Programming a quantum network

Axel Dahlberg Stephanie Wehner

July 15, 2018

1 Introduction

In this project, you will program your own quantum protocol! AXEL: Write new introduction of the project

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

- SimulaQron Website: [www.simulaqron.org]
- SimulaQron on Github: [https://github.com/StephanieWehner/SimulaQron] AXEL: Changes this?
- Code for this competition in folder: [SimulaQron/competition]
- Arxiv Paper describing the inner workings of SimulaQron: [https://arxiv.org/abs/1712.08032]
- Google form for entering the competition [https://goo.gl/forms/SQdmru1weETWdv3i2]

1.1 Important dates and deadlines

AXEL: Changes dates

- Start of competition: 19 July 2018
- Deadline for submitting your project: 19 October 2018
- Announcement of winner(s): 19 December 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: AXEL: Change link

   ```
   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 competition 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.

On can start SimulaQron using different nodes than what is configured in the config-files. To start SimulaQron using a network consisting of the nodes Adrian, Beth and Claire, simply type

```
sh run/startAll.sh Adrian Beth Claire
```

Note that this will actually change the config-files to use the nodes Adrian, Beth and Claire. Also, note that this functionality can only be used when running all the nodes simulated by SimulaQron on a single computer, as the hostname is by default set to `localhost` for all nodes and the port numbers are automatically chosen. A network with ten nodes n0 to n9 can be started by typing

