Charles University Faculty of Mathematics and Physics

Programming 2 [NPRG031]

# Simple chat with GUI

Mozharov Nazar

Prague 2023

# Content

## General information about the program.

The purpose of the task is to develop a simple chat. The program must meet the following requirements:

- Create a program with Graphic User Interface (GUI)
- Develop Client side and server side.
- Implement ability to send and receive text messages in real-time using the TCP protocol.
- Implement simple message encryption using the RSA cryptosystem

# Description of the Graphic User Interface (GUI)

I implemented the graphic user interface using the Windows Presentation Foundation (WPF), which is a graphical subsystem as part of the .NET Framework. To change the general appearance of the program, I used third-party open-source library called “ModernWPF UI Library”.


*Game GUI structure*

## Description of user interaction with the program

When a user opens the program, they enter a waiting window. They remain in this window until they can connect to the server. After that, they will need to enter their username. If they enter a name that is already taken by another person, the program will ask them to choose a different username for the current session. Afterwards, the user is taken to the room selection window. Each room represents a closed chat accessible only to those who have the password for that room. All messages written within a room remain there. Now the user needs to choose whether to create a room or join an existing one.

- If the user decides to create a room, they will need to enter its name and a password for accessing it. After creation, the program automatically enters this room.
- If the user decides to join an existing room, they are taken to the room selection window where they can choose the desired room from the list or enter its name manually. If the user enters a non-existent room name, the program will ask them to enter a correct room name. Afterwards, the user will need to enter the password for the room they are trying to join. If an incorrect password is entered, the program notifies the user and prompts them to enter the correct password.

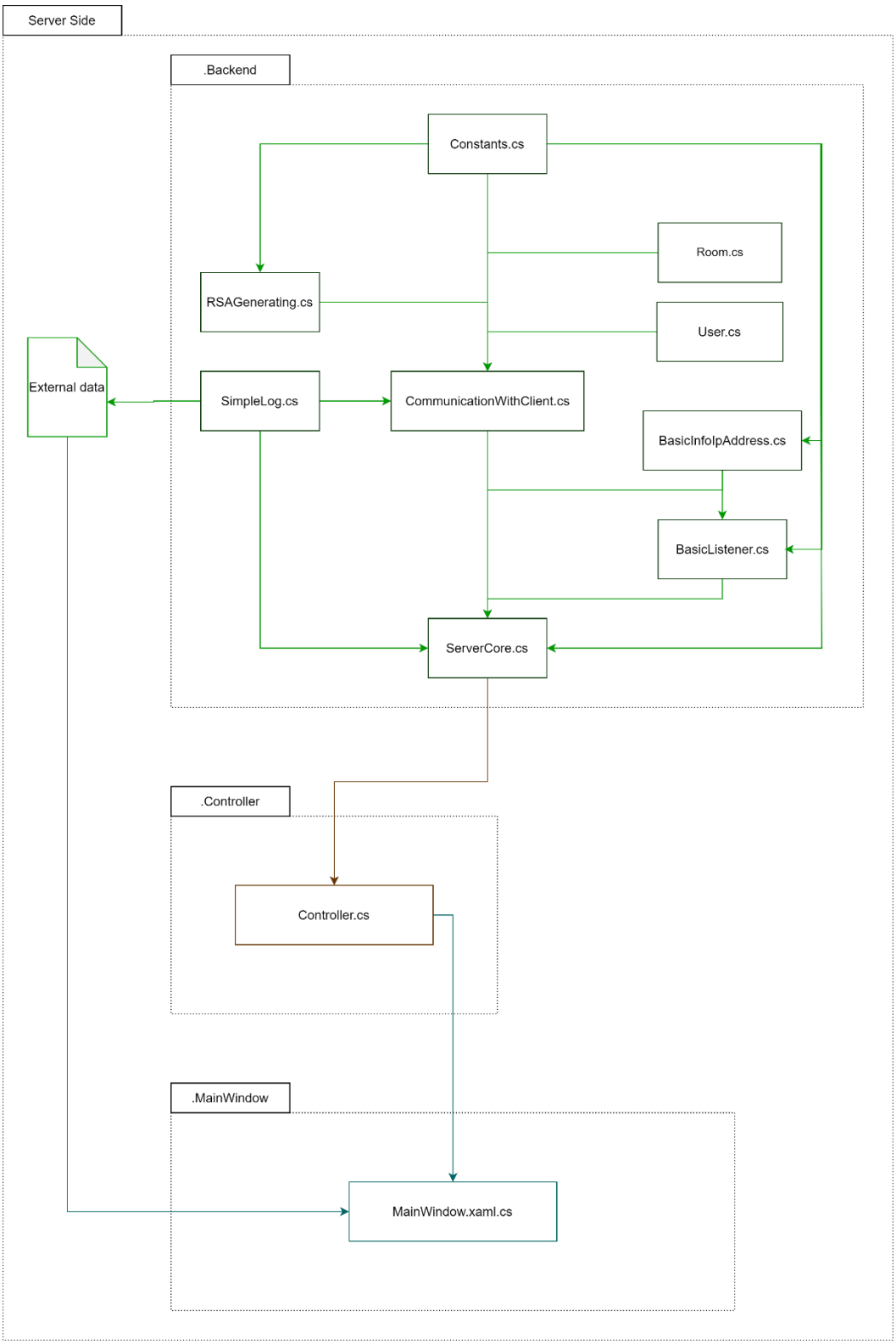
Once the user has chosen a room, they enter the chat of that room, where they can anonymously communicate with its participants. When all participants leave the room, it will be deleted for security purposes.

# Describing the technical part

## About the server part code

To implement the server part, I divided code into 3 parts: the backend, the controller, and the GUI:

- The backend is responsible for analyzing, securing and processing all the data that comes in through the TCP protocol in real time.
- The controller is responsible for the communication between the backend and the GUI.
- The GUI is responsible for displaying logs and entering basic information to configure the server before it starts.




