Quantum Network Control Center User Guide

Version 1

Gruppe 49 BP 2022

March 18, 2022

Contents

1 Introduction

This document is intended as a guide for future users and developers of the Quantumnetwork Controllcenter (QNCC). The QNCC was created as part of the Bachelorpraktikum (BP) of the winter semester 2021/2022 by BP group 49. This group consisted of Jonas Hühne, Sarah Schumann, Aron Hernandez, Sasha Petri and Lukas Dentler, and worked for Maximilian Tippmann from the department of physics at the TU Darmstadt.

2 Quick-Start Guide

2.1 Introduction

The following document is a guide aimed at first-time users of the Quantum Network Control Center Application(QNCC). It will guide you through all steps required to create a working connection and start sending messages and files.

2.2 Prerequisites

2.2.1 Installation

A copy of the application should be installed on each machine that will take part in the communication during this guide, as well as the machine that will act as the photon source server.
To install the QNCC, you simply need to place the folder containing the .jar and the QNCC folder where ever you want to run the application from.
Note that multiple instances of the application can be run on the same machine but should be located in different locations to keep their records separate.

2.2.2 Java

The application itself will be a .jar file that was compiled with Java Version 17 and as such will require the machines to have at least Version 17 of Java to run it.
Alternatively a new .jar can be generated from the source code at a lower Java Version.

You can check your Java Version by entering the command "java -version" into the command terminal.

2.2.3 Port-Forwarding

To take part in the Network, each computer needs to be able to host a server that other potential communication-partners can connect to.

This requires that you either connect to the other computer via your local network, simulate a local network with a VPN or, if you chose to communicate over the Internet, you will need to open at least one port in your Routers settings for incoming connections.

Depending on your Routers model and brand, the settings might look a little different, but in general you will need to open the settings of your Router in your browser, access its Port-Forwarding settings and add a rule/exception for a valid Port-number. This rule should be for a miscelaneos application (not FTP or https etc.) and use the TCP Protocol.

2.2.4 Running the Application

To start the application, open the command terminal in the folder that contains the .jar file and enter the following command: "java -jar QNCC.jar". Depending on your system settings, you also might be able to run the .jar file by double-clicking it.

By default, the QNCC.jar launches without any arguments, but it is possible to supply 2 or 3 optional parameters. The first and second parameter can be used to input the IP and Port on which you intend to host your Communication Server. The third parameter can be used to launch without the GUI and use a console UI instead. Using the console UI is currently discouraged, as it does not support the full functionality of the software.
An example of a launch using parameters looks like this:
"java -jar QNCC.jar 127.0.0.1 2200"
It would run the QNCC and host the server on the local ip on port 2200.
To launch the application without the GUI, you can use "noGUI" as the third parameter.
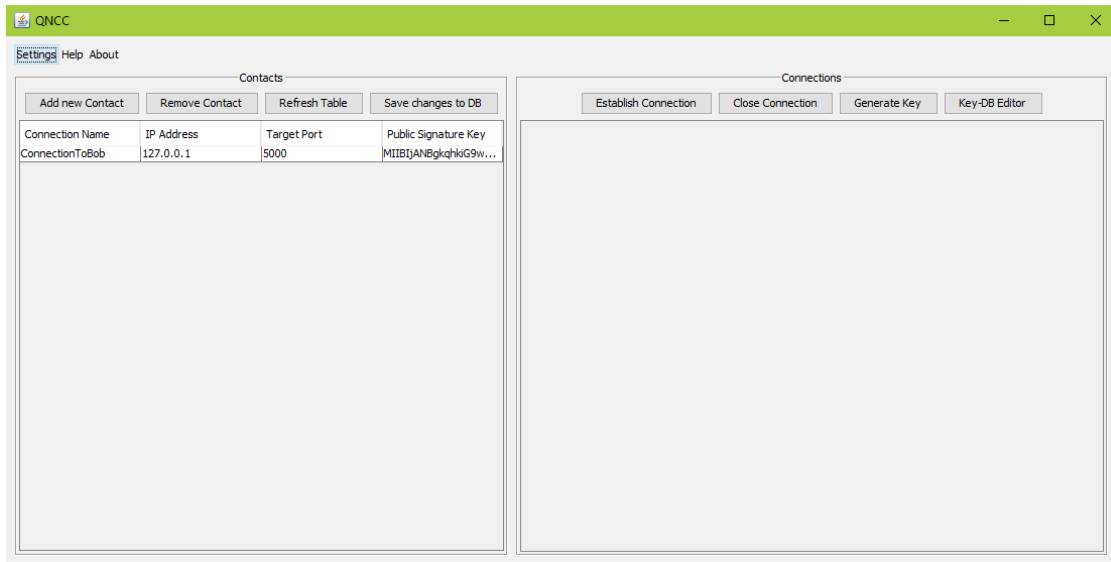
2.3 Overview of the Components

Figure 1: The GUI of the Quantum Network Control Center.

The application is made out of multiple different components. The components relevant for you as a user are:

1. The Settings Menu: Accessible via the '"Settings"' button at the top left of the application. Here you will enter all relevant information such as your IP and Port.

2. The Help Button: Located next to the '"Settings"' button. This opens the folder containing the documentation and this Quick-Start Guide.

3. The Contact Table: Located on the left side of the main window of the application. This allows you to store, edit and use persistent contact entries to make the connecting process less work-intensive.

