PAPER

SimulaQron—a simulator for developing quantum internet software

View the article online for updates and enhancements.

Recent citations

- Methods for classically simulating noisy networked quantum architectures
  Iskren Vankov *et al*

Quantum Science and Technology

PAPER

SimulaQron—a simulator for developing quantum internet software

Axel Dahlberg ⓘ **and Stephanie Wehner**

QuTech, Lorentzweg 1, 2628 CJ Delft, Netherlands

E-mail: e.a.dahlberg@tudelft.nl

Abstract

We introduce a simulator of a quantum internet with the specific goal to support software development. A quantum internet consists of local quantum processors, which are interc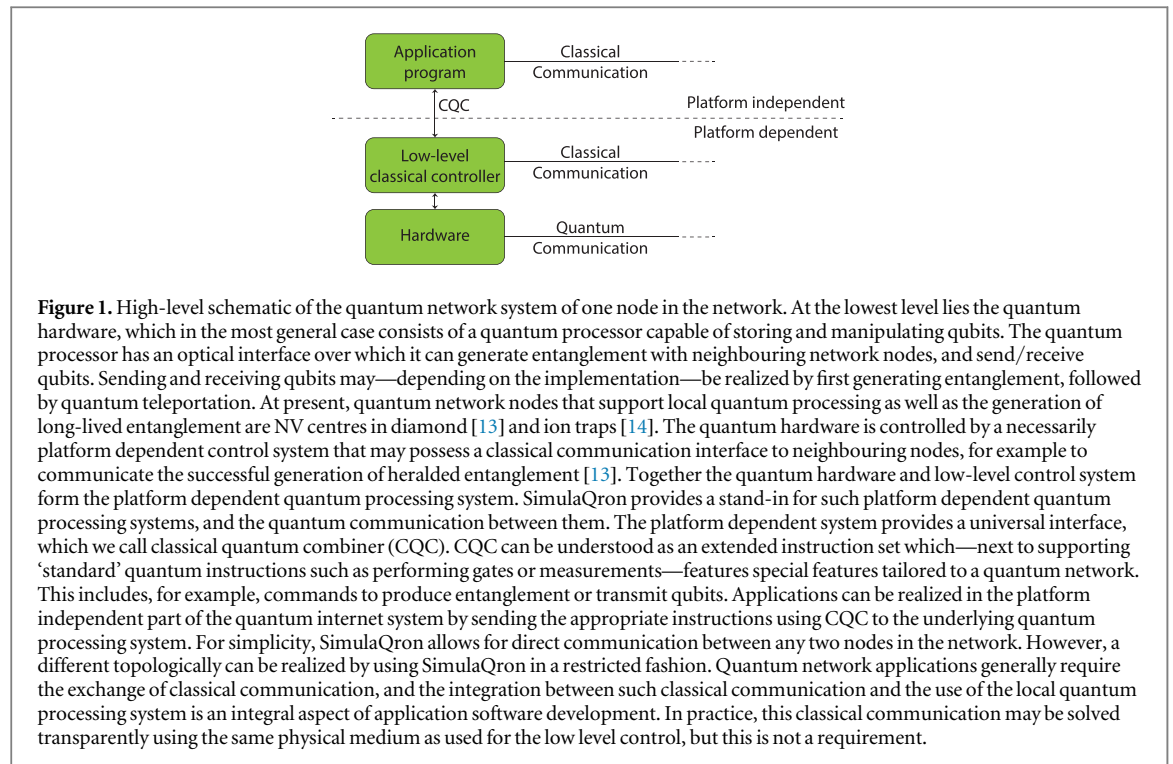onnected by quantum communication channels that enable the transmission of qubits between the different processors. While many simulators exist for local quantum processors, there is presently no simulator for a quantum internet tailored towards software development. Quantum internet protocols require both classical as well as quantum information to be exchanged between the network nodes, next to the execution of gates and measurements on a local quantum processor. This requires quantum internet software to integrate classical communication programming practises with novel quantum ones. SimulaQron is built to enable application development and explore software engineering practises for a quantum internet. SimulaQron can be run on one or more classical computers to simulate local quantum processors, which are transparently connected in the background to enable the transmission of qubits or the generation of entanglement between remote processors. Application software can access the simulated local quantum processors to execute local quantum instructions and measurements, but also to transmit qubits to remote nodes in the network. SimulaQron features a modular design that performs a distributed simulation based on any existing simulation of a quantum computer capable of integrating with Python. Programming libraries for Python and C are provided to facilitate application development.

1. Introduction

A quantum internet enables quantum communication between remote quantum processors in order to solve problems that are infeasible classically. Many applications of a quantum internet are already known, the most famous of which is quantum key distribution (QKD) [1, 2] which allows two network nodes to establish an encryption key. Examples of other applications are secure identification [3] and other two-party cryptographic tasks [4], clock synchronization [5], secure delegated quantum computation [6], and even extending the baseline of telescopes [7]. For many of these applications, only relatively simple quantum processors capable of operating on a handful of qubits are required, as they draw their power from quantum entanglement. Entanglement can be realized using already one qubit at each end point, and its capabilities cannot be replicated using classical communication. In the case of QKD, for example, quantum processors capable only of preparing and measuring single qubits can already be sufficient [1].

The first quantum networks that connect remote quantum processors capable of operating on several qubits each are expected to be deployed within the coming years. End-nodes, holding such quantum processors and on which applications run, will be able to send qubits and generate entanglement between each other using the network. Depending on the distance between the end-nodes, different ways to realize the quantum communication can be used, including the use of quantum repeaters [8–12]. In order to execute arbitrary quantum internet applications on these networks, it is essential to create a development framework in which software for these applications, running on the end-nodes, can be written and debugged. Writing software for a quantum internet shares some similarities with programming a quantum computer, but in addition poses new challenges. Similar to programming a quantum computer, we wish to execute quantum gates and measurements