*Picture of dependencies between backend, controller and GUI of the server side*

BasicInfoIpAddress.cs

The main class in the file BasicInfoIpAddress.cs is called BasicInfoIpAddress. This class is used to provide basic information about an IP address and a connection port. This data can be used later for determining server settings.

BasicInfoIpAddress:

- Property **bool DebugMode** is used to allow the developer to define different behaviors of methods depending on whether the program is currently in debugging mode.
- Property **IIpAddressProvider IpAddress** is used to provide the IP address as an object of the IIpAddressProvider interface. This allows the developer to get the IP address in the same way, but with a different method implementation.
- Property **int Port** stores the value of the port.

- Method **IPEndPoint GetIPEndPoint()** allows the developer to get an IPEndPoint with the previously declared IP address and port.
- Method **IPAddress GetIPAddress()** allows the developer to get an IPAddress with the desired implementation.

- Constructor **BasicInfoIpAddress (bool debugMode, string ipAddressString = Constants.DefaultIP, int port = Constants.DefaultPort)** takes bool debugMode, string ip and int port as input. By default, all this data is taken from the Constants.cs file. In the body of the constructor itself, all these values are assigned to class attributes.

BasicListener.cs

The main class in the file BasicListener.cs is called BasicListener. This class is responsible for creating a basic TCP listener object that can be used to listen for incoming connections on a specified IP address and port.

BasicListener:

- Property **IPEndPoint IpEndPoint** represents the endpoint (IP address and port) on which the listener will listen for incoming connections.
- Property **IPAddress IpAddress** represents the IP address extracted from the BasicInfoIpAddress object.
- Property **int Backlog** represents the maximum length of the pending connections queue.

- Method **TcpListener GetTcpListener()** this method creates a new TcpListener object using the IP endpoint obtained from the BasicInfoIpAddress object. It returns the created TcpListener object.

- Constructor **BasicListener(BasicInfoIpAddress ipAddress, int backlog = 10)** initializes a new instance of the BasicListener class. It takes a BasicInfoIpAddress object and an optional backlog value as input.

Room.cs

The main class in the file Room.cs is called Room. This class represents a room in a server-side application, where users can join and interact with each other.

Room:

- Property **string Name** represents the name of the room. It is a public property that can be accessed and modified.
- Property **string Password** represents the password of the room. It is a private property and can only be accessed within the class.
- Property **ConcurrentBag<string> UsersInRoom** represents a collection of usernames of users currently in the room. It is a public property that can be accessed and modified.

- Method **bool ComparePassword(string password)** is used to compare a given password with the stored hashed password. It takes a password as input, hashes it using the SHA1 algorithm, and compares the hashed password with the stored Password property. It returns true if the passwords match, indicating that the given password is correct for the room.

- Constructor **Room(string name, string password)** is used to create a new instance of the Room class. It takes a name and a password as input. Inside the constructor, the name is assigned to the Name property, and the password is hashed using the SHA1 algorithm. The hashed password is then stored in the Password property.

User.cs

The main class in the file User.cs is called User. This class represents a user in the server-side application.

User:

- Property **string Id** represents the unique identifier of the user.
- Property **string Name** represents the name of the user.
- Property **RSAGenerating rsaGeneratingServer** is used to store RSA keys on the server-side for encryption and decryption operations.
- Property **RSAGenerating rsaGeneratingClient** is used to store RSA keys on the client-side for encryption and decryption operations.
- Property **TcpClient TcpConnection** represents the TCP connection established with the user.
- Property **string CurrentRoomName** represents the name of the current room the user is in.

❖ **Constructor User(TcpClient client)** initializes a new instance of the User class. It takes a TcpClient object as input to establish the TCP connection with the user. Inside the constructor:

The TcpConnection property is set to the provided TcpClient object.

Two instances of the RSAGenerating class, rsaGeneratingServer and rsaGeneratingClient, are created for generating RSA keys on the server-side and client-side, respectively.

The client's IP address and port are hashed using SHA1 and converted into a unique identifier for the user, which is assigned to the Id property.

RSAGenerating.cs

The main class in the file RSAGenerating.cs is called RSAGenerating. This class is responsible for generating RSA keys, encrypting and decrypting data using RSA encryption, and converting data between byte arrays and strings.

RSAGenerating:

- Property **string PublicKey** represents the public key generated by the RSA algorithm. It has a protected setter, allowing modification within the class hierarchy.
- Property **RSAParameters KeyParams** represents the parameters of the RSA key. It has a protected setter, allowing modification within the class hierarchy.

- Method **GenerateKeys()** generates both private and public keys using the RSACryptoServiceProvider class. It sets the PublicKey property as a string representation of the public key and assigns the KeyParams property with the RSA key parameters.
- Method **SetPublicKeyFromString(string publicKey)** sets the public key from a string representation without setting a private key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then imports the public key from the given string. Finally, it exports the updated key parameters, excluding the private key.
- Method **EncryptRawData(byte[] data)** encrypts the given byte array data using the public key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then encrypts the data using the imported public key. The encrypted data is returned as a byte array.
- Method **DecryptRawData(byte[] data)** decrypts the given byte array data using the private key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then decrypts the data using the imported private key. The decrypted data is returned as a byte array.
- Method **EncryptString(string data)** encrypts the given string data. It converts the string to bytes using UTF-8 encoding, divides the data into blocks, encrypts each block using the EncryptRawData method, and combines the encrypted blocks into a single block. The resulting encrypted data is returned as a byte array.

- Method **DecryptIntoString(byte[] data)** decrypts the given byte array data and returns the decrypted string. It divides the data into blocks, decrypts each block using the DecryptRawData method, combines the decrypted blocks, and removes any padding or zero bytes. The resulting byte array is converted to a string using UTF-8 encoding and returned as the decrypted string.

❖ Constructor **RSAGenerating()** is responsible for initializing the class and generating the RSA keys. It calls the GenerateKeys method.