4. The Connections Panel: Located on the right side of the main window of the application. Here you can see and interact with all connections that involve your machine.
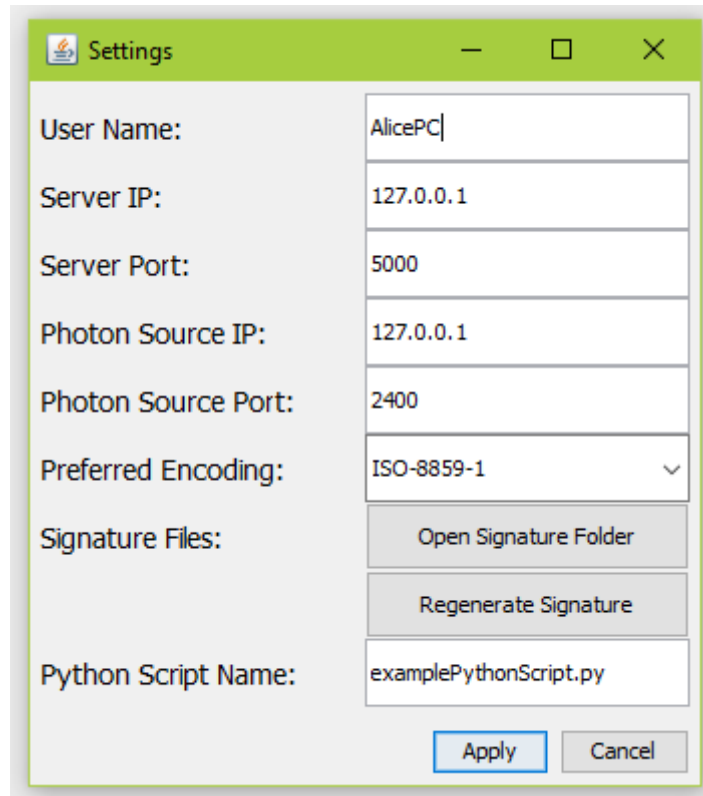
2.4 Adjusting your Settings

Figure 2: The settings menu.

When starting the application for the first time, or after your IP/Port config-uration has changed, you should begin by opening the Settings. Press the button labeled '"Settings'" in the top left corner of the application. Here you need to enter the following information:

1. **User Name:** Enter a name that represents you or your local machine. It will be sent to connected communication-partners.

2. **Server IP:** Enter the IP that other clients should connect to. Depending on the intended range of communication, this could be your localhost ip (127.0.0.1), you LAN IP 192.168.0.X or your WAN IP.

3. **Server Port:** Enter the number of the port that you opened during the prereq. steps of this guide. If you have not done so already, you need to open the port now. Your machine will be acting as a server on this port,

allowing other users of this software to establish connections towards you.

4. **The Photon Source IP and Port:** In order to generate a key for the encrypted communication, you will need an instance of this application running on the machine that can trigger the photon source. The IP and Port fields need to contain that machines IP and Port. If you do not intend to use encrypted communication or generate a key, you do not need to enter anything here yet.

5. **Preferred Encoding:** This is the encoding used when translating between Strings and byte-Arrays. Unless you need to change this, leave it at the default value.

6. **Signature Files:** Here you can open the folder where your own public and private signature keys are stored. Since this is the first time you are running the application, you will need to generate a pair of signature keys by clicking "Regenerate Signature". Now you should have 2 new files in the Signature Folder. These files will be of the types '".pub"' and '".key"'. The '".pub"' file is your public key, the '".key"' file is your private key. You need to share the contents of the public key file with anyone that you want to use authenticated communication with - this allows your communication partner to verify that a signed message was sent by you. **Never share the contents of your private key file with anyone**. The private key file allows you to authenticate yourself by signing your messages - anyone in possession of that file could use it to authenticate messages in your name.

7. **Python Script Name:** The python script needed for the actual key generation process should be placed in QNCC/python/ and it´s name should be entered in this setting. The python script should be designed to accept 2 arguments, initiative-variable that determines what part of the key gen process the script should take care of, and a path pointing to the folder in which the QNCC and the python script will exchange "in-", "out-", "terminate-" and "key.txt" files.

For now, click "Apply" to accept the changes to the settings and close the window. If you have changed your own Server Port, you will need to restart the application for the change to apply.

2.5 Adding a Contact

To make it easier to connect to a communication partner, you should add their details to your Contact Table. To do this, click "Add new Contact". A window will now open, which allows you to enter the contact information of your intended communication partner.
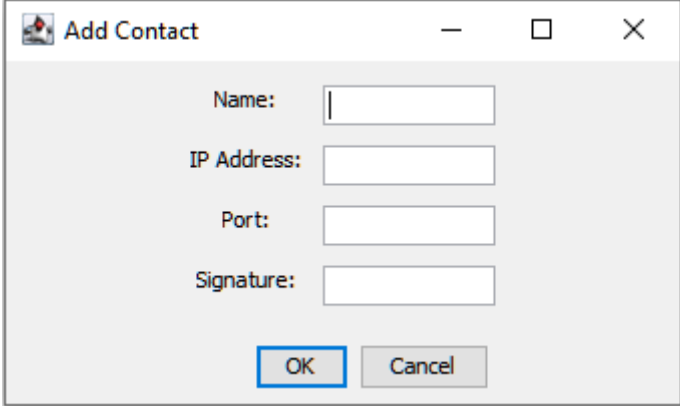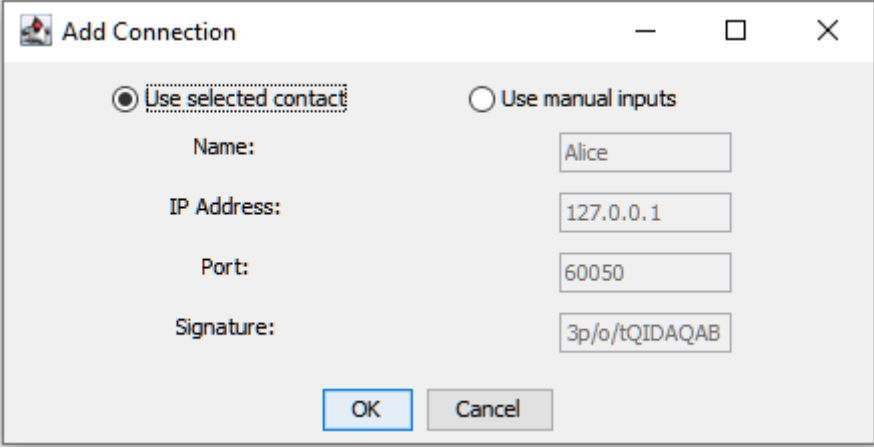

