Programming a quantum network

Axel Dahlberg Stephanie Wehner

January 21, 2018

1 Introduction

In this project, you will program your own quantum protocol! We start by a very simple protocol in which Alice will simply send $|0\rangle$ to Bob over a channel (which we will call Eve). You will get to extend this program to perform a simplified form of BB84 QKD yourself on simulated quantum internet hardware!

To pass this project (and get the highest grade), it is sufficient to complete the specific exercises at the end of this document. However, you may choose to enter your exercises, or any extension, other protocol - in short, whatever you want! - into our competition! Several prizes are available with the best individual project having the opportunity to be offered an internship in QuTech's quantum internet team this summer! We will also showcase the 10 best projects on our website.

Here, some useful links and then its time to get started!

- SimulaQron Website: [www.simulaqron.org]
- SimulaQron on Github: [https://github.com/StephanieWehner/SimulaQron]
- Code for this exercise in folder: [SimulaQron/examples/programming_q_network]
- Arxiv Paper describing the inner workings of SimulaQron: [https://arxiv.org/abs/1712.08032]
- Google form for entering the competition [https://goo.gl/forms/8TegAvM0zxCSt4bW2]

1.1 Important dates and deadlines

- You can start submitting your project to edX in Week 7!
- Deadline for submitting your project to edX: 23 January 2018
- Deadline for peer-reviews: 28 January 2018
- Deadline for filling in the form for the competition: 23 January 2018

2 Installation instructions

To install SimulaQron, follow the steps below. The first step provides the necessary prerequisites by installing Anaconda. If you are familiar with installing Python packages, this can also be done without Anaconda: To run SimulaQron you need **Python 3** with the packages **twisted**, **service_identity** and **qutip**. Note that qutip also requires additional packages but these, together with twisted and service_identity, are all included in the latest version of Anaconda (but not older ones!). Furthermore, you need for Python 3 to execute if you type `python` in a terminal.

If you do not know how to set this up, follow the instructions in step 1 below. To follow the instructions you need to have access to a terminal. We are assuming here you that you use bash as your terminal shell (e.g., standard on OSX or the GIT Bash install on Windows 10). On most Linux distributions press the keys Ctrl+Alt+T to open a terminal. For OSX you press Cmd+Space, type terminal.app and press enter. For Git Bash on Windows an icon will be installed by default on your desktop, which you can click to open bash.

Note: In the default configuration, SimulaQron starts up multiple servers on localhost (i.e., your own computer) to form the simulated quantum internet hardware. SimulaQron does not provide any access control to its simulated hardware, so you are responsible to securing access should this be relevant for you. You can also run the different simulated nodes on different computers. We do not take any responsibility for problems caused by SimulaQron.

---

1. **Prerequisites**:
   The easiest way to get the required packages to run SimulaQron is to install Anaconda. Follow the link below that corresponds to your operating system:

   - Linux
   - OSX
   - Windows

   and go through the instructions to install Anaconda. When you download the installer, make sure to chose Anaconda and the 3.X version.

   When you have installed Anaconda you should also get the Python package qutip. Do this by typing the following in a terminal:

   ```
   pip install qutip
   ```

   Note: If you already have Anaconda and want to create a new environment for using SimulaQron (maybe your current environment uses Python 2). Then type the following:

   ```
   conda create -n new_env anaconda python=3
   ```

   where `new_env` is the name of the new environment and can be chosen by you. Note that older versions of Anaconda did not include Twisted - so you may wish to update Anaconda or installed Twisted manually. You should include `anaconda` in the commands since this installs all packages in the default Anaconda to the new environment. To activate this new environment type `source activate new_env` on Linux and OSX or `activate new_env` on Windows.