SimpleLogs.cs

The main class in the file SimpleLogs.cs is called SimpleLogs. This class provides functionality for logging messages to a file.

SimpleLogs:

- Property **string FileName** represents the name of the log file. It is a get-only property that is set using the current date and time in the format "log_dd_MM_yyyy_HH_mm_ss.txt".
- Property **string PathToFile** represents the full path to the log file. It is a get-only property that is set by combining the current directory with the FileName property.

- Method **bool CreateLogFile(string pathToFile = "")** is used to create the log file. It takes an optional parameter pathToFile, which allows specifying a custom file path. If the pathToFile parameter is not provided, the default path is used (PathToFile property).
- Method **bool WriteToFile(string textToWrite, string pathToFile = "")** is used to write a text string to the log file. It takes two parameters: textToWrite, which represents the text to be written, and an optional parameter pathToFile, which allows specifying a custom file path. If the pathToFile parameter is not provided, the default path is used (PathToFile property).

CommunicationWithClient.cs

The main class in the file CommunicationWithClient.cs is called CommunicationWithClient. This class handles the communication between the server and the clients in an async manner. An additional class is InteractWithClient, which implements methods for communicating with a client via TCP.

CommunicationWithClient:

- Property **ConcurrentBag<User> clients** a concurrent bag that stores information about connected clients.
- Property **ConcurrentBag<Room> rooms** a concurrent bag that stores information about chat rooms.

- Method **CommunicationWithClient()** constructor for the CommunicationWithClient class. Initializes the clients and rooms bags.
- Method **ConcurrentBag<User> GetClients()** returns the clients bag.
- Method **ConcurrentBag<Room> GetRooms()** returns the rooms bag.
- Method **async Task HandleClientAsync(TcpClient client)** handles communication with a client asynchronously. Initializes user and interaction objects, performs initial settings, adds the user to the clients bag, and starts receiving and sending messages.
- Method **async Task StartReceiveAndSendMessagesAsync(User user, InteractWithClient interactWithClient)** starts the process of receiving and sending messages to/from a user. Handles different types of messages received from the user and performs corresponding actions.
- Method **async Task GetAndSetUsernameFromClient(string message, User user, InteractWithClient interactWithClient)** gets and sets the username from the client. Checks if the username is already taken and sends the appropriate response to the client.
- Method **async Task<Tuple<User, InteractWithClient>> InitialSetting(User user, InteractWithClient interactWithClient)** performs the initial settings for the user and interaction objects. Sends the server's public key to the client and receives the client's public key.
- Method **async Task SendRoomList(User user, InteractWithClient interactWithClient)** sends the list of available rooms to the user.

- Method **async Task JoinToRoom(User user, InteractWithClient interactWithClient, string message)** handles the user's request to join a room. Checks the room password and adds the user to the room if the password is correct.
- Method **async Task CreateRoom(User user, InteractWithClient interactWithClient, string message)** handles the user's request to create a new room. Checks if the room name is already taken and creates the room if it's available.
- Method **void BroadcastMessage(string message, User currentUser)** broadcasts a message to all users in the current room except the sender.
- Method **ConcurrentBag<T> RemoveItemFromConcurrentBag<T>(T itemToRemove, ConcurrentBag<T> oldBag)** removes an item from a concurrent bag and returns a new bag without the removed item.
- Method **void HandleException(Exception ex, string additionalText = "")** handles exceptions by writing them to a log file.
- Method **bool CloseConnection(TcpClient client, User currentUser)** closes the connection with a client. Removes the client from the clients bag and closes the associated TCP connection. Also calls CloseRoom to remove the user from the current room.
- Method **bool CloseRoom(User currentUser)** closes the user's current room. Removes the user from the room's list of users. If the room becomes empty, removes the room from the rooms bag.

InteractWithClient:

- Property **NetworkStream Stream** represents the network stream used for communication with the client.
- Property **RSAGenerating rsaGeneratingServer** an instance of the RSAGenerating class used for RSA encryption on the server side.
- Property **RSAGenerating rsaGeneratingClient** an instance of the RSAGenerating class used for RSA encryption on the client side.

- Method **async Task<string> ReceiveMessageWithEncryptionAsync()** receives an encrypted message from the client asynchronously. It reads data from the network stream, decrypts it using the server's RSA encryption, and returns the decrypted message as a string.

- Method **async Task<string> ReceiveMessageAsync()** receives a message from the client asynchronously. It reads data from the network stream, decodes it using UTF-8 encoding, and returns the received message as a string.
- Method **async Task SendMessageWithEncryptionAsync(string message)** sends an encrypted message to the client asynchronously. It encrypts the provided message using the client's RSA encryption, and writes the encrypted data to the network stream.
- Method **async Task SendMessageAsync(string message)** sends a message to the client asynchronously. It encodes the provided message using UTF-8 encoding and writes the encoded data to the network stream.
- Method **void CloseStreamConnection()** closes the network stream connection.

❖ Constructor **InteractWithClient(TcpClient client, RSAGenerating rsaClient, RSAGenerating rsaServer)** initializes the InteractWithClient object with a TcpClient instance representing the client connection, and RSA encryption instances for the client and server. The network stream is obtained from the client, and the RSA encryption instances are assigned to the corresponding properties.

ServerCore.cs

The main class in the file ServerCore.cs is called ServerCore. This class represents the core functionality of a server and handles the communication with clients.

