    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(struct timeval));

    bzero(buff, sizeof(buff));
    read(sockfd, buff, sizeof(buff));
    if(strlen(buff) != 0) {
            //printf("From Server : %s\n", buff);
            if(strcmp(&buff[1], "KE_INIT") == 0 ){
                printf("got KE_INIT from the target : %d\n", atoi(buff) );
                start_keyexchange_responder(sockfd, atoi(buff) );
                return;
            }
            target_client = atoi(buff);
            if(client_encryption[target_client] == 1) {
                    plaintext = aes_ige_decryption(target_client, buff+1);
                    printf("%d : %s", target_client, plaintext);
                    return;
            }

            printf("> %d : %s", target_client, buff+1);
//          printf("From Server : %s\n", buff);
    }
        //      printf("My sockfd :%d, sockfd passed: %d\n", mysockfd, *(int *)sockfd);
}
```

If the “**reader**” notices that there is an incoming “KE_INIT” message, it will call “**start_keyexchange_responder**” to respond and finish the key-exchange with the client that sent the “KE_INIT”

```
    if(strcmp(&buff[1], "KE_INIT") == 0 ){
        printf("got KE_INIT from the target : %d\n", atoi(buff) );
        start_keyexchange_responder(sockfd, atoi(buff) );
        return;
    }
```

If client_encryption of a particular target_client is set, the reader understands that the incoming message is an AES encrypted message, which has to be decrypted before printing on the client screen.

```
   target_client = atoi(buff);
   if(client_encryption[target_client] == 1) {
           plaintext = aes_ige_decryption(target_client, buff+1);
           printf("%d : %s", target_client, plaintext);
           return;
   }
```

Writer: simple function that captures the cli input and writes to the target socket.(server)

```
/* Input:
 *      sockfd : connection fd to write to.
 * Description: Reads from the cli and writes to the given socket.
 */
void writer(int sockfd) {
    int MAX=256;
    char buff[MAX];
    unsigned char *data_send = malloc(256);
    int n;
    int target_client;
    bzero(buff, sizeof(buff));
    if(first_time_writer_call == true)
    {
        printf("Enter the client number to talk to followed by the message string : \n");
        first_time_writer_call = false;
    }

    while ((buff[n++] = getchar()) != '\n')
       ;
    // write(*(int *)sockfd, buff, sizeof(buff));
    if(strlen(buff) != 0) {
            target_client = atoi(buff);
            if(client_encryption[target_client] == 1) {
                    data_send = aes_ige_encryption(target_client, buff+1);
                    bzero(buff, sizeof(buff) );
                    sprintf(buff, "%d%s", target_client, data_send);
                    write(sockfd, buff, MAX);
                    free(data_send);
                    return;
            }

            write(sockfd, buff, sizeof(buff) );
            free(data_send);
    }
}
```

The “writer” keeps taking input until it encounters a ‘\n’ (Enter).

```
while ((buff[n++] = getchar()) != '\n')
```

It checks if the communication with the target_client is encryption enabled. If so,
It encrypts the message with AES encryption before sending.

```
    if(client_encryption[target_client] == 1) {
            data_send = aes_ige_encryption(target_client, buff+1);
```

Key-exchange functions:

There are two main key-exchange functions, one that initiates from a client, and one that responds from another client. The logic behind their working is detailed in the **Establishing secure communication** section.

Start_keyexchange_initiator, initiates the key exchange with the remote target.
Adding here, only the function signature and doxygen header for details.