Figure 3: Adding a contact.

1. **Name:** The name you wish to save the contact under. Allowed symbols are alphanumerical symbols, - and _.

2. **IP Address**: When establishing a connection to this contact, this is the IP address you will connect to. This should be the IP address your partner entered as their Server IP in their settings.

3. **Port:** This needs to be the port your communication partner entered as their Server Port.

4. **Signature:** Optional. Enter the public Signature Key of your communication partner here. You can leave this blank if you do not plan to use authenticated or encrypted communication.
   You can also edit any of these values in the Contact Table later on.

   Now hit "OK" and you should see the new contact in your Contact Table. From there you can remove or edit the contact information. If you change any part of it, you have to hit "Save changes to DB" to make the edit permanent.

2.6 Building Connections

Figure 4: Adding a connection via a contact.

To build a connection to another machine, both your machine and the target machine need to run an instance of the QNCC application. On the target machine the port set as "Server Port" needs to be open. To start a connection to another machine, click the button labeled "Establish Connection". A window will now open and ask you for the contact details. If you have an entry in the Contact Table selected when clicking "Establish Connection", the information from the Contact Table is read automatically. Otherwise you will need to manually enter the necessary information.

If you do not plan on using authenticated messages, you can leave the Signature field empty. Hit "OK" to confirm the connection information.

After less that a second, a new Connection should have appeared in the Connection Panel. It is automatically selected as the active connection, indicated by the green border.

If the connection was successful, the connection state of the new connection should say "CONNECTED". If it says "CLOSED" or "CONNECTING" it may be that you entered the contact data incorrectly, or that the machine on the other end was not running the QNCC application. If neither of these conditions are met and a connection can still not be established, we recommend for both parties to restart the application and check the network configuration.

If a connection could be established the other communication partner should also see a new connection in his application.

The other two connection buttons, "Close Connection" and "Generate Key" always interact with the Connection that is currently selected as active, indicated by the green border.

2.7 Communication and Authentication

Figure 5: An example chat between Alice and Bob, from Alice perspective. Alice has entered Bob's public key in her contacts table, so she was able to verify messages sent and signed by him.

Once a connection has been established, you can click on "Message System". This will open the message and file-transfer log. Here you can enter and send messages, as well as read messages and file-transfers that you have received. Enter any text you wish to send in the lower text area and hit "Send Message". Your communication partner should see your message in their message log and be able to respond.

This covers basic communication. However, it is not authenticated and not encrypted. If you are using the Contacts Table, enter your partners public signature key in the signature field and hit "Save changes to DB". This will allow you to verify messages you *receive*. Next, change your Connection Mode from "UNSAFE" to "AUTHENTICATED".
If you created the Connection before setting the Signature Key in the Contac Table, you may need to enter the Signature Key of your communication-partner after switching to "AUTHENTICATED".

Messages you *send* from now on will be signed via your private signature key. If the receiver has entered your public signature key in their contacts table or when building the connection, they will be able to verify messages sent by you.
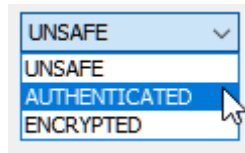
Figure 6: Changing the communication mode.

Please Note: Selecting UNSAFE/VERIFIED/ENCRYPTED determines only what kind of messages you are *sending*, not what kind of messages you are *receiving*. You can receive any kind of message at any point, depending on which mode your partner has selected.

If you did not enter a signature key when you created the connection, you will be asked to enter a key once you either receive an authenticated message or switch your own Connection Mode to "AUTHENTICATED". If you do not have a key to enter but still want to see the message, you can click "Cancel" when asked to enter a public signature key and then choose "Read" when you receive a signed message.

Now you can use the "Message System" as usual, but you will be able to verify authenticated messages received from your partner. Note the <veri-

fied> prefix in front of received messages.
If the verification fails for what ever reason, the message is not displayed, but a warning appears stating: 'SENDER-NAME' <Failed to verify Message!> ".

Should you see this warning, make sure that you gave your communication partner your correct public signature key, and that they entered it correctly.

To exchange encrypted files and messages, select ENCRYPTED as the communication mode. However, for encryption and decryption to take place, both you and your partner will need to have the same key stored in the key store database. This can currently be achieved through the Key Generator or manually edited via the Key-DB Editor (found under the connections tab).

2.8   Generating Keys

For key generation to take place, three machines are needed. First, the two machines that wish to generate a mutual key (A and B), and then a third machine with a photon source. The machine with the photon source needs to have an instance of the QNCC running, and both parties that wish to generate a key will need to have entered the photon source machine's IP and Port correctly in their settings. Additionally, both A and B will need to be able to verify messages from each other. If no Signature Key was added before or can be found in the Contact Table, it will be requested during the Key Generation Process.
For timing reasons it is recommended to make sure that both parties can send and receive authenticated messages before beginning the key generation process.