ServerCore:

- Property **CommunicationWithClient communicationWithClient** a property of the CommunicationWithClient class, which is responsible for handling communication with clients.
- Property **string LogFileName** a string property that represents the name of the log file.
- Property **BasicInfoIpAddress ipAddress** a property of the BasicInfoIpAddress class, which provides basic information about an IP address and a connection port.
- Method **async void StartServer()** a method that starts the server and begins listening for client connections. It first creates a log file using the SimpleLogs class. Then it sets the LogFileName property to the name of the created log file. After that, it initializes a TCP listener using the ipAddress field. The method then tries to start the listener and enters a loop where it accepts incoming client connections asynchronously. Each accepted client connection is passed to the HandleClientAsync method of the communicationWithClient object. If an exception occurs during the process, it is logged using the SimpleLogs class. Finally, when the loop is interrupted or finished, the listener is stopped.
- Constructor **ServerCore(string ip = Constants.DefaultIP, int port = Constants.DefaultPort)** a constructor that initializes the ServerCore object. It takes optional parameters for the IP address and port, with default values taken from the Constants class. Within the constructor, it initializes the communicationWithClient field and sets the LogFileName to an empty string. It also creates an instance of BasicInfoIpAddress using the provided IP address and port values.

Constants.cs

The main class in the file Constants.cs is called Constants. The class contains various constants that are used in different parts of the code.

Controller.cs

The main class in the file Controller.cs is called Controller. This class is responsible for controlling the server core and managing the log file. Also, it provides communication between backend and GUI parts of program.

Controller:

- Property **ServerCore ServerCore** this property represents an instance of the ServerCore class. It allows access to the server core functionality, such as starting the server.
- Property **string LogFileName** this property stores the name of the log file.

- Method **void RunServer()** this method starts the server by calling the StartServer() method of the ServerCore instance. After starting the server, the LogFileName property is updated with the value from the ServerCore.LogFileName property.

- Constructor **Controller(string ip = Constants.DefaultIP, int port = Constants.DefaultPort)** this constructor initializes a new instance of the Controller class. It takes optional parameters for the IP address and port. If no values are provided, the default values from the Constants class are used. Inside the constructor, a new instance of the ServerCore class is created, passing the IP address and port. The LogFileName is initially set to an empty string.

MainWindow.xaml.cs

The main class in the file MainWindow.xaml.cs is called MainWindow. This class implements the interaction logic for the main window of a server-side application. It has a role in creating a server and managing its communication while offering a way to view the server logs.

MainWindow:

- Property **Controller controller** is used to manage and control the operations of the server. It includes setting up the server, starting it, and logging its activities.
- Property **DateTime lastModifiedTime** is used to store the time when the server log file was last modified.

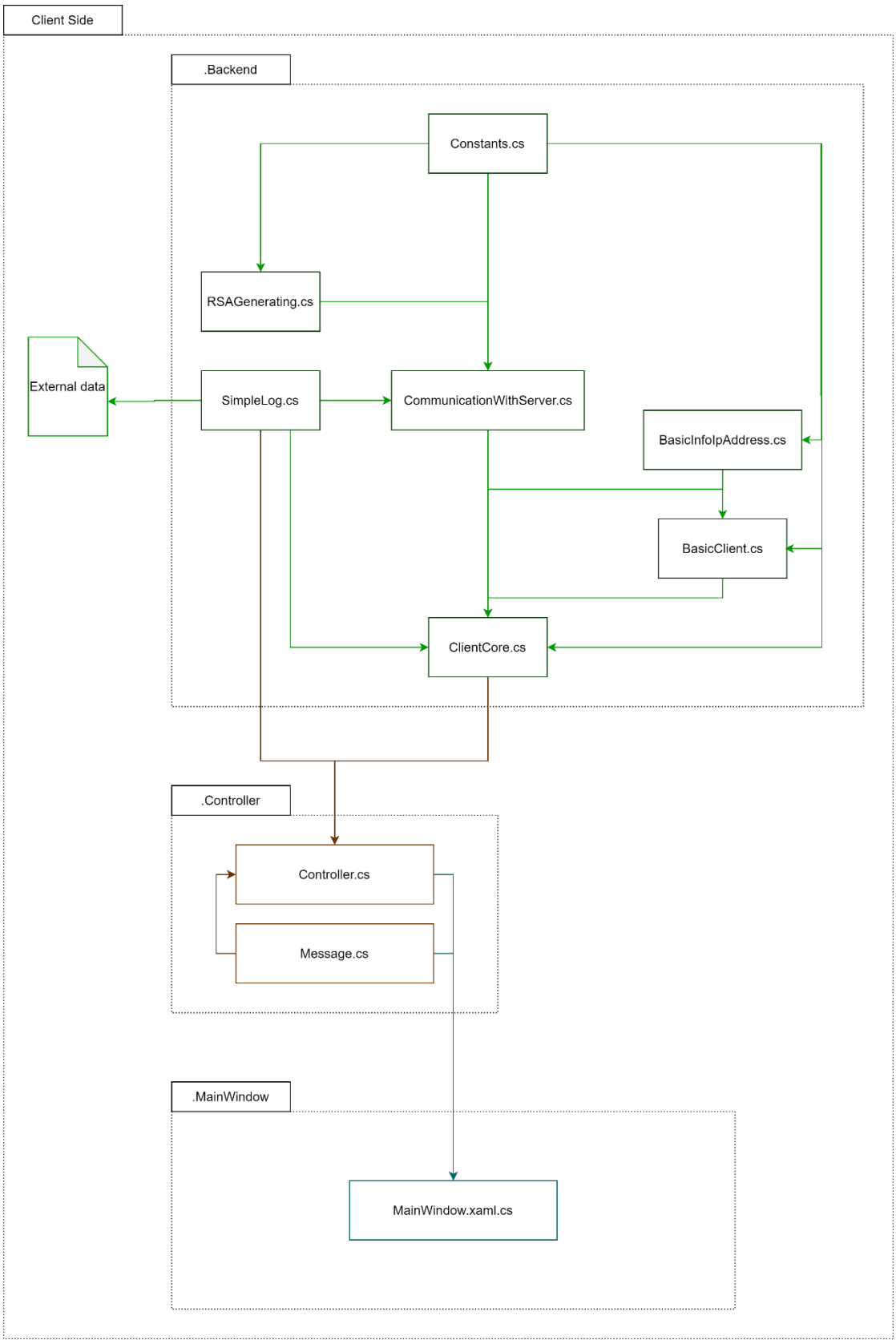
- Method void **CreateServer(string ip, int port)** is used to create a new server instance with the specified IP address and port, run the server and show the system log.
- Method void **ShowLogSystem()** starts a timer to refresh and display the system log every second.
- Method **RefreshTimer_Elapsed(object sender, ElapsedEventArgs e)** triggers every second due to a timer and refreshes the log display if the log file has been modified since the last display update.
- Method **ReadAndDisplayLog()** is used to read the current log file content and display it on the UI, also updating the last modified time of the log file.
- Method **Button_Click_Start_Server(object sender, RoutedEventArgs e)** triggers on a button click, validates and parses the IP address and port number entered by the user, disables server start and input fields after server creation, and creates a server with the provided IP and port.