   ---

2. **Download SimulaQron**:
   SimulaQron is accessible on GitHub. To download SimulaQron you need to have git installed. Installation instructions for git on your system can be found here. When you have git installed open a terminal, navigate to a folder where you wish to install SimulaQron and type:

   ```
   git clone https://github.com/StephanieWehner/SimulaQron -b BetaRelease
   ```

   The path to the folder you downloaded SimulaQron will be denoted *yourPath* below.

3. **Starting SimulaQron**:
   To run an example or the automated test below you first need to start up the necessary processes for SimulaQron. Open a terminal and navigate to the SimulaQron folder, by for example typing:

```
cd yourPath/SimulaQron
```

You also need to set the following environment variables, by typing:

```
export NETSIM=yourPath/SimulaQron
export PYTHONPATH=yourPath:$PYTHONPATH
```

Then start SimulaQron by typing:

```
sh run/startAll.sh
```

in your open terminal. This will start the necessary background processes and setup up the servers needed. It is a good idea to at this point to test if everything is working, see the next step.

4. **Running automated tests**:
   SimulaQron comes with automated tests which can be executed to see if everything is working. Assuming that you went through the previous steps, open another terminal and navigate to the SimulaQron folder. This is to separate the output from the tests and the logging information. In this new terminal, set the same environment variables as above, by typing:

   ```
   export NETSIM=yourPath/SimulaQron
   export PYTHONPATH=yourPath:$PYTHONPATH
   ```

   In this new terminal type

   ```
   sh tests/runTests.sh
   ```

   which will start the tests.

   The tests can take quite some time so be patient. If a test succeeds it will say `OK` and otherwise `FAIL`. Some of the tests are probabilistic so there is a possibility that they fail even if everything is working. So if one of the tests fails, try to run it again and see if the error persists.

5. **Configuring the network**:
   To run protocols in SimulaQron the network of nodes used need to be defined. By default SimulaQron uses five nodes: Alice, Bob, Charlie, David and Eve. For this exercise there is probably no need to change this but it is possible, if you find the need. The files used to configure this can be found in the folder `yourPath/SimulaQron/config` and information on how to do this can be found in the documentation.

3 Exercise instructions

In this exercise your task is to program protocols for the three nodes: Alice, Bob and Eve. Alice and Bob want to generate a share private key such that they can communicate without someone eavesdropping on their messages. Eve is very curious what Alice and Bob are talking about so she tries to intercept their communication.

You will program a protocol for Alice and Bob such that Eve cannot intercept their key without being detected. For simplicity, we will thereby simply assume that the communication from Alice to Bob is "authenticated", that is, Eve is not trying to impersonate Alice or Bob when classical communication is sent. In reality, we do however simply make a direct classical network connection from Alice to Bob that offers no such guarantees. If you want you can change that too!

Furthermore, you will also program an attack by Eve and hopefully see that Alice and Bob notices this. More details on what your task is and which questions you should answer is given in the next sections. Before getting started with the exercises, let us first describe the exercise and run an example.

3.1 The setup

As mentioned, there are three nodes in this setup: Alice, Bob and Eve.

Not important for you to tamper with in this exercise, but maybe useful to know, is that we will run two servers on the nodes labwlled Alice Bob and Eve (localhost in the default configuration), that realize the simulated quantum internet hardware and the CQC (classical quantum combiner) interface. In figure 1 there is a schematic overview of how the communication is realized between the nodes. Firstly, the applications in each node communicate with a CQC (classical-quantum-combiner) server that in turn talk to a SimulaQron server. CQC is an interface between the classical control information in the network and the hardware, here simulated by SimulaQron. The communication between the nodes needed to simulate the quantum hardware is handled by the SimulaQron servers, denoted SimulaQron internal communication in the figure. Note that such communication is needed since entanglement cannot be simulated locally.