With a connection between you and your partner established, click "Generate Key" at the top of the Connection Panel. This will then agree with your connection partner to generate a key and you will contact the Photon Source Server, sending it a file containing information about the two parties wanting to receive photons.
Then, it will run the python script specified in the settings.

From here on out, the connection will transmit any file names "out.txt" from its own folder in "*AppRoot*/QNCC/connections/" to the partners "*AppRoot*/QNCC/connections/" folder. Incoming transmissions for key generation will be placed as "in.txt" in that folder. This is how information can be exchanged between the python scrips via out application.
The "out.txt" is only sent if there is no file named "out.txt.lock"

The KeyGen Process can be aborted by placing a file named "terminate.txt" there and it is successfully completed if the application finds a file named "key.txt" in the folder, that contains the completed key.

The contents of this key file are then stored in the Key DB for future use.

Now you can use the Message System to sent encrypted messages or files. Please note, each bit in the key is only used once for encryption, and currently our application uses 256bit AES, meaning that 32 bytes of the key are used for each encryption.

You can also see the existing keys by clicking on the "KeyStore Editor"-Button at the top of the GUI.

2.9   The config.xml file

In the same location as the .jar file, the application will create a config.xml file when it is started for the first time. There some persistent information will be stored, e.g. the ones changeable via the settings. As it is a .xml file, it can be changed manually, but this should only be done by those who understand the basic structure of such a file, and know what the effects of the changes they make are. Tempering with the file can potentially make it unusable for the program and thus hinder or prevent the execution of the application. In this case, deleting the file will cause a new one to be generated the next time the QNCC is launched.

2.10   Conclusion

This concludes the Quick-Start Guide for the Quantumnetwork Communication Center.
You should now have a solid understanding of the functionality that our application offers.

3 Developer Guide

3.1 High Level Concept of the Program

The QNCC offers various powerful features, such as a contact Database, authenticated and secure communication channels for messages and files, a key generator and a set of easily customizable settings.

The overall network architecture utilizes one Server Socket per application and supports multiple direct connections from one application instance to another.

A connection is created by connecting from one local "Connection Endpoint" to a remote Server Socket, that then creates a new "Connection Endpoint" that is connected to the first CE.

The Network built from these connections can be used to send messages and files in one of 3 modes, "unsafe", which is just a plain-text-transmission, "authenticated", which signs each transmission with the local private signature key and requires the recipient to have the senders public signature key to verify the signature and third, "encrypted", which requires a key to be generated first and sends sends encrypted transmissions.

For the process of key generation, a suitable python script needs to be placed in QNCC/python/ and the scripts name needs to be set in the applications settings.
The script is expected to accept 2 arguments, the "initiative" and a path that indicates where the file-exchange between our application and the python script will happen, as well as where the finished key.txt will be placed by the script.
This key will then be read by the QNCC and added to the key store DB.

In order to trigger the photon source, there needs to be an instance of the QNCC running on the machine connected to the Photon Source, that accepts signals from any other instance of the program and enables the Photon Source on demand.

3.2 Overview of the Package Structure
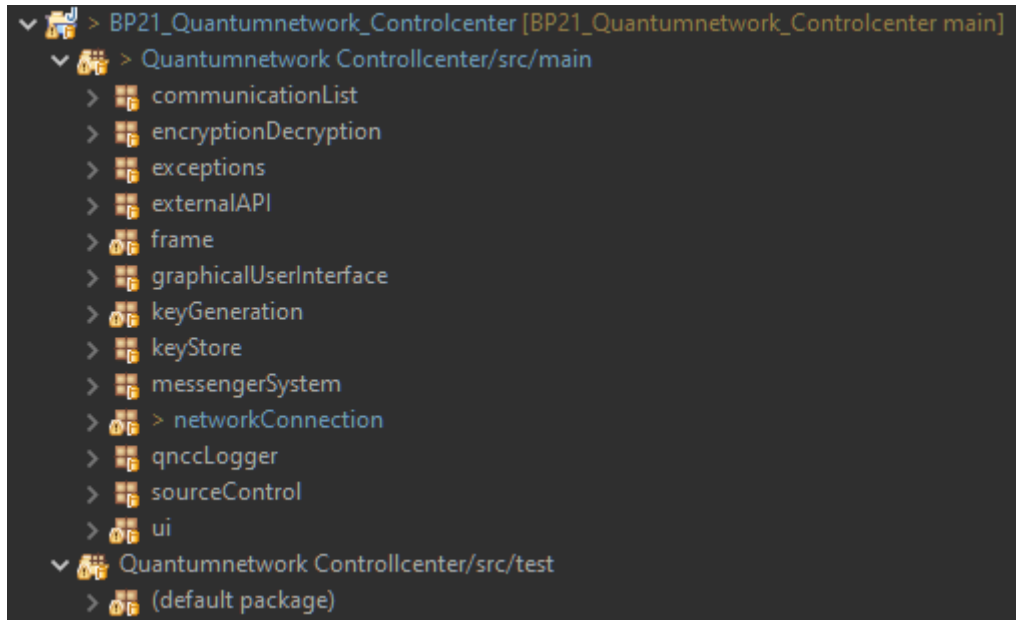
Figure 7: Package Structure.

The classes in our program are sorted into different packages according to their purpose. In alphabetical order, these packages are:

- *communicationList:* This package contains classes pertaining to the communication list / list of contacts. The Interface intended to be shared by every implementation of a communication list is located here, as well as our SQLite-based implementation, and a class representing a contact.

