Programming a quantum network

Axel Dahlberg Stephanie Wehner

August 8, 2018

1 Introduction

In this competition, you will program your own quantum protocol! Do you find yourself always losing when you play the card-game bridge? Surprisingly, you can do better if you have access to entanglement. See for example the following Youtube clip: www.youtube.com/watch?v=1YJs8AWykzs. This is the case with many *non-local* games. Your challenge for this competition will be to explore, play with and implement such games using the freely available simulator SimulaQron.

Here, some useful links and then it's time to get started!

- SimulaQron Website: [www.simulaqron.org]
- SimulaQron on Github: [github.com/StephanieWehner/QI-Competition2018]
- Code for this competition in folder: [SimulaQron/competition]
- Arxiv Paper describing the inner workings of SimulaQron: [arxiv.org/abs/1712.08032]
- Google form for entering the competition [goo.gl/forms/SQdmru1weETWdv3i2]

Note: Please make use of the GitHub repository above and NOT the standard SimulaQron repository.

1.1 Important dates and deadlines

- 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 (IMPORTANT), 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 choose 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/QI-Competition2018
   ```

   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/QI-Competition2018/SimulaQron
   ```

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

```
export NETSIM=yourPath/QI-Competition2018/SimulaQron
export PYTHONPATH=yourPath/QI-Competition2018:$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. Note that this will in fact kill all Python processes containing the strings *Test*, *setup* or *start* on your computer, before SimulaQron. This is so that SimulaQron can easily be restarted by running `sh run/startAll.sh`.

Hint: If something goes wrong with running SimulaQron and you need to kill the processes running. The easiest way to do this is to simple type `pkill python` in any terminal. Note that this will kill all of your Python processes.

Note: The default setting in SimulaQron is to only print warnings or errors. If you also want to see what is happening you can turn on the debugging output by changing the field `loglevel` from `warning` to `debug` in the file `yourPath/QI-Competition2018/SimulaQron/config/settings.ini`. You need to restart SimulaQron for the changes to take effect. A lot of information will then be printed, so it is not recommend to have debugging on unless you need to.

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, navigate to the SimulaQron folder. Make sure that `loglevel` in `yourPath/QI-Competition2018/SimulaQron/config/settings.ini` is set to `warning` before running the tests! In the terminal type

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

   which will start the tests.

   Depending on your computer, the tests can take a lot of 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/QI-Competition2018/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
```

For the base example provided below, you need at least the nodes Alice and Bob to be running.

3 The Challenge

In this edition of the programming competition, we will explore how the effects of entanglement can allow to remote players to coordinate their actions instantaneously! We will explore such coordination power using the language of so-called non-local games, of which you can find a very simple one below. Your challenge in this competition will be to implement this game in software, so it can run quantum networks - or for now - the SimulaQron simulator.

You are free to expand on this game in any direction (including programming a different application for quantum networks alltogether :-)) where here we will give you some initial ideas:

- A graphical interface to really play the game.
- An implementation of other non-local games: for example for playing bridge (see above)
- A library to play all kinds of non-local games easily.
- Introducing noise into the simulations, and seeing how your game performs....
- ... the sky is the limit.

To get started, we will provide you with the so-called Mermin game, including software to help you get started. We provide code snippets that allow you to run this, without prior knowledge of anything quantum - of course you are also welcome to expand on those :-)

Let us first describe the general form of a non-local games. In a non-local game, some number of players gets an input (i.e. challenges) each and their task is to give outputs (i.e. answers). The game has some rules that say when the answers are good given the challenges they received: that is, whenever the answers are winning answers.

The challenges can arrive at any time. To prepare, the players can agree on any strategy before the game starts. However, they must give answers immediately, in particular before being able to communicate with each other. Also, each player will only learn their own challenge and not the challenge of the other player, so each must determine by him/herself what to answer! When applying these games to achieve coordination, we can imagine that there is a judge who poses the challenges and evaluates the input. More practically, the challenges may be posed by external events, such as the players playing the card game bridge and drawing certain cards from the deck (i.e. the challenge). Here, good answers are good moves that give the players an edge in winning the bridge game.

Figure 1 gives an illustration of a general non-local game in the case of two players, which we like to call Alice and Bob.

It turns out that players can coordinate better using entanglement. In the language of games, this means that players can win with a higher probability if they make use of entanglement, compared to the optimal classical strategy. For some games, players with access to entanglement can always win. That is, their winning probability is 1, even though this is impossible for any classical strategy. One such game is the Mermin-Peres magic square games, described below.

