

Programming Project: Quantum Encryption

Presented to Elaheh Mozaffari

Prepared By:

Marc Scattolin

Raphael Pinsonneault-Thibeault

Contents

Project Overview	3
Project Story	3
Quantum Computing Basics	3
Project Description	5
System Objectives.....	5
Project Constraints and Scope	5
Tools.....	5
Critical Project Events	6
Project Design	6
Communication.....	6
QKD	9
QKD UML Diagrams	13
QKA	16
QKA UML Diagrams.....	23
Python	26
Encryption Parameters.....	27
GUI	28
Methods of Evaluation	31
Key Exchange	31
GUI	31
Results: System Quality.....	32
Project Management: Timeline	33
Conclusion	34
Annex: System Manual	35
Notes.....	35
Setting it up.....	35
Python.....	35
Java	36
Compiling and Running.....	36
User Documentation	38
Works Cited.....	48

Project Overview

Project Story

When the presenters behind talks on quantum computing brought up the idea of quantum encryption, we realized that this is in an emerging field, with possibly far-reaching implications.

Before quantum encryption, though, classical encryption needs an introduction. Much of what we do on the internet relies on it. Thanks to encryption, no one can know what we send on the internet besides ourselves and the intended recipient, even though internet packets are effectively public. This allows us to perform financial transactions or send messages over the internet without having to worry that someone is watching over our shoulder.

However, quantum computers could change this. Classical algorithms currently in use, like Diffie-Hellman (used to establish a shared private key), rely on mathematical 'trapdoors', problems that are too difficult to reverse (Denning). However, quantum computers may be able to reverse these 'trapdoors' and make current encryption much less secure. For instance, if someone could efficiently factor large numbers into its prime divisors, RSA, a public-key cryptography scheme, would fall apart. Shor's algorithm could be used to do this in only a few hours (Denning). Because of technical hurdles, this has yet to happen; the largest number factored is 15. However, it is a possibility moving forward.

This is one of the reasons behind the importance of this project. Quantum key distribution could be an encryption standard of the future, and thus creating implementations of it may be a crucial part of information security moving forward. Another reason behind the importance of it is that it will allow us to practice the programming skills we have developed over the past three semesters.

Quantum Computing Basics

To start, some of the basics in quantum computing need to be covered.

The first is the qubit. It is the unit of information that quantum computers deal in. Like a bit in a regular computer, it can be 0 or 1. However, where a quantum computer distinguishes itself is that a qubit can be in-between 0 and 1. This can be in any ratio; it could be 50% 0 and 50% 1, or 30% 0 and 70% 1 (Abraham).

However, the way that qubits are usually written is slightly different. The state of a qubit that is 50% 0 and 50% 1 would be written as $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. In this notation, $|0\rangle$ can be considered like regular '0' and $|1\rangle$ can be compared to '1'. Because the qubit is at the same time 0 and 1, it is said to be in a **superposition**.

However, when a qubit is measured, the superposition isn't measured; the output is only 0 or 1. In addition, once measured, a qubit's state is changed to a definite measurement and will produce the same result if measured again (unless altered by a subsequent quantum gate). If a qubit that is $\frac{1}{2}$ 0 and $\frac{1}{2}$ 1 is measured to be 1, then it is 1, and will always measure as 1 from then on.

As mentioned, what state is measured is based on probability. The equation determining this is:

$$p(|x\rangle) = |\langle x|\varphi\rangle|^2$$

Where $|x\rangle$ is the state that the qubit is being compared against and $|\varphi\rangle$ is the state of the qubit (Abraham).

Some linear algebra is necessary to understand this. This notation: $\begin{bmatrix} \end{bmatrix}$ represents a column vector, and $\begin{bmatrix} \end{bmatrix}$ represents a row vector. A 'row vector' and a 'column vector' mean essentially what they say on the tin: a row of numbers (like $[1 \quad 5 \quad -5]$) and a column of numbers (like $\begin{bmatrix} 5 \\ 12 \\ -5 \end{bmatrix}$) respectively.

The special states $|0\rangle$ and $|1\rangle$ are, in this notation: $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Two vectors can be added together (in this case, their corresponding entries are added). One vector can also be multiplied by a number (in that case, all numbers in the vector are multiplied by the number).

Now if we return to the example qubit of $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, this would be written as $\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{1}{\sqrt{2}}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$.

Returning to the equation, those two symbols being put next to each other means that they are multiplied. How this works is that each entry in the row is multiplied by its corresponding entry in the column; these products are then summed. It looks like this: $\begin{bmatrix} a_1 & \vdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix} = a_1 * b_1 + \dots + a_n * b_n$.

Now that the basics have been gone over, the square roots should start to make sense. The probability of a qubit $\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$, being measured as $|0\rangle$, is, according to the equation:

$$p(|x\rangle) = |\langle x|\varphi\rangle|^2 = \left| \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \right|^2 = \left| \frac{1}{\sqrt{2}} + 0 \right|^2 = \left(\frac{1}{\sqrt{2}} \right)^2 = \frac{1}{2}$$

Which is what we expect. This qubit would also have a $\frac{1}{2}$ chance of being measured as $|1\rangle$.

There's also an important subtlety here. Because of the absolute value and the squaring, each of the components of 0 and 1 don't have to be positive. Another crucial point is that a qubit doesn't have to only be measured against $|0\rangle$ or $|1\rangle$ (Abraham). Although, because measuring against them is so common, it has its own name: the Z-basis. This is the main measurement basis.

This is also the ideal time to introduce two more special qubits. They are $|+\rangle$ and $|-\rangle$. The first is the qubit that we just measured the probability of measuring. The other, $|-\rangle$, is $\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. It also has a $\frac{1}{2}$ chance of measuring as $|0\rangle$ or $|1\rangle$. Now, the X-basis can be introduced. Measuring on it means measuring against the states $|+\rangle$ or $|-\rangle$. For reasons that are outside of the scope of this paper, $|+\rangle$ is considered to represent '0' in the X basis and $|-\rangle$ represents '1' in the X-basis. Thus, a 0 or 1 can be **prepared** in either base, the Z-basis or the X-basis. It is also possible to measure a bit in **either basis**.

How do these measurement bases mix? Well, if a qubit is prepared in the Z-basis and then measured in the Z-basis¹, nothing exciting happens: the qubit is measured as it was prepared. What is important is what happens when a qubit is prepared in one base and measured in another. If a qubit is prepared as $|+\rangle$, but then measured in the Z-basis, then it has a $\frac{1}{2}$ chance of measuring $|0\rangle$ and $\frac{1}{2}$ chance of measuring $|1\rangle$, as seen before. As mentioned before, when a qubit is measured, it is changed. So, for example, if a qubit is prepared as $|+\rangle$ and measured in the Z basis to be $|0\rangle$, then **it is** now $|0\rangle$. If it's then re-measured in the X-basis, it again has a $\frac{1}{2}$ chance of being measured as $|+\rangle$ and a $\frac{1}{2}$ chance of measuring as $|-\rangle$ (Abraham).

A final key point is the no-cloning theorem. It states, essentially, that an unknown qubit cannot be cloned (Haitjema). This is what makes these protocols secure.

Project Description

System Objectives

The purpose was to create a messaging application utilizing 2 quantum key distribution protocols. This necessitates the following objectives:

- Accurately implement two quantum key distribution protocols.
- Have implementations be efficient.
- Create a simulation of a messaging application.
- Use protocols to effectively encrypt and decrypt messages in the messaging application.
- Demonstrate the security of the protocols.

Project Constraints and Scope

The conception of the project was for it to be a messaging application, encrypted using two different quantum key distribution algorithms. The application was defined to be a simulation, hence not a multiple-client program. The specific constraints are listed in the System Objectives.

The application built is a functioning simulation of a messaging application, demonstrating the security of two effectively implemented quantum key distribution algorithms. Thus, the product delivered meets outlined objectives, constraints and scope.

Tools

The tools we used to write this project are:

- **NetBeans IDE**
 - This was used as the IDE to write the Java code.
- **Spyder, Python, and Anaconda**
 - This was used to write the Python implementations of the key distribution algorithms.
- **Qiskit**
 - This Python library was used to simulate a quantum computer for the protocol.
- **JavaFX**
 - This library was used to run the GUI.

¹ Or prepared in the X basis and measured in the X basis.

- **Gradle**
 - This was used to manage dependencies.
- **SceneBuilder**
 - This was used to design most of the GUI windows.
- **Jasypt**
 - This was used to encrypt the messages with the keys that were generated with the quantum key distribution protocols.
- **Windows**
 - This is the platform we developed on.
- **Git and GitHub**
 - These were used to manage our source files.

Critical Project Events

The following dates correspond to important events.

29 Mar 2021: The implementation of the QKD was finished (including interfacing with Qiskit). This meant that with the rudimentary GUI created at that date, two users were able to message securely, encrypted with the key from the QKD algorithm.

19 Apr 2021: The complete implementation of the QKA was finished. Hence, similarly to the stated directly above, two users were able to message with encryption provided by the QKA algorithm.

21 Apr 2021: GUI was finished, marking the completion of the GUI. Thus, with precedent completion of the encryption algorithms, this was the date of completion of the application. The work on the code after this point was only minor fixes.

Project Design

The main problem we tackled in this project is, as mentioned, quantum key distribution. We found two algorithms that could do this: a quantum key distribution protocol based on BB84, and a quantum key agreement protocol. They will be described in turn.

The other main design portion was the GUI elements of the application. This project needed windows to create users, manage chat windows, and chat with another person. Each of these will be described, as well as some of the smaller helper classes that were created.

Communication

Protocol

Both the QKD and QKA will ultimately accomplish the same result. In essence, they are both key distribution protocols that make a key used to encrypt and decrypt messages. Hence, they both implement Protocol.

Figure 1: Protocol Class

<<Interface>> Protocol
+ encryptMessage(message: byte[]): byte[] + decryptMessage(message: byte[]): byte[] + stringToBytes(str: String): byte[] + bytesToString(bytes: byte[]): String

Methods of Protocol class:

- encryptMessage(message: byte[]): byte[] and decryptMessage(message: byte[]): byte[] are the default interface abstract methods to be overridden in all the following classes: QKDBob2, QKDAlice2 and QKAuser. Although not shown here, these methods each throw the KeyExchangeFailed exception. That exception is thrown when the quantum key exchange fails, either because an eavesdropper was detected or because the Python code could not run. The EncryptionException and DecryptionException are thrown by the respective methods. These exceptions wrap around any exception thrown by the library we are using to perform the encryption and decryption. For instance, the DecryptionException is thrown when a message is attempted to be decrypted with a different key than it was encrypted with.
- stringToBytes(str: String): byte[] and bytesToString(bytes: byte[]): String are static methods in the interface that convert as respective name suggests so Strings can be passed through the other methods.

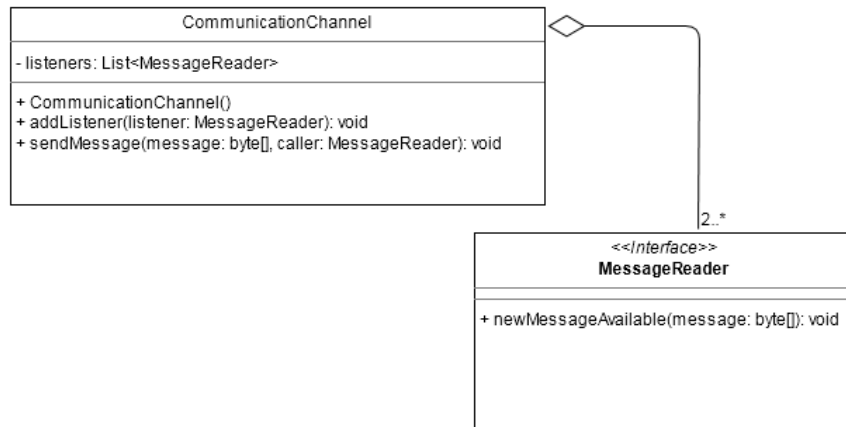
Communication Channel

Something else that we realized needed to be simulated was something through which the encrypted messages pass from user to user. To do this, we designed the CommunicationChannel class and MessageReader interface. These classes permit the two chat windows send chat messages to each other while simulating a public channel.

The CommunicationChannel allows instances of MessageReader to “register” themselves as listeners, or objects that want to be notified whenever a message is sent through the channel. These MessageReaders are added to a List. Then, when an object wants to send a message through the channel, the CommunicationChannel notifies all its listeners in the List with the method in MessageReader. Given that it takes $O(n)$ to iterate through a list of objects, this algorithm runs in $O(n)$ time to send a message.

Its diagram follows:

Figure 2: Communication Classes



Explanations for this figure:

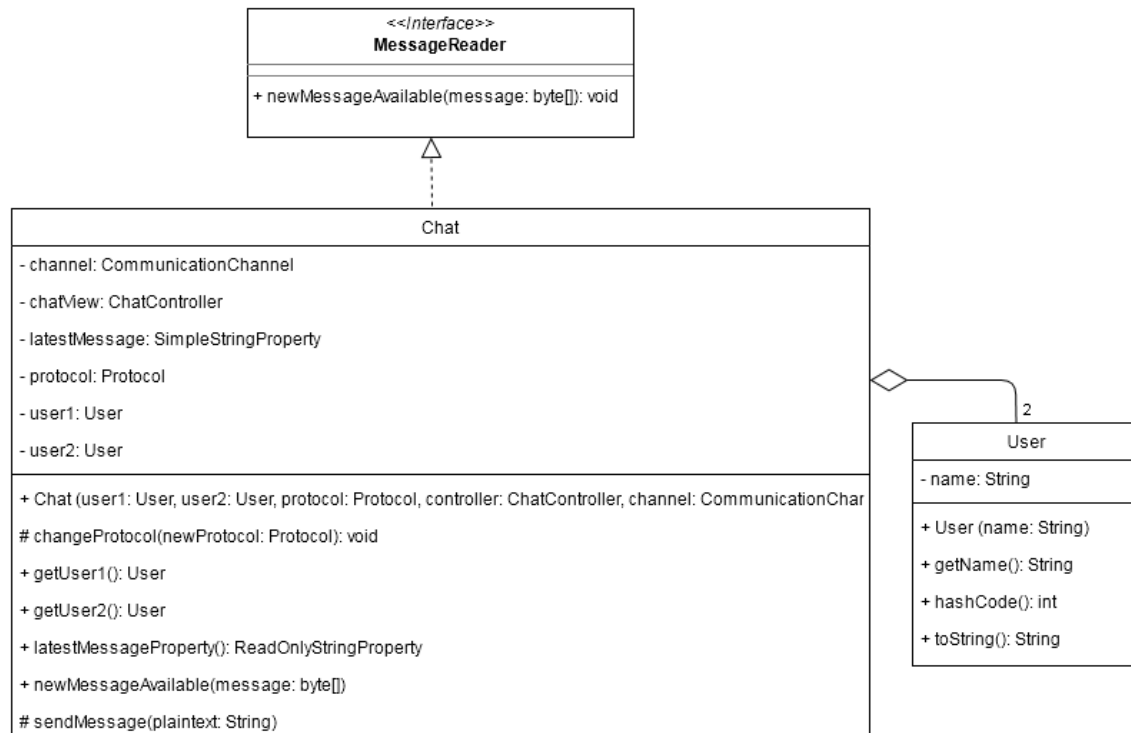
- The **MessageReader** interface, and thus the `newMessageAvailable(byte[] message)` method, is implemented by the **Chat** class. It is called by the **CommunicationChannel** when a new message is sent. The **Chat** instance decrypts and decodes the message, then passes it to the GUI to be displayed.
- The `addListener(MessageReader listener)` method adds the passed object to the listeners list in the **CommunicationChannel**. This object will be notified whenever there is a new message sent via its `newMessageAvailable` method.
- The `sendMessage(byte[] message, MessageReader caller)` method is used to send an already encrypted message to all listeners. A **MessageReader** is passed as a parameter so that the **CommunicationChannel** will not send the message back to the original sender and instead only send it to other **MessageReaders**.