```
//Input args:
//      sockfd : socket to read from and write to
//      target_client : The target client-id with which are going to perform key exchange.
//Uutput: void
//
//Description: The function that initiates the DiffieHelman key exchange with a given
//      remote host, through the server. send a KE_INIT to signal key exchange, then KE_ACK when
//      the key exchange is successful and KE_FAIL if the exchange fails.
void start_keyexchange_initiator(int sockfd, int target_client) {
```

Start_keyexchange_responder, it is called when the reader function of a client detects a “KE_INIT” from a target_client.

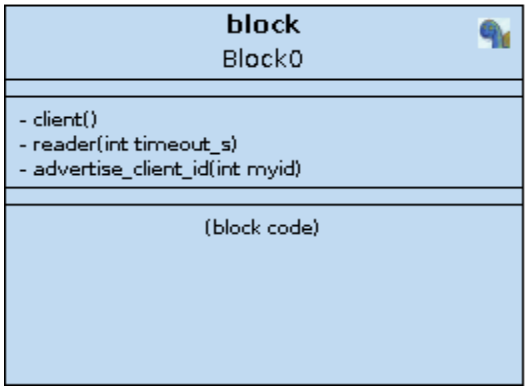
```
//Input args:
//      sockfd: socket to write to.
//      target_client: Client id of the target with whom we are performing a key exchange.
//Output: void
//
//Description: The function that responds to a Key exchange attempt from remote
void start_keyexchange_responder(int sockfd, int target_client) {
```

Modeling client applications on Ttool:

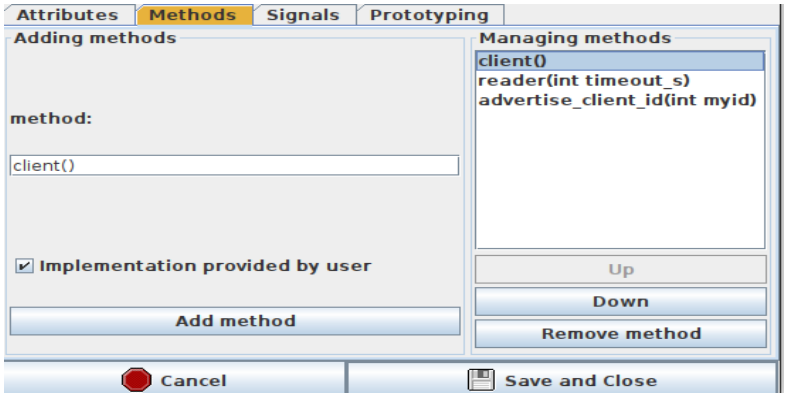
Client applications can be modeled in many ways with great flexibility due to modularized functions.

We will consider two such possible implementations.

1) Simple **ReaderOnly** client: A client that connects to a server, advertises its client_id and keeps listening for any incoming messages from its peers.



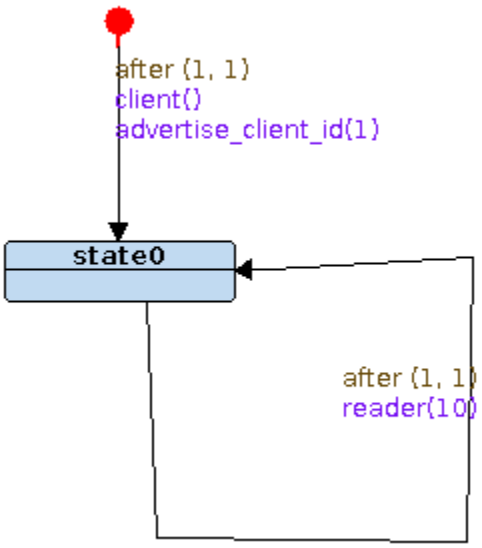
Block



Methods(make sure to check the "Implementation provided by user"

extern int sockfd; -> in global code of application.

```
extern int sockfd=0;
void __userImplemented__Block0__client(){
    struct sockaddr_in servaddr;
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
    servaddr.sin_port = htons(2222);
    sockfd = client((struct sockaddr_in*)&servaddr);
    printf("Client connected\n");
}
void __userImplemented__Block0__reader(int timeout_s){
    //second argument is the timeout of the read on the socket.
    reader(sockfd, timeout_s);
}