```
sh run/startAll.sh n0 n1 n2 n3 n4 n5 n6 n7 n8 n9
```

3 The Challenge

You're challenge will be implement a non-local game using SimulaQron. We have provided a basic example that realize a quantum strategy to win the Mermin-Peres magic square game described below. You're task is now to improve on this.

3.1 Mermin-Peres magic square game

The Mermin-Peres magic square game is a game played by two players: Alice and Bob. In each round of the game Alice and Bob will receive a challenge from a judge. Depending on their answers, they will either both win or both lose. The players are allowed to agree on a strategy prior to the game, however once the game has started they cannot communicate anymore.

The judge will challenge Alice to provide a row from a 3×3 binary matrix and Bob to provide a column. Before the game starts, neither Alice or Bob knows what the row or column will be and furthermore Alice will not know which column Bob is requested to provide and vice versa. Lets assume that the request is for the rth row and the cth column and that Alice provides the row (a_{r0}, a_{r1}, a_{r2}) and Bob the column (b_{0c}, b_{1c}, b_{2c}). Then Alice and Bob wins the round of the game if

- They agree on the intersecting element, i.e. $a_{rc} = b_{rc}$.
- The row provided by Alice has an even parity, i.e. $a_{r0} + a_{r1} + a_{r2} = 0 \pmod 2$.
- The column provided by Bob has an odd parity, i.e. $b_{0c} + b_{1c} + b_{2c} = 1 \pmod 2$.

1	1	0
1	0	1
1	0	?

If one or more of the above restrictions is not satisfied then Alice and Bob lose the round.

It can be shown that the maximal winning probability for any classical strategy for the Mermin-Peres magic square game is $\frac{8}{9}$. Surprisingly, if Alice and Bob have access to entanglement, they can in fact win the game with probability 1.

AXEL: Describe quantum strategy

3.2 The setup

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

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 labelled Alice and Bob (localhost in the default configuration), that realize the simulated quantum internet hardware and the CQC (classical quantum combiner) interface. See item 5 of section 2 for how to setup a network with different nodes. 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 send a message to Bob, Alice will simply apply the method `Alice.sendClassical("Bob",msg)`, where `msg` is the message she wish to send to Bob. The method opens a socket connection to Bob, sends the message and the closes the connection again. Note that if this method is never called, a socket connection is never opened. Bob receives the messages by `Bob.recvClassical()`.

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.

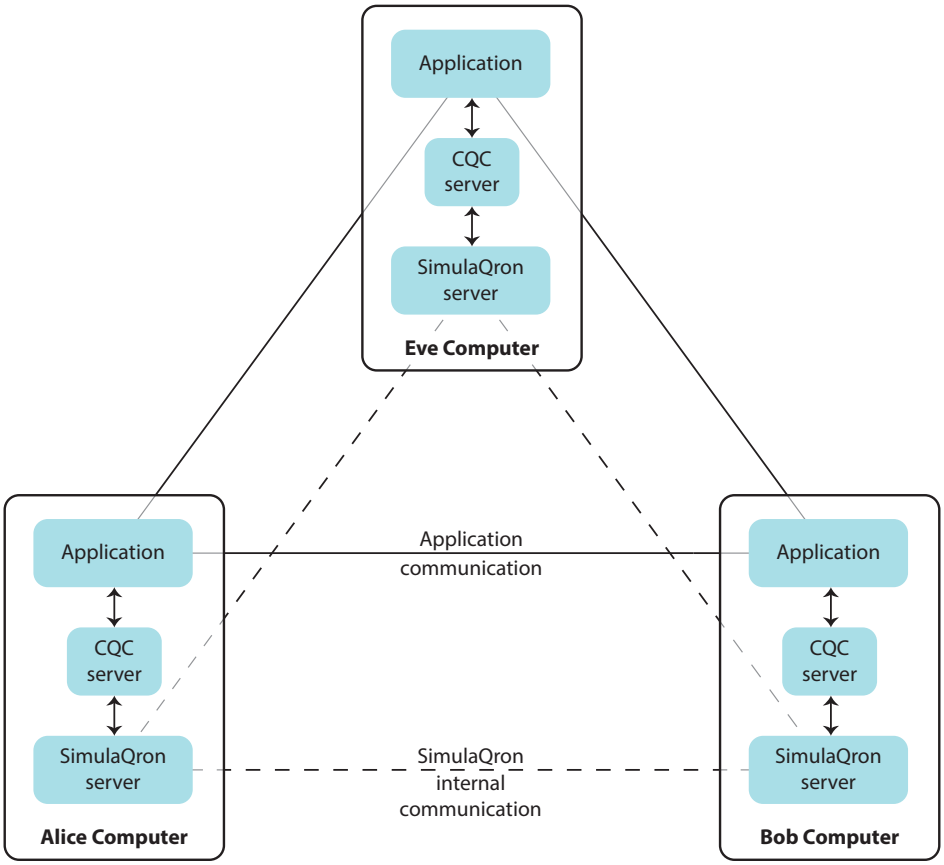

Figure 1: A schematic overview of the communication in a quantum network simulated by SimulaQron. The simulation of the quantum hardware at each node is handled by the SimulaQron server. Communication between the SimulaQron servers are needed to simulate the network, for example to simulate entanglement. 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.

3.3 A simple example

AXEL: Change to the new 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`.

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/competition/base_example` there are a few files but the ones containing the actual code is `aliceTest.py`, `bobTest.py`.

3.3.1 Alice's code

We will first look over the code for Alice. AXEL: Split up and describe the code

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

      # Create EPR pairs
      q1 = Alice.createEPR("Bob")
      q2 = Alice.createEPR("Bob")

      # Make sure we order the qubits consistently with Bob
      # Get entanglement IDs
      q1_ID = q1.get_entInfo().id_AB
      q2_ID = q2.get_entInfo().id_AB

      if q1_ID < q2_ID:
              qa = q1
              qc = q2
      else:
              qa = q2
              qc = q1

      # Put the qubits in the correct state (|++> + |-->)
      qa.H()
      qc.H()

      # Get row
      row = random.randint(0,2)

      # Perform the three measurements
      if row == 0:
              m0 = measure_XI(qa, qc)
              m1 = measure_XX(qa, qc, Alice)
              m2 = measure_IX(qa, qc)
      elif row == 1:
              m0 = (measure_XZ(qa, qc, Alice) + 1) % 2
              m1 = measure_YY(qa, qc, Alice)
              m2 = (measure_ZX(qa, qc, Alice) + 1) % 2
      elif row == 2:
              m0 = measure_IZ(qa, qc)
              m1 = measure_ZZ(qa, qc, Alice)
              m2 = measure_ZI(qa, qc)
      else:
              raise RuntimeError("Invalid row")

      # Print measuremnent outcomes
      print("\n")
      print("===========================")
      print("App {}: row is:".format(Alice.name))
      for _ in range(row):
              print("(___)")
      print("({}{}{})".format(m0, m1, m2))
      for _ in range(2-row):
              print("(___)")
```

```
print("==========================")
print("\n")

# Clear qubits
qa.measure()
qc.measure()

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

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. AXEL: Describe createEPR... 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 3.5.

3.3.2 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")

# Create EPR pairs
q1=Bob.recvEPR()
q2=Bob.recvEPR()

# Make sure we order the qubits consistently with Alice
# Get entanglement IDs
q1_ID = q1.get_entInfo().id_AB
q2_ID = q2.get_entInfo().id_AB

if q1_ID < q2_ID:
        qb=q1
        qd=q2
else:
        qb=q2
        qd=q1

# Put the qubits in the correct state (|++> + |-->)
qb.H()
qd.H()

# Get col
col = random.randint(0,2)

# Perform the three measurements
if col == 0:
        m0 = measure_XI(qb, qd)
        m1 = (measure_XZ(qb, qd, Bob) + 1) % 2
        m2 = measure_IZ(qb, qd)
elif col == 1:
        m0 = measure_XX(qb, qd, Bob)
        m1 = measure_YY(qb, qd, Bob)
        m2 = measure_ZZ(qb, qd, Bob)
elif col == 2:
        m0 = measure_IX(qb, qd)
        m1 = (measure_ZX(qb, qd, Bob) + 1) % 2
        m2 = measure_ZI(qb, qd)
else:
        raise RuntimeError("Invalid row")
```

```
print("\n")
print("==========================")
print("App {}: column is:".format(Bob.name))
print("(" + "_"*col + str(m0) + "_"*(2-col) + ")")
print("(" + "_"*col + str(m1) + "_"*(2-col) + ")")
print("(" + "_"*col + str(m2) + "_"*(2-col) + ")")
print("==========================")
print("\n")

# Clear qubits
qb.measure()
qd.measure()

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

In the first part of the code a `CQCConnection` is again initialized.

3.4 Running the example

AXEL: Change to the new 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/competition/base_example`. 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. AXEL: Described output

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.

3.5 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`).
  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 $(\text{step} \cdot \frac{2\pi}{256})$.
  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

AXEL: Change or remove exercises The goal of this exercise, will be to program some of the steps towards implementing...

- Exercise 1

You can team up with others! In this case, please include a group name in your submission to the competition (see below).

5 Submission instructions

AXEL: Look over instructions

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 you fill out the following WebForm [https://goo.gl/forms/SQdmru1weETWdv3i2] to enter in the competition.

Your submission should be a ZIP file which you will upload in the google form above.

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.