We have provided a basic example that realize a quantum strategy to win this game. You're task is now to improve on this. To get you started there are a few suggested exercises in section 5. However, what you want to improve and what you want your submission to be, is completely up to you!

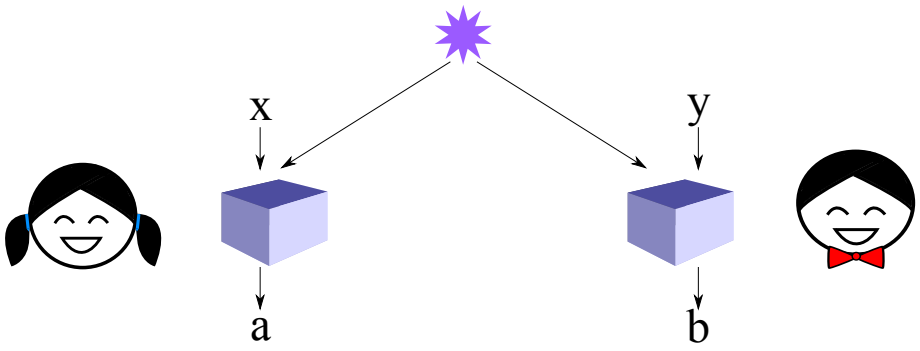

Figure 1: A non-local game with two players, Alice and Bob. We will use variables x and y to denote the challenges to the players, and a and b to denote their answers. The players must give their answers immediately, without communicating with each other - i.e. instantaneously. In the quantum regime, Alice and Bob can use entanglement shared between them (star with arrows pointing to the players) in order to give them an edge and coordinate much better than what is possible classicallly!

3.1 Mermin-Peres magic square game

The Mermin-Peres magic square game is a game played by two players: Alice and Bob. They like to play this game a lot and we will play multiple instances, i.e. rounds of the game. 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: i.e. they must try and coordinate without communication, even instantaneously.

The judge will challenge Alice to "color in" a row from a 3×3 binary matrix - i.e. a checker board - 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. We will label the rows using the numbers $r \in \{0, 1, 2\}$ and the columns using the numbers $c \in \{0, 1, 2\}$. That is, $r = 0$ is the first row, $r = 1$ the second row, $r = 2$ the third row and similarly for the columns. Each entry of this board can be either '0' (say, red), and '1' say blue.

When Alice is challenged to color in row r, she must provide values - i.e. the colors - for that row as an answer. Her answer is valid if the number of 1's in that row is even. In terms of the colors this means that either none of the squares in that row are colored blue, or exactly two of the squares are colored blue. More formally, if we use variables (a_{r0}, a_{r1}, a_{r2}) to denote her answer for challenge r, then the following condition must hold:

- The row provided by Alice has an even parity, i.e. $a_{r0} + a_{r1} + a_{r2} = 0 \pmod 2$.

Bob is challenged to color in column c, so he must provide values - i.e. the colors - for that column as answer. His answer is valid, is the numbr of 1's in the column is odd. In terms of colors this means that either exactly one of the squares in that column is colored blue, or all are colored blue. More formally, if we use variables (b_{0c}, b_{1c}, b_{2c}) to denote his answer for challenge c, then the following condition must hold:

- The column provided by Bob has an odd parity, i.e. $b_{0c} + b_{1c} + b_{2c} = 1 \pmod 2$.

So far, it would have been easy for Alice and Bob to win the game: Alice could simply return a row of all red squares, and Bob could return a column of all blue squares. The tricky part is that the rules of the game also demand that Alice and Bob agree on the intersection. For example, if Alice is challenged to color in the first row, and Bob the first column then they must give the same color for the first square - i.e. the square that appears in both Alice's row and Bob's column. More formally, the rules demand that

- They agree on the intersecting element, i.e. $a_{rc} = b_{rc}$.

Whenever Alice and Bob's answers satisfy all of these conditions, then the judge rules that they win this round of the game. Otherwise they lose.

It can be shown that the maximal winning probability for any classical strategy for the Mermin-Peres magic square game is $\frac{8}{9}$. A classical strategy means no-entanglement, but of course Alice and Bob can try and be smart. For example, they could agree before the game starts on the following coloring of the square, which would allow them to give consistent answers for many combinations of challenges, and trouble only arises if Alice is asked for the third row and Bob for the third column:

1	1	0
1	0	1
1	0	?