- *encryptionDecryption:* Classes related to encrypting and decrypting data are located here, including the AES256 class, which implements the symmetric cipher currently used for encrypted communication. It also includes the FileCrypter class, which encrypts and decrypts files.

- *exceptions:* These are custom Exceptions for use in our program.

- *externalAPI:* Currently in disuse. The intent behind the external API is to allow access to basic functionality of the program (such as key generation and encryption) without needing a GUI, which would make it easier for outside programs to make use of our implemented functionalities.

- *frame:* Contains the classes for launching and configuring the program.

14

- *graphicalUserInterface:* Contains any classes that are directly part of the GUI, through which the user interacts with the program.

- *keyGeneration:* Contains the KeyGenerator class, which contains much of the logic needed for generating symmetric keys. Could later also contain, for example, the classes used for cleaning raw key data.

- *keyStore:* The classes contained here have the purpose of storing, managing and delivering keys after generation, for use in symmetric encryption and decryption.

- *messengerSystem:* Contains classes pertaining to the sending of messages, as well as to their authentication.

- *networkConnection:* Contains most of the '"low-level'" network logic, such as the NetworkPackages which are sent through the network, the ConnectionManager which manages connections by maintaining multiple ConnectionEndpoints etc. The logic for receiving messages and handling them is also implemented here (see also 3.4.6).

- *qnccLogger:* Contains the logger used in our program as well as the classes it requires to function.

- *sourceControl:* Classes implementing logic used and required only by the machine with the photon source.

- *ui:* Classes relating to the Console UI, such as the Command enum, and classes for command parsing and execution.

Finally, there is one package which contains the unit tests used to validate our program.

3.3   Dependencies

The program uses two libraries that are not part of the standard java library. The first is the sqlite-jdbc driver, which can be found here: https://github.com/xerial/sqlite-jdbc. The second is the MiG layout for swing. The website for the MiG layout can be found here https://www.miglayout.com/, although we recommend the Maven page for downloads, which can be found here https://mvnrepository.com/artifact/com.miglayout. How to install these dependencies will differ, depending on what IDE you use, and also if you use build automation tools such as Maven. Due to the variety of different setups you may have, we can unfortunately not provide you a tutorial for your specific use case on how to integrate these components into your development environment.

3.4 Individual Components

As part of our quality assurance process, we have documented each class and function in the program with JavaDoc annotations. However, to further aid future developers, this section will give some additional information on some of the components in our program, including from a "'high-level'" point of view. We also highlight some possible future improvements and extensions.

In addition, we also offer a minimal network testing application, the "BasicNetworkTest.jar".
This utility can be used to check if two computers can connect at all.
The tool should be launched first on the machine that should play the role of the server and then on the one filling the role of client.
Just follow the instructions that the program prints in the console to perform the test.
If successfully connected, client should be able to send a message to the server, and the server should be able to respond.
A pair of computers that can connect this way will also be able to connect via the QNCC.

3.4.1 Frame (Launch and Config)

Launch   The application is launched via the class "'QuantumnetworkControllcenter'". During launch, an initialization method is run, which initializes the individual components used in the application, e.g. by creating instances of the ConnectionManager, Authentication and Encryption algorithms used in the program. These created components are currently accessible through publicly accessible fields, however, to reduce coupling, some classes receive the components they need via their constructor or certain methods instead (e.g. the MessageSystem receives its ConnectionManager here.

The way these fields are currently handled is functional, however, depending on your design principles, it may be desirable to further decouple the sub-components from the QuantumnetworkControllcenter class.

The Configuration File   During the first launch of the program, a configuration file will be created, in the form of an xml file. This file will be saved in the directory the JAR is executed in. It will also be created on any subsequent launch, should non be present at the correct place at the time. The file includes certain String variables used during the execution application and

changed if needed, so under normal circumstances, manually editing it will not be necessary.

The file consists of some header lines and tags (they should not be changed!) in addition to the key value pairs. New value pairs can be easily created with the Configuration methods. Once part of the file, the value part of the pair can be changed using a normal text editor if necessary. The ones normally present for the current implementations are:

- **'"basePathConfig"'**: This holds the absolute path leading to the folder, where the created '"QNCC"' folder is located. It should include the system specific separator at the end. If the '"QNCC"' is ever moved, or some folder in the path renamed, the path here should be adjusted to the new location to ensure that all files are still present. If the path existing here does not exist, it will hinder the application execution.

- **'"UserName"'**: This is the name of the user, that will be sent to the communication partner upon requesting a connection

- **'"UserIP"'**: This contains the IP address of the user. It should be the public IP address, but will unless changed be set to localhost at the start of the Program.

- **'"UserPort"'**: This is the port the application will listen on for connection requests, and should be set accordingly.

- **'"privateKeyFile"'**: Here the name of the file containing the private key of the user, which will be used for the signature based authentication, is saved, and can be changed. It has to be an existing file with one of the accepted file endings (checked in the '"Utils"' class of the '"messengerSystem"' package) and must be located in the '"SignatureKeys"' folder included in the '"QNCC"' folder. It should be a valid key pair together with the entry of '"publicKeyFile"'.

- **'"publicKeyFile"'**: Here the name of the file containing the public key of the user, which will be used fot the signature based authentication, is saved, and can be changed. It has to be an existing file with one of the accepted file endings (checked in the '"Utils"' class of the '"messengerSystem"' package) and must be located in the '"SignatureKeys"' folder included in the '"QNCC"' folder. It should be a valid key pair together with the entry of '"privateKeyFile"'.

