Computer Networks
Homework 2
MySSH

Nistor Marian-Sergiu

University Al. I. Cuza Iasi - Faculty of Computer Science - IIB6

Abstract. This paper illustrates the architecture of the given network and the patterns, models, technologies and techniques used in order to design it.

1 Introduction

The project consists of a network, based on the Server-Client architecture, that behaves as a UNIX-based SSH server-clients pair. The server executes commands received from the clients and echoes the commands' results back to the requesters. An encryption mechanism will be used in order to ensure the security of the system. Each client has a corresponding shell opened on the server machine, that executes the actions requested by its associated client exclusively.

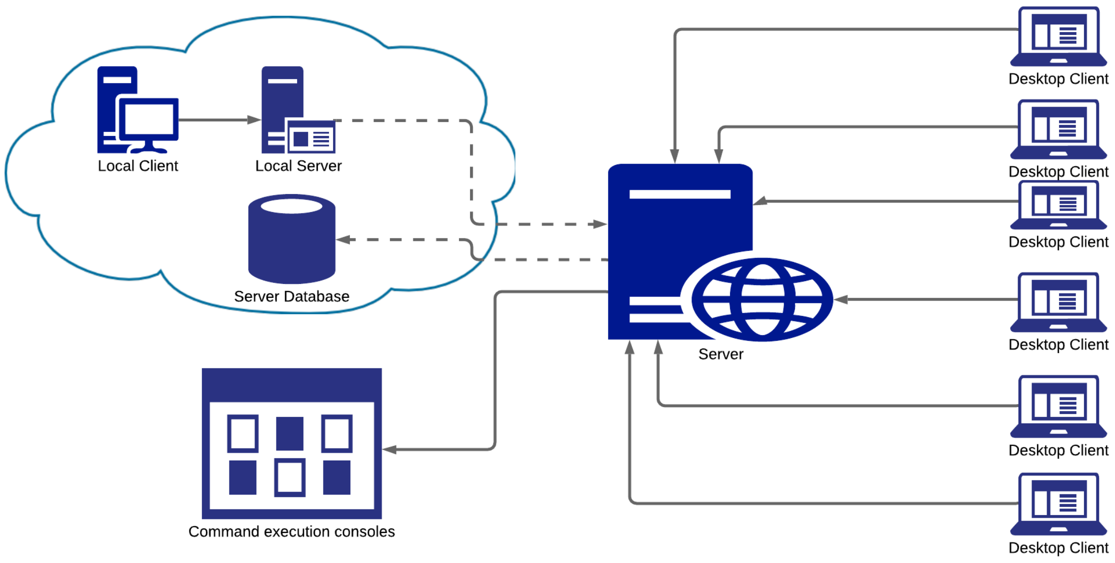
2 Used technologies

The main server was implemented using the concurrent TCP Server-Client model, in order to be able to process more requests at once and ensure the integrity of the data that is being sent from or towards the clients. A web HTTP-based implementation would not be viable as a stateful connection is mandatory for mantainting a certain state of the client(current execution physical location in the filesystem). The HTTP protocol is a stateless protocol.
The local server uses the concurrent AF_UNIX Server-Client model, in order to skip the TCP/IP network layers, therefore being able to provide or receive data faster from and towards the main server. Its primary use is stopping the main server process. This implementation was preferred over forcing a process exit on the main server, in order to ensure that the client connections are safely closed and the used memory is released accordingly. I've also opted for this design because the idea of communicating between the terminal and the main server through a local server assures easier further development of the server.
The primary hashing mechanism used by the main server and the local server is SHA-256 encryption. The secondary encryption mechanism used is the Vigenere cipher, using a generic length prefix, a unique public key shared between the main server and the client and a generic length suffix. Its only use cases are when the data cannot be predicted by the other party, making SHA-256 not

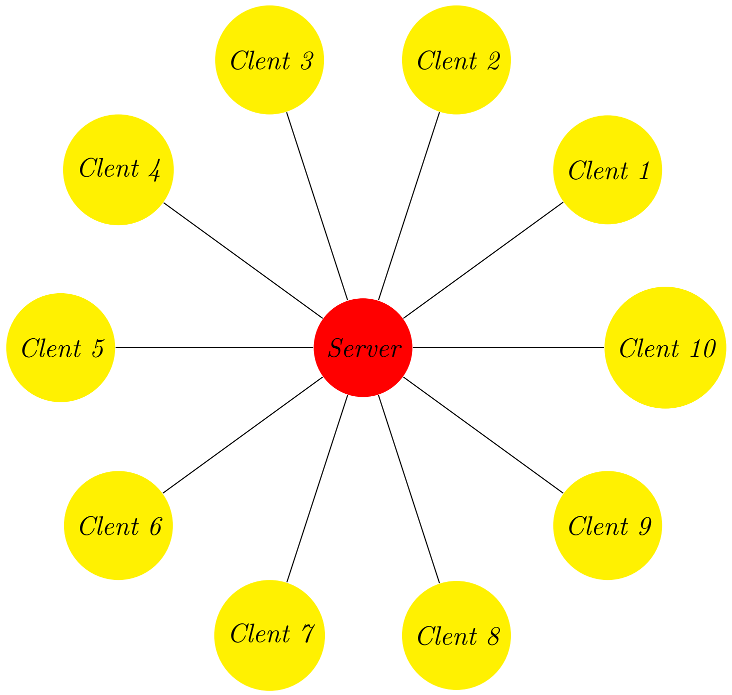
viable.
For future development, a remote desktop connection server is taken into consideration, consisting of an UDP server for desktop streaming and a TCP server for transmitting the commands(key presses and mouse clicks) towards the other party.

3 Application architecture

The detailed application architecture is being portrayed in the following diagram :



A simplified graphical representation of the network, viewed from the client's perspective(therefore, without taking the idea of the *Cloud into account), would be the the following star graph :*



This raises an issue regarding the chosen design. The main server acts as a SPOF(Single Point Of Failure), as if the given node is not available, the whole network is going to be down. Though, as the idea of the project is that the commands should be executed solely on the server machine, distributed the system would result in having to abandon the original idea of the project.