Figure 1. High-level schematic of the quantum network system of one node in the network. At the lowest level lies the quantum hardware, which in the most general case consists of a quantum processor capable of storing and manipulating qubits. The quantum processor has an optical interface over which it can generate entanglement with neighbouring network nodes, and send/receive qubits. Sending and receiving qubits may—depending on the implementation—be realized by first generating entanglement, followed by quantum teleportation. At present, quantum network nodes that support local quantum processing as well as the generation of long-lived entanglement are NV centres in diamond [13] and ion traps [14]. The quantum hardware is controlled by a necessarily platform dependent control system that may possess a classical communication interface to neighbouring nodes, for example to communicate the successful generation of heralded entanglement [13]. Together the quantum hardware and low-level control system form the platform dependent quantum processing system. SimulaQron provides a stand-in for such platform dependent quantum processing systems, and the quantum communication between them. The platform dependent system provides a universal interface, which we call classical quantum combiner (CQC). CQC can be understood as an extended instruction set which—next to supporting 'standard' quantum instructions such as performing gates or measurements—features special features tailored to a quantum network. This includes, for example, commands to produce entanglement or transmit qubits. Applications can be realized in the platform independent part of the quantum internet system by sending the appropriate instructions using CQC to the underlying quantum processing system. For simplicity, SimulaQron allows for direct communication between any two nodes in the network. However, a different topologically can be realized by using SimulaQron in a restricted fashion. Quantum network applications generally require the exchange of classical communication, and the integration between such classical communication and the use of the local quantum processing system is an integral aspect of application software development. In practice, this classical communication may be solved transparently using the same physical medium as used for the low level control, but this is not a requirement.
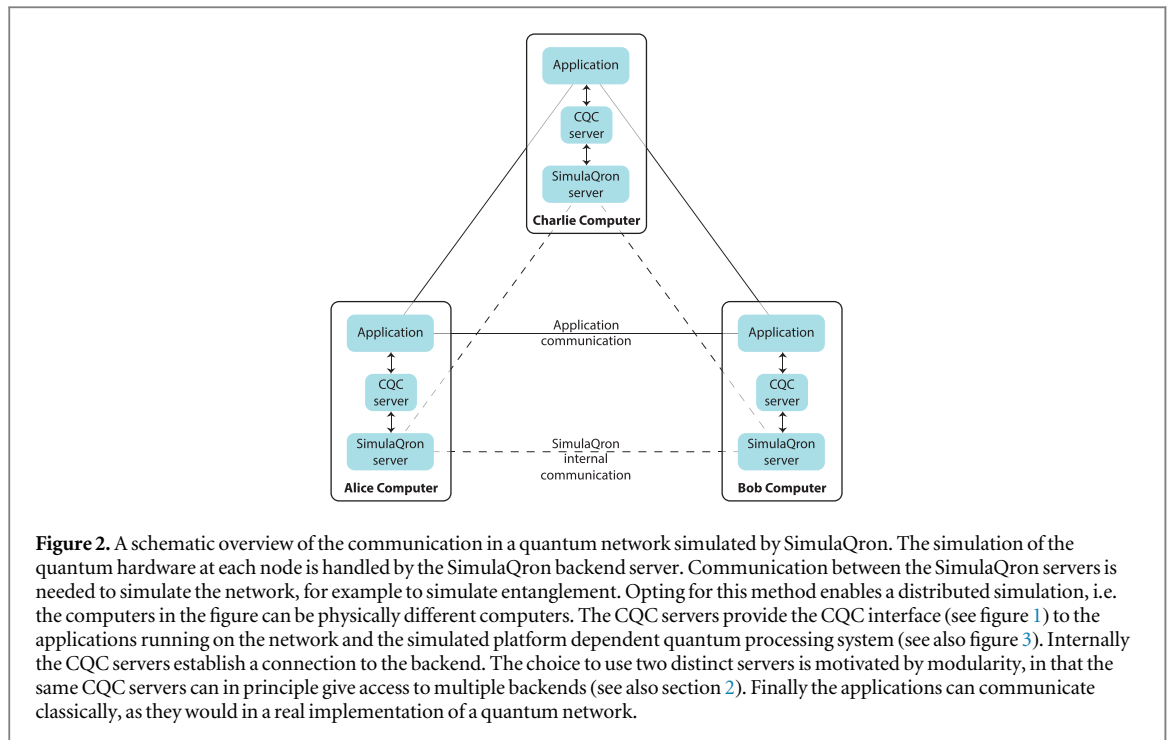
on each local quantum processor. Techniques for optimizing such gates and mapping them to the underlying hardware can be borrowed from quantum computing efforts, and are hence not the subject of this work. What differentiates programming a quantum network is the need for a close integration between classical and quantum messages exchanged during the course of the protocol. The need for such integration arises at several layers of abstraction, and poses significant design challenges. On a lower level, a quantum network stack is needed to create and track entanglement in the network, which requires classical control messages to be exchanged. On a higher level, applications which desire to create and use such entanglement do themselves exchange classical messages during the course of a protocol, requiring standard classical network programming techniques to integrate with quantum ones. At present, a full quantum network stack is missing, and no software development framework for writing quantum network applications exists.

1.1. What SimulaQron does

SimulaQron is a simulator providing a tool for software development for a quantum internet, freely available online [15]. Specifically, SimulaQron simulates several quantum processors held by the end-nodes of the network, connected by a simulated quantum communication channels. This allows the simulation of single quantum networks, as well as inter-connected quantum networks forming a quantum internet. The quantum communication channels between the end-nodes in a real implementation of the network, can be realized in different ways, for example using quantum repeaters. The simulation of quantum repeaters and their performance is an important aspect of developing a quantum network. Numerous simulations of quantum repeaters have been conducted such as for example [16–18], the objective of SimulaQron however is a rather different one in that it aims provide a platform for application development to software engineers. SimulaQron can be used to develop the software running at the end-nodes together with classical communication between these. Figure 1 provides a high level schematic of such a quantum internet system, where SimulaQron should be understood as a stand-in for the quantum processing system (platform dependent) in order to enable platform independent software development to proceed without access to quantum hardware.

SimulaQron precisely mimics a real network, and allows each simulated processor to run on a different classical computer. Each local processing system supports the execution of local quantum gates and measurements, but also network specific commands, for example to generate entanglement. SimulaQron transparently simulates the exchange of qubits and the creation of entanglement between remote processors in the background, making this functionality available to applications. This is achieved by classical communication between the computers hosting the simulated quantum processors, as depicted in the schematic of figure 2.

To perform the simulation of local quantum processors itself, SimulaQron uses existing simulators of quantum processors. What's more, SimulaQron's modular structure does in principle allow any simulation of a quantum computer accessible via Python to serve as a backend. The key novelty in SimulaQron is to leverage

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 backend server. Communication between the SimulaQron servers is 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 the CQC interface (see figure 1) to the applications running on the network and the simulated platform dependent quantum processing system (see also figure 3). Internally the CQC servers establish a connection to the backend. The choice to use two distinct servers is motivated by modularity, in that the same CQC servers can in principle give access to multiple backends (see also section 2). Finally the applications can communicate classically, as they would in a real implementation of a quantum network.

such backends into a distributed simulation that maps locally simulated entangled qubits to remote network nodes, in order to simulate the availability of entanglement between distant quantum computers. We emphasize that applications using SimulaQron's simulated entanglement, evidently do not provide the security guarantees afforded by real entanglement. The objective is instead to use SimulaQron as a development platform to write software realizing these applications which can later run on real quantum hardware and use real entanglement in order to achieve these guarantees. SimulaQron can be used as a tool for software development in all areas ranging from the implementation of the actual applications, the development of application level abstractions and programming libraries, to exploring and implementing a quantum network stack. We remark that there is presently no such stack.

1.2. What SimulaQron does not

We emphasize that SimulaQron is written as a tool for software engineers, with the objective of allowing software engineering efforts for a quantum internet to proceed. Its goal is thus to allow developers to write software that can later run with no or minimal modification on real quantum internet hardware.

For quantum experts, we remark that SimulaQron does not aim to achieve an efficient simulation of a large scale quantum internet in order to test quantum repeater schemes, error correcting codes, or study the effects of noise on distant qubits. Evidently, given that SimulaQron can use any local simulation of a quantum processor in Python as a backend, it is straightforward to let it use a backend simulating noisy qubits. Noise can also be added manually by the application, for example, by probabilistically applying Pauli gates to the qubits during the protocol. We remark however that such adhoc additions of noise do not allow us to make general and accurate statements about the performance of quantum network applications in the presence of noise. Noise in quantum devices is highly time dependent, and hence the amount of noise an application experiences is highly dependent on time delays—for example, how long it takes classical and quantum data to travel from one node to another. If SimulaQron would simply be programmed to apply a time dependent noise based on the message delays during simulation, then this would only be meaningful if these delays mirrored the exact time delays in the real network. What's more, even for executing local gates alone, the execution time of the simulation on the classical computer does not provide reliable timings.