Chat

Another class that was made in this project was the **Chat** class. It exists to manage the flow of messages between the chat windows, and provides the `StringProperty` that allows the user window (the window that shows all of a user's chats) to show the most recently sent message in its interface. It is also where the control users window changes the Protocol between two chatters.

Its diagram is:

Figure 3: Chat Class



Precisions regarding the important methods:

- The `changeProtocol(Protocol newProtocol)` method swaps out the protocol that will be used in future message sending.
- The `latestMessageProperty()` method returns a `ReadOnlyStringProperty`. This property is internally not read-only and is set by this whenever a message is sent or successfully received. The user window binds this property to a part of its interface, so it always shows the most recent message.
- The `newMessageAvailable(byte[] message)` method is implemented from `MessageReader`, and it is called when the person on the other end of the chat sends a message. Chat decrypts it using the `Protocol` and sends it to the `ChatController` (the chat window).
- The `sendMessage(String plaintext)` method is called by the chat window to encrypt and send a message. It simply uses the `Protocol` and `CommunicationChannel` it has to encrypt and send the message.

QKD

This protocol, also called BB84, allows two people (Alice and Bob) who have access to a quantum channel to establish a shared random number, specifically, a string of bits, which serves as a key. This number is used, in this case, in a symmetric encryption algorithm to safely pass messages back and forth.

Utility Methods

Before the actual process is described, a few other methods will be described. These methods are used in the implementation of this protocol; you will see them in the code listing of this protocol. These were placed in the `QKDAlice` class for historic reasons. Its diagram is:

Figure 4: Utility Class for QKD

QKDAlice
[fields omitted]
<u>+ keepAtIndices(indices: List<Integer>, str: String): String</u>
<u>+ matchingIndices(a: String, b: String): List<Integer></u>
<u>+ removeAtIndices(indices: List<Integer>, str: String): String</u>
<u>+ sampleIndices(ratio: int, length: int): List<Integer></u>

The methods are:

- The keepAtIndices(List<Integer> indices, String str) method takes the input string and returns only the string with the indices specified in the input list.
- The matchingIndices(String a, String b) method returns a list of integers of the indices where the two strings are the same
- The removeAtIndices(List<Integer> indices, String str) method takes the input string and returns the string with the indices at the specified indices removed.
- The sampleIndices(int ratio, int length) method returns a list of integers. This elements of the list are in sorted order, are in the range [0,length), and the ratio of the length of the returned list to the specified length is approximately the ratio parameter, where the ratio is a percentage out of 100.

These methods all have pretty simple implementations and will not be described further except for the sampleIndices method, which is described near the end of this section.

The Protocol

The first step is Alice creating a long string of random 1s and 0s. She also makes another random binary number. This string does not determine the bits of the key; instead, it determines the way they will be encoded into qubits (Haitjema). There are two common bases that qubits can be encoded in, the Z-basis, and the X-basis. This encoding is shown in the listing that follows.

Listing 1: Alice making Circuits

```
def makeSendCircuits(keyData):  
    circuits = []  
    for i in range(number of bits to send):  
        circuit = QuantumCircuit(1,1)  
        if (keyData.sendBits[i] == 1):  
            circuit.x(0) #invert the bit to make it 1  
        if (keyData.sendBases[i] == 1):  
            circuit.h(0) #put the bit in the X basis  
        circuits.append(circuit)  
    return circuits
```

Listing 2: Measuring QuantumCircuits

Note that this code uses the H gate, which is the same gate used to encode a qubit into the X basis, to turn it into a Z basis qubit since the measure function in qiskit only measures on the Z basis.

Then, Bob tells Alice what bases he measured the bits in (without revealing what the actual measurements were). Alice then tells Bob which of his measurements were in the same base as she prepared them using the `matchingIndices` method. At this point, Bob discards his measurements which he did in a different base than Alice, and Alice does the same, with the `removeAtIndices` method.

Table 1: Example Measurement of QKD Protocol

Alice's bits	0	1	0	0	0	1	
Alice's bases	Z	Z	X	X	Z	X	
Alice's qubits	0	1	+	+	0	-	
This is sent to Bob							
	Z	Z	X	Z	X	X	Bob's bases
	0	1	+	1	+	-	Bob's measurement
At this point, Alice and Bob compare their bases							

Where measurement bases matched	Y	Y	Y	N	N	Y	
Final key (Alice)	0	1	0			1	
	0	1	0			1	Final key (Bob)

The reason that this is secure has to do with the no-cloning theorem. It states that an unknown qubit state cannot be perfectly duplicated. Thus, if an eavesdropper Eve tried to get the key, she would have to measure the qubits. In doing so, they would be altered, and Bob's key would be different than Alice's, even where they measured in the same base (if Eve measured in a different base to Alice and Bob).

Security Check

Finally, in order to verify that the transmission happened successfully and with no eavesdropping, Alice and Bob send a sample of their measurements. For instance, they can send every other bit to each other and verify that they line up. If this check passes, Alice and Bob can be fairly sure that no eavesdropper altered the qubits by measuring them, and they use the remaining bits to form a key.

To expand on that last point, every time Eve measures a qubit, as can be seen in the table below, she has a ¼ chance of her measurement affecting the outcome. So, she has a chance of getting away with it, if a small number of bits are compared as Alice and Bob compare measurements (Haitjema). This is how this protocol is made secure: Alice and Bob send a sample of their measurements over any communication channel and see if there are any bits where their measurements are off.

Table 2: Chance that Eve will be Detected

Qubit encoded in	Eve measures in	Outcome
Z	Z	Undetected
	X	50% chance that the outcome will be affected

The way that this was implemented is a bit more complicated, because security must be configurable. For this class, the security property is taken to be an integer from 0 to 50. This describes the approximate percentage of the measured bits that are compared. In order to determine which measurements exactly to compare and send to Alice, the `sampleIndices` method, already mentioned, is used. This method works by first determining what ratio of bits the percentage corresponds to. For example: if the security property is 50, ½ of bits are compared; if the security property is 32, 1/3 of bits are compared. That denominator is called `everyN`. Then, it creates a list of integers which starts at 0 and adds all the integers which are multiples of `everyN`. For instance, if the security property is 32, the list will contain 0,3,6,9,... With this list, every third measured bit will be compared.

The number of bits that are initially sent by Alice are calculated to compensate for this, as well as the loss due to bits which are not measured in the same base. The formula that determines the number of bits to send is:

$$\text{bitsToSend} = ((\text{KEY_SIZE_BITS} * 2.5) / (1 - (\text{securityLevel} / 100.0)))$$

Note that the security level has to be limited far below 100, since if 100% of bits measurements were compared, the number of bits sent would have to be effectively infinite.

This overall key exchange process is summarized in the following listing:

Listing 3: Bob Measuring Bits and Forming Key

```
//make measurement bases
for (int i = 0; i < bitsSent; i++) {
    bob_bases += rand.nextBoolean() ? '1' : '0';
}
//make measurements
bob_measurements = getPython().getResults(bob_bases + " " + circuits).split(" ", 2)[0];
//figure out where we measured in the same basis as alice
List<Integer> matchingMeasurements = alice.measuredSameIndices(bob_bases);
bob_matching_measured = keepAtIndices(matchingMeasurements, bob_measurements);
//make and compare a sample
List<Integer> sampleIndices = sampleIndices(alice.getSecurityLevel(), bob_matching_measured.length());
String bob_sample = keepAtIndices(sampleIndices, bob_matching_measured);
if (alice.samplesMatch(bob_sample, sampleIndices)) {
    //make the key here
    this.key = removeAtIndices(sampleIndices, bob_matching_measured);
    this.textEncryptor.setPassword(key);
}
```

Runtime Efficiency

Finally, it is necessary to discuss the time efficiency of this algorithm. It is firstly important to note that sending a qubit and measuring it all take constant time.

On average, they will discard half of the bits because there is a 50% chance that Bob will happen to measure in the same base as Alice. After this, they must make sure that no one was eavesdropping, as mentioned above. If they compare half of the bits, the maximum security property, then the number of bits sent is $4n$ ($2n \cdot 2n$), where n is the length of the key. Given that $O(4n) = O(n)$, this algorithm has linear runtime complexity with respect to key length.

Python

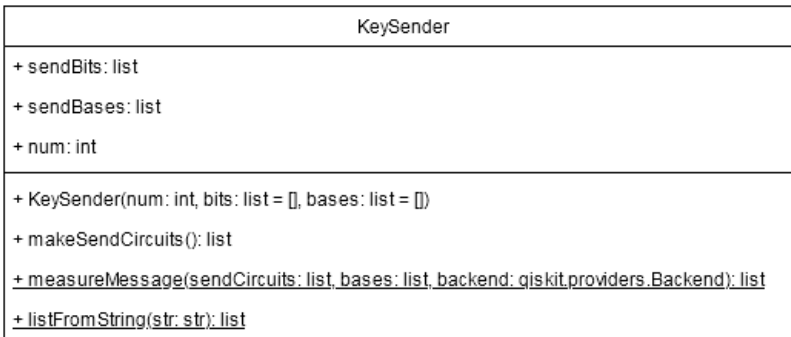
While the bulk of the key exchange happens in Java, Python is necessary to simulate the qubits. When Alice wants to send her qubits to Bob, the code first makes the bits and bases and passes it to an instance of PyScript, described later in this report. That code makes instances of QuantumCircuits, which are then serialized in Python using qasm. The whole list of circuits is passed as json back to the Java.

Then, when Bob or an eavesdropper wants to measure these circuits, the code passes those lists of serialized circuits to the instance of PyScript, which understands the different input, measures the circuits, and passes the measurement results back.

QKD UML Diagrams

The class within Python that is used to do this is pictured below:

Figure 5: UML Diagram of QKD Python Class

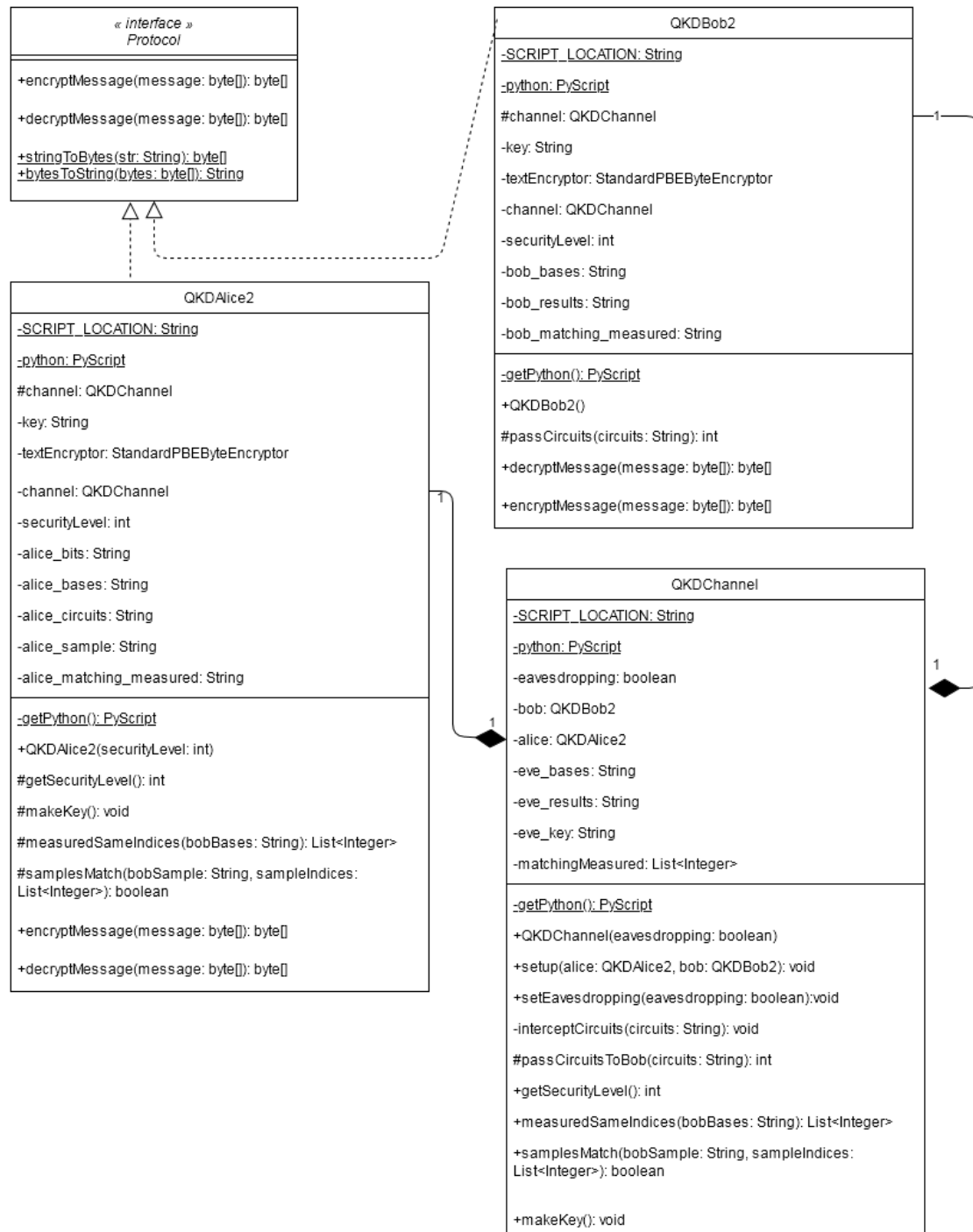


Note that while the last two methods in the class are indicated to be static methods of KeySender, in the actual code they are functions in the global scope. This notation was chosen because these functions otherwise have no place to be in a UML diagram. Also note that all data members are indicated to be public since Python does not have access modifiers.

Precisions regarding this diagram:

- The constructor requires the number of bits to send, but the bits and bases parameters are optional; they are randomly initialized if nothing is provided for them. In the actual application, the bits and bases are taken from the Java code.
- The makeSendCircuits() function returns the circuits the KeySender makes from the bits and bases it holds.
- The measureMessage(list sendCircuits, list bases, Backend backend) function measures the provided QuantumCircuits, assuming they only have one classical bit, which is the case for the circuits sent in this program.
- The listFromString(str) function splits the string character by character and returns the resulting list.

Figure 6: UML Diagram of QKD Java Classes



All of these classes have a private static String member called `SCRIPT_LOCATION`, a member of type `PyScript`, and a private static method `getPython()`. These are used to call the Python code associated with each class. This is necessary to perform the simulations of the quantum circuits, as qiskit is a python library. The String “circuits” that are referred to in these methods are lists of serialized QuantumCircuits.