3.1 Main server transactions

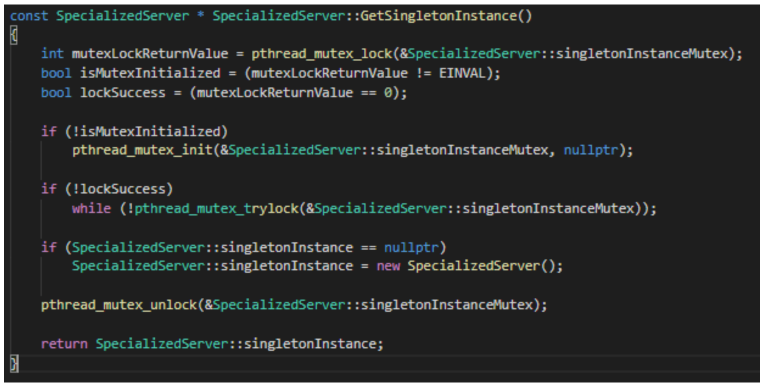
A client is uniquely identified by a tuple consisting of his IP and MAC addresses. The main server gets information from the database regarding whitelisted tuples and accepts or rejects the connection request. If the connection is accepted, it awaits a login request, containing an administrator username and password, which again are being confirmed using database information. After the whitelisted client was logged in, his requests are being processed in a new bash console process and the responses are echoed back to the client. Each client has a corresponding shell opened on the server machine, that executes the actions requested by its associated client exclusively. After executing the desired commands, the client can close the connection and inform the server regarding this action.

3.2 Cloud transactions - Local server transactions

A local server is running concurrently with the main server, awaiting commands received from a local client. The local client can stop the main server by sending *a generic message through the local server.*

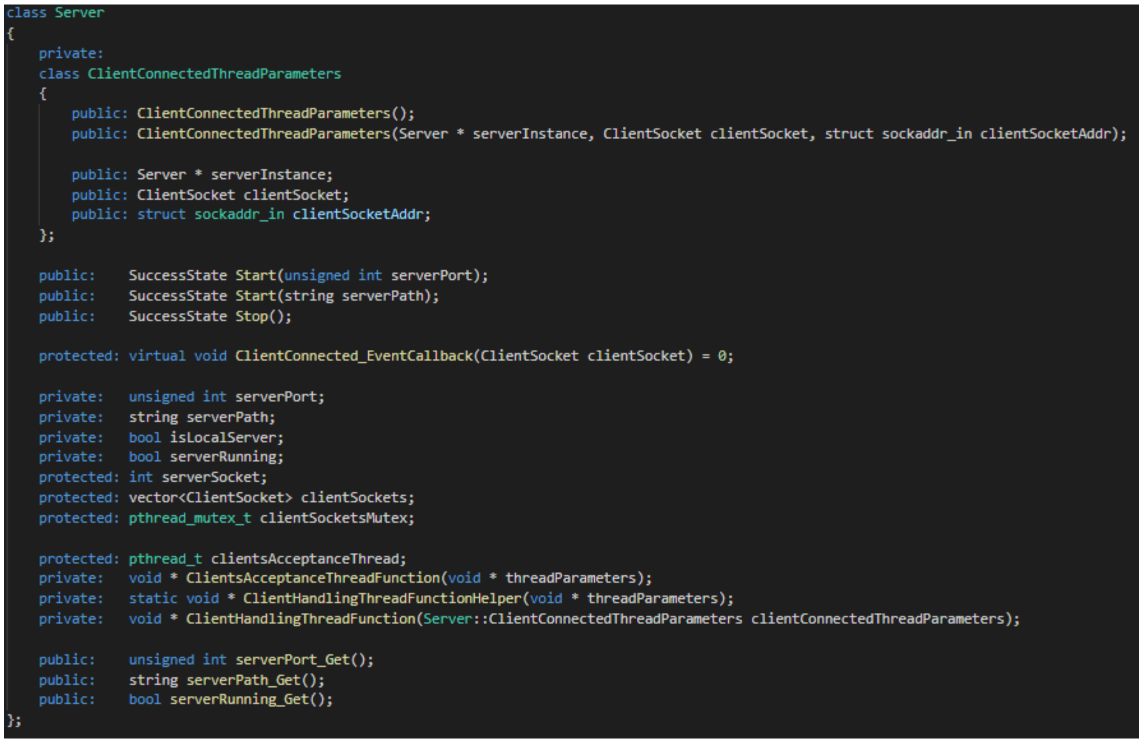
4 Implementation details

The main server implements a TCP Server-Client architecture to ensure the consistency of exchanged data.
Concurrent design patterns were used for the servers(the main and local one) in order to handle multiple client connections at the same time.
The main server does not provide IPv6 support.
The local server is using a concurrent AF_UNIX Server-Client architecture to skip the TCP/IP network layers and provide better response time from and towards the main server.
In the event of a forced process exit on the main server, resources are being released and clients are being disconnected. In order to achieve this, the native at_quick_exit C++ event trigger has been used.
The singleton class design pattern was used to prevent the existence of several instances of the main and the local server, as only one instance should be used.

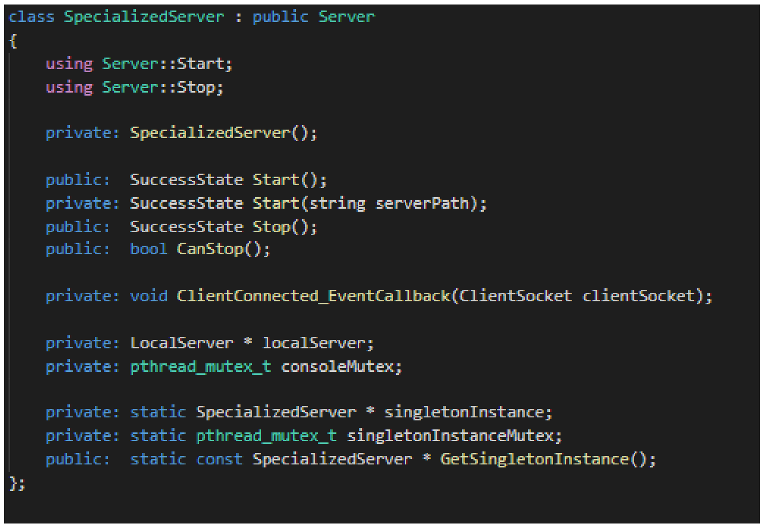


```
const SpecializedServer * SpecializedServer::GetSingletonInstance()
{
    int mutexLockReturnValue = pthread_mutex_lock(&SpecializedServer::singletonInstanceMutex);
    bool isMutexInitialized = (mutexLockReturnValue != EINVAL);
    bool lockSuccess = (mutexLockReturnValue == 0);

    if (!isMutexInitialized)
        pthread_mutex_init(&SpecializedServer::singletonInstanceMutex, nullptr);

    if (!lockSuccess)
        while (!pthread_mutex_trylock(&SpecializedServer::singletonInstanceMutex));

    if (SpecializedServer::singletonInstance == nullptr)
        SpecializedServer::singletonInstance = new SpecializedServer();

    pthread_mutex_unlock(&SpecializedServer::singletonInstanceMutex);

    return SpecializedServer::singletonInstance;
}
```