In order to explore the precise effect of noise in a quantum network, it is essential to be able to model time very precisely. This can be achieved using a technique called discrete event simulation that is well known from classical networking. This is subject of a separate simulation platform (NetSquid [19]), which performs a discrete event simulation of a large scale quantum internet capable of precisely modelling timing delays and hence study the effects of delays in quantum communication, as well as classical control communication on the performance of quantum protocols and a quantum network stack. NetSquid is not yet publicly available and differs from SimulaQron in several key aspects apart from its ability to model time: first, it is a monolithic simulation and does not provide a real simulation network on different computers as SimulaQron does. As such, it does not

mirror software engineering practices in networked programming. Second, it does not provide a real time interactive experience of working on a quantum network as provided by SimulaQron. We remark that NetSquid will feature a network emulation mode, in which applications written for SimulaQron can communicate via CQC with a—purely local—simulated quantum network using NetSquid in which time can be tightly controlled. This way their performance can be tested under controlled timing conditions, and realistic time dependent noise models of different hardware platforms.

Also for software engineers, we remark that there are some things which are purposefully not handled by SimulaQron. This includes, for example, the management and tracking of entanglement required by a quantum network stack, which may on the other hand be explored and implemented using SimulaQron. No quantum network stack, nor protocols for managing and tracking entanglement are presently known, and are still undergoing development [20]. Importantly, we also note that for the same reason no efforts have been made to secure or authenticate access to the simulated quantum processors. A SimulaQron backend server will by default happily accept requests from any client connecting to it, and in particular allow access to any qubit given its correct identifiers. It is clear that mechanisms for implementing access control to quantum nodes and qubits are important to realize in software for a quantum internet, highlighting the need for a tool like SimulaQron as a stand-in for quantum hardware in order to develop them.

1.3. Related work

There is to our knowledge no analogue of SimulaQron available for application development on a quantum internet. There are of course numerous simulators of a quantum computers available, some of which could in principle be used as a local simulation backend in conjunction with SimulaQron, with the distributed simulation handled by SimulaQron. Freely available ones include ProjectQ [21] which is written in Python and contains an optimizing compiler. Other simulators include Liquid [22] (available as a binary), Forest by Rigetti [23], QX [24], the simulation backend of the IBM Quantum Experience [25] (can be accessed in Python using QISKit), and the Microsoft Quantum Development Kit [26] (using the programming language Q#). Presently, the quantum backend of SimulaQron is simply realized using QuTip [27], which is not designed to be an efficient simulator, but more than sufficient for the purpose of simulating a small network. Using QuTip also has the advantage of providing a very easy way to extend SimulaQron to operate on noisy instead of perfect qubits, should one wish to gain insights into whether, for example, software performing error correction for a quantum protocol has been implemented correctly. Again, we emphasize that while this allows verification of the correct implementation of such error correction schemes, SimulaQron is not meant to accurately model time dependent noise in the network.
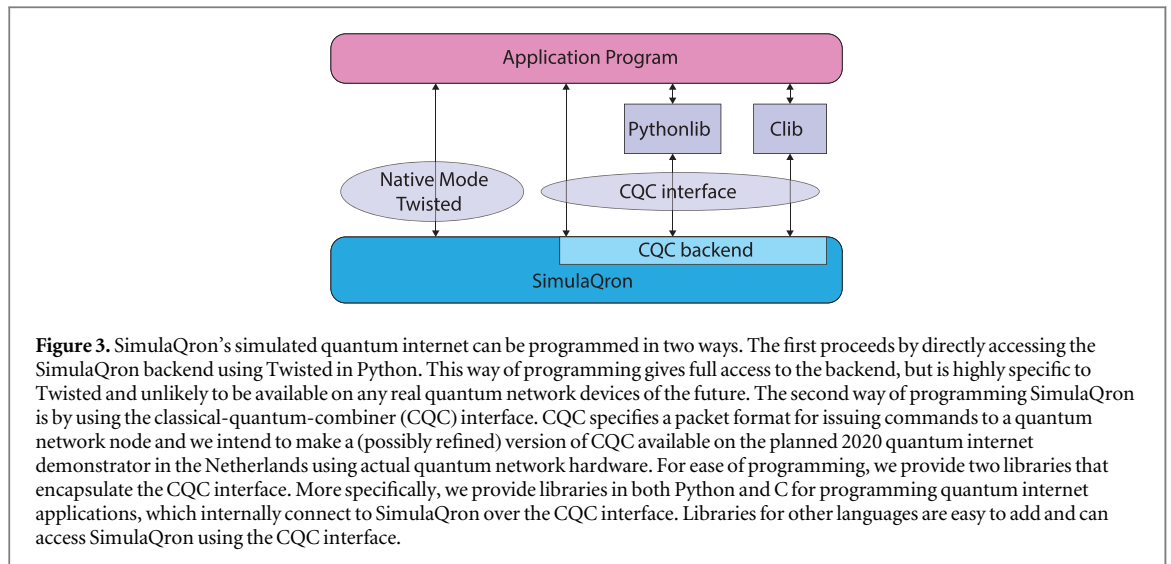
1.4. Overview

SimulaQron itself is written in Python [28] due to the popularity of this language in the quantum information community, making it easier to extend if desired. Internally, SimulaQron also makes use of the Twisted framework for Python [29]. Twisted is a library providing functions that facilitate the development of network applications in Python. SimulaQron's simulated quantum internet can be programmed in two ways as outlined in figure 3. In section 2 we will provide an overview of the design and inner workings of SimulaQron. In section 3 we discuss the integration with the CQC interface. We provide two simple examples on how to program SimulaQron using the Python CQC library in section 4, together with performance analysis of SimulaQron for some test-cases. Further examples, programming templates as well as an API documentation can be found online [15]. Finally, we discuss future developments and extensions of SimulaQron.

2. Backend

Let us now first describe the SimulaQron backend, which can be accessed in Python using Twisted's Perspective Broker [29]. We note that for programming applications there is no need to ever access the backend directly, as the Python library using CQC provides a much easier way to program applications. CQC is an interface (close to) what we intend to make available on the quantum internet demonstrator in the Netherlands that can be accessed from any programming language.

2.1. Challenges

The main challenge in providing a simulator suitable for programming quantum internet applications is to simulate quantum entanglement. Mathematically, any quantum state can be written as a density matrix $\rho \in \mathbb{C}^{d \times d}$, where d is the dimension of the quantum system (see [30], or [31], Weeks 0 and 1 for an introduction to quantum information). Crucially, for two entangled qubits A and B, we have $\rho \neq \sum_j p_j \rho_j^A \otimes \rho_j^B$, with $\rho_j^A \in \mathbb{C}^{2 \times 2}$, $\rho_j^B \in \mathbb{C}^{2 \times 2}$, and $\otimes$ denoting the tensor product. This means that we cannot factorize ρ into

Figure 3. SimulaQron's simulated quantum internet can be programmed in two ways. The first proceeds by directly accessing the SimulaQron backend using Twisted in Python. This way of programming gives full access to the backend, but is highly specific to Twisted and unlikely to be available on any real quantum network devices of the future. The second way of programming SimulaQron is by using the classical-quantum-combiner (CQC) interface. CQC specifies a packet format for issuing commands to a quantum network node and we intend to make a (possibly refined) version of CQC available on the planned 2020 quantum internet demonstrator in the Netherlands using actual quantum network hardware. For ease of programming, we provide two libraries that encapsulate the CQC interface. More specifically, we provide libraries in both Python and C for programming quantum internet applications, which internally connect to SimulaQron over the CQC interface. Libraries for other languages are easy to add and can access SimulaQron using the CQC interface.

different components ρ_j^A and ρ_j^B that could be simulated individually on two different network nodes. Instead, we need to simulate the entire matrix ρ as one, while making qubits A and B virtually available at two different nodes in the network.