void __userImplemented__Block0__advertise_client_id(int myid){
    //advertise my client id to the server.
    advertise_client_id(sockfd, myid);
}
```



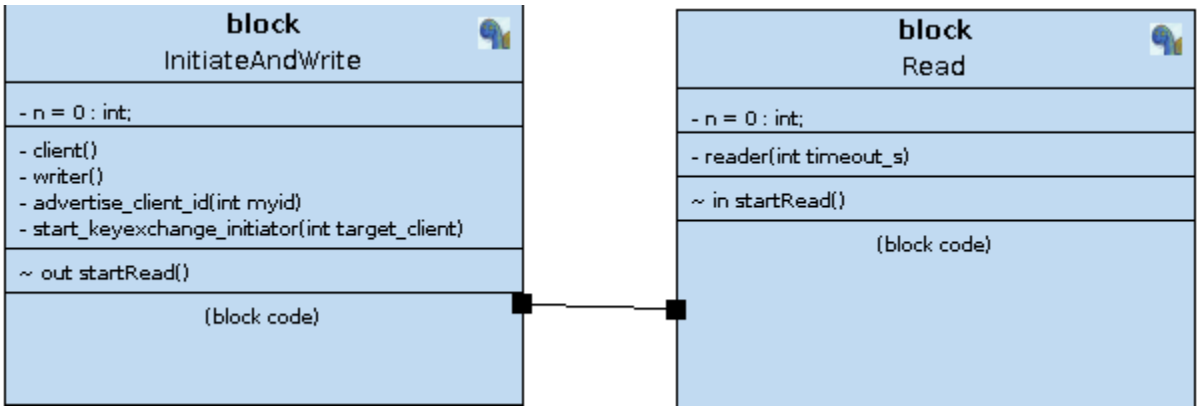
Prototyping, global code of block

As we can notice, the state machine of the reader client is very simple.

A client object is created, then a client_id of 1 is being advertised.

Then, reader is called in a loop to keep listening for incoming messages.

2) Secure **ReaderWriter** client: A client that connects to a server, advertises its client_id and keeps listening for any incoming messages from its peers and capable of taking inputs from cli and sending to the server to relay to the target_client. A true chat app.



The block diagram is an ensemble of two blocks, joined by a signal.
The purpose of two separate blocks is to enable parallel processing(threading).
First block is InititateAndWrite, which initiates client, advertises client_id and start/respond to key exchange, keep calling writer in a loop, to check for any ready outgoing messages.

Second block is Read, which keeps calling “reader” in a loop, to check for incoming messages.

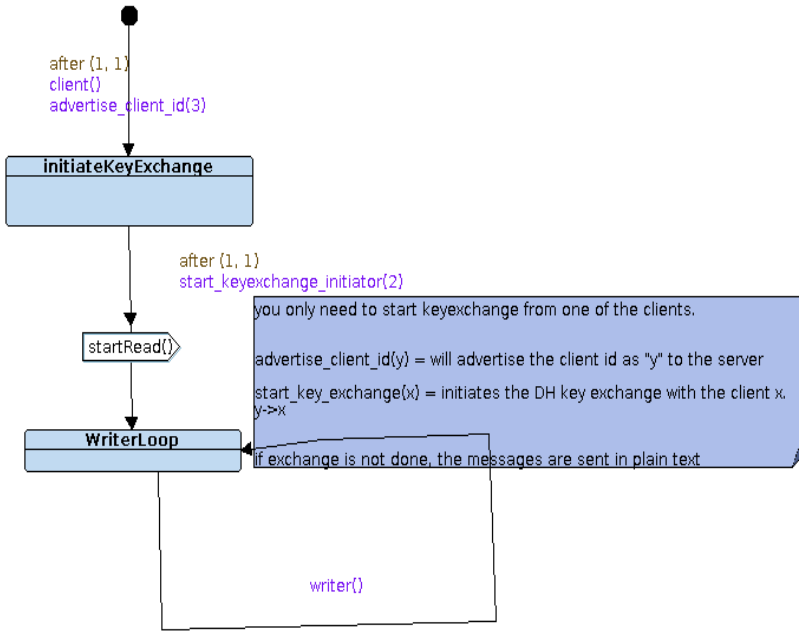
InitiateAndWrite : prototyping