The following are the methods of interest in `QKDAlice2`:

- The `getSecurityLevel()` method, which returns an integer from 0 to 50 representing what proportion of the measurements out of 100 will be used as a security check.
- The `measuredSameIndices(String bobBases)` method, which is where Alice returns a `List<Integer>` representing where they both measured in the same base.
- The `samplesMatch(String bobSample, List<Integer> sampleIndices)` method. The parameters are Bob's sample of his measurements, as well as which indices of measurements the sample are taken from. Alice then returns if her sample (taken from the same indices as Bob's sample) matches.
- The `makeKey()` method, which causes Alice to start the key exchange.

The following is the method of interest in `QKDBob2`:

- The `passCircuits(String circuits)` method, which causes Bob to perform measurements on the passed circuits and call the methods described above to perform the key exchange.

The `QKDChannel` class is the class that sits between `QKDAlice2` and `QKDBob2`. It aggregates an instance of `QKDAlice2` and `QKDBob2`, so that it can be between all communication between them, and thus attempts to eavesdrop when requested. It implements all of the methods above, to allow the key exchange to happen. When it is not eavesdropping, it just calls the corresponding method in the Alice or Bob; when it is eavesdropping, however, the following happens:

- The `passCircuitsToBob(String circuits)` method calls `interceptCircuits(String circuits)`, which measures the circuits in random bases and stores these measurements. Then it passes the measured circuits to Bob.
- The `measuredSameIndices(String bases)` method stores and then returns the list of bases where Alice and Bob measured in the same basis
- The `samplesMatch(String sample, List<Integer> sampleIndices)` method first makes a `String` consisting of the bits in locations where Alice and Bob measured in the same basis. Then, it forms Eve's attempted key by taking the aforementioned `String` and removing the entries that correspond to the sample indices. It then calls Alice's `samplesMatch` method and returns the result.

The goal of this is to attempt to follow what Bob is doing to establish a key. In practice, this always fails if the `securityLevel` is greater than 5 or so, since the chance of at least one measurement changing the outcome is near certain, as discussed above. In addition, even if the eavesdropping is not caught by the security check, Eve never gets the same key as Alice or Bob because of all the bits she measures, at least one of them will be different.

QKA

This protocol allows for two users, denoted Alice and Bob, to *agree* on an encryption key, as opposed to the QKD where Alice *distributes* the key to Bob. The protocol was created to satisfy fairness, as Alice has significantly more power in the quantum key *distribution* than Bob. Yu et al. define the properties as follows,

Security Property: Inherent within the quantum encryption, the messaging between Alice and Bob must be secure.

Weak Fairness Property: Each user contributes to the final encryption key.

Strong Fairness Property: Each user contributes to the final encryption key. Further, a user cannot have more power than another in the protocol. Consequently, no users can manipulate bits of the final encryption key.

QKD's are said to have a security property, however a weak fairness property, or no fairness properties at all (the one described has no fairness property, since Bob only measures, and does not actually contribute). Existing QKA's are said to perhaps have a security property (not being very secure), and having a weak fairness property, hence unsatisfactory fairness as well. Thus, a Quantum Key Agreement was created by Yu et al. to satisfy the security property, evidently, and the strong fairness property.

The protocol has several stages, notably, the Particle Exchange Stage, the Public Discussion Stage, and the Key Generation Stage (Yu et al.).

(To make the symbols and variables clearer, we will view Alice's steps. However, due to the strong fairness property, Bob must be doing the exact same steps as Alice, evidently in parallel.)

(Further, when encoding is mentioned, the same definition and rules stand, where a bit 0 is encoded into the state $|0\rangle$ or $|+\rangle$, and a bit 1 is encoded into the state $|1\rangle$ or $|-\rangle$)

Before the description of the protocol, the following methods are to be defined (note: the following are explained with the UML class diagram in the following section, however, for clarity, they are listed here):

Listing 4: Measure and Measure2

```
def measure(self, length, sendCircuits, recCircuits):
    mes = []
    for i in range(len(sendCircuits)):
        sendAndReceive = sendCircuits[i] + recCircuits[i]
        result = execute(sendAndReceive, self.backend, shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        mes.append(measured_bit)
    return mes

def measure2(self, ls, ba):
    sendCircuits = ls[:]
    recCircuits = Contributor.createRecCircuits(len(ba), ba)
    mes = []
    for i in range(len(sendCircuits)):
        sendAndReceive = sendCircuits[i] + recCircuits[i]
        result = execute(sendAndReceive, self.backend, shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        mes.append(measured_bit)
    return mes
```

Both output the measurements of inputted qubits. However, measure measures with sendCircuits and recCircuits (sendCircuits are the first gates corresponding to initial state changes, respecting the rules regarding $|0\rangle$, $|+\rangle$, $|1\rangle$ and $|-\rangle$) as outlined above; recCircuits are the secondary gates placed on

reception of the list by a user corresponding to state changes respective to bases, following the rules, along with a measure gate returning a measurement) while measure2 measures with the qubits and respective bases.

Particle Exchange Stage

Step 1:

Beginning, Alice creates a K_A , a string of arbitrary bits with a size of n . She then hashes K_A according to the following,

$$h_A = H(K_A)$$

Step 2.1:

Where H is an acceptable hash method, and h_A is the resultant. Next, h_A and K_A must be encoded into qubits. Note however, that K_A is a string of bits, while h_A is not, so h_A must be converted to binary. So, the following equations occur,

$$h_A \rightarrow h_A$$

$$h_A \rightarrow C_A$$

$$K_A \rightarrow S_A$$

Where the first equation denotes the conversion to binary, the second equation is the encoding of the binary hash h_A into a list of qubits, we shall call it C_A , and the third equation is the reordering of K_A , the result is called S_A . Note, the original order of S_A is kept, hence, the moves K_A made to get to S_A is kept. *Note, length $h_A \neq$ length K_A as the 32-bit of the hash is most certainly not equal to whatever length the string of bits Alice and Bob decided upon.* Next, S_A is then encoded, so $S_A \rightarrow S'_A$.

Listing 5: h_A , C_A , and S_A are Created (relevant methods are attached)

```
self.K = Contributor.cbits(n)
self.seed = np.random.randint(1,101)

Ktemp = (''.join(map(str, self.K)))
self.Kkey = int(Ktemp)

random.Random(self.seed).shuffle(self.K)
S = self.K[:]
self.K = self.unshuffle(self.K, self.seed)

H_ = Contributor.decimalToBinary(hash(Ktemp))
H = [int(x) for x in str(H_)]
```

```
def cbits(n):
    rng = np.random.default_rng()
    return rng.integers(0,1,n, endpoint=True)

def decimalToBinary(n):
    if n > 0:
        return bin(n)[2:]
    else:
        return bin(n)[3:]

def unshuffle(self, shuffled_ls, seed):
    n = len(shuffled_ls)
    p = [i for i in range(1, n + 1)]
    random.Random(seed).shuffle(p)
    zipped_ls = list(zip(shuffled_ls, p))
    zipped_ls.sort(key=lambda x: x[1])
    return [a for (a, b) in zipped_ls]
```

Listing 6: S_+ and C are Made (relevant methods attached)

```

self.bS = list(Contributor.cbits(len(S)))
self.bH = list(Contributor.cbits(len(H)))

sentS = Contributor.send(len(S), S, self.bS)
sentH = Contributor.send(len(H), H, self.bH)

S_ = sentS.makeSendCircuits()
C = sentH.makeSendCircuits()

```

```

def makeSendCircuits(self):
    circuits = []
    for i in range(self.length):
        circuit = QuantumCircuit(1,1)
        if (self.sendBases[i] == 0):
            if (self.sendBits[i] == 0):
                pass
            else:
                circuit.x(0)
        else:
            if (self.sendBits[i] == 0):
                circuit.h(0)
            else:
                circuit.x(0)
                circuit.h(0)
        circuits.append(circuit)
    return circuits

```

Step 2.2:

Afterwards, decoy particles are inserted into S'_A and C_A . The list of decoy particles must be a string of bits before being encoded. Here, we run into a complication. Since, $length\ h_A \neq length\ K_A$, there must be a list of decoys for each variable. The size of the string of decoy bits is related to the size of the list they will be inserted in, since the security against eavesdropping increases as the number of encoded qubits increases. In sum, the list of decoy particles is different for both S'_A and C_A .

Next, Alice randomly decides where to place the decoy particles into the lists, however, she must keep the positions and measurement bases of the decoys.

Decoy particles are then encoded and placed randomly into S'_A and C_A , giving,

$$D_{A1} \text{ into } C_A \rightarrow C'_A$$

$$D_{A2} \text{ into } S'_A \rightarrow S''_A$$

Where C'_A and S''_A contain both their respective decoy and non-decoy particles, and the position of the decoys will be called, respectively, $D_{C'_A}^{pos.}$ and $D_{S''_A}^{pos.}$. The measurement bases of the decoys will be called $D_{C'_A}^{bi.}$ and $D_{S''_A}^{bi.}$.

Listing 7: Decoy Particles are Created and Inserted into Corresponding Lists (with methods attached)

```

"""now for the decoys"""
len_dS = int(len(S_) * securityProperty)
len_dC = int(len(C) * securityProperty)

self.bi_dS = Contributor.cbits(len_dS)
self.ba_dS = Contributor.cbits(len_dS)

self.bi_dC = Contributor.cbits(len_dC)
self.ba_dC = Contributor.cbits(len_dC)

sentdS = Contributor.send(len_dS, self.bi_dS, self.ba_dS)
sentdC = Contributor.send(len_dC, self.bi_dC, self.ba_dC)

dS = sentdS.makeSendCircuits()
dC = sentdC.makeSendCircuits()
"""now we have decoys, time to input them into lists, remembering positions
first get positions"""
self.pos_dS = Contributor.getDecoyPos(len_dS, len(S_))
self.pos_dC = Contributor.getDecoyPos(len_dC, len(C))
"""now, insert"""
self.S_ = Contributor.insertDecoys(dS, S_, self.pos_dS)
self.C_ = Contributor.insertDecoys(dC, C, self.pos_dC)

```

```

def insertDecoys(dls, ls, pos):
    x = ls[:]
    for i in range(len(ls)):
        for j in range(len(pos)):
            if pos[j] == i:
                x[i:i] = [dls[j]]
    return x

def getDecoyPos(len_dls, len_ls):
    pos = random.sample(range(len_ls), len_dls)
    pos.sort()
    return pos

```

Public Discussion Stage

Step 3.1:

Alice and Bob recognize each other in a public channel owned by Eve (i.e. the server), who is a potential eavesdropper.

Step 3.2

For reclarification, what we discuss Alice doing, Bob does as well, in parallel.

In the public discussion phase, both Alice and Bob announce their lists of particles and the positions of the decoys, with measurement bases. Hence, Alice and Bob give each other two lists of encoded qubits each (Alice gives C'_A and S''_A , and Bob gives C'_B and S''_B) with the positions and measurement bases of the decoy particles, across a public channel.

So, continuing along with Alice's point of view, Alice receives two lists of qubits, C'_B and S''_B , along with $D_{C'_B}^{pos.}$, $D_{S''_B}^{pos.}$, $D_{C'_B}^{bi.}$ and $D_{S''_B}^{bi.}$.

Step 3.3:

Alice then measures the decoy particles since she has their position and measurement base.

Listing 8: The ReceiveData1 method (relevant methods attached below)

```
def ReceiveData1(json_strS_, json_strC_, strpos_dS, strpos_dC, strba_dS, strba_dC, backend):
#returns decoy bits + lists without decoys
    user = Contributor(None, backend, None, None, None)
    strS_ = json.loads(json_strS_)
    strC_ = json.loads(json_strC_)
    S_ = []
    C_ = []

    for i in range(len(strS_)):
        S_.append(QuantumCircuit().from_qasm_str(strS_[i]))

    for i in range(len(strC_)):
        C_.append(QuantumCircuit().from_qasm_str(strC_[i]))

    pos_dS = json.loads(strpos_dS)
    pos_dC = json.loads(strpos_dC)
    ba_dS = strToLs(strba_dS)
    ba_dC = strToLs(strba_dC)

    dS = user.receiveDecoy(S_, pos_dS, ba_dS)
    dC = user.receiveDecoy(C_, pos_dC, ba_dC)

    S_ = user.removeDecoy(S_, pos_dS)
    C_ = user.removeDecoy(C_, pos_dC)

    strS_ = []
    strC_ = []
    for i in range(len(S_)):
        strS_.append(S_[i].qasm())
    for i in range(len(C_)):
        strC_.append(C_[i].qasm())

    json_S_ = json.dumps(strS_)
    json_C_ = json.dumps(strC_)

    return str(dS), str(dC), json_S_, json_C_
```

```

def receiveDecoy(self, ls, pos, ba):
    """returns decoy measurements"""
    len_ls = len(ls)
    decoys = Contributor.findDecoy(ls, pos)
    recCircuits = Contributor.createRecCircuits(len_ls, ba)
    return self.measure(len_ls, decoys, recCircuits)

def removeDecoy(self, ls, pos):
    """returns list without decoys"""
    new_ls = []
    for i in range(len(ls)):
        if i in pos:
            pass
        else:
            new_ls.append(ls[i])
    return new_ls

```

Explanation: Input is S__ and C_ along with decoy positions and bases. Output is the measured decoy bits along with the original list with decoys having been removed. Note: json is used to transfer objects as it not possible to properly transfer objects from python, to java, and back to python otherwise.

Step 3.4

Alice then announces her results in the public channel.

Step 3.5:

Bob checks to see if Alice's results are equal to the initial list of decoy bits, if they are not equal to an agreeable error rate between the two users, then there has been an eavesdropping as the measurements were inaccurate. If this is the case, abort and start over. If this is not the case, continue. This is the identical and parallel process for Bob. Hence, the decoy particles are what allow for the security property.

[done with securityCheck(data: String[]) which receives data from ReceiveData1 (specifically, the measured decoy bits)

Also, it must be noted that for this project, no error rate was tolerated i.e. the two lists of decoy measurements must be identical]

Key Generation Stage

Step 4.1:

Now that Alice and Bob have determined that there has been no eavesdropping (they cannot know for certain, however) they move on to the generation of a shared encryption key.

They begin by announcing the measurement bases of the remaining particles, meaning the non-decoys, hence, Alice announces the measurement bases of S'_A and C_A , and Bob announces the measurement bases of S'_B and C_B . They then reveal the original order of S' , which is S.

Step 4.2:

Continuing in Alice's point of view, she has the encoded qubits of S_B and C_B . She must then measure them, and the following equations occur,

$$S_B \rightarrow K'_B$$

$$C_B \rightarrow h_B$$

Where, K'_B is Bob's original key of bits but is denoted as prime because this is Alice's version of it, and h_B is the hashed K_B , though remember that h_B is still in binary form. So, convert it back, $h_B \rightarrow h_B$.

Step 5:

Alice then, to reaffirm that there has been no eavesdropping, or that there has been no sort of error, performs the hash function,

$$h'_B = H(K'_B)$$

Where, h'_B is the hash of Alice's obtained K'_B . This is where the properties of a hash come into play. A hash, having uniformity and characteristic distinction, means that if the hash of both K_B 's are not equal, meaning if $h_B \neq h'_B$, then there has been some error or eavesdropping, i.e. something has gone wrong. If nothing has gone wrong, then, most certainly, $K'_B = K_B$, and so the process, and therefore the key, is reliable. The process is identical for Bob.