❖ Constructor **MainWindow()** initializes the MainWindow component.

## About the client part code

To implement the client part, I also divided code into 3 parts: the backend, the controller, and the GUI:

- The backend is responsible for analyzing, securing and processing all data that is sent to and from the server.
- The controller is responsible for the communication between the backend and the GUI.
- The user interface is responsible for displaying logs and entering basic information to configure the server before it starts.




*Picture of dependencies between backend, controller and GUI of the client side*

BasicInfoIpAddress.cs

The main class in the file BasicInfoIpAddress.cs is called BasicInfoIpAddress.

This class is used to provide basic information about an IP address and a connection port. This data can be used later for determining server settings.

BasicInfoIpAddress:

- Property **bool DebugMode** is used to allow the developer to define different behaviors of methods depending on whether the program is currently in debugging mode.
- Property **IIpAddressProvider IpAddress** is used to provide the IP address as an object of the IIpAddressProvider interface. This allows the developer to get the IP address in the same way, but with a different method implementation.
- Property **int Port** stores the value of the port.

- Method **IPEndPoint GetIPEndPoint()** allows the developer to get an IPEndPoint with the previously declared IP address and port.
- Method **IPAddress GetIPAddress()** allows the developer to get an IPAddress with the desired implementation.

❖ Constructor **BasicInfoIpAddress (bool debugMode, string ipAddressString = Constants.DefaultIP, int port = Constants.DefaultPort)** takes bool debugMode, string ip and int port as input. By default, all this data is taken from the Constants.cs file. In the body of the constructor itself, all these values are assigned to class attributes.

BasicClient.cs

The main class in the file BasicClient.cs is called BasicClient. This class represents a basic client that interacts with an IP address and a port.

BasicClient:

- Property **IPAddress IpAddress** this property gets the IP address associated with the client. It is publicly accessible but can only be set privately within the class.
- Property **int Port** this property gets the port number associated with the client. It is publicly accessible but can only be set privately within the class.

- Method **TcpClient GetTcpClient()** this method returns a new instance of the TcpClient class. It can be used to establish a TCP connection with a server.

- Constructor **BasicClient(BasicInfoIpAddress ipAddress)** this constructor initializes a new instance of the BasicClient class. It takes an object of the BasicInfoIpAddress class as a parameter. Within the constructor, the IP address and port are obtained from the BasicInfoIpAddress object and assigned to the corresponding properties of the BasicClient class.

RSAGenerating.cs

The main class in the file RSAGenerating.cs is called RSAGenerating. This class is responsible for generating RSA keys, encrypting and decrypting data using RSA encryption, and converting data between byte arrays and strings.

RSAGenerating:

- Property **string PublicKey** represents the public key generated by the RSA algorithm. It has a protected setter, allowing modification within the class hierarchy.
- Property **RSAParameters KeyParams** represents the parameters of the RSA key. It has a protected setter, allowing modification within the class hierarchy.

- Method **GenerateKeys()** generates both private and public keys using the RSACryptoServiceProvider class. It sets the PublicKey property as a string representation of the public key and assigns the KeyParams property with the RSA key parameters.
- Method **SetPublicKeyFromString(string publicKey)** sets the public key from a string representation without setting a private key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then imports the public key from the given string. Finally, it exports the updated key parameters, excluding the private key.
- Method **EncryptRawData(byte[] data)** encrypts the given byte array data using the public key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then encrypts the data using the imported public key. The encrypted data is returned as a byte array.
- Method **DecryptRawData(byte[] data)** decrypts the given byte array data using the private key. It creates a new instance of RSACryptoServiceProvider, imports the KeyParams, and then decrypts the data using the imported private key. The decrypted data is returned as a byte array.
- Method **EncryptString(string data)** encrypts the given string data. It converts the string to bytes using UTF-8 encoding, divides the data into blocks, encrypts each block using the EncryptRawData method, and combines the encrypted blocks into a single block. The resulting encrypted data is returned as a byte array.

CommunicationWithServer.cs

The main class in the file CommunicationWithServer.cs is called CommunicationWithServer. This class is responsible for handling communication with a server using TCP/IP. It provides methods for establishing a connection, sending and receiving messages, and managing the connection state. An additional class is InteractWithClient, which implements methods for communicating with a server via TCP.

CommunicationWithServer:

- Property **TcpClient Client** represents the TCP client used for communication with the server.
- Property **CancellationTokenSource cancellationTokenSource** is used to cancel asynchronous operations.
- Property **BasicClient BasicClientInfo** contains basic client information such as the IP address and port.
- Property **InteractWithServer interactWithServer** is an instance of the InteractWithServer class, responsible for interacting with the server.
- Property **bool EstablishedConnection** indicates whether the connection with the server has been established.
- Property **ObservableCollection<Message> chatMessages** is a collection of chat messages received from the server.

- Method **async Task ConnectClientAsync()** establishes an asynchronous connection with the server using the Client property and the IP address and port from the BasicClientInfo object.
- Method **async Task EstablishConnectionWithServer()** attempts to establish a connection with the server. It continuously tries to connect until the client is connected or an exception occurs. If the connection is successful, it sets the EstablishedConnection property to true and initializes the interactWithServer object.
- Method **async Task InitialSetting()** performs initial settings for the communication with the server. It receives the server's public key and sends the client's public key to the server.
- Method **async Task<string[]> GetRoomNamesFromServer()** sends a request to the server to get a list of room names. It receives the room names as a string, splits it into an array, and returns the array of room names.