One way of achieving this, is to let one simulating node hold all qubits in the network in the same register, consisting of a matrix of dimension $2^n \times 2^n$ where n is the number of qubits. This places a prohibitively large load on the simulating node, preventing large networks to be simulated. One step towards making the simulation more efficient is to still use only one central simulating node, but keep—as much as possible—different registers. For example, by initially placing each qubit in its own register requiring the simulation of only a 2×2 matrix per initial qubit, and only merging registers as qubits become entangled with each other. This improvement is indeed implemented in SimulaQron whenever qubits are created at one node. Nevertheless if this was the only optimization, a significant load would remain at one node in the network.

Consequently, we here go one step further and choose a distributed approach to simulation, described in detail below. In summary, each node keeps a quantum register of its own, in which a number of qubits are simulated. Qubits which are virtually available at other nodes are mapped back to registers simulated at in principle any other node, typically holding the other qubit of any entanglement. This allows the network to grow dynamically, and distributes simulation efforts amongst many computers. It is especially suited to situations where large amounts of fragmented entanglement exists in the network. This is naturally the case if the simulated network is very large but different subsets of nodes are executing protocols between them at any one time. This is well motivated from the usage of the classical internet, where many subsets of a few nodes each communicate with each other, but we do not see all nodes on the internet engaging in a joint application protocol. Another natural example, which we will also consider in a performance analysis below, is given by a situation in which a large number of nodes in a ring keeps entanglement with its neighbours—for example to transmit a qubit by forward teleportation along a chain of nodes. Our analysis shows no significant problems in simulating 60 nodes using 120 qubits on a single desktop machine, while if we had placed all 120 qubits in a single register we would need to keep track of a 2^{120} dimensional matrix.

While a distributed simulation is more efficient and allows a dynamic growth of the network, it also brings new challenges. Quantum gates performed on two qubits can cause two qubits that were previously unentangled to become entangled. Consequently, if the two qubits were previously simulated in two different registers in the network, then a register merge is required. This is made challenging by the fact, that other nodes may simultaneously try and manipulate qubits simulated in said registers, or even perform operations that require a different register merge. It is in general challenging to manage access to distributed data in the presence of many concurrent requests to use it. SimulaQron solves such concurrency issues using a relatively simple but carefully crafted internal locking mechanism.

2.2. Design overview

Together, these considerations motivate the design of the SimulaQron backend depicted in figure 4. The SimulaQron backend consists of running a client and server program on each classical computer—a `virtualNode`—that wants to participate in the simulated quantum network. All such programs connect to each other, forming the simulated quantum internet backend.
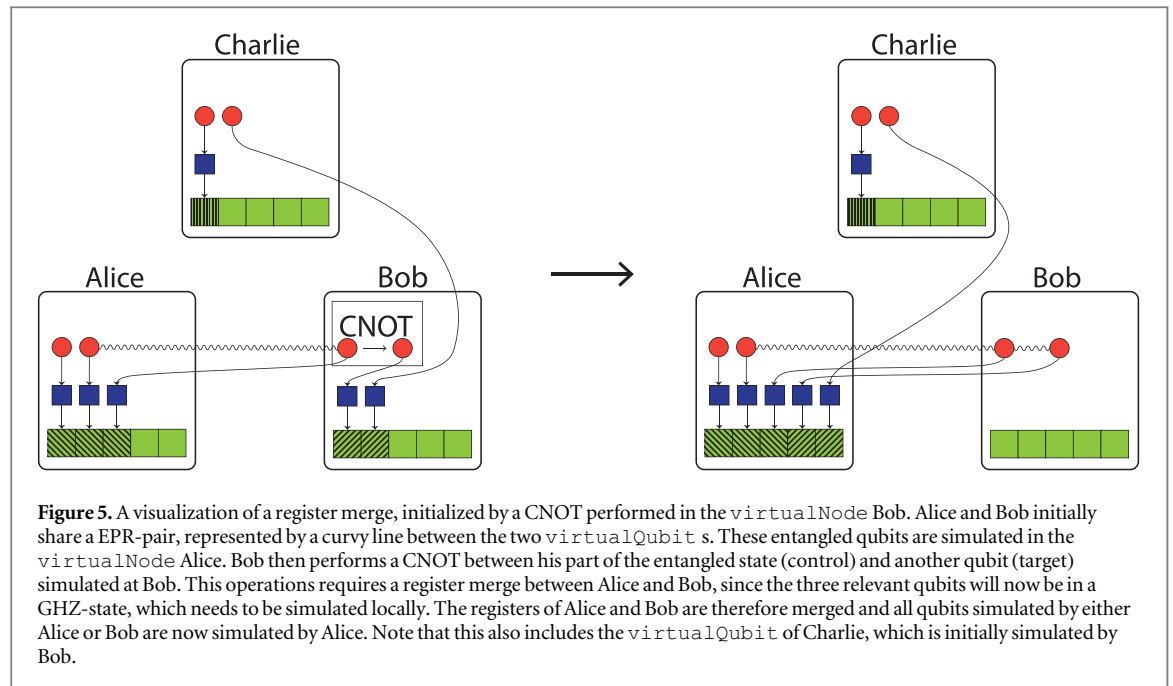
Figure 4. A visualization of the interplay between different internal components of SimulaQron. The `simulatedQubit`s (blue squares) are objects handled locally in a `virtualNode`. These `simulatedQubit`s point to a part of the `quantumRegister`, which stores the quantum state simulated by the `virtualNode`. Operations on the simulated state in the `quantumRegister` are handled by the `quantumEngine`. Additionally, a `virtualNode` also has `virtualQubit`s (red circles). These `virtualQubit`s point to `simulatedQubit`s, possibly in a different `virtualNode`. The `virtualQubit`s correspond to the actual qubits a node would have in a physical implementation of the quantum network.

SimulaQron itself does not provide a new quantum simulator, but rather builds a distributed simulation on top of an existing one using a modular design. In the initial release, we have simply used QuTip for the underlying simulation, but essentially any existing simulator (or even real quantum computing—but not yet networked—hardware) that can be addressed via Python can easily be used in conjunction with SimulaQron. An existing simulator can be made available to SimulaQron as a backend by providing an interface in the form of a `quantumEngine` to function as a `quantumRegister`. This register supports addressing individual qubits by their position in the register, that is, for a register with n qubits. For example, if a Hadamard gate H is requested on qubit j, `quantumRegister` is responsible for applying this operation on the underlying quantum simulation. In the simple case of QuTip, this means simply applying the unitary $\mathrm{id}^{\otimes j-1} \otimes H \otimes \mathrm{id}^{\otimes n-j}$ to the matrix representing ρ (see `crudeSimulator.py`).

Building on top of an underlying `quantumRegister`, SimulaQron uses a `simulatedQubit` object to represent each qubit simulated in the underlying `quantumRegister`. Manipulation of qubits then follows exclusively by manipulating such `simulatedQubit`s without interacting directly with the `quantumRegister`. In particular, each `simulatedQubit` keeps track of its own position in the register, allowing easy manipulation and update of qubits that are physically simulated at a particular node. For example, if a qubit is measured it can be removed from the underlying `quantumRegister`, effectively shrinking the size of the matrices. This allows the simulation to proceed without having the underlying register grow arbitrarily while new qubits are being created and discarded. Removal from the `quantumRegister` can be done by updating only the simulated qubits (and not the corresponding virtual ones that may be held by other nodes in the simulation), and hence the rest of the simulation can proceed to access the simulated qubits without being aware that the underlying register has been shrunk. Figure 4 illustrates the relationship between the `quantumRegister` and the `simulatedQubit`s. We note that `simulatedQubit` objects are local to the node performing their actual simulation in the `quantumRegister`.