Step 6:

Thus, Alice can now form a key (Bob as well, from his point of view which is identical to Alice's. Note: An XOR functions without considering order; implies as long as both keys are identical, there will be no error),

$$K' = K_A \text{ XOR } K'_B$$

Listing 9: The makeKey function [i.e., ReceiveData2] and the securityCheck2 function

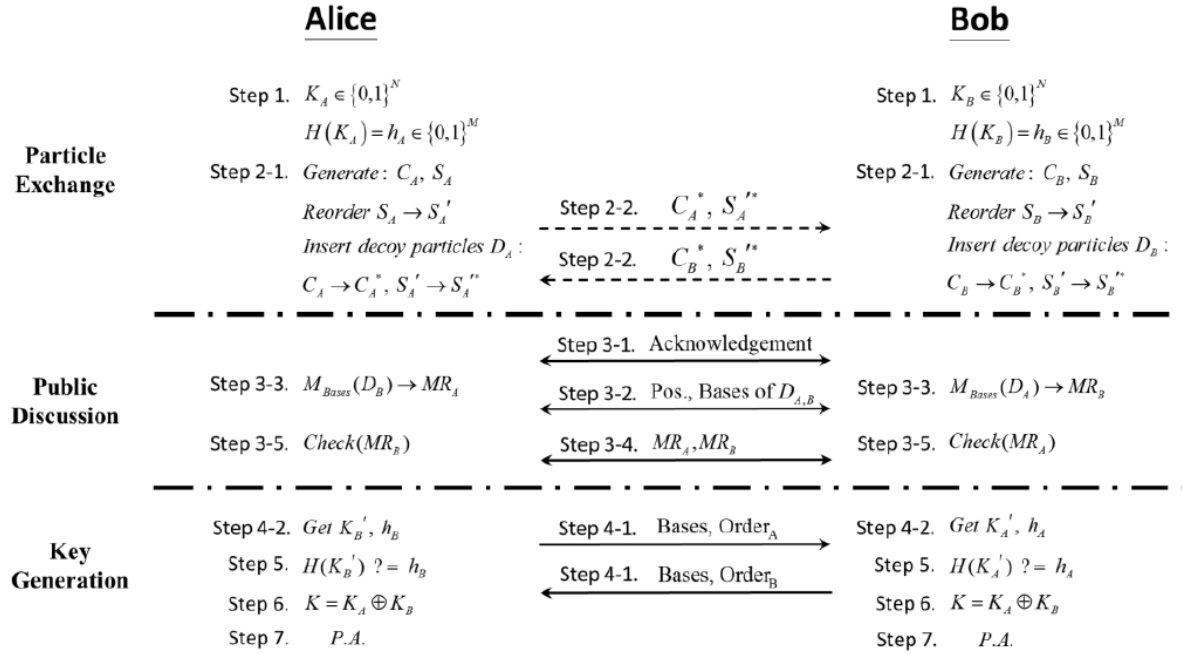
```
#ReceiveData2 ~ MakeKey
def makeKey(ownKkey, jS_, jC, seed, bS, bH, backend):
    ownKkey = int(ownKkey)
    sS_ = json.loads(jS_)
    sC = json.loads(jC)
    S_ = []
    C = []
    for i in range(len(sS_)):
        S_.append(QuantumCircuit().from_qasm_str(sS_[i]))
    for i in range(len(sC)):
        C.append(QuantumCircuit().from_qasm_str(sC[i]))
    seed = int(seed)
    bS = json.loads(bS)
    bH = json.loads(bH)
    user = Contributor(None, backend, None, None, ownKkey)
    S = user.measure2(S_, bS)
    K = (''.join(map(str, user.unshuffle(S, seed))))
    securityCheck2(Contributor.decimalToBinary(hash(K)),(''.join(map(str, user.measure2(C, bH)))))
    K=int(K)
    print(K^user.Kkey)

def securityCheck2(H_,H):
    if H_ == H:
        pass
    else:
        print('Fails')
```

These two methods perform the entire key generation stage [exception: 4.1].

Figure 7: QKA Protocol (Yu et al.)

Note: Step 7 is omitted from the protocol.



Runtime Efficiency

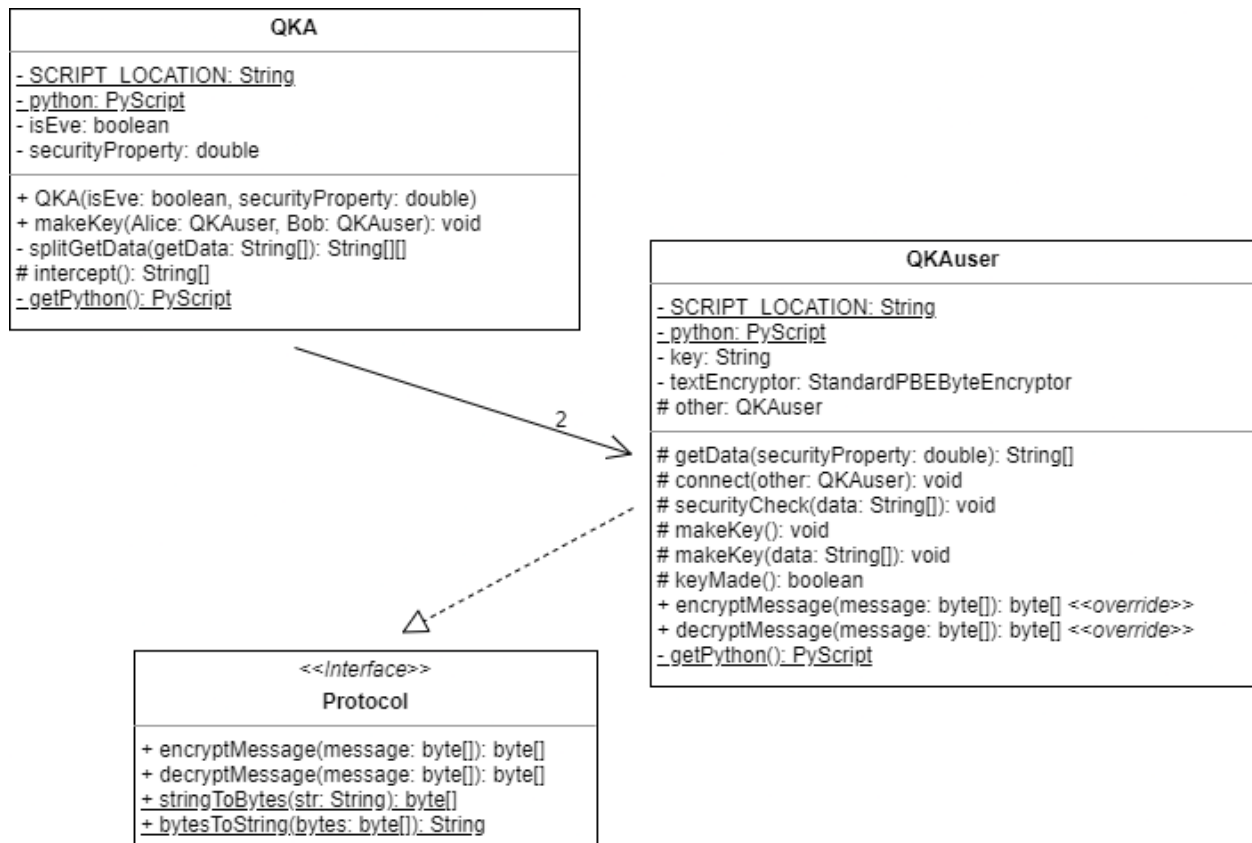
As mentioned before, sending and measuring qubits takes constant time. For the particle exchange stage, this algorithm requires that $n * D_n + h * D_h$ qubits be sent, where n is the number of bits, h is the length of the hash, and D_n and D_h are the ratios of decoy particles sent with them. In big O notation, this simplifies to $O(n * D_n + h * D_h) = O(n)$.

The public discussion stage requires that the measurement bases be sent, which is also $O(n)$. It also requires that the position of the original list be sent, which is also $O(n)$ since each bit has one number sent along with it (its original position).

Given that this algorithm requires many $O(n)$ steps be completed in succession, its runtime efficiency is $O(n)$.

QKA UML Diagrams

Figure 8: UML Diagram of QKA Classes



Methods of importance for class QKAuser:

Note: the QKAuser class simulates a user and is necessary for a realistic scenario of passing information in an actual key agreement.

- `getData(securityProperty: double): String[]` calls `qkaimplementationFINAL.py`'s `GiveData` method to give **all** necessary data for one user. Evidently, both users do this separately.
- `securityCheck(data: String[]): void` calls `qkaimplementationFINAL.py`'s `ReceiveData1` to get the decoy bits. Then, checks to see if measured decoy bits are equal to corresponding user's decoy bits.
- `makeKey(data: String[]): void` calls `qkaimplementationFINAL.py`'s `makeKey` (i.e. `ReceiveData2`) method which makes a key given own key and data necessary to find other user's key. Once the key has been made, initializes `textEncryptor` with the key.
- `keyMade(): boolean` checks to see if a key has been made or not.
- `encryptMessage(message: byte[]): byte[]` overrides `Protocol`'s method. Encrypts a message with the `textEncryptor` field's `.encrypt` method.
- `decryptMessage(message: byte[]): byte[]` overrides `Protocol`'s method. Decrypts a message with the `textEncryptor` field's `.decrypt` method.

Methods of importance for class QKA:

Note: the class simulates a communication channel between QKAusers.

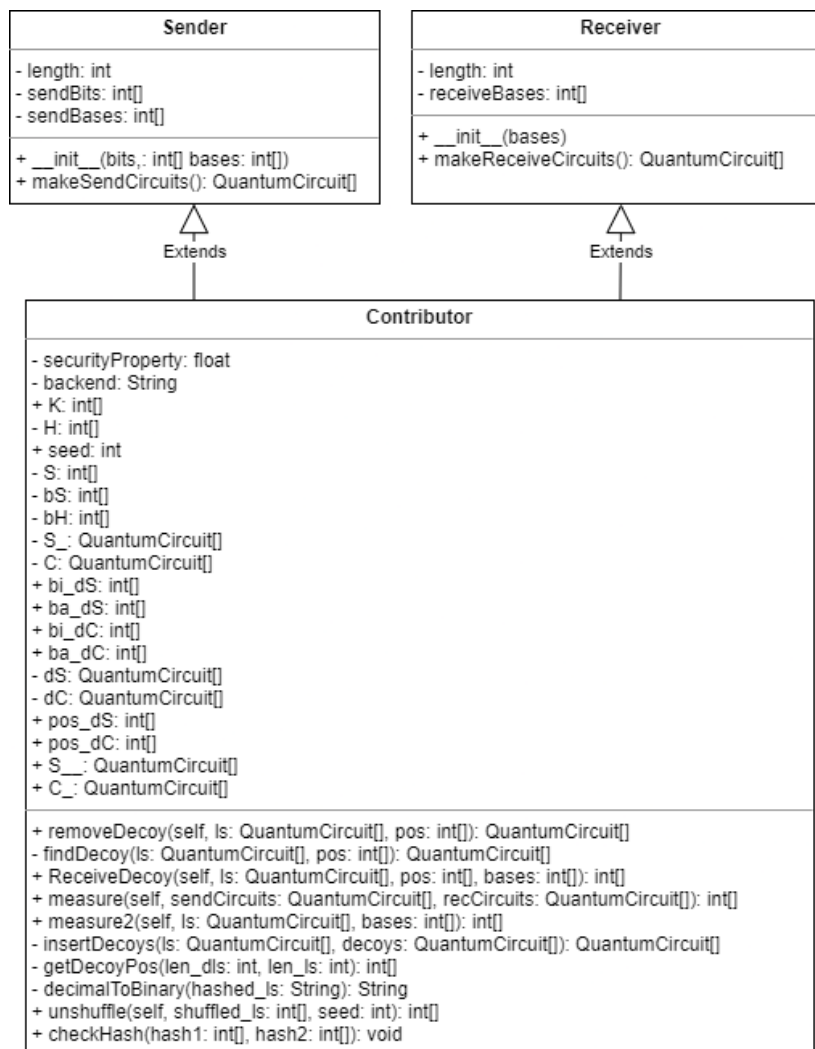
- `splitGetData(String[] data): String[][]` splits output of `GetData` into a `String` array of two usable `String` arrays.
- `makeKey(Alice: QKAuser, Bob: QKAuser): void` commences and performs the quantum key agreement protocol between two users. Calls both user's `GetData`, own `splitGetData` respective to users, `SecurityCheck`, and then both user's `makeKey` to finish.
- `intercept(): String[]` is the implementation of the simple intercept and resend eavesdropping.

The Protocol class was explained previously.

Since the QKA python script has great importance, it will be described as well.

The critical fields methods were included in the following diagram,

Figure 9: qkaImplementationFINAL.py (not including non-class methods as those are listed in the protocol description)



Methods of the classes within `qkaImplementationFINAL.py`,

RemoveDecoy(self, ls: QuantumCircuit[], pos: int[]): QuantumCircuit[] returns list without decoy given a list and the decoy positions.

findDecoy(ls: QuantumCircuit[], pos: int[]): QuantumCircuit[] returns list of decoys found within given list.

ReceiveDecoy(self, ls: QuantumCircuit, pos: int[], bases: int[]): int[] returns decoy measurements using findDecoy and measure methods.

measure(self, sendCircuits: QuantumCircuit[], recCircuits: QuantumCircuit[]): int[] returns measured qubits given a qubit list with gates corresponding to it being “sent” and a qubit list with gates corresponding to a “reception”.

measure2(self, ls: QuantumCircuit[], bases: int[]): int[] returns measured qubits given qubit list and bases.

insertDecoys(ls: QuantumCircuit[], decoys: QuantumCircuit[]): QuantumCircuit[] inserts decoys into list.

getDecoyPos(len_dls: int, len_ls: int): int[] returns a sorted list of decoys’ positions.

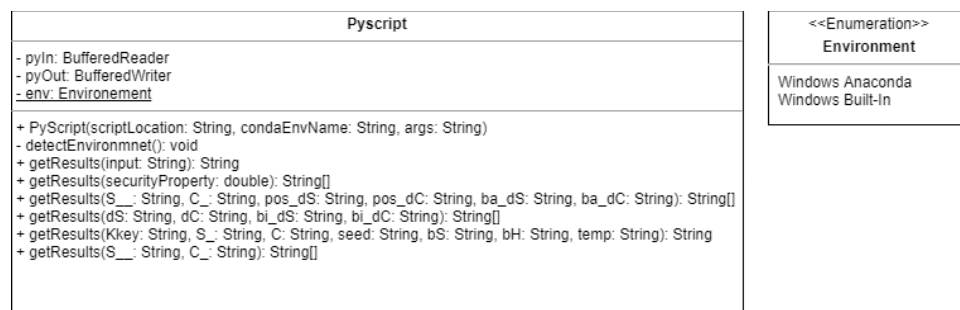
decimalToBinary(hashd_ls: String): String converts a hash into binary string.

unshuffle(self, shuffled_ls: int[], seed: int): int[] will unshuffle a shuffled list with seed.

checkHash(hash1: int[], hash2[]):void is not used in the application but it was important in testing, so it is included here. Is self explanatory.

Python

Figure 10: UML Diagram of Pyscript class



As mentioned in the explanations for the QKD and QKA protocols, this project requires the use of the Python library qiskit, so integration with Python is essential. This was done with the PyScript class.