- Method **void StartToReceiveMessages()** starts a background task to asynchronously receive messages from the server.
- Method **async Task<bool> SetUsernameToServer(string username)** sends a request to the server to set the username. It sends the username to the server and waits for a response. If the response is not the success message, it returns false. If an exception occurs during the process, it returns false as well.
- Method **async Task SendMessageFromClientToServer(string username, string messageText)** sends a message from the client to the server. It adds the message to the chatMessages collection and sends the message text to the server.
- Method **void GetMessageFromAnotherClient(string message)** processes a message received from another client. It splits the message into elements and constructs a Message object from the elements. The constructed Message object is added to the chatMessages collection.
- Method **async Task<bool> SendJoinRoomAttemptToServer(string roomName, string password)** sends a request to the server to join a room. It sends the room name and password to the server and waits for a response. If the response is not the success message, it returns false. If an exception occurs during the process, it returns false as well.
- Method **async Task<bool> SendCreateRoomAttemptToServer(string roomName, string password)** sends a request to the server to create a room. It sends the room name and password to the server and waits for a response. If the response is not the success message, it returns false. If an exception occurs during the process, it returns false as well.
- Method **async Task ReceiveMessagesAsync(CancellationToken cancellationToken)** receives messages from the server asynchronously. It continuously receives messages until the cancellation token is requested or an empty message is received. Depending on the message type, it performs different actions, such as processing received messages from other clients and adding them to the chatMessages collection.
- Method **bool CloseConnection()** closes the connection with the server. It closes the stream connection, cancels the cancellationTokenSource, and closes the TCP client. It returns true if the connection is closed successfully and false if an exception occurs.
- Method **async Task SendToServerMessageAboutClosingConnection()** sends a message to the server indicating that the connection is being closed.
- Method **bool IsClientConnected()** checks if the TCP client is connected to the server.

- ❖ Constructor **CommunicationWithServer(TcpClient client, BasicClient basicClientInfo)** initializes the CommunicationWithServer object with a provided TcpClient and BasicClient object. It sets the client, basic client information, and initializes the cancellationTokenSource. The EstablishedConnection property is set to false.
- ❖ Constructor **CommunicationWithServer()** creates an empty CommunicationWithServer object.

InteractWithClient:

- o Property **NetworkStream Stream** represents the network stream used for communication with the client.
- o Property **RSAGenerating rsaGeneratingServer** an instance of the RSAGenerating class used for RSA encryption on the server side.
- o Property **RSAGenerating rsaGeneratingClient** an instance of the RSAGenerating class used for RSA encryption on the client side.
- ▪ Method **async Task<string> ReceiveMessageWithEncryptionAsync()** receives an encrypted message from the client asynchronously. It reads data from the network stream, decrypts it using the server's RSA encryption, and returns the decrypted message as a string.
- ▪ Method **async Task<string> ReceiveMessageAsync()** receives a message from the client asynchronously. It reads data from the network stream, decodes it using UTF-8 encoding, and returns the received message as a string.
- ▪ Method **async Task SendMessageWithEncryptionAsync(string message)** sends an encrypted message to the client asynchronously. It encrypts the provided message using the client's RSA encryption, and writes the encrypted data to the network stream.
- ▪ Method **async Task SendMessageAsync(string message)** sends a message to the client asynchronously. It encodes the provided message using UTF-8 encoding and writes the encoded data to the network stream.
- ▪ Method **void CloseStreamConnection()** closes the network stream connection.
- ❖ Constructor **InteractWithClient(TcpClient client, RSAGenerating rsaClient, RSAGenerating rsaServer)** initializes the InteractWithClient object with a TcpClient instance representing the client connection, and RSA encryption instances for the client and server. The network stream is obtained from the client, and the RSA encryption instances are assigned to the corresponding properties.

SimpleLogs.cs

The main class in the file SimpleLogs.cs is called SimpleLogs. This class provides functionality for logging messages to a file.

SimpleLogs:

- Property **string FileName** represents the name of the log file. It is a get-only property that is set using the current date and time in the format "log_dd_MM_yyyy_HH_mm_ss.txt".
- Property **string PathToFile** represents the full path to the log file. It is a get-only property that is set by combining the current directory with the FileName property.

- Method **bool CreateLogFile(string pathToFile = "")** is used to create the log file. It takes an optional parameter pathToFile, which allows specifying a custom file path. If the pathToFile parameter is not provided, the default path is used (PathToFile property).
- Method **bool WriteToFile(string textToWrite, string pathToFile = "")** is used to write a text string to the log file. It takes two parameters: textToWrite, which represents the text to be written, and an optional parameter pathToFile, which allows specifying a custom file path. If the pathToFile parameter is not provided, the default path is used (PathToFile property).

ClientCore.cs

The main class in the file ClientCore.cs is called ClientCore. This class represents the core functionality of a client application that communicates with a server.

ClientCore:

- Property **CommunicationWithServer communicationWithServer** is used to establish communication with the server. It is an object of the CommunicationWithServer class.
- Property **BasicInfoIpAddress ipAddress** is used to store basic information about the IP address and connection port. It is an object of the BasicInfoIpAddress class.

- Method **StartCommunicateWithServer()** is an asynchronous task that represents the process of starting communication with the server.

- Constructor **ClientCore(string ip, int port)** initializes the communicationWithServer property by creating a new instance of the CommunicationWithServer class.

Constants.cs

The main class in the file Constants.cs is called Constants. The class contains various constants that are used in different parts of the code.

Message.cs

The main class in the file Message.cs is called Message. This class represents a message object with various properties and methods related to messaging.