- "'SourceIP'": This is the set IP address of the computer controlling the photon source, which has to be contacted during the key generation.

- "'SourcePort'": This is the set port of the computer controlling the photon source, which has to be contacted during the key generation.

- "'PythonName'": The name here is the name of the python script used during the key generation. It should be located in the subfolder "'python'" of the "'QNCC'" folder, and is responsible for handling the python part of the generation, including calling other python files.
  The script is called with 2 arguments: the "initiative" that indicates which side of the key generation process the script should take care of and the "path" to the folder in which all files exchanged during the generation process will be written to and read from.

- "'Encoding'": This is the name of the encoding used for creating or reading files and Strings. It will be used in different parts of the application to ensure no problems arising because of differences in the encoding and decoding. The default is "'ISO-8859-1'", which should cover all of the needed characters. When changing this value manually one must make sure to use the correct notation of the name, meaning the one understood by the corresponding java methods. This is also the reason, why the settings use a drop down menu for this, to avoid complications because of typos or wrong notations.

The "'Configuration'" class also has a method for creating the folder structure needed by the program, which will be called on application start. It will create the folders at the path specified in the config file, unless they already exist. Thus it will not delete any data in the the process. The list of subdirectories can be extended by adding more directory names to the variable "'DIRECTORY_LIST'", which is in the form of a String array.

3.4.2 The GUI

The GUI is built using the swing framework. For editing the GUI, we recommend using a tool such as the WindowBuilder plugin for Eclipse.

The main GUI Window is divided in 3 parts: A bar at the top with multiple buttons, a Table on the left half of the window that allows the user to interact with the Contact Database and an area on the right half that will be populated with representations of all existing connections.

18

The GUI package also contains various smaller windows, each for a specific task such as adding a new contact or creating a new connection. There are also different dialogs windows for things like warning pop-ups.

3.4.3 The Console UI

Due to time constraints during the internship in which the original software was developed, the console UI was abandoned. The methods associated with it are depecrated, and the JavaDoc is not guaranteed to be up to date. It does not currently support the full functionality of the program, such as sending messages and exchanging keys.

The idea behind the console UI was that it allows users to control the program via the command line / a custom UI styled in the way of a command line. For example, via '"contacts show'" a user could have the console display the contacts in the communication list. With '"contacts add Alice 127.0.0.1 9000'" they could add Alice as a contact, with the IP/Port pair 127.0.0.1:9000. The console also supports some utility functions, such as UP_ARROW to repeat a previous command, or SHIFT+RIGHT_ARROW to auto complete command names.

Should you wish to continue developement by adding additional commands, you would need to give each new command an entry in the Command enum, add it to the switch statement in CommandHandler and then implement a function, which represents the command. You may wish to group similar functions into a class. For example, if you were to implement commands for sending text messages and files, you could add '"SEND_TEXT'" and '"SEND_FILE'" to the Command enum and then create a class '"MessageCommandHandler'" with methods *handleSendText*(...) and *handleSentFile*(...). In '"CommandHandler'" you could then call these methods in the appropriate case of the switch statement.

In the current implementation, methods only inform the user through one returned String (which is displayed in the console) about their execution. This could be improved to, for example, use OutputStreams and Multi-Threading. In particular for Messaging related applications this may be relevant.

3.4.4 The Key Storage

The key storage, also referred to as the key store, is where the keys used for symmetric encryption are saved. We currently use the implementation provided in '"KeyStoreDbManager'". The key store allows to locally store keys in a SQL database, using SQLite through JDBC. An entry in the database has

seven parameters:

- The key ID (identifies the key, unique)

- The key buffer (byte array, actual contents of the key)

- An index, which is the # of bytes of the key that have been used already

- A source parameter

- A destination paremeter

- A boolean indicating whether the key has been marked as used (ready to delete)

- An initiative parameter.

The naming is based on the ETSI Standard '"ETSI GS QKD 004 V2.1.1'". During the initial internship it was planned that we adhere to the ETSI standards where possible, however, strict adherence was not possible due to time constraints and other design requirements. In particular, rather than the key stream based approach proposed in the paper, we have decided on a more basic approach using a database.

The source and destination parameter are currently set during generation (see class '"KeyGenerator'"), but not otherwise used. The initiative parameter is set during key generation: the one who initiated key generation will set it to *true*, the partner will set it to *false*. It is currently only handed over as an argument to the python script when the key generation is started, however, the idea has been proposed that during conflicts regarding concurrent encryption using the same key bytes, the party with initiative has priority, i.e. they get to '"use up'" the key bytes.

3.4.5   Connection Management

The Connection Manager is created when the application is launched and is responsible for keeping track of all existing connections. It offers methods to create and remove connections, as well as querying information about the currently existing ones. In general, only the connection manager should be used to add or remove connections to ensure that the rest of the application has reliant access to the existing connections.
　　Connections themselves are realized through a peer-to-peer network that uses Java's Sockets and ServerSockets. Every connection is represented by an instance of the ConnectionEndpoint class. When a member A of the

network attempts to connect to another member B, A's client socket will try to connect to B's server socket. The server socket is managed by the ConnectionManager, which has the *accept*() method running concurrently on a seperate thread. If A's client socket connects to the server socket, a ConnectionEndpointServerHandler (CESH) is created, which takes care of accepting and answering the connection request, as well as creating a connection endpoint in B's connection manager, which allows B to then also communicate with A. Further details can be found in the constructors of the ConnectionEndpoint class, as well as the *createConnectionEndpoint*() methods in the ConnectionManager.