One key component of this class is its ability to determine how Python is installed: either installed normally (where python can be directly run from the command line with the py command), or installed in an Anaconda environment. This is the detectEnvironment() method, and works as follows. It first attempts to run a piece of Python code which simply imports the necessary libraries (from qiskit import QuantumCircuit, for instance) from normally installed Python. If that script runs without errors, it indicates that in the static env member of PyScript. If that piece of code did not

run successfully, it assumes that Python is installed in an Anaconda environment and indicates that in env. Note that this step is only run the first time a PyScript is created to save time.

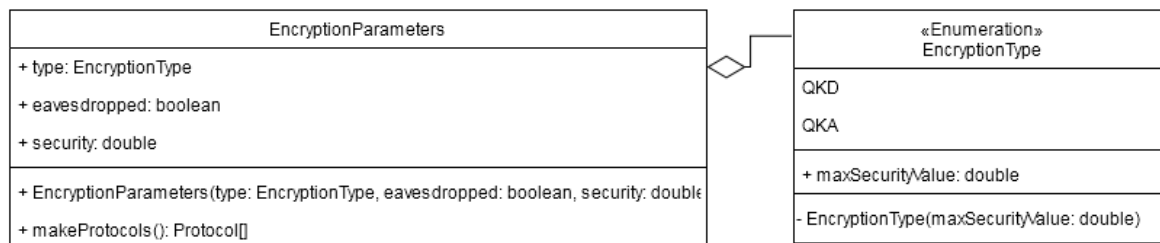
When a PyScript is created, it first checks what the Python environment is (if the check has not been performed, it performs the detection routine described above). Then, it creates a Process from a ProcessBuilder that runs the Python code with the given command line arguments.

Once a PyScript has been instantiated, the user of the class can call one of the implementations of the overloaded `getResults()` methods, as can be seen in the UML diagram above. Each of these methods passes a series of Strings to the standard input channel of the Python script and returns what the script returned on its standard output stream. Thus, the Python code needs to be written in such a way that it will take the input it needs in its standard input stream and output a result that will be understood by the corresponding Java class that called it. This process of serialization and deserialization seems to be what causes the quantum key distribution to be so slow.

Encryption Parameters

We needed a way to create and pass around encryption parameters. That is the role the EncryptionParameters class plays. The UML diagram for it, as well as for the EncryptionType enum it uses, are below.

Figure 11: Encryption Parameters Class



The `EncryptionParameters` class has 3 data members. I wanted to make them publicly available and non-changing, so I made them public final data members of the class. I chose to do this instead of making them private and giving them accessors only to avoid unnecessary code. The members are:

- `Type`: this is the type of key exchange to perform
- `Eavesdropped`: whether or not the key exchange will be eavesdropped on
- `Security`: how secure to make the key exchange
- The `makeProtocols()` method. This makes a pair of `Protocol` instances, in a 2-element array, that use the key exchange specified in the type.

Of note is the security member. Because the QKA and QKD implementations were written to take different maximum values for security, the `EncryptionType` enumeration specifies a `maxSecurityValue` that each key exchange implementation expects. This value is used to set the maximum value of the slider used to configure the security. This is also a public final data member, for the same reason as for `EncryptionParameters`.

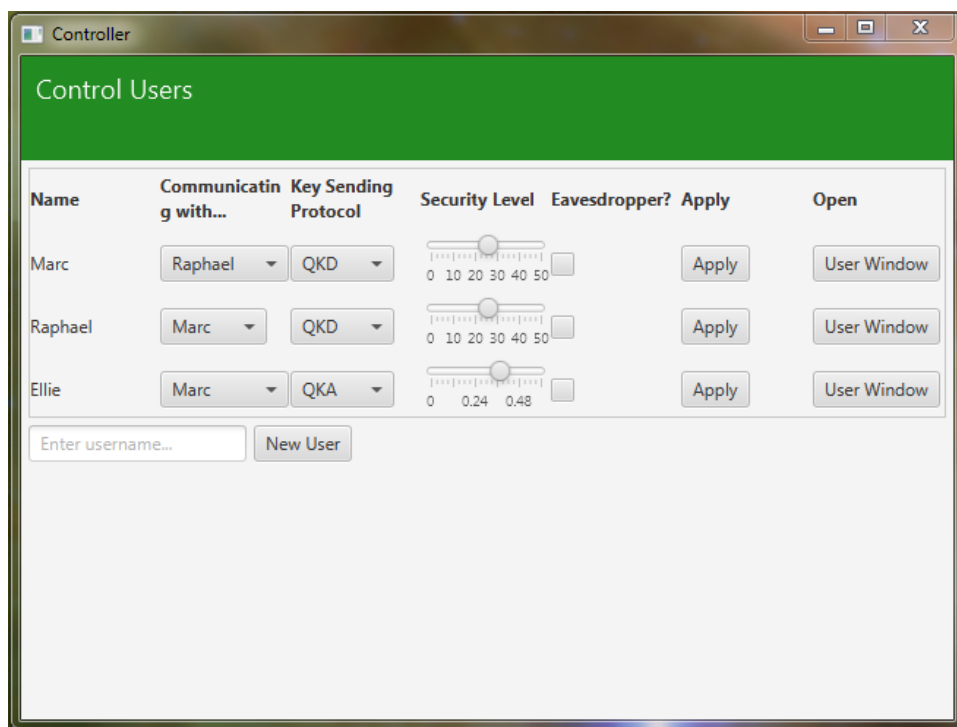
GUI

The overall goal of the GUI was to make it simple to use and yet provide all essential features. To do this, we picked a consistent colour scheme of green and white and laid out the controls as logically as possible.

The initial window is the Chat Control window; it allows the user to create Users which can chat among themselves. The first step is to add the user by typing their name and then clicking “New User”. Then, once at least two users have been made, configuring the key exchange is possible. Next to each user, the dropdown “Communicating with” switches to which user the key exchange should be configured with. The next three controls configure the key exchange (QKD or QKA, as described above, the security level, and presence of eavesdropper). Selecting the security level, as seen below, is done with a slider. Its maximum value changes depending on the protocol because of the different way the classes interpret the security level.

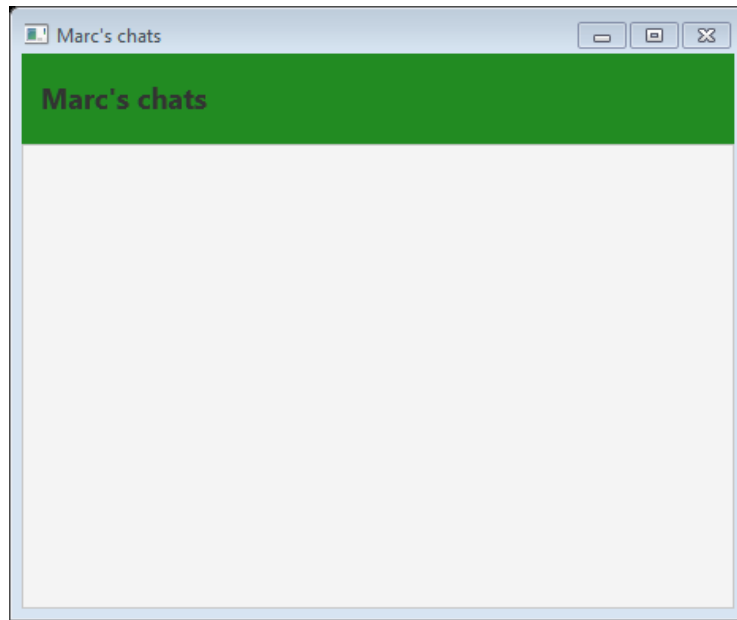
Then, the Apply button applies the changes, so the two users will now communicate with a key generated from the given settings. Finally, the “User Window” buttons open the User windows corresponding to each User if that window was closed.

Figure 12: Control Users Window



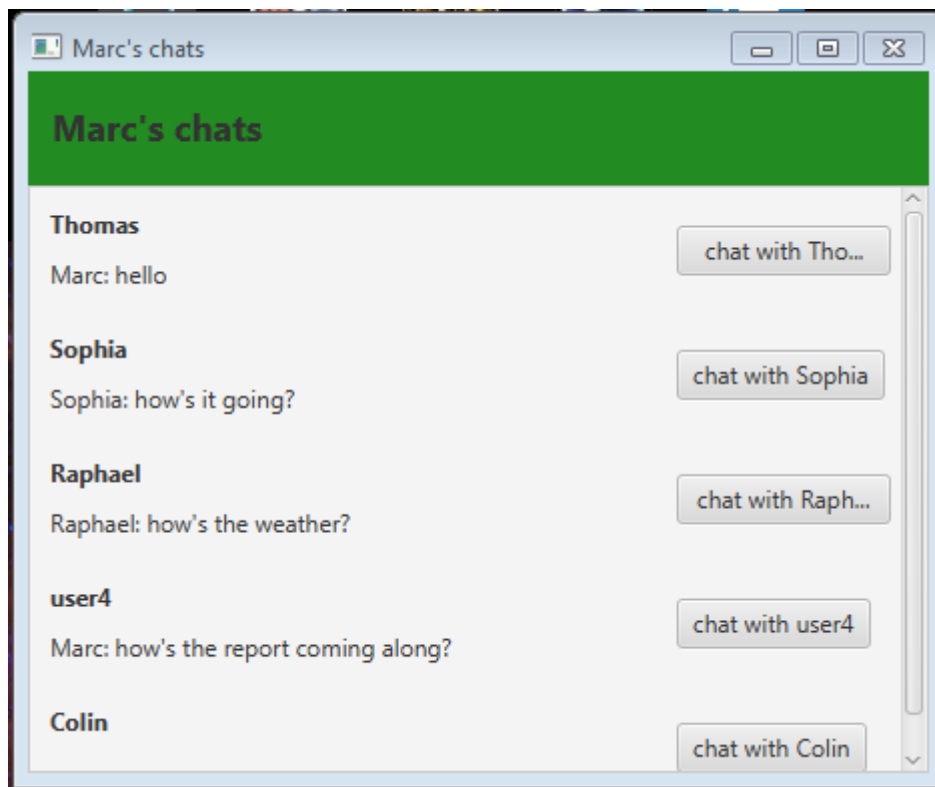
The following is the User Window. It initially opens when a new user is created; subsequently, it can be reopened with the User Window button respective to the user.

Figure 13: User Window Initialized



Evidently, when a user (e.g. Marc) is initialized, no chats have been created. Hence, the window is empty.

Figure 14: User Window with Chats

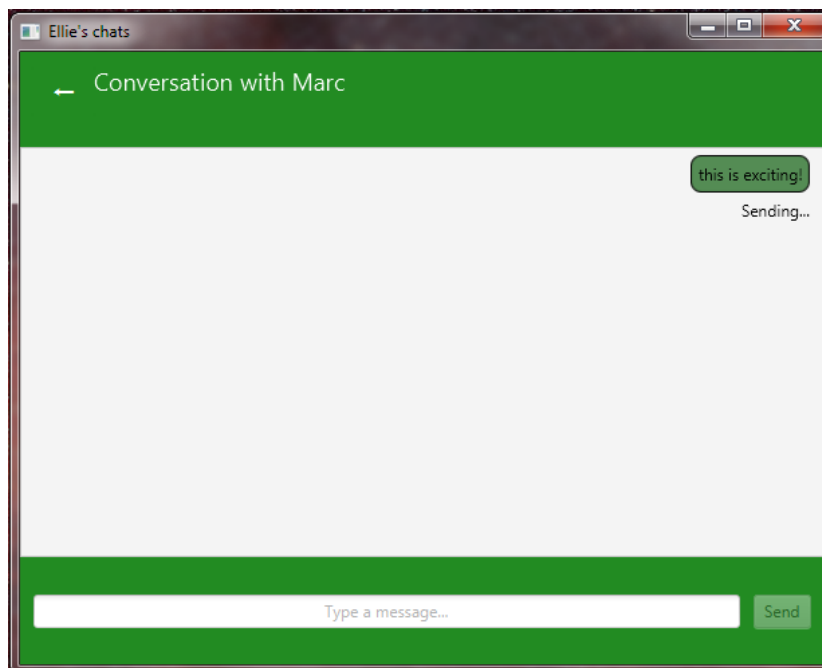


When chats have been created via the Control Users Window, the chats will appear in corresponding User Windows. The chat buttons will open the chat respective to the user. Notice, the

text underneath the bold name is the last message sent in the chat. Notice as well when the chat window has been initialized, there is no last message, visible with user Colin.

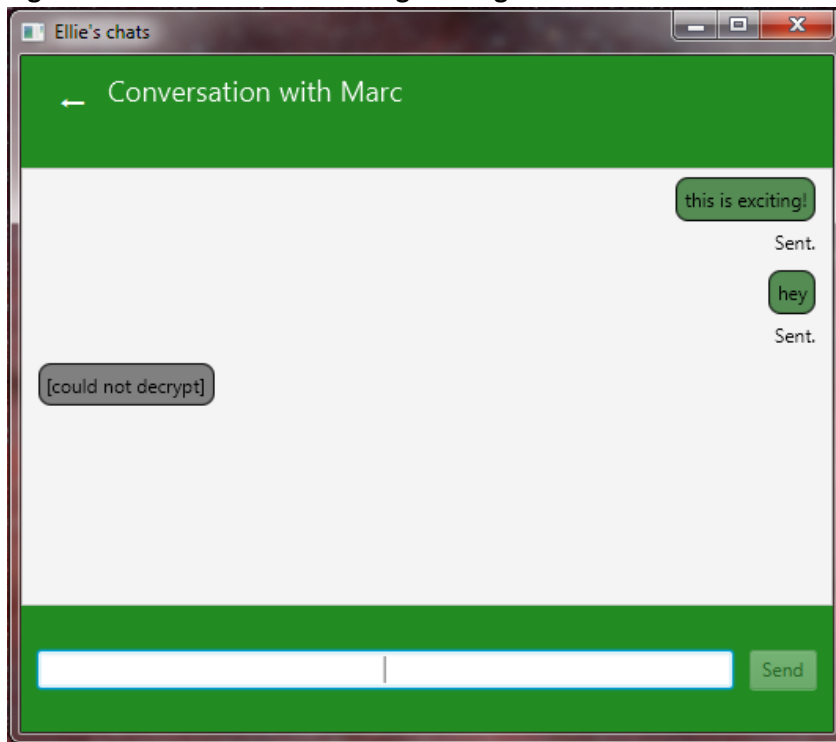
The following is the chat window. When a message is typed on the bottom and sent by pressing the “Send” button (alternatively, pressing the enter key), it is sent to the other user. Until it has been sent, text below the chat bubble says “Sending...”. The most significant time delay happens when the first message is sent when the key exchange protocols have been changed, since the key exchange takes some time.

Figure 15: Chat Window with a Message in Transit



The following screenshot shows that when a message has been successfully encrypted and sent, a label below it says “Sent.”. In addition, it shows what happens when a message is successfully received but could not be decrypted. That happens when the keys generated by Alice and Bob are different, which is due to eavesdropping in the key exchange (meaning Bob’s measurements were different to Alice’s bits in some places) but with such a low security property that it was not detected.

Figure 16: Chat Window Showing Messages that Sent and that Failed to Decrypt



Methods of Evaluation

Key Exchange

Test classes were made for each of the key exchange protocols, and for the encryption library we were using.