Message:

- Property **string Id** a string representing the unique identifier of the message.
- Property **string Username** a string representing the username of the sender or receiver of the message.
- Property **string Text** a string representing the content of the message.
- Property **bool SenderOrReceiver** a boolean value indicating whether the message is sent by the user or received by the user.
- Property **string Time** a string representing the timestamp of the message.

❖ Constructor **Message(string username, string text, bool senderOrReceiver)** a constructor that initializes a new instance of the Message class. It takes the username, text, and senderOrReceiver as input parameters. It sets the Username, Text, SenderOrReceiver, and Time properties based on the provided values. Additionally, it generates a unique Id for the message using SHA-1 hashing algorithm and the current timestamp, username, and text.

Controller.cs

The main class in the file Controller.cs is called Controller. This class is used for managing client-side actions such as connecting to the server, setting usernames, joining rooms, creating rooms, sending and receiving messages, and updating room names and chat messages. Also, it provides communication between backend and GUI parts of program.

Controller:

- Property **ObservableCollection<Message> chatMessagesOld** and chatMessagesNew are used to store and track the old and new chat messages respectively.
- Property **ObservableCollection<string> RoomsNames** is used to store the names of all chat rooms.
- Property **string CurrentRoomName** is used to store the name of the current chat room that the client has joined.
- Property **string Username** is used to store the username of the client.
- Property **ClientCore ClientCore** is used to handle the client-side core functionalities.

- Method **TryToSetUsername(string nickname)** allows the client to set a username and communicates this to the server.
- Method **TryToJoinRoom(string roomName, string password)** allows the client to join a chat room on the server.
- Method **TryToCreateRoom(string roomName, string password) allows** the client to create a new chat room on the server.
- Method **SetNewMessage()** updates the chatMessagesNew property with the latest messages from the server.
- Method **SendMessage(string messageText)** allows the client to send a message to the server.
- Method **UpdateRoomsNames()** updates the RoomsNames property with the latest room names from the server.
- Method **VariableUpdateChatMessages()** updates the chatMessagesOld property with the values from chatMessagesNew.
- Method **GetNewMessages()** returns the new messages that have been received by comparing message IDs between old and new messages.

- Method **IsClientConnectToTheServer()** checks if the client is connected to the server.
- Method **StartCommunicate()** initiates communication with the server.
- Method **StartReceiveMessagesFromUsers()** initiates the reception of messages from other users.
- Method **CloseCommunication()** sends a message about closing the connection to the server and closes the connection.

❖ Constructor **Controller(string ip = Constants.DefaultIP, int port = Constants.DefaultPort)** takes string ip and int port as input. By default, all this data is taken from the Constants.cs file. In the body of the constructor itself, a new instance of ClientCore is created with these values.

Message.cs

The main class in the file Message.cs is called Message. This class represents a message object with various properties and methods related to messaging.

Message:

- Property **string Id** a string representing the unique identifier of the message.
- Property **string Username** a string representing the username of the sender or receiver of the message.
- Property **string Text** a string representing the content of the message.
- Property **bool SenderOrReceiver** a boolean value indicating whether the message is sent by the user or received by the user.
- Property **string Time** a string representing the timestamp of the message.

❖ Constructor **Message(string username, string text, bool senderOrReceiver)** a constructor that initializes a new instance of the Message class. It takes the username, text, and senderOrReceiver as input parameters. It sets the Username, Text, SenderOrReceiver, and Time properties based on the provided values. Additionally, it generates a unique Id for the message using SHA-1 hashing algorithm and the current timestamp, username, and text.

MainWindow.xaml.cs

The main class in the file MainWindow.xaml.cs is called MainWindow. It represents the main window of a client-side chat application, and its key roles involve managing user interface interactions, messaging, connection handling, and other client-side operations.

MainWindow:

- Property **string TextBoxPreviewText** it is a string property. Its default value is "Type here...", which is used as a placeholder in the chat input box.
- Attribute **DispatcherTimer messageTimer** this is instance of DispatcherTimer class which manage task for checking for new messages.
- Attribute **DispatcherTimer connectionTimer** this is instance of DispatcherTimer class which manage task for attempting server connections.
- Attribute **DispatcherTimer getRoomNamesTimer** this is instance of DispatcherTimer class which manage task for updating room names respectively.
- Attribute **Controller controller** this is an instance of the Controller class, which manages the core logic behind sending, receiving, and managing messages and user information.

- Method **TryToConnectToTheServer(), TryToConnectToTheServer_Tick(object sender, EventArgs e)** these methods attempt to establish a connection to the server. A timer is set up in TryToConnectToTheServer() that periodically checks the connection status until it is successful.
- Method **UpdateRoomNames(), UpdateRoomNames_Tick(object sender, EventArgs e)** these methods update the list of room names in the chat application. A timer is set up in UpdateRoomNames() that periodically updates the room names.
- Method **ShowMessages(), MessageTimer_Tick(object sender, EventArgs e)** these methods handle the display of new messages in the chat window. A timer is set up in ShowMessages() that periodically checks for new messages and displays them.