Surprisingly, using a quantum strategy, Alice and Bob can do better! A quantum strategy means that they can establish entanglement before the game starts, and then use this entanglement to give answers instantaneously that allows them to win with a higher probability. In the example, of the game, it turns out that Alice and Bob can win the game all the time! That is, they can win with probability 1. This game thus provides an example in which Alice and Bob can perfectly coordinate their actions without communicating using quantum entanglement.

3.1.1 An optimal quantum strategy

We will now explain to you what the quantum strategy of Alice and Bob is exactly. If you are unfamiliar with quantum information, you can safely skip this section as we provide the necessary code for you that you can build upon. However if you want to extend and explore we encourage you to read.

As mentioned, Alice and Bob will establish entanglement ahead of time. For the game above, Alice and Bob will each have two qubits in the state described below. We also refer to this state as two EPR-pairs. Alice and Bob will now each choose a quantum measurement to perform on their qubits, that depends on the row and column. That is, for each challenge they may choose a specific measurement.

Let us look at the state. Formally, it can be written as

$$\left(\frac{1}{\sqrt{2}}\left|0\right\rangle_a\left|0\right\rangle_b + \frac{1}{\sqrt{2}}\left|1\right\rangle_a\left|1\right\rangle_b\right) \otimes \left(\frac{1}{\sqrt{2}}\left|0\right\rangle_c\left|0\right\rangle_d + \frac{1}{\sqrt{2}}\left|1\right\rangle_c\left|1\right\rangle_d\right), \tag{1}$$

where qubits a and c are held by Alice and b and d by Bob. See the base example below for how to create such entangled states.

Let us now look at the measurements. There exist a set of measurements that Alice and Bob can perform of their qubits such they win the game with certainty. One such set if given in table 1.

If you do not know how measurements work in quantum information, or if you do not know how to perform such parity measurements in the above table, do not worry, there is a function in SimulaQron that can be used to perform these measurements. This function is called `parity_meas` and is described in the next section. If you do want to know more about the basics of quantum information you can look at week 0 of the edX course Quantum Cryptography at www.edx.org/course/quantum-cryptography-caltechx-delftx-qucryptox-0.

For those that do know about measurement in quantum information, note that all elements for a given row (or column) commute. This implies that these observables can be simultaneously measured, implictly giving rise to one big measurement for each row (or column). For row $r = 0$, Alice measures $X \otimes I$ to determine the value of the first square, $X \otimes X$ to determine the value of the second square, $I \otimes X$ to determine the last square. These measurements return $+1$ which we will label '0' (red) and '-1' which we will map to '1' (blue). Similarly, if Bob's challenge is to specify the first column $c = 0$, then he measures $X \otimes I$ to determine the first square in his column, $-X \otimes Z$ to determine the second

$+X \otimes I$	$+X \otimes X$	$+I \otimes X$
$-X \otimes Z$	$+Y \otimes Y$	$-Z \otimes X$
$+I \otimes X$	$+Z \otimes Z$	$+Z \otimes I$

Table 1: Measurement operators that can be used to win Mermin-Peres magic square game with unit probability. If there is a "-" in front, it means that the measurement outcome should be flipped.

square where the "-" indicates that he inverts the outcome (i.e. 0 becomes 1 and vice versa), and $I \otimes X$ to determine the last square.

Can you see that, if Alice and Bob performs the measurement corresponding to the rows and column that they are given, Alice will obtain an even number of +1 measurement outcomes and Bob an odd number? (Hint: What happens if you multiply the measurement operators?)

3.2 Getting started without QM

In the base example code we have provided the correct measurements when the requested row is 0 and the requested column is 0. Check out Section 4.2 for details. This uses the function `parity_meas` with which you can implement the measurements marked in the table above. Your first exercise will be to implement the rest of the cases! You may want to get started by changing Bob's column to be $c = 1$, try and adapt the measurements!

3.3 Further reading

Below is a list of references that you can use to expand your submission or for further reading if you are curious:

- Wikipedia page on the Mermin-Peres magic square game and others: en.wikipedia.org/wiki/Quantum_pseudo-telepathy
- Original paper on the Mermin-Peres magic square game: [6].
- A survey on non-local games: [4].
- Generalization of the magic square game to larger matrices: [5].
- Generalization to games on graphs: [1].
- Some non-local games with more players: [2].
- Extensive survey given quantum background on Bell inequalities - and such games: [3].

4 How to program SimulaQron

This section gives a brief explanation on how to write application programs for a quantum network using SimulaQron. First, an overview of the setup of a network in SimulaQron is given 4.1. A basic example on how to program the Mermin-Peres magic square game is provided in 4.2. Finally in 4.4 is a list of useful commands.