If you have trouble following the connection creation process, it may be helpful to use an IDE tool such as Eclipse's '"open call hierarchy'" to follow the call hierarchy of certain transmission types, such as CONNECTION_REQUEST.

3.4.6 Messaging and Message Processing

At a high level, messaging is done via the '"MessageSystem'" and its methods. These methods allow the sending of text messages and files, optionally verified and encrypted. At a low level, message handling happens in each '"ConnectionEndpoint'" (CE) and the '"NetworkPackageHandler'" class. Messages in the network are transmitted via '"NetworkPackages'", which a CE sends to its connected partner through the *pushMessage*(*NetworkPackagemsg*) method.

As for processing, each CE has a thread running, which concurrently checks the InputStream of the CE's socket for new NetworkPackages. Should one arrive, the method *processMessage*(..) is called, which handles the message if it's a connection confirmation, or otherwise passes it to the '"NetworkPackageHandler'" (NPH). The NPH handles each package based on its type and on its arguments. The type and arguments give '"meta-information'" about the package. For example, a message with a type of '"CONNECTION_TERMINATION'" will be interpreted as a request to shut down the connection. Or, for a message of type '"FILE_TRANSFER'", the MessageArguments will be examined to get the name of the contained file via the *fileName* message argument. The message arguments are also used when receiving encrypted messages, as the *keyIndex* argument tells the receiver at which index in the mutual key they should start using bytes for decryption.

For details on the types, see '"TransmissionTypeEnum'".

Regarding verification, the NPH will attempt to verify every package it receives where the signature is not null. If a package has a signature that is not null, and it can not be verified for whatever reason (invalid public key, no public key for sender, ...) it is discarded. Authentication is performed with

whichever Authentication algorithm is currently instanced in the MessageSystem. NetworkPackages are signed and verified through use of their own *sign* and *verify* methods, which take all their fields into account. This is because it is not sufficient to sign and verify their byte array content, because, as mentioned before, their type and arguments are also relevant for control flow. When party A receives a signed package from party B, and uses B's public key to verify it, party A needs to be sure that the package was not manipulated on the way, or created by someone impersonating A. For example, if only the contents were signed and verified, party C could intercept a package from B to A, change the type or arguments, send it to A, and A would not be able to tell the package was manipulated.

3.4.7 Encryption and Decryption

The intent behind the QNCC software is that, once the Quantum Key Distribution (QKD) has taken place, that the mutually generated keys can then be used for encryption and decryption. For this, symmetric ciphers are used. Our software currently implements one such cipher (AES256). However, it could easily be extended to implement any number of symmetric ciphers, as long as they extend the SymmetricCipher class found in the *encryptionDecryption* package.

For encryption and decryption itself, key bytes from entries of the key store are used. Entries in the keystore are likely going to be very large (thousands of bytes), and so for use in encryption and decryption only the neccessary amount of bytes is retrieved. When encrypting or decrypting data sent through a connection, the ID of the key to use is acquired through *ce.getKeyStoreID*(), where *ce* is the ConnectionEndpoint representing the connection. The *keyStoreID* is currently equivalent to the connection endpoints ID, and a possible point of improvement would be to allow users to set it through the GUI.

The process of sending and receiving encrypted data works roughly as follows:

1. The sender and receiver have agreed upon a mutual key.

2. The sender executes methods *sendEncryptedTextMessage*(...) or *sendEncryptedFile*(...) of the MessageSystem class.

3. In these methods, some amount of data is encrypted. For this, bytes of the mutual key are retrieved, starting at the current index *n*. The index is how many bytes have been used yet (0 at the start). The index is incremented by the key length in bytes.

4. The sender informs the receiver that they have encrypted data starting at index n, through a KEY_USE_ALERT message, where the *keyIndex* argument is set to n. This message is signed to guarantee its integrity. They will not send the data unless they receive a verifiable KEY_USE_ACCEPT message, where the content is equal to the ID of the KEY_USE_ALERT message.

5. If the receiver gets the KEY_USE_ALERT message, they compare the *keyIndex n* with their local index value m for the mutual key. This is important, because the receiver may have encrypted something before receiving the key use alert, and their index m might be greater than n now.

   If $n >= m$ this means that the sender is only using bits for encryption that the receiver hasn't used yet. In that case, the receiver sets m to $n + key\_length\_in\_bytes$ and sends back a KEY_USE_ACCEPT message.

   If $m > n$ that means the sender would use bits for encryption which the receiver has already used. This isn't permissible, because each bit should only be used once for encryption. In that case, the receiver sends back a KEY_USE_REJECT message, with the *keyIndex* parameter set to m, to inform the encrypter about what the new mutual index value should be.

   In either case, the message is signed and has the ID of the KEY_USE_ALERT it is responding to as its content, so the encrypter knows which request it is a response to.

6. If the encrypter receives a KEY_USE_ACCEPT message (that is verified and has the correct contents), they send the encrypted message. The encrypted message has its *keyIndex* parameter set to n. If the encrypter receives a KEY_USE_REJECT message (that is verified and has the correct contents) they discard the encrypted message and set their own index to m. If they receive neither a confirm nor reject after three seconds, they discard the message.

7. When the receiver gets an encrypted message and can verify it, they get a key from the keystore starting at the index specified in the messages *keyIndex* argument. They then use this key to decrypt the message. Because both the encrypter and decrypter use bytes from the same key starting at the same index, the decryption will be successful and the decrypter will be able to read the contents of the message.

This protocol could be improved through the sender asking for permission to encrypt first and only encrypting when permission is received. However, a

solution of that nature would likely require some non-trivial parallel program-
ming.