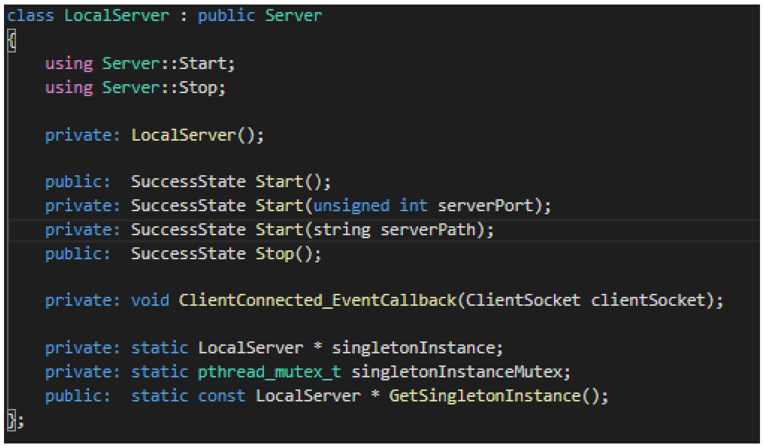
Encapsulation and inheritance(Object Oriented Design Principles) were used to enforce similarity in the behaviour of given entities. For example, both the LocalServer and the SpecializedServer classes inherit the Server class.



```
class Server
{
    private:
    class ClientConnectedThreadParameters
    {
        public: ClientConnectedThreadParameters();
        public: ClientConnectedThreadParameters(Server * serverInstance, ClientSocket clientSocket, struct sockaddr_in clientSocketAddr);

        public: Server * serverInstance;
        public: ClientSocket clientSocket;
        public: struct sockaddr_in clientSocketAddr;
    };

    public:    SuccessState Start(unsigned int serverPort);
    public:    SuccessState Start(string serverPath);
    public:    SuccessState Stop();

    protected: virtual void ClientConnected_EventCallback(ClientSocket clientSocket) = 0;

    private:   unsigned int serverPort;
    private:   string serverPath;
    private:   bool isLocalServer;
    private:   bool serverRunning;
    protected: int serverSocket;
    protected: vector<ClientSocket> clientSockets;
    protected: pthread_mutex_t clientSocketsMutex;

    protected: pthread_t clientsAcceptanceThread;
    private:   void * ClientsAcceptanceThreadFunction(void * threadParameters);
    private:   static void * ClientHandlingThreadFunctionHelper(void * threadParameters);
    private:   void * ClientHandlingThreadFunction(Server::ClientConnectedThreadParameters clientConnectedThreadParameters);

    public:    unsigned int serverPort_Get();
    public:    string serverPath_Get();
    public:    bool serverRunning_Get();
};
```



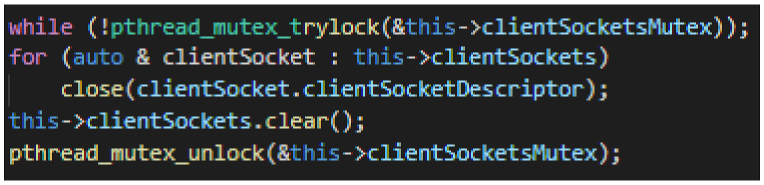
```
class SpecializedServer : public Server
{
    using Server::Start;
    using Server::Stop;

    private: SpecializedServer();

    public:  SuccessState Start();
    private: SuccessState Start(string serverPath);
    public:  SuccessState Stop();
    public:  bool CanStop();

    private: void ClientConnected_EventCallback(ClientSocket clientSocket);

    private: LocalServer * localServer;
    private: pthread_mutex_t consoleMutex;

    private: static SpecializedServer * singletonInstance;
    private: static pthread_mutex_t singletonInstanceMutex;
    public:  static const SpecializedServer * GetSingletonInstance();
};
```



```
class LocalServer : public Server
{
    using Server::Start;
    using Server::Stop;

    private: LocalServer();

    public:  SuccessState Start();
    private: SuccessState Start(unsigned int serverPort);
    private: SuccessState Start(string serverPath);
    public:  SuccessState Stop();

    private: void ClientConnected_EventCallback(ClientSocket clientSocket);

    private: static LocalServer * singletonInstance;
    private: static pthread_mutex_t singletonInstanceMutex;
    public:  static const LocalServer * GetSingletonInstance();
};
```

In order to ease further development and make the whole project a maintainable one, several functions were marked as virtual, as their implementation could vary, depending on the technologies that are being used(for example, an UI-based application wouldn't have any means of receiving input from a console).
The event callback functions are implemented as virtual methods on the base class, meant to be inherited by the child classes, for the same aforementioned reasons.

Several mutexes were used in addition to the C POSIX Threads(from the pthread.h library), to avoid memory corruption when two threads would be trying to write to a specific memory block.



```
while (!pthread_mutex_trylock(&this->clientSocketsMutex));
for (auto & clientSocket : this->clientSockets)
    close(clientSocket.clientSocketDescriptor);
this->clientSockets.clear();
pthread_mutex_unlock(&this->clientSocketsMutex);
```

4.1 Centralized system

The system is a centralized one, this model being preferred over the distributed one for this particular project. This ensures database consistency, as only one storage location is being used. However, this brings into discussion the idea that the main server becomes, therefore, a single point of failure(SPOF) in our system.