2.3. Virtual qubits

Each `simulatedQubit` object is then associated with a `virtualQubit` object. Importantly, a `virtualQubit` does not need to reside at the same network node as the corresponding `simulatedQubit`, but can be transferred to other nodes than the one performing the backend simulation. Specifically, each `virtualNode` can hold a number of `virtualQubit`s which can either be simulated locally (i.e., the simulated qubit and the virtual qubit are located at the same node), or at a remote node. Twisted's Perspective Broker is used to marshal the mapping of virtual qubits back to simulated qubit objects at remote nodes on the simulated network.

This way it is possible for SimulaQron to simulate entanglement between network nodes: if simulated nodes A and B hold entanglement, then each `virtualNode` has a `virtualQubit` that can be processed at nodes A and B as if they were truly entangled. In the background, however, the two virtual qubits are mapped back to two `simulatedQubits` within the same `quantumRegister` that may be located at either A or B, or even at some other node.

Figure 5. A visualization of a register merge, initialized by a CNOT performed in the `virtualNode` Bob. Alice and Bob initially share a EPR-pair, represented by a curvy line between the two `virtualQubit`s. These entangled qubits are simulated in the `virtualNode` Alice. Bob then performs a CNOT between his part of the entangled state (control) and another qubit (target) simulated at Bob. This operations requires a register merge between Alice and Bob, since the three relevant qubits will now be in a GHZ-state, which needs to be simulated locally. The registers of Alice and Bob are therefore merged and all qubits simulated by either Alice or Bob are now simulated by Alice. Note that this also includes the `virtualQubit` of Charlie, which is initially simulated by Bob.

Any application wishing to use SimulaQron consists of a client program at each computer that connects to the SimulaQron server backend. SimulaQron will make virtual qubits available to any client connecting to the `virtualNode` server, as well as allow the creation of new qubits and registers. Using Twisted's Perspective Broker, the client has full access to the specific `virtualQubit`, allowing it to perform gates, measurements, or send the qubit to other nodes in the simulated quantum network.

2.4. Register merges

One simple way to deal with virtual qubits is to have them simulated at one single classical computer in the network. Evidently, this puts a large strain on that specific computer, which then has to simulate every single qubit in the network. Here, we have instead chosen a distributed simulation, which is more challenging to realize, but does allow many nodes to take part in the simulation, as long as the global entanglement in each quantum state is not too large. This is typically the case, when investigating how to program applications that may each run on only a subset of the nodes in the network.

We hence allow any `virtualNode` to hold a `quantumRegister` containing simulated qubits that may eventually be sent as virtual qubits to other nodes in the network. This enables a distributed simulation in which each classical computer participating in the simulated quantum internet contributes its share to realizing the overall simulation. Yet, it is clear that this approach brings additional challenges. Specifically, consider two virtual qubits A and B at one network node, which are mapped back to two simulated qubits in *different* `quantumRegisters`. An example of such a situation is given in figure 5. If the protocol now requests the `virtualNode` to perform an entangling gate between A and B, qubits A and B may now become entangled and can thus no longer be simulated in different registers.

Entangling gates may thus require a register merge in order for the simulation to proceed. More precisely, entangling A and B then requires that the two distinct quantum registers which hold the corresponding simulated qubits to be merged into one single quantum register, followed by an update of the corresponding `simulatedQubit`s that the `virtualQubit`s are mapped to.

SimulaQron solves this problem by transparently merging the registers in the background. For the application dealing with only the `virtualQubit`s such a merge is invisible, but the simulated qubits representing the virtual qubits in question are updated.

It is clear that locking is required to ensure consistency in performing such register merges in case multiple requests to merge a register arise in the network at the same time. At present, SimulaQron implements a very simple and relatively inefficient locking mechanism tailored to acquiring the minimal set of locks required to perform and update. As two separate locks for registers and simulated qubits are used, a deadlock can arise which is dealt with using a simple randomized backoff to marshall competition for locks. Future versions of SimulaQron may be enhanced by a more sophisticated locking mechanism.

Table 1. Options that can be specified when sending a
`COMMAND` message.

Option	Effect
`OPT_NOTIFY`	Send a notify when command is done
`OPT_ACTION`	Execute further commands when done
`OPT_IFTHEN`	Execute further commands based on result
`OPT_BLOCK`	Block until command done

3. CQC

The CQC instruction format provides a low level language programmable at the level of quantum gates and measurements, tailored specifically to include certain commands and behaviour that is useful in the quantum internet domain. The CQC interface provides a way for applications to be developed independently of the underlying platform and for these to be executed on any platform which provide the CQC interface. On top of the backend, SimulaQron realizes a CQC interface (classical–quantum combiner) as implemented by the so-called CQC backend. SimulaQron can therefore be programmed using any language capable of connecting to the CQC server backend over a TCP connection, and sending packets of the required form, specified by the CQC message format. In this section we describe the CQC interface and its messages in more detail.

Two libraries are included to program SimulaQron via the CQC interface in Python and C. Programming SimulaQron via this Python library is the easiest way to make use of SimulaQron and the recommended way to get started using SimulaQron. Below, we provide two simple examples; many more examples can be found in the online documentation [15].

We expect that the present CQC interface will undergo further evolution until its use in the Dutch demonstration network. As such, we describe the main ideas and what functionality exists. The current version of the CQC interface including the precise message format is available online [15] and will be updated when a newer version exist. CQC specifies a particular instruction format for requests and replies from the CQC backend. Requests may be given as a single message to the CQC interface, or an entire batch of messages at once specifying a complex operation consisting of many actions in succession.

Messages in the CQC interface can be of different *types*, as described in section 3.1. One of these types is `COMMAND` which instructs SimulaQron (or the physical hardware) to perform a certain command. This command could for example be a gate or to send a qubit to another node. More details on the message type `COMMAND` is given in section 3.2.

In addition to the type of message and what command to perform, CQC messages will also contain information regarding for example the CQC version, an application identifier, the length of the message (including additional commands if any) and—if applicable—the qubit identifiers a command concerns, the IP and port-number the node a qubit should be sent to. Other options that can be specified is whether notification should be returned when the command is finished and if the node should be blocked during the execution of the command, see table 1.

There is also the possibility to attach a list of commands that should be executed directly after the command specified in the CQC message is completed, or if, for example, a measurement outcome takes on a certain value. As such, it is assumed that the CQC backend provides a rudimentary form of classical logic next to the quantum specific instructions that allow certain simple processing to be executed in the CQC backend, or indeed the actual hardware. That is, these commands can be executed without having to send messages back and fourth through the CQC interface between each command. Avoiding these messages between each command allows the sequence of commands to be executed faster, which is important for a physically implemented quantum network, subject to decoherence.

3.1. Message types

We describe in this section message types currently implemented in the CQC interface. As mentioned above, in future version of the interface, changes may be made and the latest version can be found online. The types of messages sent from the application to the hardware (either simulated by SimulaQron or physically implemented) are specified in table 2.

A `HELLO` message can be used to check that the connection to the hardware/SimulaQron is up and also to get some specifications about the hardware. The commands that can be specified using the type `COMMAND` is discussed in the following section. `FACTORY` is a type similar to `COMMAND`, but here a certain command should be executed a specified number of times. An example of the use of `FACTORY` is to instruct the hardware to

Table 2. Types of messages from the application to the hardware/SimulaQron.

Type	Effect
TP_HELLO	Alive check (get hardware spec.)
TP_COMMAND	Execute (list of) command
TP_FACTORY	Execute (list of) command repeatedly
TP_GET_TIME	Get creation time of qubit

Table 3. Types of messages from the hardware/SimulaQron to the application.