```
extern int sockfd=0;
void __userImplemented__InitiateAndWrite__client(){
      struct sockaddr_in servaddr;
      servaddr.sin_family = AF_INET;
      servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
      servaddr.sin_port = htons(2222);
      sockfd = client((struct sockaddr_in*)&servaddr);
      printf("Client connected\n");
}

void __userImplemented__InitiateAndWrite__writer(){
      writer(sockfd);
}

void __userImplemented__InitiateAndWrite__advertise_client_id(int my_id){
      advertise_client_id(sockfd, my_id);
}

void __userImplemented__InitiateAndWrite__start_keyexchange_initiator(int target_client) {
      start_keyexchange_initiator(sockfd, target_client);
}
```

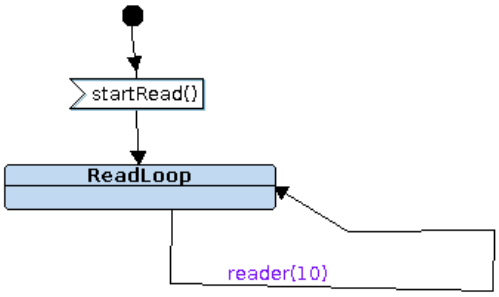
Read: prototyping

```
extern int sockfd;

void __userImplemented__Read__reader(int timeout_s){
    //second argument is the timeout of the read on the socket.
    reader(sockfd, timeout_s);
}
```

State Machine of InitiateAndWrite: (client_id=3)



State Machine of Read:



In InitiateAndWrite, we notice that we are doing a “start_keyexchange_initiator(2), which means, we(id=3) are initiating a key exchange with(id=2) (make sure that the client app ID=2 is running) Now, send a signal to the Read block to start read on the socked in a loop, and parallelly, call writer in a loop. Thus achieving continuous read and write capability, chat.

From the above two examples, we can notice that we can realize a wide range of client behaviors by modifying the client state machines. Thus the prototyping of client behaviors becomes quick and easy.

Execution output

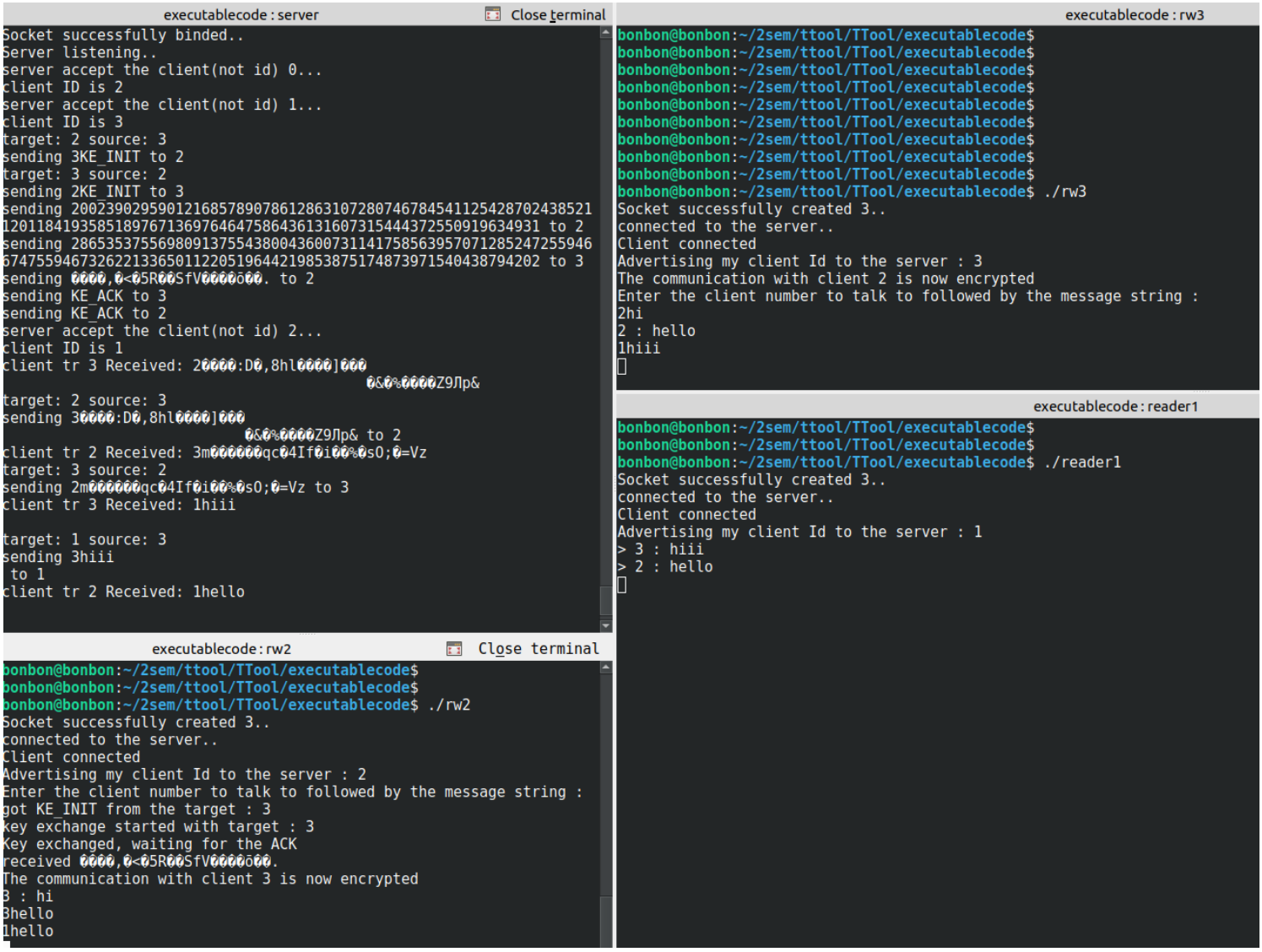
Executables generated:

server - the server binary

reader1 - simple Reader(only)

rw2 - simple read&write binary

rw3 - simple read&write binary that initiates key-exchange with client-id=2.



As we can notice, the clients registered to the server and advertised their client_id.

rw3 initiated and finished KE with rw2(now the communication between them is secure).

Every message going to rw2 from rw3(example “2hi”) is encrypted before sending to the server, the server only sees the target_client_id 2 and relays it to the rw2, by overwriting the first byte as 3.

So, we can notice that the communication is secure even from the server.

However, since reader1(client_id = 1) has no such secure encryption enabled, every message sent to 1 from 2 and 3 are in plain text.(which is expected and as per design).