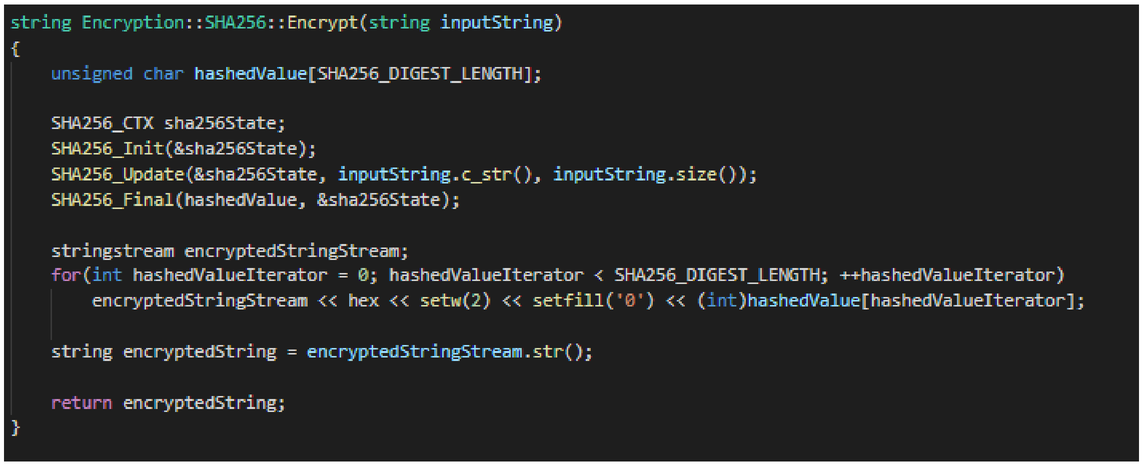
4.2 Single point of failure(SPOF)

The chosen network design implies a Single point of failure(SPOF) - the main server, as the system is a centralized one, having only one master node.

4.3 Data encryption and security

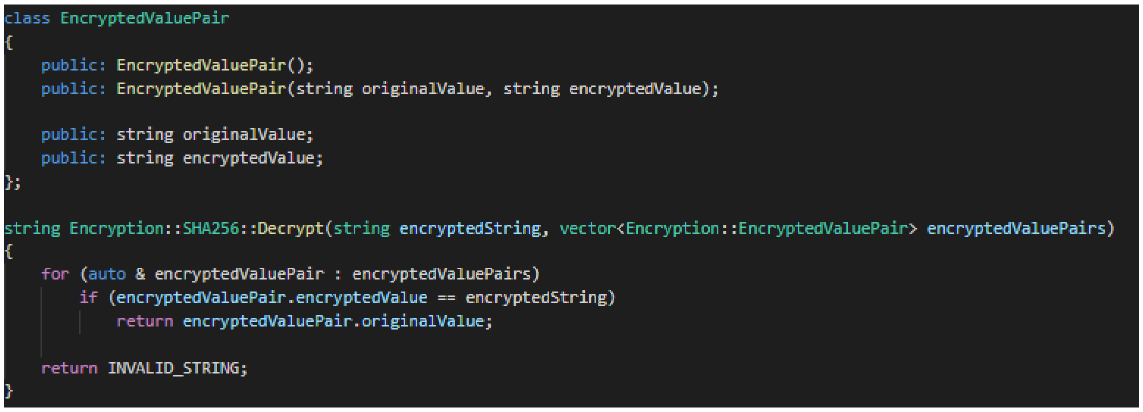
SHA-256 hashing *The primary hashing mechanism used by the main server and the local server is SHA-256 encryption. The secondary encryption mechanism used is the Vigenere cipher, using a generic length prefix, a unique public key shared between the main server and the client and a generic length suffix. Its only use cases are when the data cannot be predicted by the other party, making SHA-256 not viable. Therefore, SHA-256 hashing is used during the initial stage of the connection life-cycle, when the client IP, MAC and the administrator credentials are being sent towards the server for validation and the response is echoed back to the client. After this stage, Vigenere cipher encryption is mostly used, as the commands received from the client and their execution results are not predictable by the other party.*
SHA-256 hashing is being implemented using the openssl/sha.h library.

The hashing function is implemented as follows :



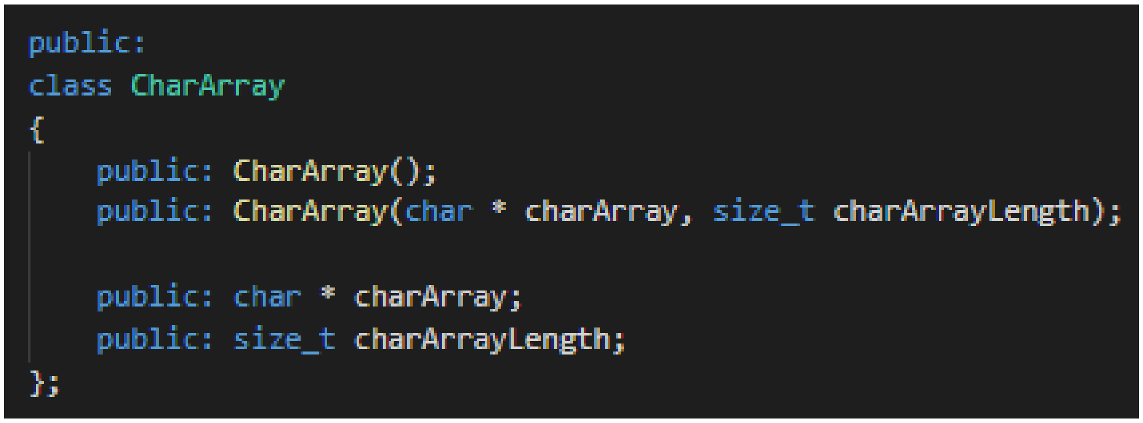
```
string Encryption::SHA256::Encrypt(string inputString)
{
    unsigned char hashedValue[SHA256_DIGEST_LENGTH];

    SHA256_CTX sha256State;
    SHA256_Init(&sha256State);
    SHA256_Update(&sha256State, inputString.c_str(), inputString.size());
    SHA256_Final(hashedValue, &sha256State);

    stringstream encryptedStringStream;
    for(int hashedValueIterator = 0; hashedValueIterator < SHA256_DIGEST_LENGTH; ++hashedValueIterator)
        encryptedStringStream << hex << setw(2) << setfill('0') << (int)hashedValue[hashedValueIterator];

    string encryptedString = encryptedStringStream.str();

    return encryptedString;
}
```

The decryption function is implemented as follows :