4.1 The setup

In the standard version of the Mermin-Peres magic square game, there are two nodes: Alice and Bob.

Not necessarily important during this competition, 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 2 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.

4.2 Base example

We have implemented a base example to help you get started that can be found in the folder `yourPath/QI-Competition2018/competition/base_example`.

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/QI-Competition2018/competition/base_example` there are a few files but the ones containing the actual code is `aliceTest.py`, `bobTest.py`.

4.2.1 Alice's code

We will first look over the code for Alice, step-by-step.

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

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. When an operation is applied to a `qubit`, the `CQCConnection` is used to

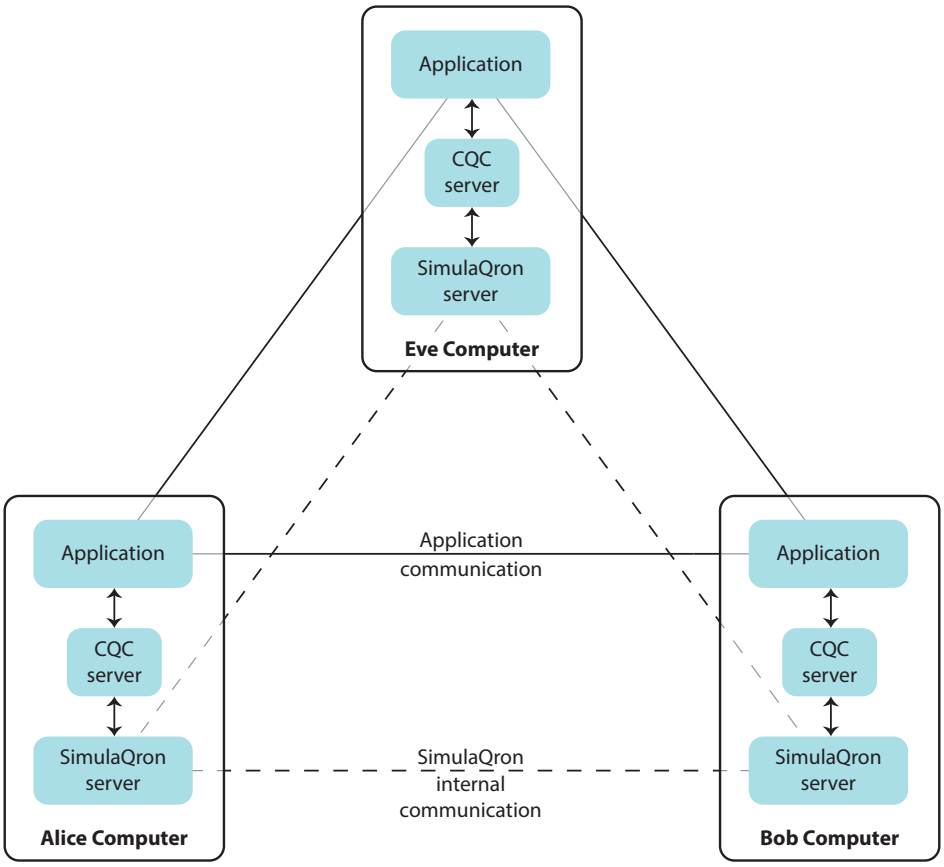

Figure 2: A schematic overview of the communication in a quantum network simulated by SimulaQron. The simulation of the quantum hardware at each node is handled by the SimulaQron server. Communication between the SimulaQron servers are needed to simulate the network, for example to simulate entanglement. 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.

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 4.4. Next, Alice creates two EPR-pairs (entangled pairs) with Bob.

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

The call to create EPR pairs is asymmetric and initialized by one of the nodes. As we will see below, Bob will answer to this request by calling the method `recvEPR`. When the above code has been executed (together with the code on Bob's side), Alice and Bob have two qubits each, which are pairwise entangled. By design, SimulaQron will not guarantee that commands are executed in the order they are called, as in a real network. Because of this, we cannot assume that Alice's qubit `q1` is entangled with Bob's qubit `q1`. To consistently order the qubits between the nodes, we will make use of the provided *entanglement information*, and in particular the sequence ID. There are more things

contained in the *entanglement information*, but for now you only need to know about the sequence ID. The sequence ID is an integer, which is unique for every EPR created between Alice and Bob[1].