The only thing relevant for you doing the exercise, is that SimulaQron comes with a Python library that handles all the communication between the application and the CQC server. In this library, the object `CQCConnection` takes care of this communication from your application to the CQC backend of SimulaQron. This allows your application to issue instructions to the simulated quantum internet hardware, such as creating qubits, making entanglement, etc. Any operation applied to the qubits in this Python library is automatically translated to a message sent to the CQC server, by the `CQCConnection`. For performing quantum operations, you thus only need to understand the Python CQC library supplied with SimulaQron.

In your application protocol, you may wish to send some classical information yourself. For example, Alice might wish to tell Bob which basis she measured in in BB84 QKD. On top of the quantum network there will thus be classical communication between the applications, denoted Application communication in the figure. Such communication would also be present in a real implementation of a quantum network. It is your responsibility as the application programmer to realize this classical communication. One way to do this is via standard socket programming in Python.

However, for convenience we have included a built-in feature in the Python library that realizes this functionality, which have been developed for ease of use for someone not familiar with a client/server setup. This communication is also handled by the object `CQCConnection`. Let assume that Alice wants to send a classical message to Bob and that `Alice` and `Bob` are instances of `CQCConnection` at the respective nodes. For Alice to be able to send a message to Bob, Bob needs to first start a server by running `Bob.startClassicalServer()`. Alice would then apply the method `Alice.sendClassical("Bob",msg)`, where `msg` is the message she wish to send to Bob. The method checks if a socket connection is already opened to Bob and if not opens one and sends the message. Note that if this method is never called, a socket connection is never opened. Both the socket connection on Alice's side and the server on Bob's side can be closed by the methods `closeClassicalChannel` and `closeClassicalServer`, respectively. Alternatively all connections set up by the `CQCConnection`, including the connection to the CQC server, can be closed by the method `close`

We emphasise that to have classical communication between the applications, one is not forced to use the built-in functionality realized by the `CQCConnection`. You can just as well setup your own client/server communication using the method of your preference.

3.2 A simple example

We have implemented a very simple example to help you get started that can be found in the folder `yourPath/SimulaQron/examples/programming_q_network`. This simple scheme is definitely not secure QKD and your task is to improve it. In the example, Alice will generate a random bit k which will be used as a shared key between Alice and Bob. Alice encodes the key in a qubit by applying X^k to the qubit $|0\rangle$ and sends this qubit to Bob. In the example, Alice actually sends the qubit to Eve (representing the channel), and Eve passes it on to Bob. Bob then measures the qubit and can recover