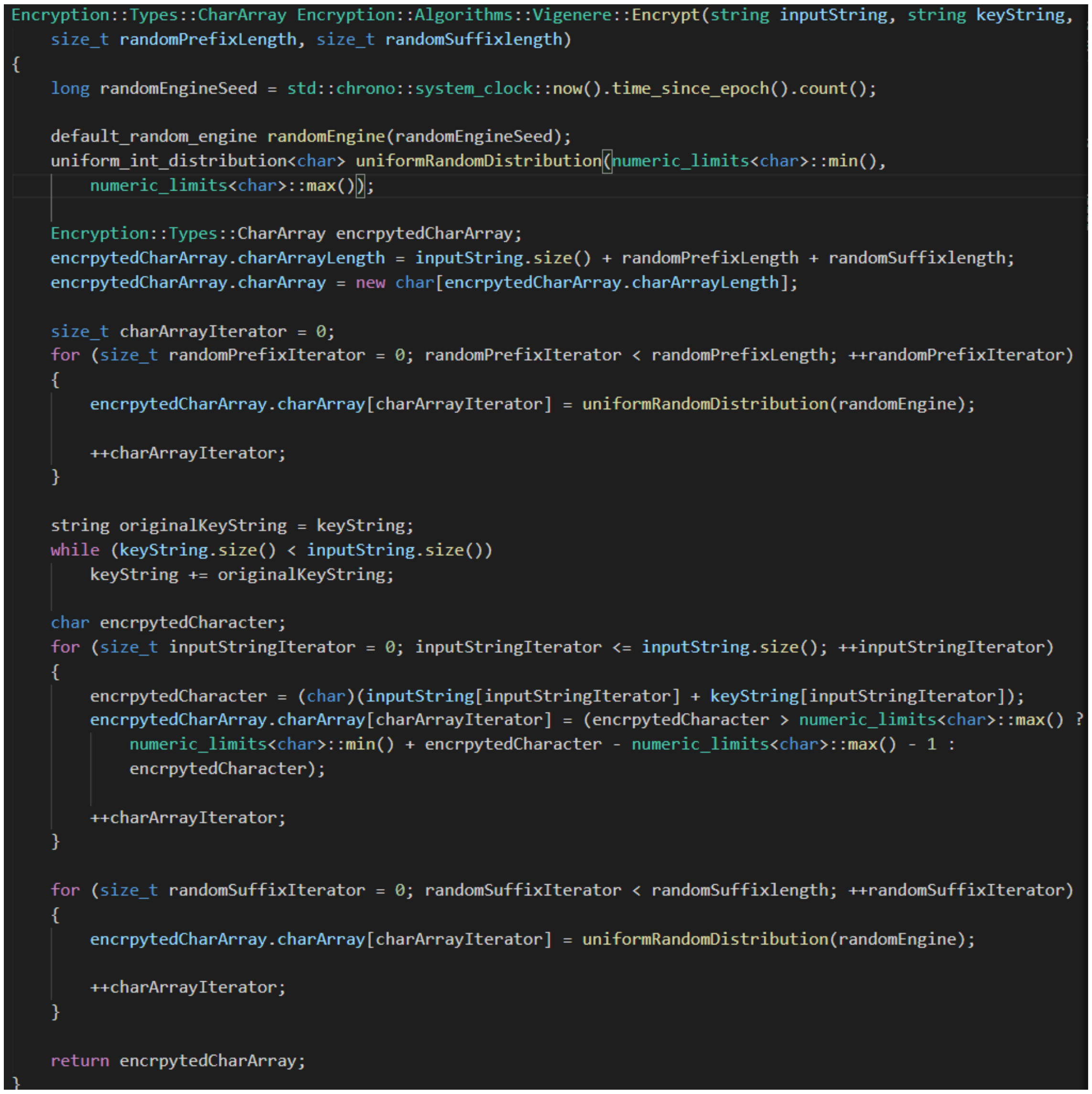
```
class EncryptedValuePair
{
    public: EncryptedValuePair();
    public: EncryptedValuePair(string originalValue, string encryptedValue);

    public: string originalValue;
    public: string encryptedValue;
};

string Encryption::SHA256::Decrypt(string encryptedString, vector<Encryption::EncryptedValuePair> encryptedValuePairs)
{
    for (auto & encryptedValuePair : encryptedValuePairs)
        if (encryptedValuePair.encryptedValue == encryptedString)
            return encryptedValuePair.originalValue;

    return INVALID_STRING;
}
```

Vigenere encryption *For the Vigenere encryption implementation I've opted for a custom CharArray class, because in a Vigenere-encrpyted string, the end-of-string character doesn't necessarily point towards the end of the string. Also, the customCharArray class doesn't represent a null-terminated string.*



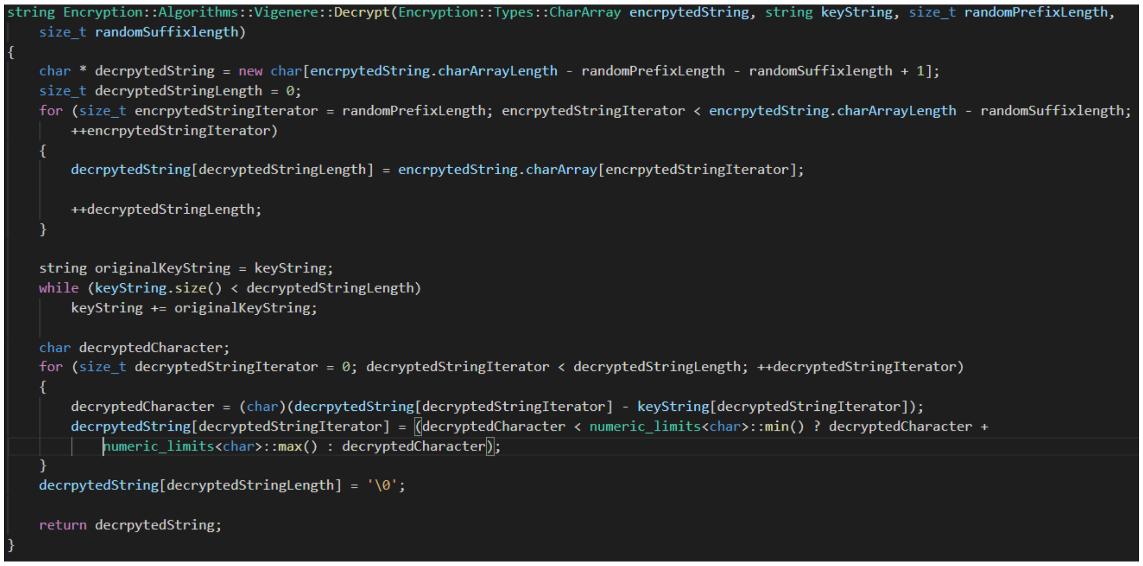
```
public:
class CharArray
{
    public: CharArray();
    public: CharArray(char * charArray, size_t charArrayLength);

    public: char * charArray;
    public: size_t charArrayLength;
};
```