We will re-name the qubits such that `qa` is Alice's qubit with the lowest sequence ID and `qc` the other qubit held by Alice. Similarly for Bob and the qubits `qb` and `qd`. We then have that Alice's qubit `qa` is entangled with Bob's qubit `qb` and similarly for qubits `qc` and `qd`.

```
# Make sure we order the qubits consistently with Bob
# Get sequence 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
```

Next Alice receives her challenge from the judge in the game, see section 3.1. Currently this will always be row 0. Your task will be to extend this to make the game more exciting.

```
# Get row
row = 0
```

Depending on the challenge from the judge, Alice will perform certain actions. The correct measurements for the case where the row is 0, is already implemented below. You will need to fill in the other two cases.

Some of the measurements that Alice needs to perform are so called parity measurements. Do not worry if you do not know what a parity measurement is, you can simply use the function `parity_meas`.

```
# Perform the three measurements
if row == 0:
        m0 = parity_meas([qa, qc], "XI", Alice)
        m1 = parity_meas([qa, qc], "XX", Alice)
        m2 = parity_meas([qa, qc], "IX", Alice)
else:
        m0 = 0
        m1 = 0
        m2 = 0
```

Next the measurement outcomes are printed.

```
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")
```

To not leave qubits in the simulation that will take up memory we measure these out. Using the argument `inplace=False` will perform a destructive measurement and remove the qubit from the simulation.

```
# Clear qubits
qa.measure(inplace=False)
qc.measure(inplace=False)
```

[1]The sequence ID is not globally unique in the network, however together with the information that the entanglement is between the nodes Alice and Bob, this effectively gives a globally unique entanglement ID.

Finally, we close the connection to the simulation backend.

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

4.2.2 Bob's code

We will now take a look what happens on Bob's side. The code is essentially the same as for Alice, except that Bob calls the method `recvEPR` instead of `createEPR` as mentioned above. 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 sequence 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

# Get col
col = 0

# Perform the three measurements
if col == 0:
        m0 = parity_meas([qb, qd], "XI", Bob)
        m1 = parity_meas([qb, qd], "XZ", Bob, negative=True)
        m2 = parity_meas([qb, qd], "IZ", Bob)
else:
        m0 = 0
        m1 = 0
        m2 = 0

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()
```

4.3 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. If this is not already up and running, start the background processes in a terminal by following step 3

in section 2. Navigate to the folder `yourPath/QI-Competition2018/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/QI-Competition2018/SimulaQron
export PYTHONPATH=yourPath/QI-Competition2018:$PYTHONPATH
```

To run the example, type:

```
sh run_example.sh
```

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.

4.4 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`).
  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.

---

5 Exercises

Below are a few exercises and suggestions on how to get started. As mentioned, it is completely up to you what you choose your submission to be. Have fun and be creative!

- Complete the base example such that the correct measurements will be done for all combinations of rows and columns.
- Extend the base example such that rows and columns other than 0 can actually be requested by Alice and Bob. Do you want to randomize this or maybe be able to input this yourself to the game?
- Make it a real game! Let someone playing the game actually choose the measurements to be performed.
- Optimize the code?
- Make a graphical interface?
- Generalize to other non-local games?
- Include noise and see what effect this has?
- Implement a game of Bridge and include the option to use entanglement to be play better.

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

6 Submission instructions

You may submit any protocol you wish to our competition: a beautiful solution to the exercises or suggestions above, extending them in any way you find useful, 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)
    - 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.

7 Discussions and issues

If you have any issues or problems use the issue tracker on GitHub, do not send issues by email! This allows us to keep track of issues and allow anyone to contribute in solving these. Please use the issue tracker for the competition-repository github.com/StephanieWehner/QI-Competition2018

References

[1] Alex Arkhipov. Extending and Characterizing Quantum Magic Games. pages 1–20, 2012.

[2] Gilles Brassard, Anne Broadbent, and Alain Tapp. Recasting Mermin's multi-player game into the framework of pseudo-telepathy. *Quant. Inf. Comp.*, 5(7), 2005.

[3] N. Brunner, D. Cavalcanti, S. Pironio, V. Scarani, and S. Wehner. Bell non-locality. *Reviews of Modern Physics*, 86(419), 2014.

[4] Richard Cleve, Peter Hoyer, Ben Toner, and John Watrous. Consequences and Limits of Nonlocal Strategies. pages 1–25, 2004.

[5] Samir Kunkri, Guruprasad Kar, Sibasish Ghosh, and Anirban Roy. Winning strategies for pseudo-telepathy games using single non-local box. *Strategy*, pages 1–9, 2008.

[6] N. David Mermin. Hidden variables and the two theorems of John Bell. *Reviews of Modern Physics*, 65(3):803–815, 1993.