Type	Effect
TP_NEW_OK	Qubit was allocated
TP_EXPIRE	Qubit has expired
TP_DONE	Done with command
TP_RECV	Received qubit
TP_EPR_OK	Created EPR pair
TP_MEASOUT	Measurement outcome
TP_INF_TIME	Return timing information
ERR_GENERAL	General purpose error
ERR_NOQUBIT	No more qubit
ERR_UNKNOWN	Unknown qubit ID
ERR_UNAVAILABLE	Cannot allocate qubit
ERR_DENIED	No access to qubit
ERR_VERSION	CQC version not supported
ERR_UNSUPP	Sequence not supported
ERR_TIMEOUT	Timeout

continuously produce a specified number of entangled EPR-pairs. An EPR-pair could then ready to be used whenever it is needed by the protocol running on the network.

There are also message types returned by the hardware to the application, as specified in table 3.

3.2. Possible commands

The different commands currently supported in the CQC interface when using the message type COMMAND are specified in table 4. The full CQC packet format includes qubit and entanglement identifiers required by the commands below [15]. We remark that we include a command to create pairwise entanglement, since this reflects the way two nodes are entangled by a heralded entanglement generation scheme (see figure 1 for a high level implementation schematic and information). When a qubit is received or entanglement has been generated, a message, TP_RECV or TP_EPR_OK respectively, is returned to the application. This message notifies the application that the command was successful and that possible corrections required by the entanglement generation scheme has been applied.

The angle of rotation for the single-qubit rotations are currently specified by one byte and is therefore discretized to 256 possible angles. When the creation of an EPR pair is requested, a message containing an entanglement identifier will be returned to the application. At present, there is no protocol to track and manage entanglement in a network available. We are currently developing and testing such a protocol, which will be referenced on the SimulaQron website once released [15].

4. Examples

We provide here a simple example of how to program SimulaQron using the Python CQC library (see figure 3). More examples, also using the C CQC library, can be found in the full online documentation [15]. Before running the example presented here, the simulated network we consider needs to be configured and the severs that does the communication in the backend of SimulaQron has to be setup. Information on how to configure the simulated network and setup the servers can be found in the online documentation [15]. In what follows, we will hence assume that the SimulaQron and CQC backends have been setup already, simulating the hardware of two quantum internet nodes labelled Alice and Bob. We remark that the names Alice and Bob are translated by the Python CQC library into IP addresses according the configuration file specified on [15].

Table 4. Commands that can be specified when using the type
`COMMAND`.

Command	Effect
CMD_NEW	Ask to allocate a new qubit
CMD_ALLOCATE	Ask to allocate multiple qubits
CMD_RELEASE	Release qubit to be used by other app.
CMD_RESET	Reset qubit to $|0\rangle$
CMD_MEASURE	Measure qubit (demolition)
CMD_MEASURE_INPLACE	Measure qubit (non-demolition)
CMD_SEND	Send qubit to another node
CMD_RECV	Ask to receive qubit
CMD_EPR	Create EPR pair with another node
CMD_RECV_EPR	Ask to receive half of EPR pair
CMD_SWAP	Entanglement swapping
CMD_I	Identity
CMD_X	Pauli X
CMD_Y	Pauli Y
CMD_Z	Pauli Z
CMD_H	Hadamard
CMD_K	K gate (Z to Y)
CMD_T	T gate
CMD_ROT_X	Rotation around X (multiple of $\frac{2\pi}{256}$)
CMD_ROT_Y	Rotation around Y (multiple of $\frac{2\pi}{256}$)
CMD_ROT_Z	Rotation around Z (multiple of $\frac{2\pi}{256}$)
CMD_CNOT	CNOT (this qubit as control)
CMD_CPHASE	CPHASE (this qubit as control)

4.1. Sending BB84 states

The first example we consider has no classical communication on the application level between Alice and Bob. This implies that we do not need to set up a separate client/server interaction to exchange information at the level of applications. This means that Alice and Bob only connect locally to the CQC backend of SimulaQron, which functions as the quantum hardware at their network node, to issue quantum instructions. In this example, Alice will send a single qubit to Bob.

4.1.1. Code for Alice

The first thing we need to do is to initialize an object called a `CQCConnection`, which does the communication with SimulaQron using the CQC interface. Once this connection is set up, Alice has access to her own locally simulated quantum hardware.

```
from SimulaQron.cqc.pythonLib.cqc import *
# Establish connection to SimulaQron
Alice=CQCConnection("Alice")
```

The argument that `CQCConnection` takes should be the name specified in the configuration file for the CQC-network. Once the connection is set up we can then create our first qubit. The `qubit`-object takes a `CQCConnection` as argument when initialized. When operations are applied to the `qubit`, the `CQCConnection` is used to communicate with SimulaQron, such that the corresponding simulatedQubit is updated.

```
# Create new qubit
q=qubit(Alice)
```

Alice will then send Bob one out of the four states

$$|0\rangle, \quad |1\rangle, \quad |+\rangle, \quad |-\rangle, \tag{1}$$

where $|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$, depending on the choice of the bits h_A and x set in the program. The states in equation (1) are usually called BB84-states and can be described by two binary variables as follows

$$H^{h_A}X^x|0\rangle \qquad h_A, x \in \{0, 1\}, \tag{2}$$

where *H* is the Hadamard operation and *X* is the Pauli-X operation, defined as

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \qquad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \tag{3}$$

The second part of the code on Alice's side will then apply the operation $H^{h_A}X^x$ and send the qubit to Bob. In our simple example, Alice is done using the quantum internet and hence we close the `CQCConnection`.

```
# Determine which BB84 state to use
h_A=1; x = 0
# if x = 1, flip |0> to |1>
if x==1: q.X()
# if h_A==1, convert to Hadamard basis
if h_A==1: q.H()
# Send qubit to Bob
Alice.sendQubit(q, "Bob")
# Close connection to SimulaQron
Alice.close()
```

4.1.2. Code for Bob

We will now describe the code on Bobs side. In our example, Bob does nothing but wait for a qubit to arrive. Once he receives one, he will measure it in the standard- ($\{|0\rangle, |1\rangle\}$) (for $h_B = 0$) or the Hadamard-basis ($\{|+\rangle, |-\rangle\}$) (for $h_B = 1$) and print the measurement outcome. The code on Bobs side can be seen below, where h_B determines that basis Bob measures in.

```
from SimulaQron.cqc.pythonLib.cqc import *
# Establish a connection to SimulaQron
Bob=CQCConnection("Bob")
# Choose which basis we measure in
h_B=1
# Wait to receive a qubit
q=Bob.recvQubit()
# If we chose the Hadamard basis
# to measure in, apply H
if h_B==1: q.H()
# Measure the qubit in the standard
# basis and store the outcome in m
m = q.measure()
# Print measurement outcome
print("Bobs meas. outcome: {}".format(m))
# Close connection to SimulaQron
Bob.close()
```

Note that if $h_A = h_B$ the measurement outcome for Bob will be Alice's choice of *x* with probability one.

4.2. Teleporting a qubit

In our second example, Alice teleports a qubit to Bob. This example demonstrates how to create shared EPR pairs, and also how to perform additional classical communication from Alice and Bob on the application level. This classical communication can be realized using standard client/servers programming. An inefficient, but convenient testing tool, to perform classical communication tool is provided by the Python library that does not require knowledge of classical client/server programming.

4.2.1. Code for Alice

As in the previous example, the first thing that happens is that a `CQCConnection` is initialized to handle the communication to SimulaQron via the CQC interface.