For QKD, there are two classes that test it. The first is QKD2test. It creates instances of QKDAlice2, QKDBob2, and QKDChannel, and tests under three different scenarios: with no eavesdropping, with eavesdropping and standard security check, with eavesdropper and limited security check, and with eavesdropper and no security check. It allowed me to verify that the key exchange was happening, and that the eavesdropper was getting caught when we expected.

The second test class is Qkd2PythonTest. It tests the Python code that runs the simulation of the quantum key exchange. For instance, it verifies that when Alice prepares a bit in the same base that Bob measures it in, she measures it the same way as Alice. This allowed me to catch a bug with the eavesdropper, which was not correctly measuring the qubits in the X basis; after their measurement, the qubit was put in the Z basis, instead of being returned to the X basis.

There is also a test class for QKA, QKAtest. It creates instances of QKA and QKAuser to test that it performs as expected under different conditions (eavesdropping or not, different security levels).

GUI

We evaluated the GUI with manual testing. On one hand this involved creating many users with different security configurations and ensuring that all parts of the GUI, and encryption, behaved as expected. It also involved more specific testing, like resizing the windows to ensure that their contents

would resize appropriately and attempting to create multiple users with the same name to verify it wouldn't allow that.

Results: System Quality

Overall, we accomplished most of our objectives. The specific goals are discussed below.

Accurate Implementation

The implementations are accurate, to the best of our knowledge, as discussed in the methods of evaluation section. We can also be fairly sure since we used a quantum circuit simulator that is widely known and used. In addition, by using an actual quantum computer library, this code may be helpful to others who wish to implement similar protocols.

Use the Protocols to Encrypt and Decrypt Messages

The keys generated are used to encrypt and decrypt the messages, since they the generated keys are passed as passwords to encryptor classes in Jasypt.

Efficient Implementation

The key exchange implementation is not very fast, taking around 20 to 30 seconds to do one key exchange, and this only gets worse when QKD makes repeated attempts after a failure. By doing some quick performance testing, it was determined that the decoding of the QASM strings was the main bottleneck in this code. That functionality was delegated to the built-in functions in qiskit, so I was worried it would be hard to solve. I (Marc) wrote a small function that could parse a subset of QASM, the portion we use to encode the qubits. This made the key exchange for QKD take a few seconds at most. However, given that there was little time to test and add this to the QKA implementation, I decided to leave it out so the implementations would effectively have a level playing field. This is one shortcoming of our implementation.

Viable Demo of Messaging Application

As we have shown in the design section, the GUI allows any individual user to chat with any other user, so long as encryption settings have been determined between them. It also has many visual indicators: the grey text boxes for messages that could not be decrypted, the indicator at the bottom of each message bubble to indicate if or when a message has been successfully sent.

However, this application is still just a demo. Of course, this was necessary since we do not have the ability to connect this code to a real quantum network. However, some features that would have made it more full-featured are absent, like group chat support.

Project Management: Timeline

Task	Planned Date	Assigned Person	Actual Date	Notes
Features	8 Feb 2021	Marc and Raphael		These tasks were all completed for the initial report, so the original date was the date they were completed.
QKD Algorithm Report Explanation	10 Feb 2021	Marc		
QKA Algorithm Report Explanation	10 Feb 2021	Raphael		
GUI Design	15 Feb 2021	Marc		
UML Design	17 Feb 2021	Raphael	20 Feb 2021	
Finish implementation of QKD (with security and eavesdropper simulation)	28 Feb 2021	Marc	11 Mar 2021	
Finish implementation of QKA (with security and eavesdropper simulation)	6 Mar 2021	Raphael	19 Apr 2021	
Write Java code to interface with Qiskit (QKD and QKA classes)	13 Mar 2021	Marc and Raphael	29 Mar 2021(QKD) 19 Apr 2021 (QKA)	
Implement chat window with encryption	27 Mar 2021	Marc and Raphael	17 Mar 2021	This works with an old and simpler implementation of QKD, which is why it was finished before QKD was.
Implement chat control window and chat view windows	10 Apr 2021	Marc and Raphael	7 Apr 2021	
Implement user window (this marks the completion of the GUI)	21 Apr 2021	Marc and Raphael	21 Apr 2021	
Collect data, screenshots, and make new UML diagrams for final report.	24 Apr 2021	Marc and Raphael	8 May 2021	
Write final report	30 Apr 2021	Marc and Raphael	14 May 2021	

Conclusion

The project was a success. The application is as designed, and all designated tasks were completed. Further, it has been tested and it functions. Building this application was difficult, as it took knowledge not taught in our courses; specifically, quantum computing (taught by guest lecturers), Qiskit and Python. This provided many learning experiences. We are now comfortable in both Python and Java, whilst our peers are presumably only comfortable in Java. The tasks themselves were challenging as well (e.g. implementing a complex protocol such as the QKA). Thus, it should be said the quality of the tasks was remarkable, all considered. Further, the quantity was excellent, as we are only two students working on a large project whilst groups of three and four exist.

Annex: System Manual

Notes

Given the way this project runs Python code, it only works on Windows.

Setting it up

Python

The first dependency is Python, at least version 3.6 is required. While this project works if Python and qiskit are installed within an Anaconda environment, installing Python the more standard way is simpler, so that is what is shown here. Python can be downloaded here:

<https://www.python.org/downloads/>. To test that it is installed correctly, from a command prompt (search cmd in the start menu), and run this command:

```
py
```

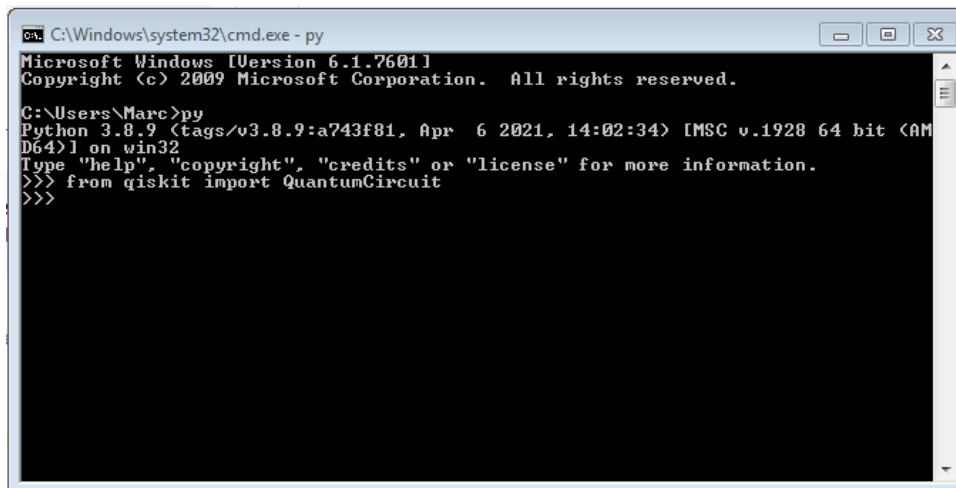
If Python is successfully installed, that will start a Python shell, with >>> at the last line. Type “quit()” (without the quote marks) and press enter to exit this. Then, the Qiskit library must be installed. To do so, type:

```
py -m pip install qiskit
```

in a command prompt as before. You should see a message that it was successfully installed. To test this, open a python shell as before (open a prompt, type “py” then press enter), then type the following line:

```
from qiskit import QuantumCircuit
```

If, after a few seconds it returns you to the >>> prompt, these dependencies have successfully been installed, as can be seen in the following image:



```
C:\Windows\system32\cmd.exe - py
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Marc>py
Python 3.8.9 (tags/v3.8.9:a743f81, Apr  6 2021, 14:02:34) [MSC v.1928 64 bit <AMD64>] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> from qiskit import QuantumCircuit
>>>
```

If not, there may be an issue with the way qiskit was installed. Searching the error may help, but otherwise there are instructions at qiskit’s website: <https://qiskit.org/> or https://qiskit.org/documentation/getting_started.html. Note: if you decide to use an anaconda

environment, to install Python, ensure that it is named QiskitEngine, that you install Qiskit within it, and that it is possible to use it from cmd (i.e. it is possible to run “conda activate QiskitEngine” within cmd).

Java

A JDK of version 11-14 is required to run this project. JDK versions 15 and 16 have worked, but they have sometimes caused issues. Oracle provides installers here: <https://www.oracle.com/java/technologies/javase-jdk11-downloads.html> (others, like AdoptOpenJDK, are available).

There are two ways to compile and run the project: from within Netbeans (the simpler method), or by compiling it with Gradle and running it from there. This section describes those methods.

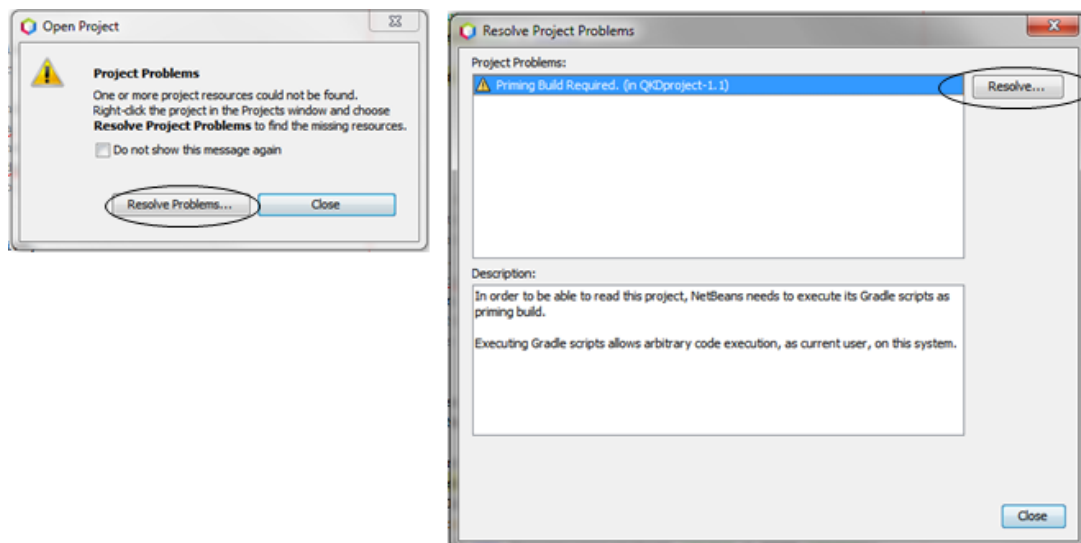
Regardless of method, the code needs to be downloaded. Use the zip that was submitted for this project on Léa or download from the GitHub repository (<https://github.com/vexandmore/QKDproject> no login required) for a more recent version: this report deals with version **1.1**. To download that version in particular download the source zip at <https://github.com/vexandmore/QKDproject/releases/tag/1.1>. Later versions may work differently. Version 1.1 added a better GUI to pick the security property and has the extra files so it can be compiled and run without Netbeans. If the version submitted April 26th is used, it **must** be run within Netbeans.

The first step is to unzip this file (more specifically, the javaFX_Practice folder), in some location you know.

Compiling and Running

Within Netbeans

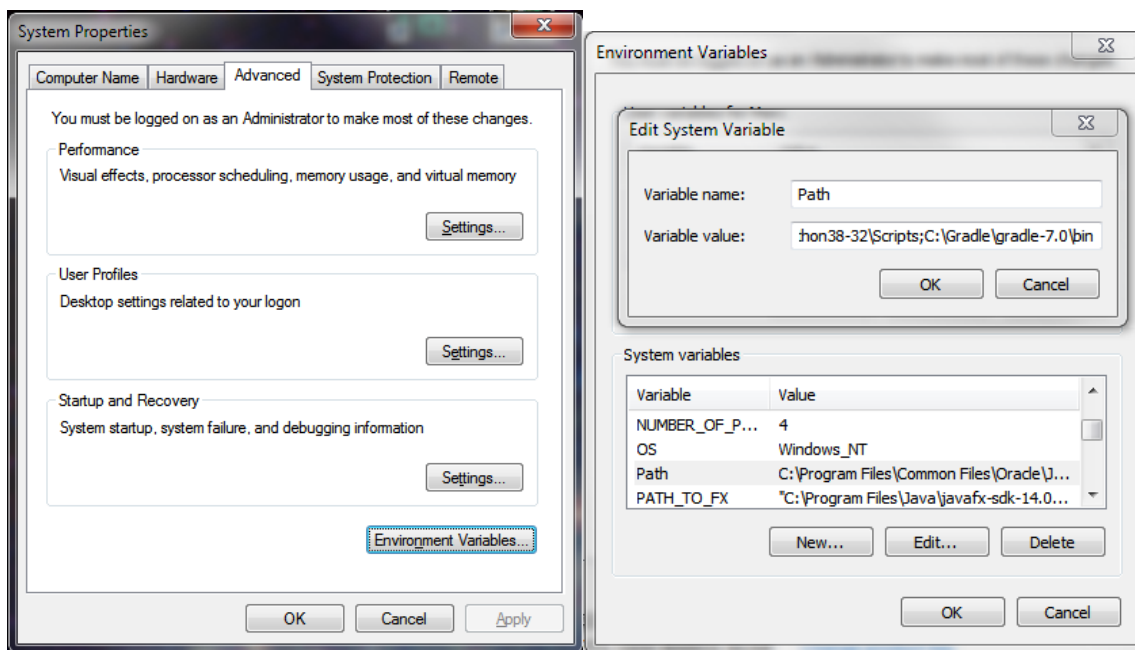
Open the project in Netbeans. It will say that there is a problem with the project. Click on “Resolve Problem” and “Resolve”, as shown in the screenshots, and it will run the priming build and fix the error.



From here, it should be possible to click on the “Run” button and the project will run. If there is an error regarding class version, try changing the Java platform of the project to a JDK in the range of 11-14.

From Gradle

Install the latest version of Gradle. On Windows, it is necessary to download and unzip the binary distribution. It is available here: <https://gradle.org/releases/>; select the binary-only download. Open the zip folder and copy the gradle-7.0 folder into a new folder, like C:\Gradle. Then, it is necessary to add it to the PATH system variable. First, right-click on “Computer” or “This PC”, then click “Properties”. In the window that opens, click on “Advanced System Settings”, then on “Environment Variables”, as can be seen below. Then, scroll to the “PATH” variable in “System Variables”, click “Edit”, and add a semicolon followed by this new path:



Then, close and reopen cmd and navigate to the project folder. To do so, verify where the project folder is, then run the “cd” command on it, like in this example:

```
C:\Users\Marc>cd "Documents\Other Code\Projects\javaFX_Practice"  
C:\Users\Marc\Documents\Other Code\Projects\javaFX_Practice>
```

From here, type “gradle distZip”. If there is an error about Gradle not being a valid command, verify that it has correctly been added to the PATH, as seen above.

```
C:\Windows\system32\cmd.exe

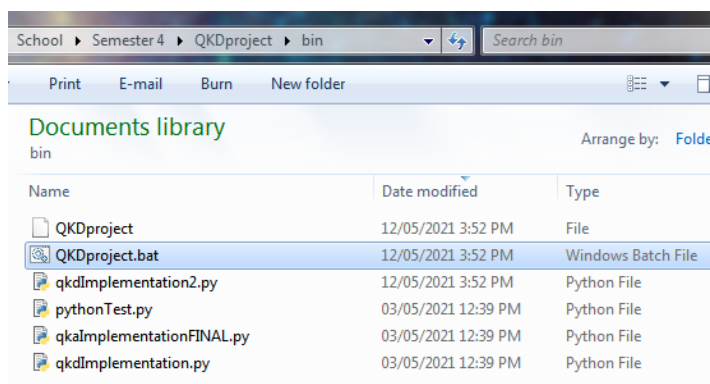
C:\Users\Marc>cd C:\Users\Marc\Documents\Other Code\Projects\javaFX_Practice
C:\Users\Marc\Documents\Other Code\Projects\javaFX_Practice>gradle distZip

> Configure project :
Project : => 'QKDproject' Java module

BUILD SUCCESSFUL in 1s
5 actionable tasks: 5 up-to-date
C:\Users\Marc\Documents\Other Code\Projects\javaFX_Practice>
```

Then, return to the project folder, and open the “build” folder, then “distributions”. There will be the “QKDproject.zip” file. This is the compiled project. In order to run it, it needs to be decompressed and a couple of files must be moved manually.