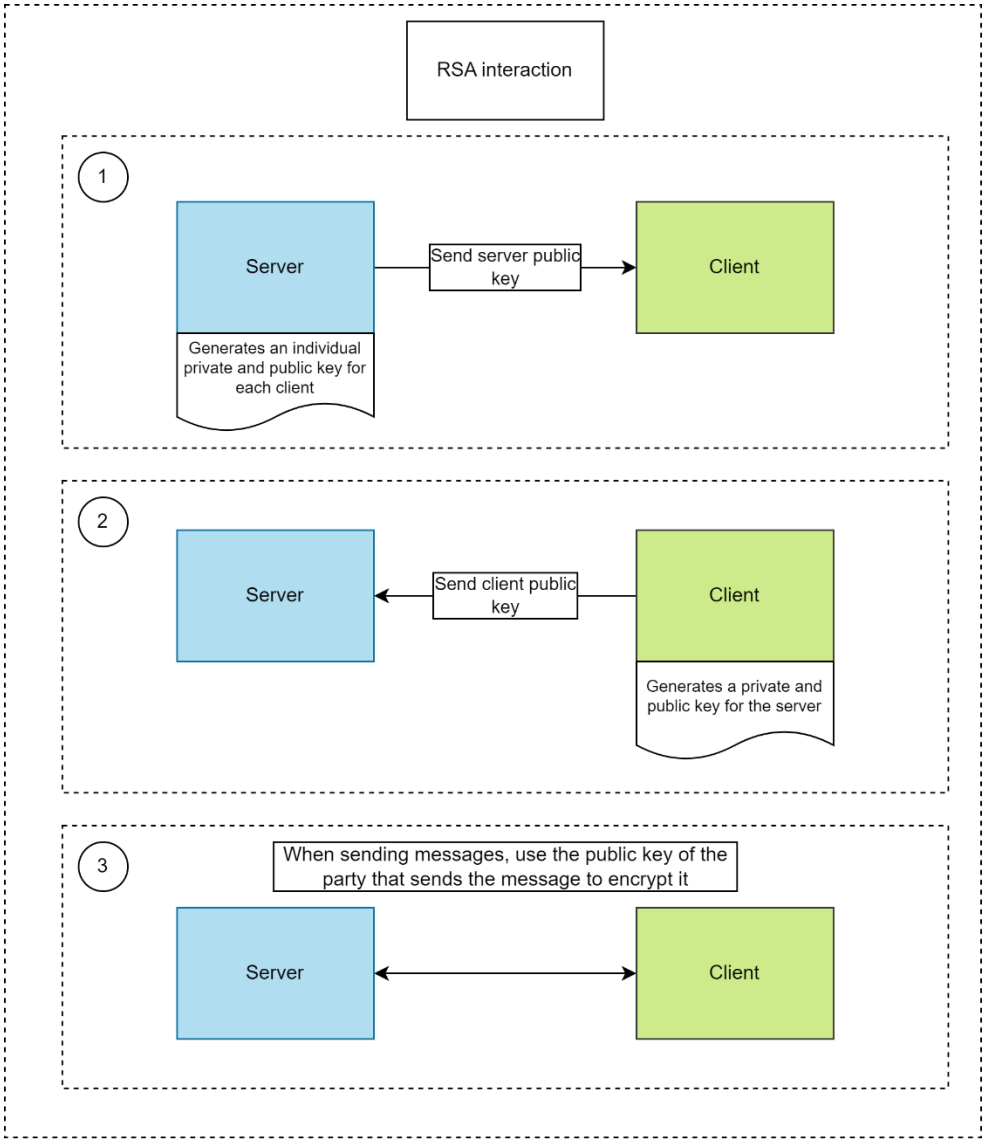
- Method **AddMessageToChat(string userNickname, string message, bool senderOrReceiver)** this method takes in the user nickname, message, and a boolean indicating whether the user is a sender or receiver. It creates and formats a new message to be added to the chat window.
- Method **GeneratePastelColorFromString(string inputString)** this method generates a pastel color based on an input string. It's used for generating unique colors for user nicknames.
- Method **JenkinsHash(string input)** this method generates a hash value from an input string, which is used in color generation.
- Method **MyGrid_VisibilityChanged(object sender, DependencyPropertyChangedEventArgs e)** this method starts receiving messages from users when the chat grid becomes visible and stops the getRoomNamesTimer.
- These are various event handlers managing button clicks, room joining/creation, and ensuring proper disconnection when the window is closed: **Button_ClickAsync(), ButtonContinue_Click(), JoinButton_ClickAsync(), JoinChoosingButton_Click(), CreateChoosingButton_Click(), CreateButton_ClickAsync(), ButtonBackToChoose_Click(), Window_Closed(), SendMessageAsync(), TextBox_GotFocus(), RoomsListBox_SelectionChanged()**

❖ Constructor **MainWindow()** initializes the user interface, establishes initial communication, sets up timers for handling messages and connections, and assigns the current instance as the data context for binding operations. It also binds the 'roomsListBox' to the controller.

About security and client-server interaction

To exchange messages between the client and the server over TCP I use RSA encryption, here are some reasons why I chose this RSA algorithm:

- Security: RSA is one of the most widespread and proven cryptographic algorithms that provides a high level of security.
- If the key is intercepted during transmission, and then the message encrypted with that key is intercepted, the attacker will not be able to decrypt it because the public key is only intended for message encryption.




*Picture of client-server encryption key exchange*

- By default, the program is set up to exchange data at 6KB per transfer. Of which 3 KB is available for useful information, while the rest of the message space is spent on encryption.

- The server and the client have predefined commands that are added to the beginning of a message before sending it. These commands are necessary for the client and server to understand what they want to obtain or accomplish with a particular message.
  One example of such a command set is the application of a user name. The client sends a message with a command at the beginning indicating that the client wants to set a user name. The server recognizes this and responds with either a positive command if the user's name is available or a negative command if the user's name is already taken by someone else.

- During operation, the server identifies users based on their unique ID generated by hashing their input data using the SHA algorithm. This provides additional security on the server side.

- The server does not store any user data. All conversations conducted between users are local and are immediately deleted upon the closure of all clients involved in the conversation. This ensures additional anonymity and security.

## About code testing

To verify the correctness of the code, you have written a series of unit tests for the most important and necessary parts of the code. Some of these unit tests are integration tests, which means they require a running server or client. If a test is an integration test, the necessary steps for its proper activation are described in code comments.

## Conclusion about the created program

During the development of a small chat application, I learned how RSA encryption works, how to write client and server components, and how to work with the WPF graphical interface. Additionally, I gained experience in writing unit tests, some of which were integration tests.

In the future, I would like to add new usage scenarios to the program. These include an improved user interface and enhanced user experience. I want to add new commands to the client and server parts to enable file transfer. I also want to give users the option to choose whether they want to save their conversations in a database for future reference. To achieve this, I will need to write secure and reliable interactions between the server and the database.