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

By calling the method `createEPR`, Alice makes a request to generate an EPR-pair with Bob, i.e. to generate the state $\frac{1}{\sqrt{2}}(|00\rangle_{AB} + |11\rangle_{AB})$.

```
# Make an EPR pair with Bob
qA=Alice.createEPR("Bob")
```

When the EPR-pair has been generated, Alice prepares another qubit q in the state $|+\rangle = H|0\rangle$, which she wants to teleport to Bob.

```
# Create a qubit to teleport
q=qubit(Alice)
# Prepare the qubit to teleport in |+>
q.H()
```

Alice makes a Bell-measurement between the qubit qA in the EPR-pair shared with Bob and the qubit q. The Bell-measurement is done by applying a CNOT gate with q as control and qA as target, followed by a Hadamard gate on q and measuring both qubits in the standard basis.

```
# Apply the local teleportation operations
q.cnot(qA)
q.H()
# Measure the qubits
a = q.measure()
b=qA.measure()
print("Alice meas. out.: a = {}, b = {}".format(a,b))
```

At this point Alice needs to communicate to Bob what her measurement outcomes was, such that Bob can recover the state which is teleported. This classical communication can be realized by setting up a client/server interaction between Alice and Bob. There is a built-in feature in the Python library that realize 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. For Alice to sent a message to Bob, the method Alice.sendClassical("Bob",msg) is simple called, where msg is the message she wish to send to Bob. The method opens a socket connection to Bob, sends the message and closes the connection again. Note that if this method is never called, a socket connection is never opened.
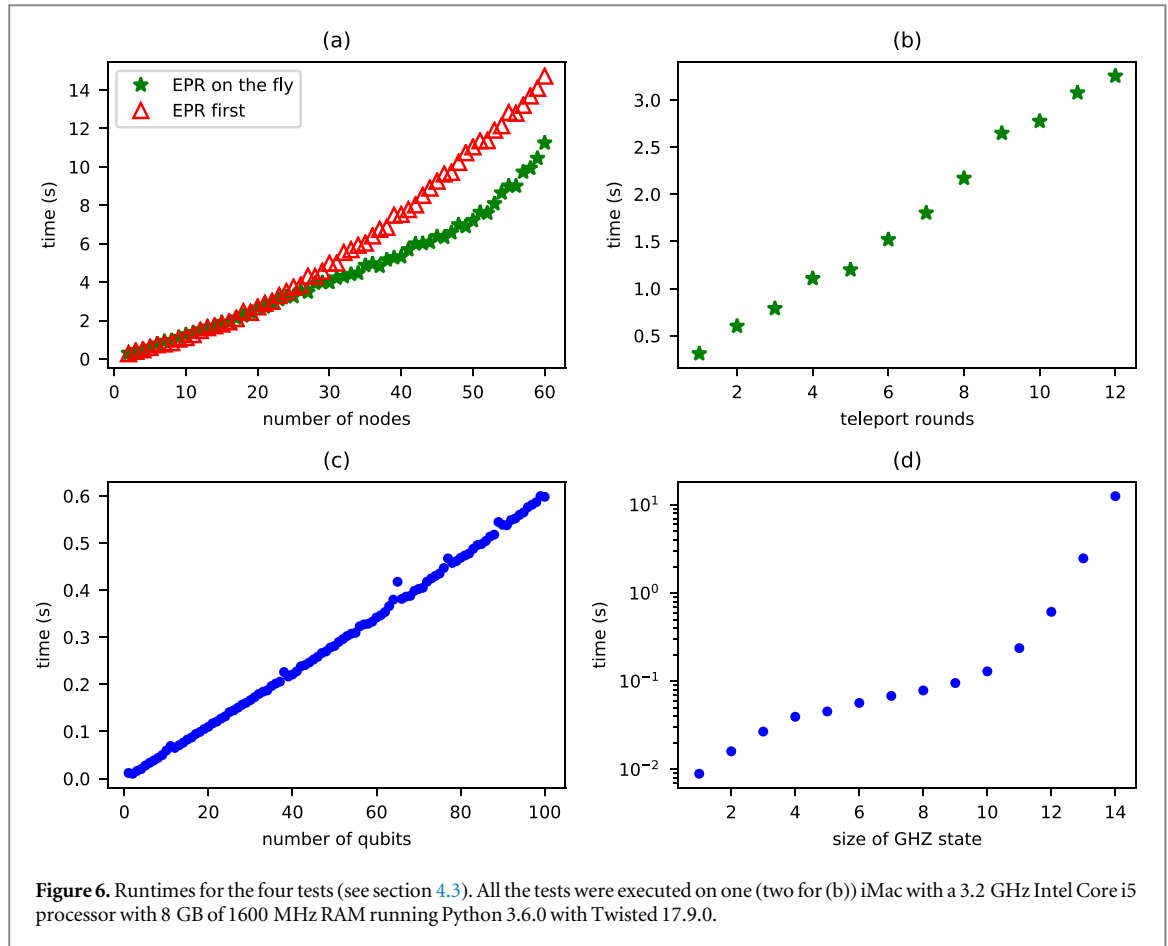
We emphasize that to have classical communication between the applications, one is not forced to use the built-in functionality realized by the CQCConnection. A standard client/server setup can also be used.

```
# Send corrections to Bob
Alice.sendClassical("Bob",[a,b])
# Stop the connections
Alice.close()
```

4.2.2. Code for Bob

As mentioned, Bob will need to know the measurement outcomes from Alice and will therefore setup a server to be able to receive these. Bob will then receive the qubit qB which is part of the EPR-pair generated with Alice. By calling the method recvClassical, Bob receives the measurement outcomes that Alice sent. Corrections are then performed, depending on these measurement outcomes. The qubit qB will then be in the state Alice prepared, i.e. the state $|+\rangle$. Finally, Bob measures the qubit qB which gives 0 or 1 with equal probability.

```
# Initialize the connection
Bob=CQCConnection("Bob")
# Make an EPR pair with Alice
qB=Bob.recvEPR()
# Receive info about corrections
data=Bob.recvClassical()
message=list(data)
a=message[0]
b=message[1]
# Apply corrections
if b==1: qB.X()
if a==1: qB.Z()
# Measure qubit
m=qB.measure()
print("Bob meas. out.: {}".format(m))
# Stop the connection
Bob.close()
```

Figure 6. Runtimes for the four tests (see section 4.3). All the tests were executed on one (two for (b)) iMac with a 3.2 GHz Intel Core i5 processor with 8 GB of 1600 MHz RAM running Python 3.6.0 with Twisted 17.9.0.

4.3. Performance

In this section we present a practical performance analysis of SimulaQron, which has been performed by running the following four tests:

(a) A network with n nodes in a ring is used to teleport a qubit n times from a sender, through all the nodes and back to the sender again. The sender records the time it took for the qubit to traverse the network. We test two different cases, depending on when the nodes create the EPR-pairs for teleportation: (1) Each node creates an EPR-pair with the next node on the ring only once the qubit to be teleported is received. (2) The creation of all EPR-pairs starts in advance. The qubit is teleported forward as soon as the next EPR-pair becomes available. Case (1) is denoted 'EPR on the fly' in figure 6 and case (2) as 'EPR first'. This test is executed on the same computer.

(b) A network consisting of two nodes is used to teleport a qubit back and forth between the nodes n times. This means teleporting a qubit $2n$ times. In this test each node is simulated on its own physical computer, which both are on the same Ethernet network. The time it takes to perform all the teleportations of the qubit is again recorded.

(c) Here we test how much time it takes for one node to initialize n qubits and later measure them. No two-qubit gate, that can entangle the qubits, is used in this test.

(d) One node initializes a GHZ state on n qubits, i.e. $|\text{GHZ}_n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$, and measures all the qubits. The time this takes is recorded.

The runtime for these tests can be found in figure 6. Note that for these simulations there are three processes running for each node: one performing the actual simulation, one listening to incoming CQC messages and the application program which sends the CQC messages, see figure 2. Thus, in the case of 60 nodes in test (a), there are in fact 180 processes running on a single computer. These processes communicate over TCP, which creates some delay in the runtime of the simulation.