The encryption function is implemented as follows :



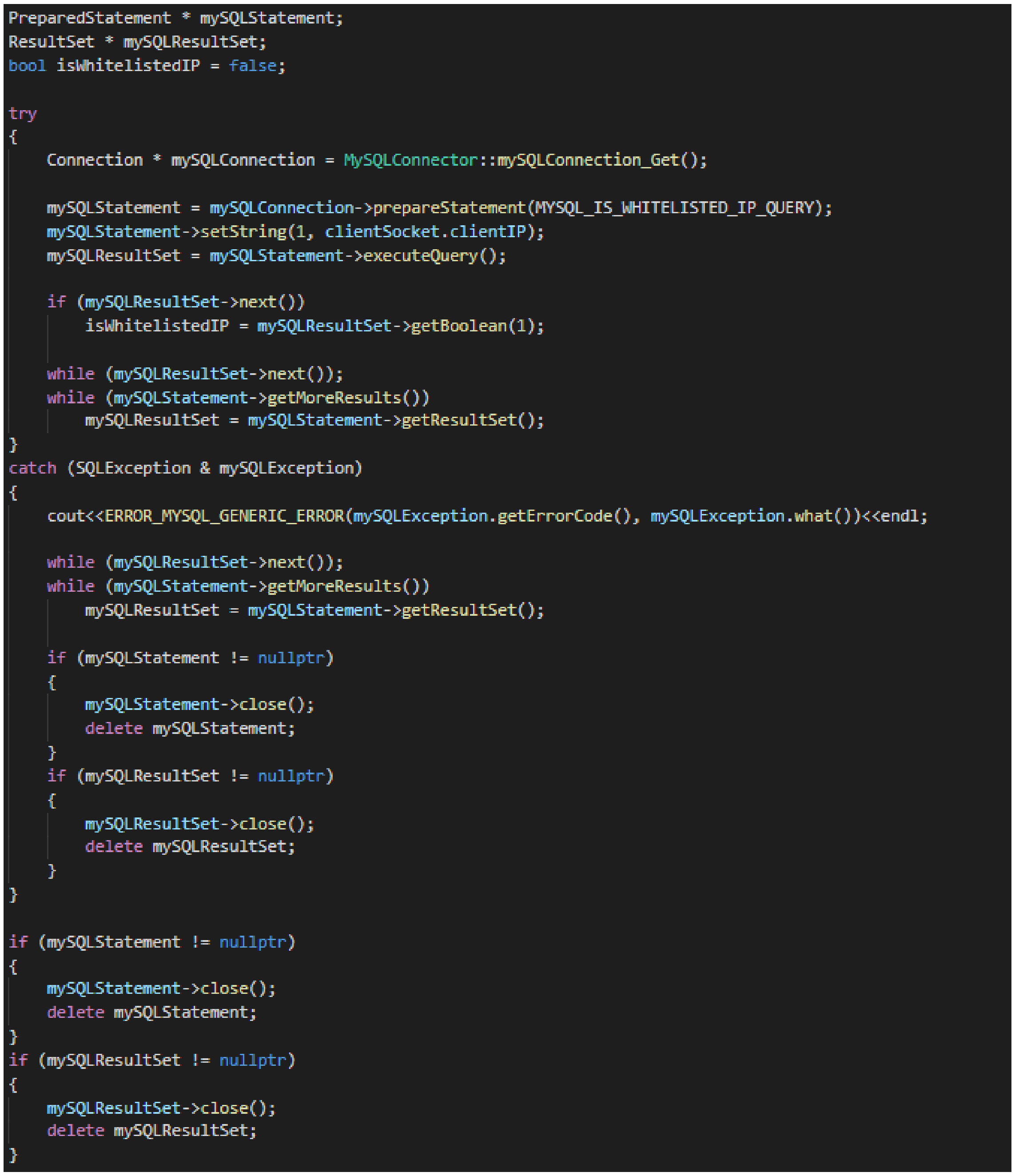
```
Encryption::Types::CharArray Encryption::Algorithms::Vigenere::Encrypt(string inputString, string keyString,
    size_t randomPrefixLength, size_t randomSuffixlength)
{
    long randomEngineSeed = std::chrono::system_clock::now().time_since_epoch().count();

    default_random_engine randomEngine(randomEngineSeed);
    uniform_int_distribution<char> uniformRandomDistribution(numeric_limits<char>::min(),
        numeric_limits<char>::max());

    Encryption::Types::CharArray encrpytedCharArray;
    encrpytedCharArray.charArrayLength = inputString.size() + randomPrefixLength + randomSuffixlength;
    encrpytedCharArray.charArray = new char[encrpytedCharArray.charArrayLength];

    size_t charArrayIterator = 0;
    for (size_t randomPrefixIterator = 0; randomPrefixIterator < randomPrefixLength; ++randomPrefixIterator)
    {
        encrpytedCharArray.charArray[charArrayIterator] = uniformRandomDistribution(randomEngine);

        ++charArrayIterator;
    }

    string originalKeyString = keyString;
    while (keyString.size() < inputString.size())
        keyString += originalKeyString;

    char encrpytedCharacter;
    for (size_t inputStringIterator = 0; inputStringIterator <= inputString.size(); ++inputStringIterator)
    {
        encrpytedCharacter = (char)(inputString[inputStringIterator] + keyString[inputStringIterator]);
        encrpytedCharArray.charArray[charArrayIterator] = (encrpytedCharacter > numeric_limits<char>::max() ?
            numeric_limits<char>::min() + encrpytedCharacter - numeric_limits<char>::max() - 1 :
            encrpytedCharacter);

        ++charArrayIterator;
    }

    for (size_t randomSuffixIterator = 0; randomSuffixIterator < randomSuffixlength; ++randomSuffixIterator)
    {
        encrpytedCharArray.charArray[charArrayIterator] = uniformRandomDistribution(randomEngine);

        ++charArrayIterator;
    }

    return encrpytedCharArray;
}
```

The decryption function is implemented as follows :