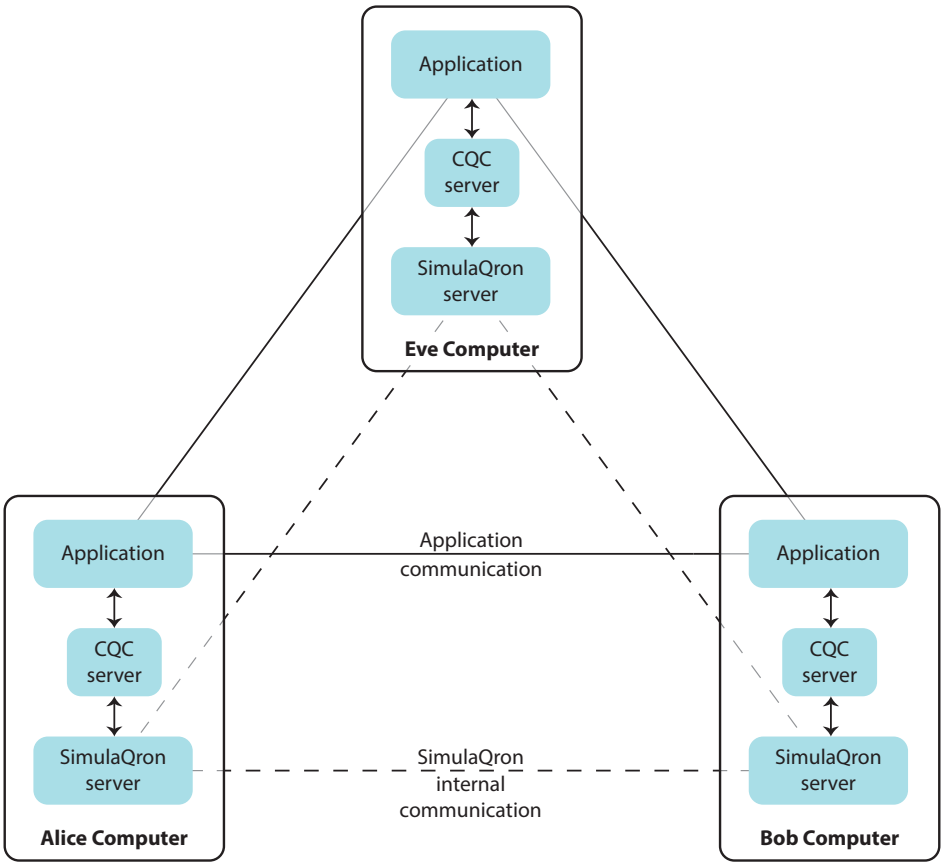

Figure 1: A schematic overview of the communication in a quantum network simulated by SimulaQron. The simulation of the quantum hardware at each node is handled by the SimulaQron server. Communication between the SimulaQron servers are needed to simulate the network, for example to simulate entanglement. Opting for this method enables a distributed simulation, i.e. the computers in the figure can be physically different computers. The CQC servers provide an interface between the applications running on the network and the simulated hardware. Finally the applications can communicate classically, as they would do in a real implementation of a quantum network.

the key. Evidently, here Eve can also measure the qubit on the way to Bob to get the key. Importantly, Eve can do this without disturbing the state of the qubit, by measuring in the standard basis.

Let us now look more in detail on the actual code. How to run the code is described in the next section. In the folder `yourPath/SimulaQron/examples/programming_q_network` there are a few files but the ones containing the actual code is `aliceTest.py`, `bobTest.py` and `eveTest.py`.

3.2.1 Alice's code

We will first look over the code for Alice. In the first part of the code Alice generates a random bit and encodes this in a qubit, as seen below:

```
# initialize the connection
Alice=CQCConnection("Alice")

#generate a key
k=random.randint(0,1)
```

```
# create a qubit
q=qubit(Alice)

# encode the key in the qubit
if k==1:
        q.x()
```

First an object called `CQCConnection` is initialized. The `CQCConnection` is responsible for all the communication between the node Alice and SimulaQron and also to other nodes, as described in the previous section. Then a `qubit` object is initialized, taking the `CQCConnection` as argument. When an operation is applied to a `qubit`, the `CQCConnection` is used to communicate with SimulaQron. Operations can be applied to the qubit by for example writing `q.X()`, `q.H()` or `q1.cnot(q2)`, where `q1` and `q2` are different `qubit` objects initialized with the same `CQCConnection`. More useful commands are given in the section 3.4.

Alice will now send the qubit to Bob. As discussed above, all the quantum communication will go through Eve (since we cannot be sure it does not).

```
#Send qubit to Bob (via Eve)
Alice.sendQubit(q,"Eve")
```

To send a qubit the `CQCConnection` is called with the method `sendQubit` which takes the qubit as argument and the name of the node to send it to.

In the last part of Alice's code she encodes a message $m = 0$ by computing $m + k \pmod 2$ and sends this classical message to Bob. This is done by calling the method `sendClassical` which takes as argument the node to send the message to and the message itself. The message can either be a integer between 0 and 255 or a list of such integers. Finally the connections are closed by calling `Alice.close()`.

```
# Encode and send a classical message m to Bob
m=0
enc=(m+k)%2
Alice.sendClassical("Bob",enc)

print("Alice send the message m={} to Bob".format(m))

# Stop the connections
Alice.close()
```