3.4.8 Logging

The logger used for this project uses a singlet instance of Java-utils File-Handler to save all logging events into a log file created when starting the program and named with the current date and time.
For each class that should write content to the log file an instance of the class *Log* needs to be crated as one of the classes fields. The constructor needs

- the Name of the class in the Format of "*NameOfClass.class.getName*()

- and the sensitivity of the logger for this class (from the enumeration *LogSensitivity* that determines the needed severity of log events to be saved to the log file.

3.4.9 Communication List

The communication list is used for storing information about communication partners in a persistent way. These can then for example be used to initiate a connection. For this, an entry (also called contact) stores these four pieces of information:

- the name of the contact

- the IP address of the contact

- the port of the contact

- the public key of the contact, used to verify the signature potentially sent with a message

The information is currently stored in a SQLite Database using JDBC. An interface called '"CommunicationList'" provides the needed public methods. If a different kind of saving method is desired, this should be implemented, and a change of the initiated object in the '"QuantumnetworkControlcenter'" class should be sufficient for it to work. The methods there include an insert method, setter methods for the information of the entries by name, a query method both by name and by the combination of IP and port, as well as a query method to get all entries currently in the list.
The current implementation uses Regex to check requirements for the information, such as a syntax check for the IP address (accepted are both ipv4 and ipv6) and one for the name, which can be used to limit the symbols allowed there. Additionally a minimum and a maximum for the port number can be set, to ensure the entered number could be valid. A unique constraint

set during the table creation ensures, that each name is unique, and a specific pair of IP address and port can only be entered once.

The interface as well as the implementing class '''SQLiteCommunicationList''' also use the immutable object '''Contact''' for returning the entry information during a query. There, the four parts of the contact information can be set on object creation and not be changed afterwards. Getter methods for the variables are included.

3.4.10   Source Control

When generating a key, it is necessary to have access to a machine that can trigger the photon source.
This machine needs to run an instance of the QNCC and the IP and Port of this instance need to be entered into the settings of the two applications that want to generate the key.
During the key generation process, the user that initiated the key generation (i.e. has initiative = 1) will create a connection to the QNCC that is acting as the Source Server and send a signal containing the IP and port of both parties involved in the key generation.
The received information is written to a file and then the connection is closed and destroyed by the Source Server.
The contents of these files is used to determine where to send the photons for the key generation.

3.4.11   External API

Using the existing methods the external API provides 3 main types of functionality for outside programs

1. provide a key for a given communication partner as

    - *SecretKey*
    - *bytearray*
    - *String*

2. any given files can be

    - encrypted for a given communication partner
    - decrypted after it has been encrypted by a given communication partner

3. the existing network can be used to send and receive files (including encryption with following decryption)

3.5 How to Compile

To compile the project in Eclipse, for the SQLite database access you need to include a driver by adding an external jar as referenced library to the project. The file is called sqlite-jdbc-VERSION.jar and can be downloaded here: https://github.com/xerial/sqlite-jdbc

To compile the program in Intellij, you need to add the driver for the SQLite database access. This can be done by including an external jar to the libraries via the '"Project Structure'" dialog located in the '"File'" dropdown menu, in the subcategory modules. The jar is included in the plugins of the IDE, in the subfolder plugins/svn4ide/ and is called sqlite-jdbc-VERSION.jar, or can be alternatively be downloaded here: https://github.com/xerial/sqlite-jdbc

Additionally you need another library for the layout. Adding it is the same as for the jdbc library. Please refer to the dependency section 3.3.

3.5.1 Creating a jar

Intellij:
File -> Project Structure -> Artifacts
Create a new Aritfact -> JAR -> From modules with dependencies
Module should already be "Quantumnetwork Controllcenter", otherwise choose it
Select the main class ("QuantumnetworkControllcenter (frame)")
Leave the selection on "extract to the target JAR"
Choose the correct Directory (should be the one above the "META-INF" folder, currently "Quantumnetwork Controllcenter"
Make sure the box infront of "Include tests" is not checked
OK Remove all unnecessary libraries from the included parts (left side) (Currently, only the sqlite-jdbc-(ver).jar, with (ver) replaced with the correct number code, and the one for the miglayout should be necessary)
Change the name to not have a space (better for file usage)
OK
Build -> Build Artifacts
Select the correct one and choose "Build"
The Jar is located in the folder Quantumnetwork Controllcenter/out/ artifacts/QuantumnetworkControllcenter_jar/
(File separators might vary between different operating systems)
(This way includes the META-INF for the jdbc library, including the license)

For less difficulty while executing the file, rename it eg '"QNCC.jar"'

Eclipse:
Right-click in the project -> Export -> Runnable jar
Choose the main class "Quantumnetwork Controllcenter" of the package "frame" for Launch configuration
Choose an Export destination
Choose the option "Extract required libraries into the generated JAR"
Click "Finish"

VisualStudioCode: (untested)
You need the "Project Manager for Java" Extension, eg included in the "Java Extension Pack"
To export as a jar, you have to click the arror in the topline of the Java Projects Section (should read "Export Jar")
Choose the main class (here: QuantumnetworkControllcenter in the frame Package)
Make sure to include the jdbc lib and the miglayout lib in the next step, as well as the target classes, but not include any other libraries
Click okay
You should now have a jar file

3.5.2 Additional Note

During the creation of the jar, the IDE might include files with the endings .RSA, .DSA or .SF into the subfolder META-INF, which can in some cases hinder the program execution. To prevent this, they should be deleted beforehand. On Linux this can be done by using a standard archive handling program, eg Ark, to delete these files. On Windows, it works with tools like 7zip. In both cases unpacking it should not be necessary.