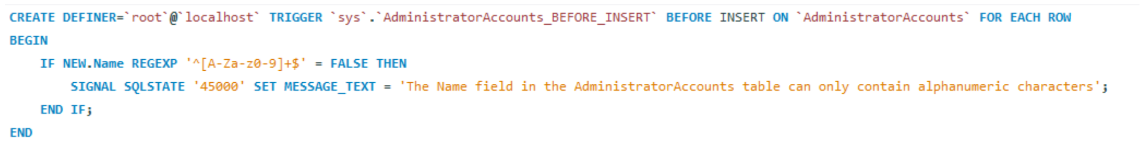
```
string Encryption::Algorithms::Vigenere::Decrypt(Encryption::Types::CharArray encrpytedString, string keyString, size_t randomPrefixLength,
    size_t randomSuffixlength)
{
    char * decrpytedString = new char[encrpytedString.charArrayLength - randomPrefixLength - randomSuffixlength + 1];
    size_t decryptedStringLength = 0;
    for (size_t encrpytedStringIterator = randomPrefixLength; encrpytedStringIterator < encrpytedString.charArrayLength - randomSuffixlength;
        ++encrpytedStringIterator)
    {
        decrpytedString[decryptedStringLength] = encrpytedString.charArray[encrpytedStringIterator];

        ++decryptedStringLength;
    }

    string originalKeyString = keyString;
    while (keyString.size() < decryptedStringLength)
        keyString += originalKeyString;

    char decryptedCharacter;
    for (size_t decryptedStringIterator = 0; decryptedStringIterator < decryptedStringLength; ++decryptedStringIterator)
    {
        decryptedCharacter = (char)(decrpytedString[decryptedStringIterator] - keyString[decryptedStringIterator]);
        decrpytedString[decryptedStringIterator] = (decryptedCharacter < numeric_limits<char>::min() ? decryptedCharacter +
            numeric_limits<char>::max() : decryptedCharacter);
    }
    decrpytedString[decryptedStringLength] = '\0';

    return decrpytedString;
}
```

Prepared MySQL statements and constraints *Prepared MySQL statements are being used as form of security against SQL Injection attacks, as such :*



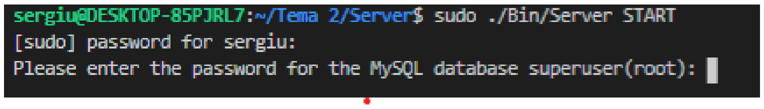
```
PreparedStatement * mySQLStatement;
ResultSet * mySQLResultSet;
bool isWhitelistedIP = false;

try
{
    Connection * mySQLConnection = MySQLConnector::mySQLConnection_Get();

    mySQLStatement = mySQLConnection->prepareStatement(MYSQL_IS_WHITELISTED_IP_QUERY);
    mySQLStatement->setString(1, clientSocket.clientIP);
    mySQLResultSet = mySQLStatement->executeQuery();

    if (mySQLResultSet->next())
        isWhitelistedIP = mySQLResultSet->getBoolean(1);

    while (mySQLResultSet->next());
    while (mySQLStatement->getMoreResults())
        mySQLResultSet = mySQLStatement->getResultSet();
}
catch (SQLException & mySQLException)
{
    cout<<ERROR_MYSQL_GENERIC_ERROR(mySQLException.getErrorCode(), mySQLException.what())<<endl;

    while (mySQLResultSet->next());
    while (mySQLStatement->getMoreResults())
        mySQLResultSet = mySQLStatement->getResultSet();

    if (mySQLStatement != nullptr)
    {
        mySQLStatement->close();
        delete mySQLStatement;
    }
    if (mySQLResultSet != nullptr)
    {
        mySQLResultSet->close();
        delete mySQLResultSet;
    }
}

if (mySQLStatement != nullptr)
{
    mySQLStatement->close();
    delete mySQLStatement;
}
if (mySQLResultSet != nullptr)
{
    mySQLResultSet->close();
    delete mySQLResultSet;
}
```

Database constraints were implemented using MySQL table triggers, to ensure that the fields have certain patterns.



```
CREATE DEFINER=`root`@`localhost` TRIGGER `sys`.`AdministratorAccounts_BEFORE_INSERT` BEFORE INSERT ON `AdministratorAccounts` FOR EACH ROW
BEGIN
    IF NEW.Name REGEXP '^[A-Za-z0-9]+$' = FALSE THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'The Name field in the AdministratorAccounts table can only contain alphanumeric characters';
    END IF;
END
```

Additional security measures *The main server cannot be started or stopped unless the requester is logged in as superuser on the server machine. The local server is being controlled only by the main server.*



```
sergiu@DESKTOP-85PJRL7:~/Tema 2/Server$ sudo ./Bin/Server START
[sudo] password for sergiu:
Please enter the password for the MySQL database superuser(root):
```

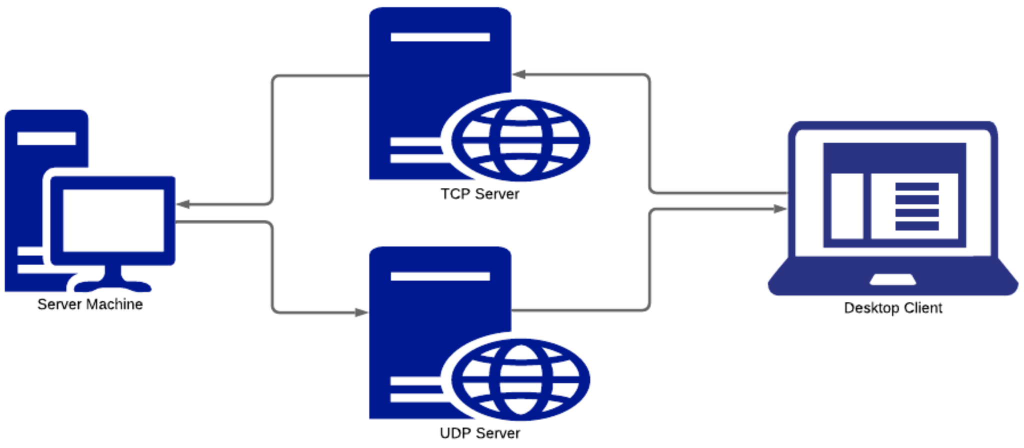
5 Conclusions

The centralized system is the easier and the more viable implementation for the given project, as the client's interest would be executing commands solely on the server machine. A centralized database would also be the better fitting choice, as the data load is low, storing mostly whitelisted IP addresses, MAC addresses and administrator credentials. The main use case of the application would be sending remote controls to a server, in the context of a company. Therefore, the number of clients with the ability to connect to the server would be a low one.