3.2.2 Bob's code

We will now take a look what happens on Bob's side. Bob's code is given as follows:

```
# Initialize the connection
Bob=CQCConnection("Bob")
Bob.startClassicalServer()

# Receive qubit from Alice (via Eve)
q=Bob.recvQubit()

# Retreive key
k=q.measure()

# Receive classical encoded message from Alice
enc=Bob.recvClassical()[0]

# Calculate message
m=(enc+k)%2

print("Bob retrived the message m={} from Alice.".format(m))
```

```
# Stop the connection
Bob.close()
```

In the first part of the code a `CQCConnection` is again initialized. What is also done here is to execute the command `Bob.startClassicalServer()`. This starts up a server which allows Bob to receive classical messages from other nodes. Bob receives messages by calling the method `recvClassical`, which is done after he received and measured the qubit from Alice. In the end Bob decrypts the message he received by using the key he measured and finally closes the connections.

3.2.3 Eve's code

We already seen all commands used by Eve in the codes of Alice and Bob. Eve opens `CQCConnection`, receives a qubit, sends the qubit to Bob and closes the connections.

```
# Initialize the connection
Eve=CQCConnection("Eve")

# Receive qubit from Alice
q=Eve.recvQubit()

# Forward the qubit to Bob
Eve.sendQubit(q,"Bob")

# Stop the connection
Eve.close()
```

3.3 Running the example

Now that we have seen what the code of Alice, Bob and Eve does it is time to run it and see what happens. It is again a good idea to have **two** terminals to separate the output from the protocols and the logging information from the background processes. If this is not already up and running, start the background processes in a terminal by following step 3 in section 2. Open a **new** terminal and navigate to the folder `yourPath/SimulaQron/examples/programming_q_network`. As in section 2, make sure the environment variables are set by for example typing in the new terminal:

```
export NETSIM=yourPath/SimulaQron
export PYTHONPATH=yourPath:$PYTHONPATH
```

To run the example, type:

```
sh run_example.sh
```

in the new terminal. In the terminal running the background processes there will be a lot of logging information describing what happens in the background, which is not important at this point. In the terminal where you executed the example, there will also be some information describing the communication between the nodes and the background processes. These outputs can in fact be turned of by for example typing `q.X(print_info=False)` or similarly for other commands. Two important outputs of the example concern the message sent from Alice to Bob. Hopefully the message Bob received is the message Alice actually sent.

Now it is up to you to improve the code. In the next sections there are some instructions and questions to get going. First we list some commands that can be useful to realize this.

3.4 Useful commands

Here we list some useful methods that can be applied to a `CQCConnection` object or a `qubit` object below. The `CQCConnection` is initialized with the name of the node (`string`) as an argument.

`CQCConnection.`:

- `sendQubit(q,name)` Sends the qubit q (`qubit`) to the node name (`string`).
  Return: `None` .
- `recvQubit()` Receives a qubit that has been sent to this node.
  Return: `qubit` .
- `createEPR(name)` Creates an EPR-pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ with the node name (`string`).
  Return: `qubit` .
- `recvEPR()` Receives qubit from an EPR-pair created with another node (that called `createEPR`).
  Return: `qubit` .
- `sendClassical(name,msg)` Sends a classical message msg (`int` in range(0,256) or list of such `int` s) to the node name (`string`). Opens a socket connection if not already opened.
  Return: `None` .
- `startClassicalServer()` Starts a server that can receive classical messages sent by `sendClassical` .
  Return `None` .
- `recvClassical()` Receives a classical message sent by another node by `sendClassical` .
  Return `bytes` .

Here are some useful commands that can be applied to a `qubit` object. A `qubit` object is initialized with the corresponding `CQCConnection` as input and will be in the state $|0\rangle$.