Open the file and paste its contents (a folder named “QKDproject”) into another folder. Then, open the “QKDproject” folder, copy all of the files ending in .py (“pythonTest.py”, etc.) and paste them in the “bin” folder. Finally, open the “bin” folder, and double-click “QKDproject.bat”. The control users window should open along with a command prompt.



User Documentation

User Guide: Description of Interaction with Application

First, follow the steps in the System Manual, then:

The opening page of the application is the Control Users Window.

Controller

Control Users

Name	Communicatin g with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
<input type="text"/>						

New User

To use the application, user names must be inputted. This is done by typing in the name into designated text field.

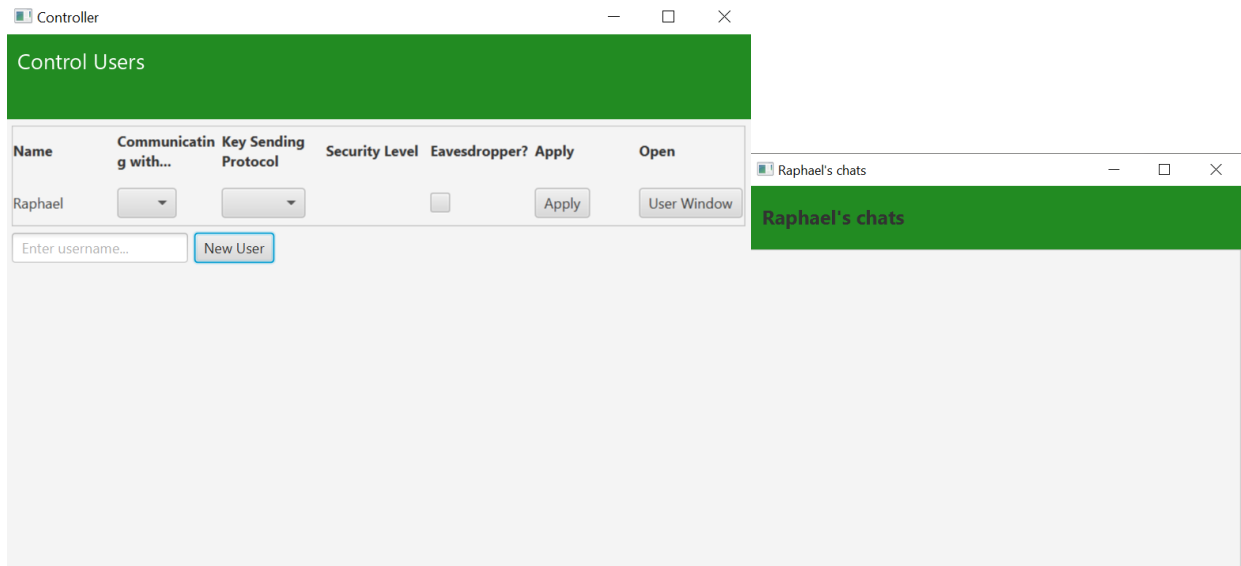
Controller

Control Users

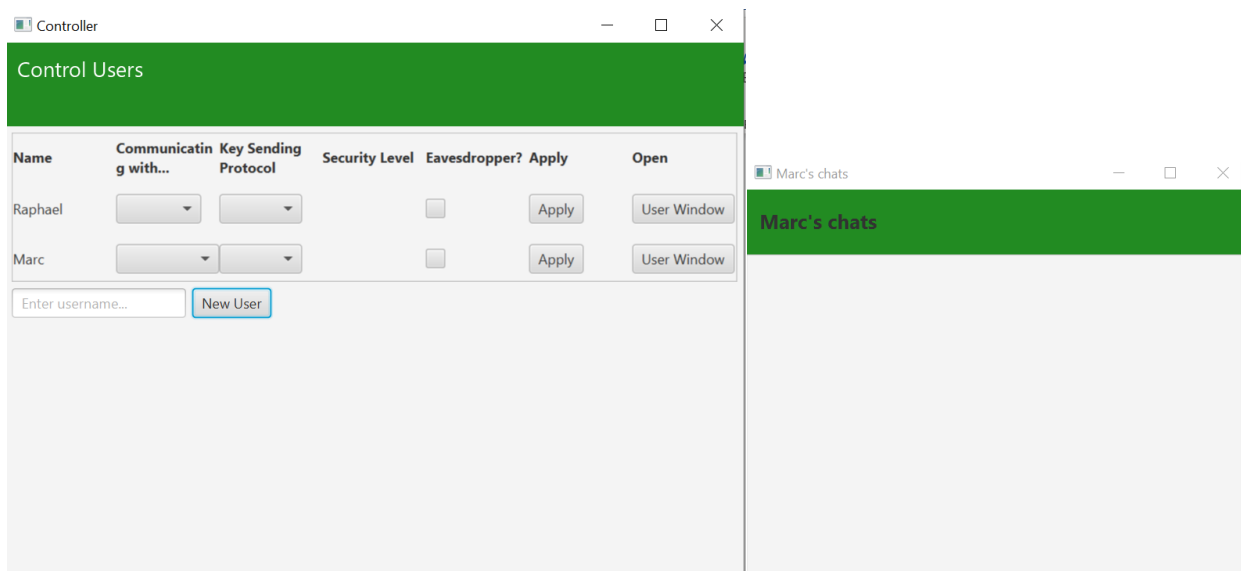
Name	Communicatin g with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
<input type="text" value="Raphael"/>						

New User

Press the enter key to input the typed name or click the “New User” Button. This will input the new user and open the corresponding user window.



Do the same and input another user.



It must be said that these users' windows can be closed and opened at any point in time. Just press the "open window" button corresponding to the window previously closed, and the same window will reappear.

Now, it is necessary to designate which user another will be chatting with. To do this, select from the dropdown menu under the bold text “Communicating with...” the user desired.

Name	Communicating with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
Raphael	Marc			<input type="checkbox"/>	Apply	User Window
Marc	Marc			<input type="checkbox"/>	Apply	User Window

Enter username... New User

Next, choose the form of encryption is to be used. To do this, select from the dropdown menu under the bold text “Key Sending Protocol” the desired protocol.

Name	Communicating with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
Raphael	Marc	QKA	0 0.24 0.48	<input type="checkbox"/>	Apply	User Window
Marc		QKA		<input type="checkbox"/>	Apply	User Window

Enter username... New User

Now, the security level can be adjusted. This determines how “secure” the key sending protocol is, as explained previously. This is done by moving the slider under the bold text “Security Level”. In the

following image, it has been set to maximum security.

Name	Communicating with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
Raphael	Marc	QKA	0 0.24 0.48	<input type="checkbox"/>	Apply	User Window
Marc				<input type="checkbox"/>	Apply	User Window

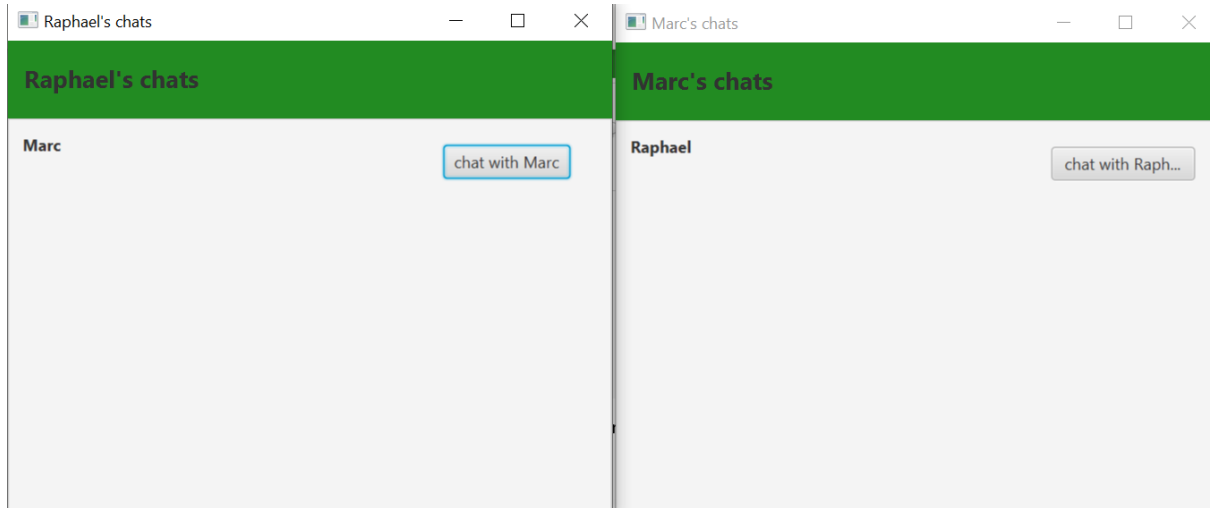
Enter username... New User

Next, under the bolt text “Eavesdropper?”, decide on whether to test the security of the protocol by including an eavesdropper, or choose to simply message between the two users. This is done by, respectively, checking the little box, or not. In the following image, eavesdropping has been set to on.

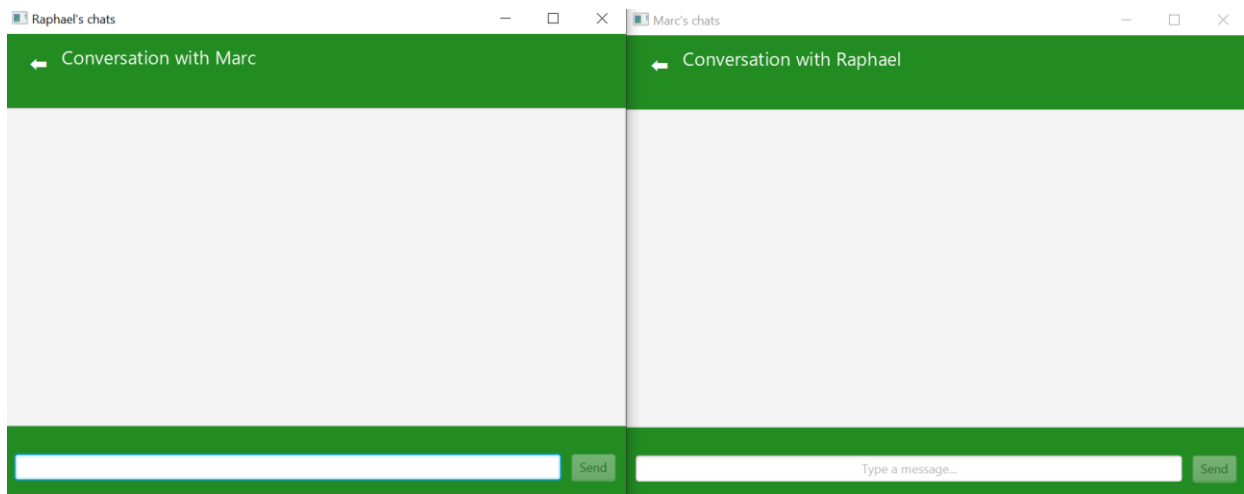
Name	Communicating with...	Key Sending Protocol	Security Level	Eavesdropper?	Apply	Open
Raphael	Marc	QKA	0 0.24 0.48	<input checked="" type="checkbox"/>	Apply	User Window
Marc				<input type="checkbox"/>	Apply	User Window

Enter username... New User

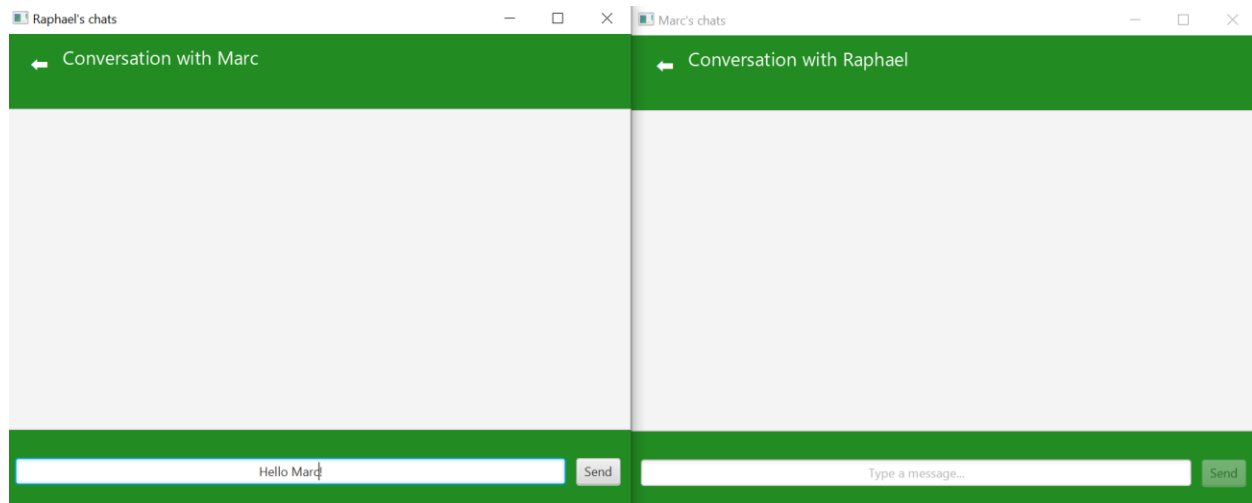
Here, we click the “apply” button to create the chats. What this will do is create options in the respective user’s windows to chat with the corresponding user, as shown below.



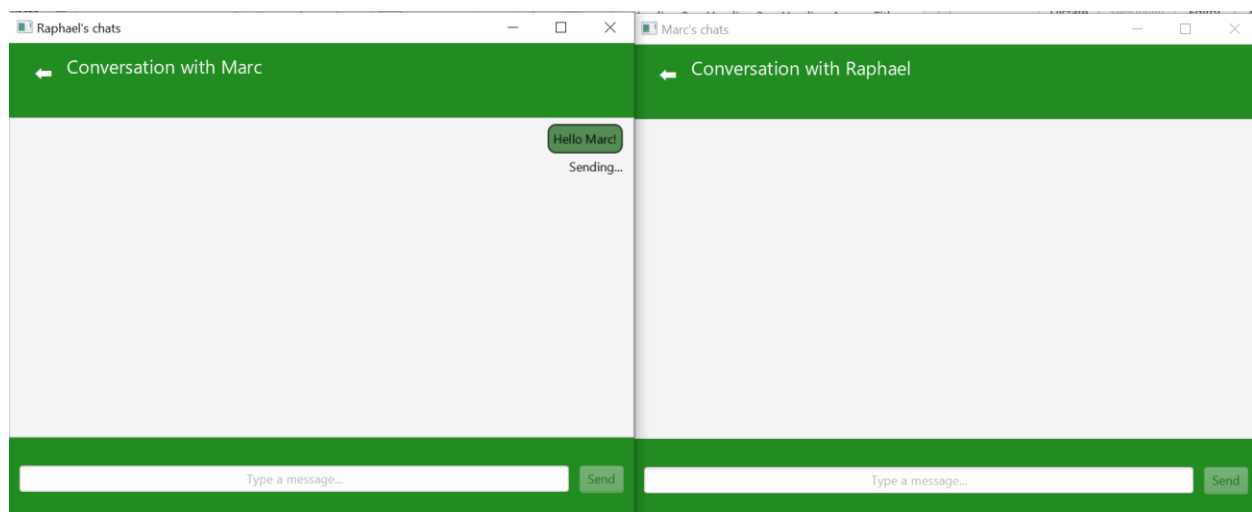
Now, press the “chat with ___” button on the user/s. This will change the window the respective chat window.