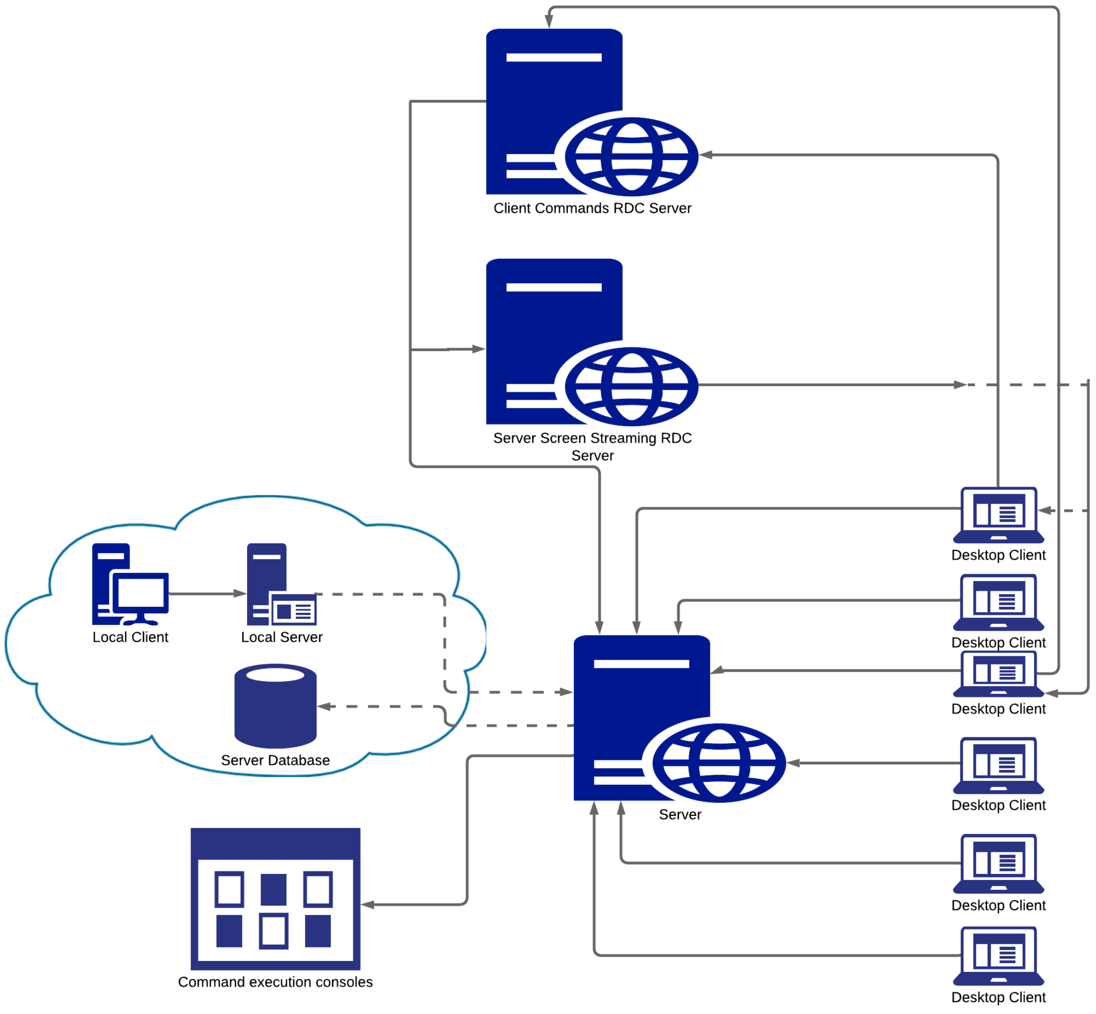
5.1 Remote desktop connection application

The idea of a remote desktop connection application could also be taken into consideration, giving the client the possibility of connecting to the server machine, for a better user experience(UX).
This would bring into discussion two more servers : an unidirectional TCP server and an unidirectional UDP server. No more than one remote desktop connection should be allowed. Therefore, the TCP and UDP servers would block any other incoming connections after one has been established.
The TCP server would transmit the information regarding pressed keys and control triggers from the client, towards the server machine. The UDP server would send the desktop data stream towards the client.
The first server would use the TCP/IP protocol in order to ensure the integrity

of the data sent towards the server, regarding the control triggers. This wouldn't affect the performance of the server anyhow, as the number of commands would not be a high one.
The second server would use the UDP/IP protocol for streaming the server's desktop, as lots of packages would have to be sent in a small amount of time. Data integrity is not crucial in this situation, while performance is a point of focus.



The network diagram containing the RDP sub-network would be the following :



The two servers dedicated to RDC are being started and stopped automatically by the main server. Whenever a client received clearing for RDC connection from the main server, he is marked as "whitelisted" on the RDC servers and he will be prompted to connect to the two servers.
In order to send the data regarding the server's screen the UDP protocol is being used, as mentioned previously!
The data is being serialized and sent in chunks of 4096 bytes, in order not to exceed the maximum socket buffer length.

References

1. *Bal, H.E., Steiner, J.G., Tanenbaum, A.S.: Programming Languages for Distributed Computing Systems, Association for Computing Machinery(ACM) (1989)*
2. *Tanenbaum, A.: Computer Networks. Prentice Hall (1981)*
3. *Computer Networks - Faculty of Computer Science https://profs.info.uaic.ro/~computernetworks/index.php*
4. *Wikipedia - Secure Hash Algorithms https://en.wikipedia.org/wiki/Secure_Hash_Algorithms*
5. *Wikipedia - SHA-2 https://en.wikipedia.org/wiki/SHA-2*
6. *Wikipedia - Vigenere cipher https://en.wikipedia.org/wiki/Vigenere_cipher*
7. *Wikipedia - Distributed Computing https://en.wikipedia.org/wiki/Distributed_computing*
8. *Wikipedia - Single point of failure https://en.wikipedia.org/wiki/Single_point_of_failure*
9. *Wikipedia - Event-driven architecture https://en.wikipedia.org/wiki/Event-driven_architecture*
10. *Oracle Academy - Database Design and Programming with SQL https://academy.oracle.com/pages/database_design_course.pdf*
11. *Peter Coad - Object-Oriented Patterns https://courses.cs.washington.edu/courses/cse503/04sp/readings/designpattern.pdf*