Since `quantumRegisters` are only merged when needed, i.e when a two-qubit gate is performed between qubits in different `quantumRegisters`, the runtime highly depends on how large multi-partite entangled

states are produced in the simulation. For example, in test (c) non-entangled qubits are generated in one node and the runtime then scales linearly with the number of qubits. On the other hand in test (d), all qubits are in a single entangled state which requires all qubits to be in the same `quantumRegister` and the runtime then scales super-exponentially with the number of qubits. We emphasize that this runtime is a direct consequence of using QuTip as a backend for the simulation. Creating a GHZ-state directly in QuTip gives the same runtime-scaling as in figure 6(d). Thus, by using a different backend for the simulation, the runtime to create larger entangled states can certainly be improved.

As mentioned, new qubits are always put in different registers and these registers are only merged if a two-qubit gate is performed between qubits in different registers. Currently fixed decisions are used to decide which node will store the register after the merge, depending on which qubit is the target and which is control in the two-qubit gate. There is clearly a room for improvement of the performance, if the direction of the register merge is choosen in a more dynamic way, depending on the protocol being simulated. At this point we do not have enough usage data to know what good decisions for the direction of the register merge are. We also point out that we make no effort in trying to split registers if qubits later become unentangled by some other gate, since this is unlikely to be feasible in general and would require a lot of extra computations.

We emphasize that SimulaQron is not intended to substitute simulators dedicated to handle large multipartite states, as in the case for quantum computation. Despite this, developing software realizing for example distributed quantum computation using SimulaQron, is in principle possible since universal quantum computation is supported by the provided operations. However, SimulaQron's use-cases are for larger (or smaller) networks where far from all nodes are entangled in a single state. Nonetheless, this does not exclude cases where each node share some entanglement with another node, for example as in test (a) where all the nodes share two EPR-pairs with two other nodes, since these EPR-pairs can be simulated in different `quantumRegisters` and does not require a single `quantumRegister` for the whole network.

5. Conclusion

SimulaQron provides access to simulated quantum internet hardware, enabling application development. SimulaQron is undergoing continuous development and improvements and new features may be added over time.

CQC itself can be understood as a very low level language by itself, supported directly by SimulaQron, and tailored specifically to the needs and behaviour of quantum internet application programs. At present, CQC is hidden away from view by the Python and C libraries that enable application development at a much higher level. Indeed, programming using the Python CQC library is the fastest way to get started writing a quantum internet application using SimulaQron. It is conceivable that CQC will be integrated into higher level languages directly by a means of a compiler, which produces CQC commands instead of using a dedicated library.

We might also imagine that more convenient higher level programming environments and libraries become available in the future. While the Python CQC library already offers many conveniences, such as dealing with qubits as Python objects, performing specific gates to—for example—generate BB84 states, it is still a relatively low level approach to programming the quantum internet. Indeed, many protocols depend on exchanging BB84 states, and their creation and processing could in the future be handled by a set of quantum internet software libraries.

Acknowledgments

ORCID iDs

Axel Dahlberg   https://orcid.org/0000-0003-2479-7424

References

[1] Bennett C H and Brassard G 1984 Quantum cryptography: public key distribution, and coin-tossing *Proc. 1984 IEEE Int. Conf. on Computers* pp 175–9
[2] Ekert A K 1991 Quantum cryptography based on Bell's theorem *Phys. Rev. Lett.* **67** 661–3

[3]   Damgård I, Fehr S, Salvail L and Schaffner C 2014 Secure identification and QKD in the bounded-quantum-storage model *Theor. Comput. Sci.* **560** 12–26

[4]   Kaniewski J and Wehner S 2016 Device-independent two-party cryptography secure against sequential attacks *New J. Phys.* **18** 055004

[5]   Jozsa R, Abrams D S, Dowling J P and Williams C P 2000 Quantum clock synchronization based on shared prior entanglement *Phys. Rev. Lett.* **85** 2010–3

[6]   Childs A M 2005 Secure assisted quantum computation *Quantum Inf. Comput.* **5** 456–66

[7]   Kellerer A 2014 Quantum telescopes *Astron. Geophys.* **55** 1–12

[8]   Muralidharan S, Li L, Kim J, Lütkenhaus N, Lukin M D and Jiang L 2016 Optimal architectures for long distance quantum communication *Sci. Rep.* **6** 1–10

[9]   van Meter R 2014 *Quantum Networking* (Chichester: Wiley)

[10] Sangouard N, Simon C, De Riedmatten H and Gisin N 2011 Quantum repeaters based on atomic ensembles and linear optics *Rev. Mod. Phys.* **83** 33–80

[11] Azuma K, Tamaki K and Lo H K 2015 All-photonic quantum repeaters *Nat. Commun.* **6** 1–7

[12] Munro W J, Azuma K, Tamaki K and Nemoto K 2015 Inside quantum repeaters *IEEE J. Sel. Top. Quantum Electron.* **21** 78–90

[13] Humphreys P C, Kalb N, Morits J P J, Schouten R N, Vermeulen R F L, Twitchen D J, Markham M and Hanson R 2018 Deterministic delivery of remote entanglement on a quantum network *Nature* **558** 268–73

[14] Inlek I V, Crocker C, Lichtman M, Sosnova K and Monroe C 2017 Multispecies trapped-ion node for quantum networking *Phys. Rev. Lett.* **118** 1–5

[15] SimulaQron http://simulaqron.org

[16] Rozpędek F, Goodenough K, Ribeiro J, Kalb N, Vivoli V C, Reiserer A, Hanson R, Wehner S and Elkouss D 2018 Parameter regimes for a single sequential quantum repeater *Quantum Sci. Technol.* **3** 34002

[17] Krovi H, Guha S, Dutton Z, Slater J A, Simon C and Tittel W 2016 Practical quantum repeaters with parametric down-conversion sources *Appl. Phys.* B **122** 1–8

[18] Muralidharan S, Kim J, Lütkenhaus N, Lukin M D and Jiang L 2014 Ultrafast and fault-tolerant quantum communication across long distances *Phys. Rev. Lett.* **112** 1–6

[19] NetSQUID: Currently in preparation

[20] Wehner S, Dahlberg A, van Zwet E and Hanson R 2018 A link layer protocol for quantum networks. in preparation

[21] Steiger D S, Häner T and Troyer M 2018 ProjectQ: an open source software framework for quantum computing *Quantum* **2** 49

[22] Wecker D and Svore K M 2014 LIQUi|>: a software design architecture and domain-specific language for quantum computing arXiv:1402.4467

[23] Forest by Rigetti. https://rigetti.com/forest

[24] Khammassi N, Ashraf I, Fu X, Almudever C G and Bertels K 2017 QX: a high-performance quantum computer simulation platform *Design, Automation & Test in Europe Conf. and Exhibition (DATE), 2017* pp 464–9

[25] IBM Q Experience. https://quantumexperience.ng.bluemix.net/qx/experience

[26] Microsoft Quantum Development Kit. https://microsoft.com/en-us/quantum/development-kit

[27] Johansson J R, Nation P D and Nori F 2012 QuTiP: an open-source Python framework for the dynamics of open quantum systems *Comput. Phys. Commun.* **183** 1760–72

[28] Python. https://python.org/

[29] Twisted. https://twistedmatrix.com/trac/

[30] Nielsen M A and Chuang I L 2010 *Quantum Computation and Quantum Information* (Cambridge: Cambridge University Press) p 10

[31] MOOC: Quantum Cryptography on. https://edx.org