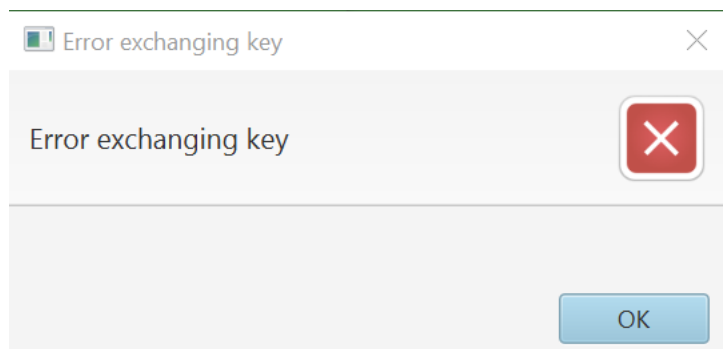
Now, the users can message each other. This is done by typing in the text box as demonstrated.



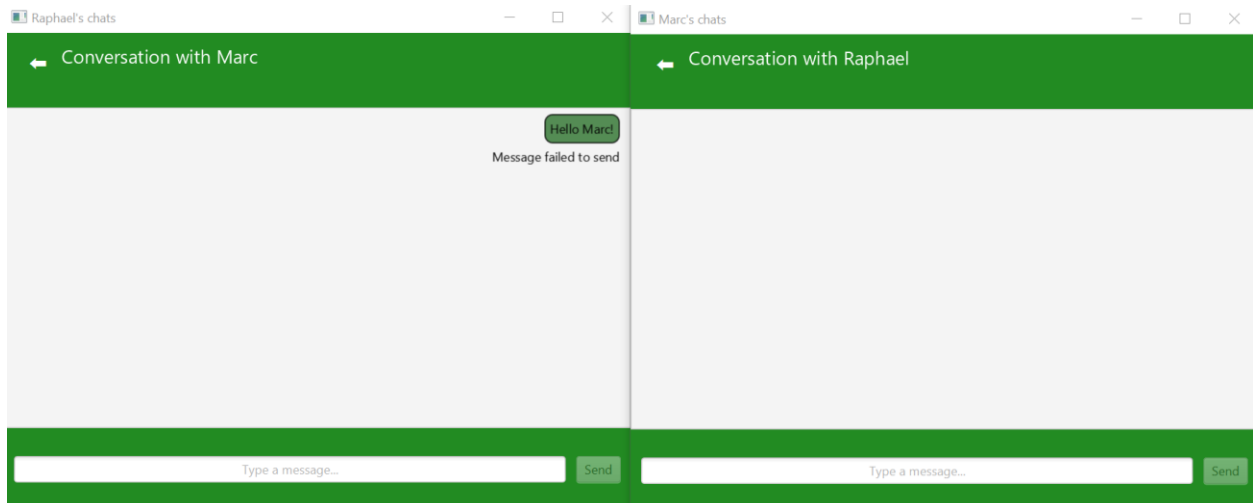
Press the enter key or click send.



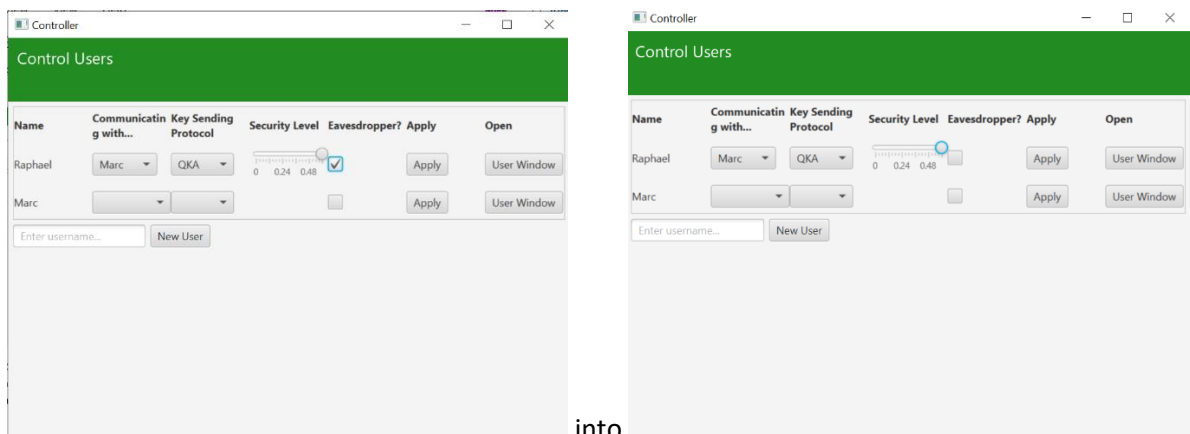
Oh no, there is an error!



This is fine, remember, we selected the presence of an eavesdropper. The algorithm simply detected it. We also observe the message did not send.

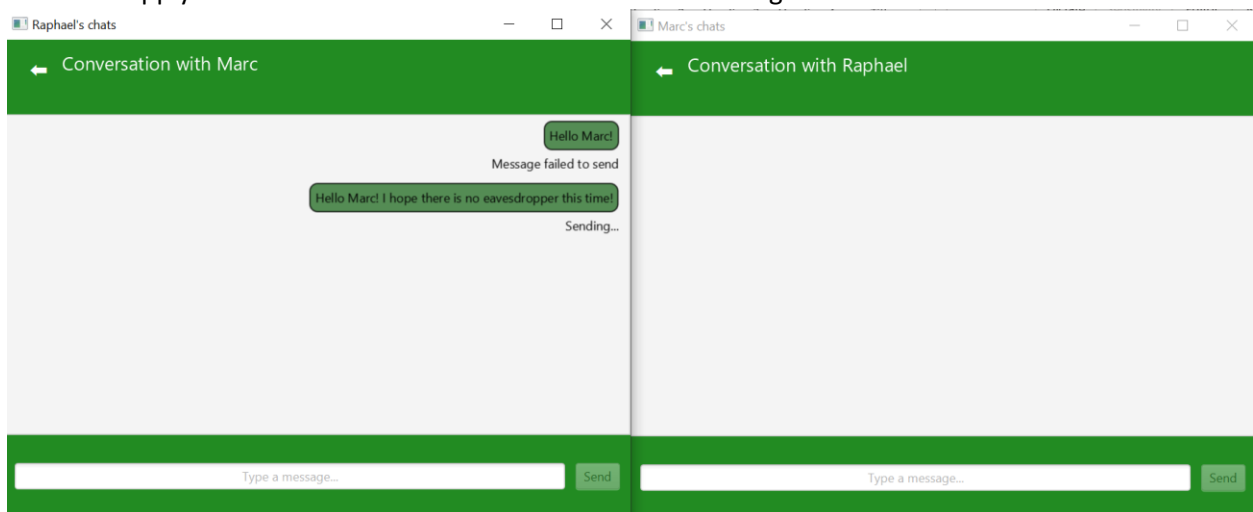


Now, the users want to message. So, go back to the Control Users Window, and deselect eavesdropper.

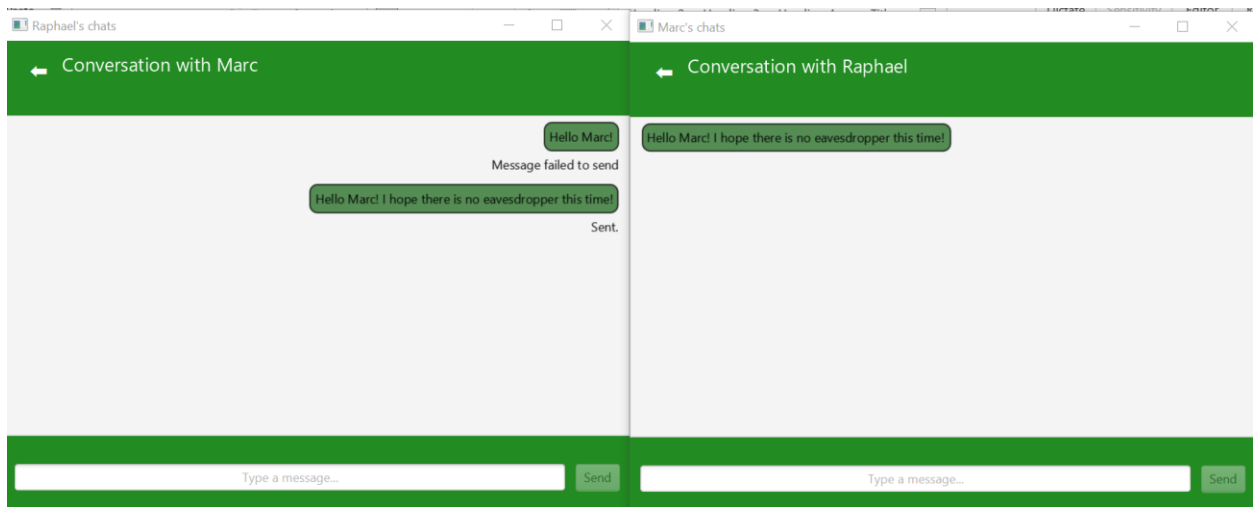


This into

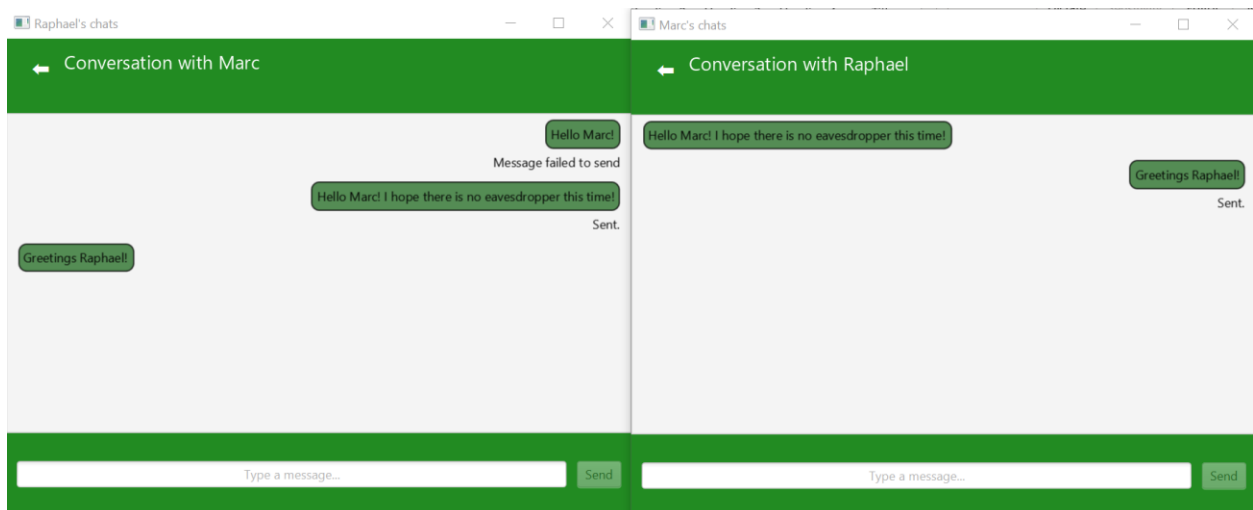
Now click apply. Go back to the chat windows and send a message.



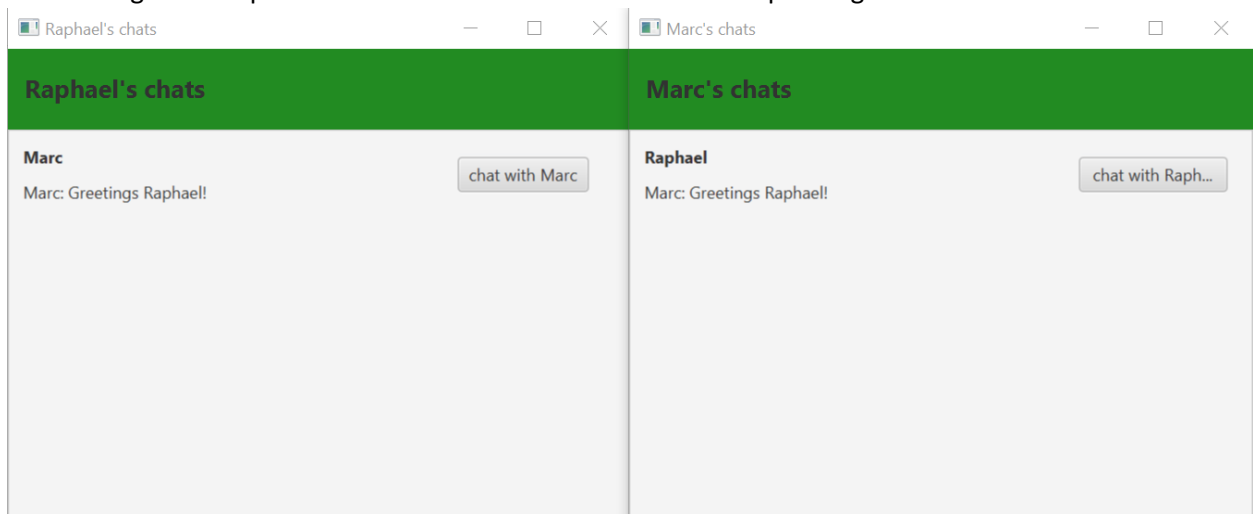
Success!



Now the other user will message.



Users can go back to their user windows by clicking the back button at the top left. Notice, the chat's last message will be present underneath the name of the corresponding other user.



Now, other users can be added and messaged with. However, this has already been covered in the guide.

Works Cited

- Abraham, Héctor, et al. Qiskit: An Open-source Framework for Quantum Computing. doi: 10.5281/zenodo.2562110
- Denning, Dorothy E. "Is Quantum Computing a Cybersecurity Threat?" *American Scientist*, American Scientist, 14 June 2019, www.americanscientist.org/article/is-quantum-computing-a-cybersecurity-threat.
- Haitjema, Mart. "A Survey of the Prominent Quantum Key Distribution Protocols." Quantum Key Distribution - QKD, McKelvey School of Engineering, 2 Dec. 2007, www.cse.wustl.edu/~jain/cse571-07/ftp/quantum/#bb84.
- Yu, Kun-Fei, et al., "Design of Quantum Key Agreement Protocols with Strong Fairness Property". National Cheng Kung University, 2017, arxiv.org/ftp/arxiv/papers/1510/1510.02353.pdf