`qubit.`:

- `X()` , `Y()` , `Z()` , `H()` , `K()` , `T()` Single-qubit gates.
  Return `None` .
- `rot_X(step)` , `rot_Y(step)` , `rot_Z(step)` Single-qubit rotations with the angle $\left(\text{step} \cdot \frac{2\pi}{256}\right)$.
  Return `None` .
- `CNOT(q)` , `CPHASE(q)` Two-qubit gates with q (`qubit`) as target.
  Return `None` .
- `measure(inplace=False)` Measures the qubit and returns outcome. If inplace (`bool`) then the post-measurement state is kept afterwards, otherwise the qubit is removed (default).
  Return `int` .

Note: A qubit simulated by SimulaQron is only removed from the simulation if it is measured (inplace=False) or if it is sent to another node. If you therefore run a program multiple times which generates qubits without measuring them you will quickly run out of qubits that a node can store. If this happens you need to restart the background processes to reset the simulation. This can be done by running `sh run/startAll.sh` again, as in step 3 in section 2.

---

4 Exercises

The goal of this exercise, will be to program some of the steps towards implementing QKD in simulation!

- Extend the program above to let Alice send a random BB84 state to Bob, and let Bob measure it in a random basis.
- Extend the program above to let Alice send n random BB84 states to Bob.
- Extend your program to let Alice and Bob extract one bit of key $k \in \{0, 1\}$:
  - Alice and Bob determine when they measured in the same basis. Let x denote Alice's string where they measure in the same basis.
  - Alice now picks a simple extractor for 1 bit of key k: she randomly xors the bits in x. That is, $k = \text{Ext}(x, r) = x \cdot r = \sum_{j=1}^{m} x_j r_j \mod 2$, where m is the length of x.
  - Alice sends the seed to Bob who uses it to obtain the key as well.
- Implement some "attack" on the channel (i.e. in Eve). Let Alice and Bob estimate the error rate in the standard and hadamard basis. What do you observe?

You can team up with others! In this case, please submit on edX for each participant and include a group name in your submission to the competition (see below).

5 Guidelines for peer review

When you peer review other people's submissions a few tips and guidelines:

- Completeting the exercises above is sufficient for full grade. Very special projects can take part in the competition and do not need to get extra or more credit in your peer review.
- When judging projects, please note that you are executing code written by other people on your computer. If you execute the code to test it, please use due diligence to make sure that what you are running safe to execute. We do not take any responsibility for running other people's code on your computer.
- Team projects should be judged the same as individual projects.

6 And beyond... the competition!

You may submit any protocol you wish to our competion: a beautiful solution to the exercises above, extending them in any way you find useful, making the protocol device independent, implementing any other quantum internet protocol,....

We will hand out several prizes in our competition and showcase the best projects online. See the website [www.simulaqron.org] for details on the different prizes - including the best prize for an individual project that can win an internship with us here at QuTech in the summer!

If you want to participate in the competition, we ask that - in addition to submitting your project on edX - also fill out the following WebForm [https://goo.gl/forms/8TegAvM0zxCSt4bW2] to enter in the competition. The competition is optional and not a requirement for doing the edX assignment.

Your submission should be a ZIP file which you will upload to edX containing the following:

1. Some information we will use on our website when showcasing the best and winning projects. Please include this with your submission as a txt format in the folder INFO:
   - Your real name: First Name, Last Name

- Name of group (if applicable)
- Email address
- Age (not relevant for prize selection)
- Occupation (not relevant for prize selection)
- School/University/Company (not relevant for prize selection)
- Title of your project
- Abstract describing your project (max. 200 words, will be used on the competition website if you win, or make the top projects list)
- Your edX username
- 300x300 Picture of yourself/your group (will be used on website if you win, or make the top projects list). Called yourname.PNG (PNG format only)

2. In addition to all the code making up your submission, please include the following files into a folder called "DESCRIPTION":

   - A file called README.txt describing all files in your submission.
   - A PDF describing the objective, summary and design overview of your submission (max. 2 pages)
   - A script call run_submission.sh which will execute your project.