Results

This project demonstrates the capability of Ttool to quickly prototype a chat application ecosystem, with ready to use end-user applications generated. The generated app ecosystem supports both encrypted and unencrypted chatting services, and the functionality is highly flexible as per the user's usecase.

Reproduction of the results

Files: Kindly refer to the git repo for all the codes.

-The seccom.c and seccom.h goes in the Ttool/executablecode/src/
-Makefile and Makefile.defs are placed in TTool/executablecode/lib/generated_src
-TaskFile.java is placed in TTool/src/main/java/avatartranslator/toexecutable. this needs a "make ttoolnotest"

Load ChapApps.xml for client and server code:
-Generate server app(rename and store it before generating other apps)
-Generate client app, give appropriate id number(advertise_client_id(x)).
-Generate another client app, give appropriate id number(advertise_client_id(y)) and if you wish to, you can do a start_keyexchange_initiator(x), so that the client does a DH key exchange with the client x.(make sure client x is running before starting client y)

Scope for further improvement

Although we achieved the goal of a flexible chat app generator, there is still scope for further improvement of the same.

- Try to bring the modularization to the server side and decouple the server functions to have more flexibility on the server side like client side.
- Make the input/output of the clients interface with a GUI for more human friendly usage.
- Could try to implement the same for other security protocols by further decoupling the security protocol from the client application.

References

Cryptography Basics:

Ms. Önen: Secure Communications, course at Eurecom, 2020

UML and TTool:

L. Apvrille: UML for Embedded Systems, course at Eurecom, 2021
L. Apvrille: Code generation from Avatar Design Diagrams in TTool, 2020
L. Apvrille: Use of TTool explained through the creation of a coffee machine, 2021

Charles CHREIM, Jacques de MATHAN: Fast Prototyping of Secure Applications, 2021

The Telegram encryption protocol: https://core.telegram.org/api/end-to-end
https://www.geeksforgeeks.org/tcp-server-client-implementation-in-c/
https://broux.developpez.com/articles/c/sockets/
https://man7.org/linux/man-pages/man2/read.2.html
https://man7.org/linux/man-pages/man2/write.2.html